Proof-Oriented Programming in F*

Nikhil Swamy, Guido Martínez, and Aseem Rastogi

Jun 02, 2024

CONTENTS:

F* is a dependently typed programming language and proof assistant. This book describes how to use F* for *proof-oriented programming*, a paradigm in which one co-designs programs and proofs to provide mathematical guarantees about various aspects of a program's behavior, including properties like functional correctness (precisely characterizing the input/output behavior of a program), security properties (e.g., ensuring that a program never leaks certain secrets), and bounds on resource usage.

Although a functional programming language at its core, F* promotes programming in a variety of paradigms, including programming with pure, total functions, low-level programming in imperative languages like C and assembly, concurrent programming with shared memory and message-passing, and distributed programming. Built on top of F*'s expressive, dependently typed core logic, no matter which paradigm you choose, proof-oriented programming in F* enables constructing programs with proofs that they behave as intended.

A note on authorship: Many people have contributed to the development of F* over the past decade. Many parts of this book too are based on research papers, libraries, code samples, and language features co-authored with several other people. However, the presentation here, including especially any errors or oversights, are due to the authors. That said, contributions are most welcome and we hope this book will soon include chapters authored by others.

Structure of this book

This book is a work in progress

The first four parts of this book explain the main features of the language using a variety of examples. You should read them sequentially, following along with the associated code samples and exercises. These first four parts are arranged in increasing order of complexity—you can stop after any of them and have a working knowledge of useful fragments of F*.

The remaining parts of the book are more loosely connected and either provide a reference guide to the compiler and libraries, or develop case studies that the reader can choose depending on their interest. Of course, some of those case studies come with prerequisites, e.g., you must have read about effects before tackling the case study on parsers and formatters.

- Part 1: Basic Functional Programming and Proofs

The first part of this book provides a basic introduction to programming with pure total functions, refinement types, and SMT-based proofs, and how to compile and execute your first F* program. This part of the book revises a previous online tutorial on F* and is targeted at an audience familiar with programming, though with no background in formal proofs. Even if you are familiar with program proofs and dependent types, it will be useful to quickly go through this part, since some elements are quite specific to F*.

- Part 2: Inductive Types for Data, Proofs, and Computations

We turn next to inductive type definitions, the main mechanism by which a user can define new data types. F*'s indexed inductive types allow one to capture useful properties of data structures, and dependently types functions over these indexed types can be proven to respect several kinds of invariants. Beyond their use for data structures, inductive data types are used at the core of F*'s logic to model fundamental notions like equality and termination proofs, and can also be used to model and embed other programming paradigms within F*.

- Part 3: Modularity with Interfaces and Typeclasses

We discuss two main abstraction techniques, useful in structuring larger developments: interfaces and typeclasses. Interfaces are a simple information hiding mechanism built in to F*'s module system. Typeclasses are suitable for more advanced developments, providing more flexible abstraction patterns coupled with custom type inference.

- Part 4: Computational Effects

We introduce F*'s effect system, starting with its primitive effects for total, ghost, and divergent computations. We also provide a brief primer on Floyd-Hoare logic and weakest precondition calculi, connecting them to Dijkstra monads, a core concept in the design of F*'s effect system.

- Part 5: Tactics and Metaprogramming

We introduce Meta-F*, the metaprogramming system included in F*. Meta-F* can be used to automate the construction of proofs as well as programmatically construct fragments of F* programs. There's a lot to cover here—the material so far presents the basics of how to get started with using Meta-F* to target specific assertions in your program and to have their proofs be solved using a mixture of tactics and SMT solving.

- Under the hood: F* & SMT

In this part of the book, we cover how F* uses the Z3 SMT solver. We present a brief overview of F*'s SMT encoding paying attention in particular to F* use of fuel to throttle SMT solver's unfolding of recursive functions and inductive type definitions. We also cover a bit of how quantifier instantiation works, how to profile Z3's quantifier instantiation, and some strategies for how to control proofs that are too slow because of excessive quantifier instantiation.

- Planned content

The rest of the book is still in the works, but the planned content is the following:

- Part 4: User-defined Effects
  - State
  - Exceptions
  - Concurrency
  - Algebraic Effects
- Part 5: Tactics and Metaprogramming
  - Reflecting on syntax
  - Holes and proof states
  - Builtin tactics
  - Derived tactics
  - Interactive proofs
  - Custom decision procedures
  - Proofs by reflection
  - Synthesizing programs
  - Tactics for program extraction
- Part 6: F* Libraries
- Part 7: A User's Guide to Structuring and Maintaining F* Developments
  - **The Build System**
    – Dependence Analysis – Checked files – Sample project
  - Using the F* editor
  - **Proofs by normalization**
    - Normalization steps
    - Call-by-name vs. call-by-value
    - Native execution and plugins
  - **Proof Engineering**
    - Building, maintaining and debugging stable proofs
  - **Extraction**

Part I

Introduction

CHAPTER

ONE

A CAPSULE SUMMARY OF F*

F* is a dependently type programming language that aims to play several roles:

- A general purpose programming language, which encourages higher-order functional programming with effects, in the tradition of the ML family of languages.
- A compiler, which translates F* programs to OCaml or F#, and even C or Wasm, for execution.
- A proof assistant, in which to state and prove properties of programs.
- A program verification engine, leveraging SMT solvers to partially automate proofs of programs.
- A metaprogramming system, supporting the programmatic construction of F* programs and proof automation procedures.

To achieve these goals, the design of F* revolves around a few key elements, described below. Not all of this may make sense to you—that's okay, you'll learn about it as we go.

- A core language of total functions with full dependent types, including an extensional form of type conversion, indexed inductive types, and pattern matching, recursive functions with semantic termination checking, dependent refinement types and subtyping, and polymorphism over a predicative hierarchy of universes.
- A system of user-defined indexed effects, for modeling, encapsulating, and statically reasoning about various forms of computational effects, including a primitive notion of general recursion and divergence, as well as an open system of user-defined effects, with examples including state, exceptions, concurrency, algebraic effects, and several others.
- A built-in encoding of a classical fragment of F*'s logic into the first order logic of an SMT solver, allowing many proofs to be automatically discharged.
- A reflection within F* of the syntax and proof state of F*, enabling Meta-F* programs to manipulate F* syntax and proof goals and for users to build proofs interactively with tactics.

1.1 DSLs Embedded in F*

In practice, rather than a single language, the F* ecosystem is also a collection of domain-specific languages (DSLs). A common use of F* is to embed within it programming languages at different levels of abstraction or for specific programming tasks, and for the embedded language to be engineered with domain-specific reasoning, proof automation, and compilation backends. Some examples include:

- Low*, an shallowly embedded DSL for sequential programming against a C-like memory model including explicit memory management on the stack and heap; a Hoare logic for partial correctness based on implicit dynamic frames; and a custom backend (Karamel) to compile Low* programs to C for further compilation by off-the-shelf C compilers.

- EverParse, a shallow embedding of a DSL (layered on top of the Low* DSL) of parser and serializer combinators, for low-level binary formats.
- Vale, a deeply embedded DSL for structured programming in a user-defined assembly language, with a Hoare logic for total correctness, and a printer to emit verified programs in a assembly syntax compatible with various standard assemblers.
- Steel, a shallow embedding of concurrency as an effect in F*, with an extensible concurrent separation logic for partial correctness as a core program logic, and proof automation built using a combination of Meta-F* tactics, higher-order unification, and SMT.
- Pulse, a successor of Steel, a DSL with custom syntax and typechecking algorithm, providing proofs in a small but highly expressive core logic for mutable state and concurrency called PulseCore, formalized entirely in terms of pure and ghost functions in F*.

To get a taste of F*, let's dive right in with some examples. At this stage, we don't expect you to understand these examples in detail, though it should give you a flavor of what is possible with F*.

1.2 F* is a dependently typed language

Dependently typed programming enables one to more precisely capture properties and invariants of a program using types. Here's a classic example: the type `vec a n` represents an `n`-dimensional vector of `a`-typed elements; or, more simply, a list of `n` values each of type `a`. Like other dependently typed languages, F* supports inductively defined definitions of types.

```
type vec (a:Type) : nat -> Type =
  | Nil : vec a 0
  | Cons : #n:nat -> hd:a -> tl:vec a n -> vec a (n + 1)
```

Operations on a vectors can be given types that describe their behavior in terms of vector lengths.

For example, here's a recursive function `append` to concatenate two vectors. Its type shows that the resulting vector has a length that is the sum of the lengths of the input vectors.

```
let rec append #a #n #m (v1:vec a n) (v2:vec a m)
  : vec a (n + m)
  = match v1 with
    | Nil -> v2
    | Cons hd tl -> Cons hd (append tl v2)
```

Of course, once a function like `append` is defined, it can be used to define other operations and its type helps in proving further properties. For example, it's easy to show that reversing a vector does not change its length.

```
let rec reverse #a #n (v:vec a n)
  : vec a n
  = match v with
    | Nil -> Nil
    | Cons hd tl -> append (reverse tl) (Cons hd Nil)
```

Finally, to get an element from a vector, one can program a selector whose type also includes a *refinement type* to specify that the index `i` is less than the length of the vector.

```
let rec get #a #n (i:nat{i < n}) (v:vec a n)
  : a
  = let Cons hd tl = v in
```

(continues on next page)

(continued from previous page)

```
    if i = 0 then hd
    else get (i - 1) tl
```

While examples like this can be programmed in other dependently typed languages, they can often be tedious, due to various technical restrictions. F* provides a core logic with a more flexible notion of equality to make programming and proving easier. For now, a takeaway is that dependently typed programming patterns that are quite technical in other languages are often fairly natural in F*. You'll learn more about this in *a later chapter*.

1.3 F* supports user-defined effectful programming

While functional programming is at the heart of the language, F* is about more than just pure functions. In fact, F* is a Turing complete language. That this is even worth mentioning may come as a surprise to readers with a background in general-purpose programming languages like C# or Scala, but not all dependently typed languages are Turing complete, since nontermination can break soundness. However, F* supports general recursive functions and non-termination in a safe manner, without compromising soundness.

Beyond nontermination, F* supports a system of user-defined computational effects which can be used to model a variety of programming idioms, including things like mutable state, exceptions, concurrency, IO, etc.

Here below is some code in an F* dialect called Low* which provides a sequential, imperative C-like programming model with mutable memory. The function `malloc_copy_free` allocates an array `dest`, copies the contents of an array of bytes `src` into a `dest`, deallocates `src` and returns `dest`.

```
let malloc_copy_free (len:uint32 { 0ul < len })
                     (src:lbuffer len uint8)
  : ST (lbuffer len uint8)
      (requires fun h ->
        live h src /\
        freeable src)
      (ensures fun h0 dest h1 ->
        live h1 dest /\
        (forall (j:uint32). j < len ==> get h0 src j == get h1 dest j))
  = let dest = malloc 0uy len in
    memcpy len 0ul src dest;
    free src;
    dest
```

It'll take us until much later to explain this code in full detail, but here are two main points to take away:

- The type signature of the procedure claims that under specific constraints on a caller, `malloc_copy_free` is *safe* to execute (e.g., it does not read outside the bounds of allocated memory) and that it is *correct* (i.e., that it successfully copies `src` to `dest` without modifying any other memory)
- Given the implementation of a procedure, F* actually builds a mathematical proof that it is safe and correct with respect to its signature.

While other program verifiers offer features similar to what we've used here, a notable thing about F* is that the semantics of programs with side effects (like reading and writing memory) is entirely encoded within F*'s logic using a system of user-defined effects.

Whereas `malloc_copy_free` is programmed in Low* and specified using a particular kind of Floyd-Hoare logic, there's nothing really special about it in F*.

Here, for example, is a concurrent program in another user-defined F* dialect called Steel. It increments two heap-allocated references in parallel and is specified for safety and correctness in concurrent separation logic, a different kind of Floyd-Hoare logic than the one we used for `malloc_copy_free`.

```
let par_incr (#v0 #v1:erased int) (r0 r1:ref int)
  : SteelT _ (pts_to r0 v0 `star` pts_to r1 v1)
             (fun _ -> pts_to r0 (v0 + 1) `star` pts_to r1 (v1 + 1))
  = par (incr r0) (incr r1)
```

As an F* user, you can choose a programming model and a suite of program proof abstractions to match your needs. You'll learn more about this in the section on *user-defined effects*.

1.4 F* proofs use SMT solving, symbolic computation and tactics

Stating a theorem or lemma in F* amounts to declaring a type signature and a doing a proof corresponds to providing an implementation of that signature. Proving theorems can take a fair bit of work by a human and F* seeks to reduce that burden, using a variety of techniques.

SMT Solving

Proving even a simple program often involves proving dozens or hundreds of small facts, e.g., proving that bounded arithmetic doesn't overflow, or that ill-defined operations like divisions by zero never occur. All these little proofs can quickly overwhelm a user.

The main workhorse for proofs in F* is an automated theorem prover, known as a *Satisfiability Modulo Theories*, or SMT, solver. The F* toolchain integrates the Z3 SMT Solver.

By default, the F* typechecker collects all the facts that must be proven in a program and encodes them to the SMT solver, an engine that is capable of solving problems in various combinations of mathematical logics—F* encodes problems to Z3 in a combination of first-order logic, with uninterpreted functions and integer arithmetic.

Z3 is remarkably effective at solving the kinds of problems that F* generates for it. The result is that some F* programs enjoy a high level of automation, e.g., in `memcpy`, we specified a pre- and postcondition and a loop invariant, and the system took care of all the remaining proofs.

You'll learn more about how to use leverage Z3 to prove theorems in F* in *this chapter*.

That said, Z3 cannot solve all problems that F* feeds to it. As such, F* offers several other mechanisms with varying levels of user control.

Symbolic computation

SMT solvers are great at proofs that involve equational rewriting, but many proofs can be done simply by computation. In fact, proofs by computation are a distinctive feature of many dependently typed languages and F* is no exception.

As a very simple example, consider proving that `pow2 12 == 4096`, where `pow2` is the recursive function shown below.

```
let rec pow2 (n:nat) : nat =
  if n = 0 then 1
  else 2 * pow2 (n - 1)

let proof_by_normalization ()
  : Lemma (pow2 12 == 4096)
  = normalize_term_spec (pow2 12)
```

An easy way to convince F* of this fact is to ask it (using `normalize_term_spec`) to simply compute the result of `pow2 12` on an interpreter that's part of the F* toolchain, which it can do instantly, rather than relying on an SMT solvers expensive equational machinery to encode the reduction of a recursive function.

This reduction machinery (called the *normalizer*) is capable not only of fully computing terms like `pow2 12` to a result, but it can also partially reduce symbolic F* terms, as shown in the proof below.

```
let partially_reduce_fold_right f more
  : (fold_right f ([1;2;3]@more) 0 == f 1 (f 2 (f 3 (fold_right f more 0))))
  =  _ by (T.trefl())
```

The proof invokes the F* normalizer from a tactic called `T.trefl`, another F* feature that we'll review quickly, next.

Tactics and Metaprogramming

Finally, for complete control over a proof, F* includes a powerful tactic and metaprogramming system.

Here's a simple example of an interactive proof of a simple fact about propositions using F* tactics.

```
let a_very_explicit_tactic_proof (a b : prop) : (a ==> b ==> b /\ a)
  = _ by
       (let ha = implies_intro () in
        let hb = implies_intro () in
        split ();
        hyp hb;
        hyp ha;
        qed ())
```

This style of proof is similar to what you might find in systems like Coq or Lean. An F* tactic is just an F* program that can manipulate F* proof states. In this case, to prove the theorem `a ==> b ==> (b /\ a)`, we apply commands to transform the proof state by applying the rules of propositional logic, building a proof of the theorem.

Tactics are an instance of a more general metaprogramming system in F*, which allows an F* program to generate other F* programs.

1.5 F* programs compile to OCaml and F#, C and Wasm

Of course, you'll want a way to actually execute the programs you write. For this, F* provides several ways to compile a program to other languages for execution, including support to compile programs to OCaml, F#, C and Wasm.

As such, a common way to use F* is to develop critical components of larger software systems in it, use its proof-oriented facilities to obtain assurances about those components, and then to integrate those formally proven components into a larger system by compiling the F* program to C, OCaml, or F# and linking the pieces together.

In this case, using a tool called KaRaMeL, a compiler used with F*, we can produce the following C code for `memcpy`.

```
uint8_t *MemCpy_malloc_copy_free(uint32_t len, uint8_t *src)
{
  KRML_CHECK_SIZE(sizeof (uint8_t), len);
  uint8_t *dest = KRML_HOST_CALLOC(len, sizeof (uint8_t));
  MemCpy_memcpy(len, (uint32_t)0U, src, dest);
  KRML_HOST_FREE(src);
  return dest;
}
```

Notice that the code we get contains no additional runtime checks: the detailed requires and ensures clauses are all gone and what's left is just a plain C code. Later we'll see how to actually write loops, so that you're not left with recursive functions in C. The point is that all the proof and specification effort is done before the program is compiled, imposing no runtime overhead at all.

1.6 To F*, or not to F*?

We've quickly seen a bit of what F* has to offer—that may have been bit overwhelming, if you're new to program proofs. So, you may be wondering now about whether it's worth learning F* or not. Here are some things to consider.

If you like programming and want to get better at it, no matter what your level is, learning about program proofs will help. Proving a program, or even just writing down a specification for it, forces you to think about aspects of your program that you may never have considered before. There are many excellent resources available to learn about program proofs, using a variety of other tools, including some of the following:

- Software Foundations: A comprehensive overview of programming language semantics and formal proofs in the Coq proof assistant.
- A Proof Assistant for Higher-Order Logic: A tutorial on the Isabelle/HOL proof assistant.
- Certified Programming with Dependent Types: Provides an introduction to proof engineering in Coq.
- Type-driven Development: Introduces using dependent types to developing programs correctly in Idris.
- Theorem Proving in Lean: This is the standard reference for learning about the Lean theorem prover, though there are several other resources too.
- Dafny resources: A different flavor than all of the above, Dafny is an SMT powered program verifier for imperative programs.
- Liquid Haskell: This tutorial showcases proving programs with refinement types.

All of these are excellent resources and each tool has unique offerings. This book about F* offers a few unique things too. We discuss a few pros and cons, next.

Dependent Types and Extensionality

F*'s dependent types are similar in expressiveness to Coq, Lean, Agda, or Idris, i.e., the expressive power allows formalizing nearly all kinds of mathematics. What sets F* apart from these other languages (and more like Nuprl) is its extensional notion of type equality, making many programming patterns significantly smoother in F* (cf. the *vector* example). However, this design also makes typechecking in F* undecidable. The practical consequences of this are that F* typechecker can time-out and refuse to accept your program. Other dependently typed languages have decidable typechecking, though they can, in principle, take arbitrarily long to decide whether or not your program is type correct.

A Variety of Proof Automation Tools

F*'s use of an SMT solver for proof automation is unique among languages with dependent types, though in return, one needs to also trust the combination of F* and Z3 to believe in the validity of an F* proof. Isabelle/HOL provides similar SMT-assisted automation (in its Sledgehammer tool), for the weaker logic provided by HOL, though Sledgehammer's design ensures that the SMT solver need not be trusted. F*'s use of SMT is also similar to what program verifiers like Dafny and Liquid Haskell offer. However, unlike their SMT-only proof strategies, F*, like Coq and Lean, also provides symbolic reduction, tactics, and metaprogramming. That said, F*'s tactic and metaprogramming engines are less mature than other systems where tactics are the primary way of conducting proofs.

A Focus on Programming

Other dependently typed languages shine in their usage in formalizing mathematics—Lean's mathlib and Coq's Mathematical Components are two great examples. In comparison, to date, relatively little pure mathematics has been formalized in F*. Rather, F*, with its focus on effectful programming and compilation to mainstream languages like C, has been used to it produce industrial-grade high-assurance software, deployed in settings like the Windows and Linux kernels, among many others.

Maturity and Community

Isabelle/HOL and Coq are mature tools that have been developed and maintained for many decades, have strong user communities in academia, and many sources of documentation. Lean's community is growing fast and also has excellent tools and documentation. F* is less mature, its design has been the subject of several research papers, making it somewhat more experimental. The F* community is also smaller, its documentation is more sparse, and F* users are usually in relatively close proximity to the F* development team. However, F* developments also have a good and growing track record of industrial adoption.

1.7 A Bit of F* History

F* is an open source project at GitHub by researchers at a number of institutions, including Microsoft Research, MSR-Inria, Inria, Rosario, and Carnegie-Mellon.

The name The F in F* is a homage to System F (https://en.wikipedia.org/wiki/System_F) which was the base calculus of an early version of F*. We've moved beyond it for some years now, however. The F part of the name is also derived from several prior languages that many authors of F* worked on, including Fable, F7, F9, F5, FX, and even F#.

The "*" was meant as a kind of fixpoint operator, and F* was meant to be a sort of fixpoint of all those languages. The first version of F* also had affine types and part of the intention then was to use affine types to encode separation logic—so the "*" was also meant to evoke the separation logic "*". But, the early affine versions of F* never really did have separation logic. It took until almost a decade later to have a separation logic embedded in F* (see Steel), though without relying on affine types.

Part II

Programming and Proving with Total Functions

The core design philosophy of F* is that the type of a term (a program fragment) is a specification of its runtime behavior. We write `e : t` to mean that a term `e` has type `t`. Many terms can have the same type and the same term can have many types.

One (naive but useful) mental model is to think of a type as describing a set of values. For instance, the type `int` describes the set of terms which compute integer results, i.e., when you have `e : int`, then when `e` is reduced fully it produces a value in the set `{..., -2, -1, 0, 1, 2, ...}`. Similarly, the type `bool` is the type of terms that compute or evaluate to one of the values in the set `{true,false}`. Unlike many other languages, F* allows defining types that describe arbitrary sets of values, e.g., the type that contains only the number `17`, or the type of functions that factor a number into its primes.

When proving a program `e` correct, one starts by specifying the properties one is interested in as a type `t` and then trying to convince F* that `e` has type `t`, i.e., deriving `e : t`.

The idea of using a type to specify properties of a program has deep roots in the connections between logic and computation. You may find it interesting to read about propositions as types, a concept with many deep mathematical and philosophical implications. For now, it suffices to think of a type `t` as a specification, or a statement of a theorem, and `e : t` as computer-checkable claim that the term `e` is a proof of the theorem `t`.

In the next few chapters we'll learn about how to program total functions and prove them correct.

CHAPTER

TWO

GETTING OFF THE GROUND

To start writing some F* programs, we'll need to learn some basics about the syntax of the language and some core concepts of types and functions.

2.1 Text Editors

F* can be used as a command line tool with any text editor. If you're viewing this in the interactive online tutorial, you can use the Ace-based text editor alongside, which provides some basic conveniences like syntax highlighting. However, beyond casual use, most users of F* rely on one of the follwing IDE plugins.

- fstar-mode.el, which provides several utilities for interactively editing and checking F* files in emacs.
- fstar-vscode-assistant, which also provides interactive editing and checking support in VS Code.

The main benefit to using these IDE plugins is that they allow you to incrementally check just the changing suffix of an F* file, rather than rechecking the entire file in batch mode. They also provide standard things like jumping to definitions, type of a symbol etc.

Both these plugins rely on a generic but custom interaction protocol implemented by the F* compiler. It should be possible to implement IDE support similar to fstar-mode.el or fstar-vscode-assistant in your favorite plugin-capable editor.

2.2 Basic syntactic structure

An F* program is a collection of modules, with each module represented by a single file with the filename extension `.fst`. Later, we'll see that a module's interface is in a separate `.fsti` file and allows hiding details of a module's implementation from a client module.

A module begins with the module's name (which must match the name of its file, i.e., `module A` is in `A.fst`) and contains a sequence of top-level signatures and definitions. Module names always begin with a capital letter.

- Signatures ascribe a type to a definition, e.g., `val f :  t`.

Definitions come in several flavors: the two main forms we'll focus on when programming with total functions are

- possibly recursive definitions (let bindings, `let [rec] f = e`)
- and, inductive type definitions (datatypes, `type t = | D1 :  t1 | ... | Dn :  tn`)

In later sections, we'll see two other kinds of definition: user-defined indexed effects and sub-effects.

2.3 Comments

Block comments are delimited by `(*` and `*)`. Line comments begin with `//`.

```
(* this is a
   block comment *)


//This is a line comment
```

2.4 Primitives

Every F* program is checked in the context of some ambient primitive definitions taken from the core F* module `Prims`.

2.4.1 False

The type `False` has no elements. Since there are no terms that satisfy `e : False`, the type `False` is the type of unprovable propositions.

2.4.2 Unit

The type `unit` has a single element denoted `()`, i.e., `() : unit`.

2.4.3 Booleans

The type `bool` has two elements, `true` and `false`. Note, the lowercase `false` is a boolean constant, distinct from the uppercase `False` type.

The following primitive boolean operators are available, in decreasing order of precedence.

- `not`: Boolean negation (unary, prefix)
- `&&`: Boolean conjunction (binary, infix)
- `||`: Boolean disjunction (binary, infix)

Conditionals

You can, of course, branch on a boolean with `if/then/else`

```
if b then 1 else 0

if b1 && b2 || b3
then 17
else 42
```

2.4.4 Integers

The type `int` represents unbounded, primitive mathematical integers. Its elements are formed from the literals `0, 1, 2, ...`, and the following primitive operators, in decreasing order of precedence.

- `-`: Unary negation (prefix)
- `-`: Subtraction (infix)
- `+`: Addition (infix)
- `/`: Euclidean division (infix)
- `%`: Euclidean modulus (infix)
- `op_Multiply`: Unfortunately, the traditional multiplication symbol `*` is reserved by default for the tuple type constructor. Use the module `FStar.Mul` to treat `*` as integer multiplication.
- < : Less than (infix)
- <=: Less than or equal (infix)
- > : Greater than (infix)
- >=: Greater than or equal (infix)

Note: F* follows the OCaml style of no negative integer literals, instead negate a positive integer like `(- 1)`.

2.5 Boolean refinement types

The F* core library, `Prims`, defines the type of natural numbers as follows

```
let nat = x:int{x >= 0}
```

This is an instance of a boolean refinement type, whose general form is `x:t { e }` where `t` is a type, and `e` is a `bool`-typed term that may refer to the `t`-typed bound variable `x`. The term `e` *refines* the type `t`, in the sense that the set `S` denoted by `t` is restricted to those elements `x` ∈ `S` for which `e` evaluates to `true`.

That is, the type `nat` describes the set of terms that evaluate to an element of the set `{0, 1, 2, 3, ...}`.

But, there's nothing particularly special about `nat`. You can define arbitrary refinements of your choosing, e.g.,

```
let empty = x:int { false } //the empty set
let zero = x:int{ x = 0 } //the type containing one element `0`
let pos = x:int { x > 0 } //the positive numbers
let neg = x:int { x < 0 } //the negative numbers
let even = x:int { x % 2 = 0 } //the even numbers
let odd = x:int { x % 2 = 1 } //the odd numbers
```

If you're coming from a language like C or Java where a type primarily describes some properties about the representation of data in memory, this view of types as describing arbitrary sets of values may feel a bit alien. But, let it sink in a bit—types that carve out precise sets of values will let you state and check invariants about your programs that may otherwise have only been implicit in your code.

Note: Refinement types in F* trace their lineage to F7, a language developed at Microsoft Research c. 2007 – 2011. Liquid Haskell is another language with refinement types. Those languages provide additional background and resources for learning about refinement types.

Boolean refinements are a special case of a more powerful form of propositional refinement type in F*. Refinement types, in conjunction with dependent function types, are, in principle, sufficient to encode many kinds of logics for program correctness. However, refinement types are just one among several tools in F* for program specification and proof.

2.5.1 Refinement subtyping

We have seen so far how to define a new refinement type, like `nat` or `even`. However, to make use of refinement types we need rules that allow us to:

1. check that a program term has a given refinement type, e.g., to check that `0` has type `nat`. This is sometimes called *introducing* a refinement type.
2. make use of a term that has a refinement type, e.g., given `x : even` we would like to be able to write `x + 1`, treating `x` as an `int` to add `1` to it. This is sometimes called *eliminating* a refinement type.

The technical mechanism in F* that supports both these features is called *refinement subtyping*.

If you're used to a language like Java, C# or some other object-oriented language, you're familiar with the idea of subtyping. A type `t` is a subtype of `s` whenever a program term of type `t` can be safely treated as an `s`. For example, in Java, all object types are subtypes of the type `Object`, the base class of all objects.

For boolean refinement types, the subtyping rules are as follows:

- The type `x:t { p }` is a subtype of `t`. That is, given `e : (x:t{p})`, it is always safe to *eliminate* the refinement and consider `e` to also have type `t`.
- For a term `e` of type `t` (i.e., `e : t`), `t` is a subtype of the boolean refinement type `x:t { p }` whenever `p[e / x]` (`p[e/x]` is notation for the term `p` with the variable `x` replaced by `e`), is provably equal to `true`. In other words, to *introduce* `e : t` at the boolean refinement type `x:t{ p }`, it suffices to prove that the term `p` with `e` substituted for bound variable `x`, evaluates to `true`.

The elimination rule for refinement types (i.e., the first part above) is simple—with our intuition of types as sets, the refinement type `x:t{ p }` *refines* the set corresponding to `t` by the predicate `p`, i.e., the `x:t{ p }` denotes a subset of `t`, so, of course `x:t{ p }` is a subtype of `t`.

The other direction is a bit more subtle: `x:t{ p }` is only a subtype of `p`, for those terms `e` that validate `p`. You're probably also wondering about how to prove that `p[e/x]` evaluates to `true`—we will look at this in detail later. But, the short version is that F*, by default, uses an SMT solver to prove such fact, though you can also use tactics and other techniques to do so.

2.5.2 An example

Given `x:even`, consider proving `x + 1 : odd`; it takes a few steps:

1. The operator + is defined in F*'s library. It expects both its arguments to have type `int` and returns an `int`.
2. To prove that the first argument `x:even` is a valid argument for +, we use refinement subtyping to eliminate the refinement and obtain `x:int`. The second argument `1:int` already has the required type. Thus, `x + 1 : int`.
3. To conclude that `x + 1 : odd`, we need to introduce a refinement type, by proving that the refinement predicate of `odd` evaluates to true, i.e., `x + 1 % 2 = 1`. This is provable by SMT, since we started with the knowledge that `x` is even.

As such, F* applies subtyping repeatedly to introduce and eliminate refinement types, applying it multiple times even to check a simple term like `x + 1 : odd`.

2.6 Functions

We need a way to define functions to start writing interesting programs. In the core of F*, functions behave like functions in maths. In other words, they are defined on their entire domain (i.e., they are total functions and always return a result) and their only observable behavior is the result they return (i.e., they don't have any side effect, like looping forever, or printing a message etc.).

Functions are first-class values in F*, e.g., they can be passed as arguments to other functions and returned as results. While F* provides several ways to define functions, the most basic form is the λ term, also called a function literal, an anonymous function, or a simply a *lambda*. The syntax is largely inherited from OCaml, and this OCaml tutorial provides more details for those unfamiliar with the language. We'll assume a basic familiarity with OCaml-like syntax.

2.6.1 Lambda terms

The term `fun (x:int) -> x + 1` defines a function, a lambda term, which adds 1 to its integer-typed parameter `x`. You can also let F* infer the type of the parameter and write `fun x -> x + 1` instead.

2.6.2 Named functions

Any term in F* can be given a name using a `let` binding. We'll want this to define a function once and to call it many times. For example, all of the following are synonyms and bind the lambda term `fun x -> x + 1` to the name `incr`

```
let incr = fun (x:int) -> x + 1
let incr (x:int) = x + 1
let incr x = x + 1
```

Functions can take several arguments and the result type of a function can also be annotated, if desired

```
let incr (x:int) : int = x + 1
let more_than_twice (x:int) (y:int) : bool = x > y + y
```

It's considered good practice to annotate all the parameters and result type of a named function definition.

Note: In addition to decorating the types of parameters and the results of function, F* allows annotating any term `e` with its expected type `t` by writing `e <:  t`. This is called a *type ascription*. An ascription instructs F* to check that the term `e` has the type `t`. For example, we could have written

```
let incr = fun (x:int) -> (x + 1 <: int)
```

2.6.3 Recursive functions

Recursive functions in F* are always named. To define them, one uses the `let rec` syntax, as shown below.

```
open FStar.Mul
let rec factorial (n:nat)
  : nat
  = if n = 0
    then 1
    else n * factorial (n - 1)
```

This syntax defines a function names factorial with a single parameter n:nat, returning a nat. The definition of factorial is allowed to use the factorial recursively—as we'll see in a later chapter, ensuring that the recursion is well-founded (i.e., all recursive calls terminate) is key to F*'s soundness. However, in this case, the proof of termination is automatic.

Note: Notice the use of *open FStar.Mul* in the example above. This brings the module *FStar.Mul* into scope and resolves the symbol * to integer multiplication.

F* also supports mutual recursion. We'll see that later.

2.7 Arrow types

Functions are the main abstraction facility of any functional language and their types are pervasive in F*. In its most basic form, function types, or arrows, have the shape:

```
x:t0 -> t1
```

This is the type of a function that

1. receives an argument e of type t0, and
2. always returns a value of type t1[e / x], i.e., the type of the returned value depends on the argument e.

It's worth emphasizing how this differs from function types in other languages.

- F*'s arrows are dependent—the type of the result depends on the argument. For example, we can write a function that returns a bool when applied to an even number and returns a string when applied to an odd number. Or, more commonly, a function whose result is one greater than its argument.
- In F*'s core language, all functions are total, i.e., a function call always terminates after consuming a finite but unbounded amount of resources.

Note: That said, on any given computer, it is possible for a function call to fail to return due to resource exhaustion, e.g., running out of memory. Later, as we look at *effects*, we will see that F* also supports writing non-terminating functions.

2.7.1 Some examples and common notation

1. Functions are *curried*. Functions that take multiple arguments are written as functions that take the first argument and return a function that takes the next argument and so on. For instance, the type of integer addition is:

   ```
   val (+) : x:int -> y:int -> int
   ```

2. Not all functions are dependent and the name of the argument can be omitted when it is not needed. For example, here's a more concise way to write the type of (+):

   ```
   val (+) : int -> int -> int
   ```

3. Function types can be mixed with refinement types. For instance, here's the type of integer division—the refinement on the divisor forbids division-by-zero errors:

```
val (/) : int -> (divisor:int { divisor <> 0 }) -> int
```

4. Dependence between the arguments and the result type can be used to state relationships among them. For instance, there are several types for the function `let incr = (fun (x:int) -> x + 1)`:

   ```
   val incr : int -> int
   val incr : x:int -> y:int{y > x}
   val incr : x:int -> y:int{y = x + 1}
   ```

   The first type `int -> int` is its traditional type in languages like OCaml.

   The second type `x:int -> y:int{y > x}` states that the returned value `y` is greater than the argument `x`.

   The third type is the most precise: `x:int -> y:int{y = x + 1}` states that the result `y` is exactly the increment of the argument `x`.

5. It's often convenient to add refinements on arguments in a dependent function type. For instance:

   ```
   val f : x:(x:int{ x >= 1 }) -> y:(y:int{ y > x }) -> z:int{ z > x + y }
   ```

   Since this style is so common, and it is inconvenient to have to bind two names for the parameters `x` and `y`, F* allows (and encourages) you to write:

   ```
   val f : x:int{ x >= 1 } -> y:int{ y > x } -> z:int{ z > x + y }
   ```

6. To emphasize that functions in F*'s core are total functions (i.e., they always return a result), we sometimes annotate the result type with the effect label "`Tot`". This label is optional, but especially as we learn about *effects*, emphasizing that some functions have no effects via the `Tot` label is useful. For example, one might sometimes write:

   ```
   val f : x:int{ x >= 1 } -> y:int{ y > x } -> Tot (z:int{ z > x + y })
   ```

   adding a `Tot` annotation on the last arrow, to indicate that the function has no side effects. One could also write:

   ```
   val f : x:int{ x >= 1 } -> Tot (y:int{ y > x } -> Tot (z:int{ z > x + y }))
   ```

   adding an annotation on the intermediate arrow, though this is not customary.

2.8 Exercises

This first example is just to show you how to run the tool and interpret its output.

```
module Part1.GettingOffTheGround

let incr (x:int) : int = x + 1
```

Notice that the program begins with a `module` declaration. It contains a single definition named `incr`. Definitions that appear at the scope of a module are called "top-level" definitions.

You have several options to try out these examples.

F* online

To get started and for trying small exercises, the easiest way is via the online tutorial. If that's where you're reading this, you can just use the in-browser editor alongside which communicates with an F* instance running in the cloud. Just click on this link to load the code of an exercise in the editor.

That said, the online mode can be a bit slow, depending on the load at the server, and the editor is very minimalistic.

For anything more than small exercises, you should have a working local installation of the F* toolchain, as described next.

F* in batch mode

You can download pre-built F* binaries from here.

Once you have a local installation, to check a program you can run the `fstar` at the command line, like so:

```
$ fstar Sample.fst
```

In response `fstar` should output:

```
Verified module: Sample
All verification conditions discharged successfully
```

This means that F* attempted to verify the module named `Sample`. In doing so, it generated a some “verification conditions”, or proof obligations, necessary to prove that the module is type correct, and that is discharged, or proved, all of them successfully.

F* in emacs

Rather than running `fstar` in batch mode from the command line, F* programmers using the emacs editor often use fstar-mode.el, an editor plugin that allows interactively checking an F* program. If you plan to use F* in any serious way, this is strongly recommended.

2.8.1 Many types for `incr`

Here are some types for `incr`, including some types that are valid and some others that are not.

This type claims that `incr` result is greater than its argument and F* agrees—remember, the `int` type is unbounded, so there’s no danger of the addition overflowing.

```
let incr1 (x:int) : y:int{y > x} = x + 1
```

This type claims that `incr` always returns a natural number, but it isn’t true, since incrementing a negative number doesn’t always produce a non-negative number.

```
let incr2 (x:int) : nat = x + 1
```

F* produces the following error message:

```
Sample.fst(11,26-11,31): (Error 19) Subtyping check failed; expected type
Prims.nat; got type Prims.int; The SMT solver could not prove the query, try to
spell your proof in more detail or increase fuel/ifuel (see also prims.fst(626,
18-626,24))
Verified module: Sample
1 error was reported (see above)
```

Source location

The error message points to `Sample.fst(11,26-11,31)`, a source range mentioned the file name, a starting position (line, column), and an ending position (line, column). In this case, it highlights the `x + 1` term.

Severity and error code

The `(Error 19)` mentions a severity (i.e., `Error`, as opposed to, say, `Warning`), and an error code (19).

Error message

The first part of the message stated what you might expect:

```
Subtyping check failed; expected type Prims.nat; got type Prims.int
```

The rest of the message provides more details, which we'll ignore for now, until we've had a chance to explain more about how F* interacts with the SMT solver. However, one part of the error message is worth pointing out now:

```
(see also prims.fst(626,18-626,24))
```

Error messages sometimes mention an auxiliary source location in a "see also" parenthetical. This source location can provide some more information about why F* rejected a program—in this case, it points to the constraint `x>=0` in the definition of `nat` in `prims.fst`, i.e., this is the particular constraint that F* was not able to prove.

So, let's try again. Here's another type for `incr`, claiming that if its argument is a natural number then so is its result. This time F* is happy.

```
let incr3 (x:nat) : nat = x + 1
```

Sometimes, it is convenient to provide a type signature independently of a definition. Below, the `val incr4` provides only the signature and the subsequent `let incr4` provides the definition—F* checks that the definition is compatible with the signature.

```
val incr4 (x:int) : int
let incr4 x = x + 1
```

Try writing some more types for `incr`. (Load exercise.)

Some answers

```
let incr5 (x:int) : y:int{y = x + 1} = x + 1
let incr6 (x:int) : y:int{x = y - 1} = x + 1
let incr7 (x:int) : y:int{if x%2 = 0 then y%2 = 1 else y%2 = 0} = x + 1
```

2.8.2 Computing the maximum of two integers

Provide an implementation of the following signature:

```
val max (x:int) (y:int) : int
```

There are many possible implementations that satisfy this signature, including trivial ones like:

```
let max x y = 0
```

Provide an implementation of `max` coupled with a type that is precise enough to rule out definitions that do not correctly return the maximum of `x` and `y`.

Some answers

```
val max (x:int) (y:int) : int
let max x y = if x >= y then x else y

val max1 (x:int) (y:int)
  : z:int{ z >= x && z >= y && (z = x || z = y)}
```

(continues on next page)

(continued from previous page)

```
let max1 x y = if x >= y then x else y

let max2 (x:int) (y:int)
  : z:int{ z = max x y }
  = if x > y
    then x
    else y
```

2.8.3 More types for factorial

Recall the definition of `factorial` from earlier.

```
open FStar.Mul
let rec factorial (n:nat)
  : nat
  = if n = 0
    then 1
    else n * factorial (n - 1)
```

Can you write down some more types for factorial?

Some answers

```
let rec factorial1 (n:nat)
  : int
  = if n = 0
    then 1
    else n * factorial1 (n - 1)

let rec factorial2 (n:nat)
  : y:int{y>=1}
  = if n = 0
    then 1
    else n * factorial2 (n - 1)
```

2.8.4 Fibonacci

Here's a doubly recursive function:

```
let rec fibonacci (n:nat)
  : nat
  = if n <= 1
    then 1
    else fibonacci (n - 1) + fibonacci (n - 2)
```

What other types can you give to it?

Some answers

```
val fibonacci_1 : x:int -> y:int{y >= 1 /\ y >= x /\ (if x>=3 then y >= 2 else true)}
let rec fibonacci_1 n =
```

(continues on next page)

(continued from previous page)

```
  if n <= 1 then 1 else fibonacci_1 (n - 1) + fibonacci_1 (n - 2)

(* Try these other types too *)
(* val fibonacci_1 : int -> int *)
(* val fibonacci_1 : int -> nat *)
(* val fibonacci_1 : int -> y:int{y>=1} *)
(* val fibonacci_1 : x:int -> y:int{y>=1 /\ y >= x} *)
(* val fibonacci_1 : int -> Tot (x:nat{x>0}) *)
```

CHAPTER

THREE

POLYMORPHISM AND TYPE INFERENCE

In this chapter, we'll learn about defining type polymorphic functions, or how to work with generic types.

3.1 Type: The type of types

One characteristic of F* (and many other dependently typed languages) is that it treats programs and their types uniformly, all within a single syntactic class. A type system in this style is sometimes called a *Pure Type System* or PTS.

In F* (as in other PTSs) types have types too, functions can take types as arguments and return types as results, etc. In particular, the type of a type is `Type`, e.g., `bool : Type`, `int : Type`, `int -> int : Type` etc. In fact, even `Type` has a type—as we'll see when we learn about *universes*.

3.2 Parametric polymorphism or generics

Most modern typed languages provide a way to write programs with generic types. For instance, C# and Java provide generics, C++ has templates, and languages like OCaml and Haskell have several kinds of polymorphic types.

In F*, writing functions that are generic or polymorphic in types arises naturally as a special case of the *arrow types* that we have already learned about. For example, here's a polymorphic identity function:

```
let id : a:Type -> a -> a = fun a x -> x
```

There are a several things to note here:

- The type of `id` is an arrow type, with two arguments. The first argument is `a : Type`; the second argument is a term of type `a`; and the result also has the same type `a`.
- The definition of `id` is a lambda term with two arguments `a : Type` (corresponding to the first argument type) and `x : a`. The function returns `x`—it's an identity function on the second argument.

Just as with any function, you can write it instead like this:

```
let id (a:Type) (x:a) : a = x
```

To call `id`, one can apply it first to a type and then to a value of that type, as shown below.

```
let _ : bool = id bool true
let _ : bool = id bool false
let _ : int = id int (-1)
let _ : nat = id nat 17
```

(continues on next page)

(continued from previous page)

```
let _ : string = id string "hello"
let _ : int -> int = id (int -> int) (id int)
```

We've defined a function that can be applied to a value `x:a` for any type `a`. The last line there maybe requires a second read: we instantiated `id` to `int -> int` and then applied it to `id` instantiated to `int`.

3.3 Exercises

Let's try a few simple exercises. Click here for the exercise file.

Try defining functions with the following signatures:

```
val apply (a b:Type) (f:a -> b) : a -> b
val compose (a b c:Type) (f: b -> c) (g : a -> b) : a -> c
```

Answer

```
let apply a b f = fun x -> f x
let compose a b c f g = fun x -> f (g x)
```

How about writing down a signature for `twice`:

```
let twice a f x = compose a a a f f x
```

Answer

```
val twice (a:Type) (f: a -> a) (x:a) : a
```

It's quite tedious to have to explicitly provide that first type argument to `id`. Implicit arguments and type inference will help, as we'll see, next.

3.4 Type inference: Basics

Like many other languages in the tradition of Milner's ML, type inference is a central component in F*'s design.

You may be used to type inference in other languages, where one can leave out type annotations (e.g., on variables, or when using type-polymorphic (aka generic) functions) and the compiler determines an appropriate type based on the surrounding program context. F*'s type inference includes such a feature, but is considerably more powerful. Like in other dependently typed language, F*'s inference engine is based on higher-order unification and can be used to infer arbitrary fragments of program text, not just type annotations on variables.

Let's consider our simple example of the definition and use of the identity function again

```
let id (a:Type) (x:a) : a = x
```

```
let _ : bool = id bool true
let _ : bool = id bool false
let _ : int = id int (-1)
let _ : nat = id nat 17
let _ : string = id string "hello"
let _ : int -> int = id (int -> int) (id int)
```

Instead of explicitly providing that first type argument when applying `id`, one could write it as follows, replacing the type arguments with an underscore _.

```
let _ : bool = id _ true
let _ : bool = id _ false
let _ : int = id _ (-1)
let _ : nat = id _ 17
let _ : string = id _ "hello"
let _ : int -> int = id _ (id _)
```

The underscore symbols is a wildcard, or a hole in program, and it's the job of the F* typechecker to fill in the hole.

Note: Program holes are a very powerful concept and form the basis of Meta-F*, the metaprogramming and tactics framework embedded in F*—we'll see more about holes in a later section.

3.5 Implicit arguments

Since it's tedious to write an _ everywhere, F* has a notion of *implicit arguments*. That is, when defining a function, one can add annotations to indicate that certain arguments can be omitted at call sites and left for the typechecker to infer automatically.

For example, one could write

```
let id (#a:Type) (x:a) : a = x
```

decorating the first argument `a` with a `#`, to indicate that it is an implicit argument. Then at call sites, one can simply write:

```
let _ : bool = id true
let _ : bool = id false
let _ : int = id (-1)
let _ : nat = id 17
let _ : string = id "hello"
let _ : int -> int = id id
```

And F* will figure out instantiations for the missing first argument to `id`.

In some cases, it may be useful to actually provide an implicit argument explicitly, rather than relying on the F* to pick one. For example, one could write the following:

```
let _ = id #nat 0
let _ = id #(x:int{x == 0}) 0
let _ = id #(x:int{x <> 1}) 0
```

In each case, we provide the first argument of `id` explicitly, by preceding it with a `#` sign, which instructs F* to take the user's term rather than generating a hole and trying to fill it.

CHAPTER

FOUR

EQUALITY

Equality is a subtle issue that pervades the design of all dependent type theories, and F* is no exception. In this first chapter, we briefly touch upon two different kinds of equality in F*, providing some basic information sufficient for the simplest usages. In a *subsequent chapter*, we'll cover equality in much greater depth.

4.1 Decidable equality and `eqtype`

We've implicitly used the equality operator = already (e.g., when defining `factorial`). This is the *boolean* equality operator. Given two terms $e_1 : t$ and $e_2 : t$, so long as `t` supports a notion of decidable equality, $(e_1 = e_2) : \texttt{bool}$.

To see why not all types support decidably equality, consider `t` to be a function type, like `int -> int`. To decide if two functions $f_1, f_2 : \texttt{int -> int}$ are equal, we'd have to apply them to all the infinitely many integers and compare their results—clearly, this is not decidable.

The type `eqtype` is the type of types that support decidably equality. That is, given $e_1 : t$ and $e_2 : t$, it is only permissible to compare $e_1 = e_2$ if `t : eqtype`.

For any type definition, F* automatically computes whether or not that type is an `eqtype`. We'll explain later exactly how F* decides whether or not a type is an `eqtype`. Roughly, for F* has built-in knowledge that various primitive types like integers and booleans support decidable equality. When defining a new type, F* checks that all values of the new type are composed structurally of terms that support decidable equality. In particular, if an `e : t` may contain a sub-term that is a function, then `t` cannot be an `eqtype`.

As such, the type of the decidable equality operator is

```
val ( = ) (#a:eqtype) (x:a) (y:a) : bool
```

That is, `x = y` is well-typed only when `x : a` and `y : a` and `a : eqtype`.

Note: We see here a bit of F* syntax for defining infix operators. Rather than only using the `val` or `let` notation with alphanumeric identifiers, the notation `( = )` introduces an infix operator defined with non-alphanumeric symbols. You can read more about this here.

4.2 Propositional equality

F* offers another notion of equality, propositional equality, written ==. For *any type* t, given terms $e_1, e_2 : t$, the proposition $e_1 == e_2$ asserts the (possibly undecidable) equality of e_1 and e_2. The type of the propositional equality operator is shown below:

```
val ( == ) (#a:Type) (x:a) (y:a) : prop
```

Unlike decidable equality (=), propositional equality is defined for all types. The result type of (==) is prop, the type of propositions. We'll learn more about that in the *next chapter*.

CHAPTER

FIVE

INTERFACING WITH AN SMT SOLVER

As mentioned *at the start of this section*, a type `t` represents a proposition and a term `e : t` is a proof of `t`. In many other dependently typed languages, exhibiting a term `e : t` is the only way to prove that `t` is valid. In F*, while one can do such proofs, it is not the only way to prove a theorem.

By way of illustration, let's think about *Boolean refinement types*. As we've seen already, it is easy to prove `17 : x:int{x >= 0}` in F*. Under the covers, F* proves that `(x >= 0) [17/x]` reduces to `true`, yet no explicit term is given to prove this fact. Instead, F* encodes facts about a program (including things like the semantics of arithmetic operators like >=) in the classical logic of an SMT solver and asks it (Z3 typically) to prove whether the formula `17 >= 0` is valid in a context including all encoded facts about a program. If Z3 is able to prove it valid, F* accepts the formula as true, without ever constructing a term representing a proof of `17 >= 0`.

This design has many important consequences, including, briefly:

- Trust: F* implicitly trusts its encoding to SMT logic and the correctness of the Z3 solver.
- Proof irrelevance: Since no proof term is constructed for proofs done by SMT, a program cannot distinguish between different proofs of a fact proven by SMT.
- Subtyping: Since no proof term is constructed, a term like 17 can have many types, `int`, `nat`, `x:int{x = 17}`, etc. As mentioned *earlier*, F* leverages this to support refinement subtyping.
- Undecidability: Since Z3 can check the validity of formulas in the entirety of its logic, including things like quantifying universally and existentially over infinite ranges, F* does not restrict the the formulas checked for validity by Z3 to be boolean, or even decidable. Yes, typechecking in F* is undecidable.

In this chapter, we'll learn about the the classical logic parts of F*, i.e., the parts that allow it to interface with an SMT solver.

Note: The beginning of this chapter is a little technical, even though we're not telling the full story behind F*'s classical logic yet. If parts of it are hard to understand right now, here's what you need to know to before you *jump ahead*.

F* let's you write quantified formulas, called propositions, like so

```
forall (x1:t1) ... (xn:tn). p
exists (x1:t1) ... (xn:tn). p
```

You can build propositions from booleans and conjunctions, disjunctions, negations, implications, and bi-implications:

```
p /\ q   //conjunction
p \/ q   //disjunction
~p       //negation
p ==> q  //implication
p <==> q //bi-implication
```

For example, one can say (as shown below) that for all natural numbers `x` and `y`, if the modulus `x % y` is `0`, then there exists a natural number `z` such that `x` is `z * y`.

```
forall (x:nat) (y:nat). x % y = 0 ==> (exists (z:nat). x = z * y)
```

F* also has a notion of propositional equality, written ==, that can be used to state that two terms of any type are equal. In contrast, the boolean equality = can only be used on types that support decidable equality. For instance, for `f1, f2 : int -> int`, you can write `f1 == f2` but you cannot write `f1 = f2`, since two functions cannot be decidably compared for equality.

5.1 Propositions

The type `prop` defined in `Prims` is F*'s type of proof-irrelevant propositions. More informally, `prop` is the type given to facts that are provable using the SMT solver's classical logic.

Propositions defined in `prop` need not be decidable. For example, for a Turing machine `tm`, the fact `halts tm` can be defined as a `prop`, although it is impossible to decide for an arbitrary `tm` whether `tm` halts on all inputs. This is contrast with `bool`, the type of booleans `{true, false}`. Clearly, one could not define `halts tm` as a `bool`, since one would be claiming that for `halts` is function that for any `tm` can decide (by returning true or false) whether or not `tm` halts on all inputs.

F* will implicitly convert a `bool` to a `prop` when needed, since a decidable fact can be turned into a fact that may be undecidable. But, when using propositions, one can define things that cannot be defined in `bool`, including quantified formulae, as we'll see next.

5.2 Propositional connectives

Consider stating that `factorial n` always returns a positive number, when `n:nat`. In the *previous section* we learned that one way to do this is to give `factorial` a type like so.

```
val factorial (n:nat) : x:nat{x > 0}
```

Here's another way to state it:

```
forall (n:nat). factorial n > 0
```

What about stating that `factorial n` can sometimes return a value that's greater than `n * n`?

```
exists (n:nat). factorial n > n * n
```

We've just seen our first use of universal and existential quantifiers.

5.2.1 Quantifiers

A universal quantifier is constructed using the forall keyword. Its syntax has the following shape.

```
forall (x1:t1) ... (xn:tn) . p
```

The x1 ... xn are bound variables and signify the domain over which one the proposition p is quantified. That is, forall (x:t). p is valid when for all v : t the proposition p[v/x] is valid.

And existential quantifier has similar syntax, using the exists keyword.

```
exists (x1:t1) ... (xn:tn) . p
```

In this case, exists (x:t). p is valid when for some v : t the proposition p[v/x] is valid.

The scope of a quantifier extends as far to the right as possible.

As usual in F*, the types on the bound variables can be omitted and F* will infer them. However, in the case of quantified formulas, it's a good idea to write down the types, since the meaning of the quantifier can change significantly depending on the type of the variable. Consider the two propositions below.

```
exists (x:int). x < 0
exists (x:nat). x < 0
```

The first formula is valid by considering x = -1, while the second one is not—there is not natural number less than zero.

It is possible to quantify over any F* type. This makes the quantifiers higher order and dependent. For example, one can write

```
forall (n:nat) (p: (x:nat{x >= n} -> prop)). p n
```

Note: The SMT solver uses a number of heuristics to determine if a quantified proposition is valid. As you start writing more substantial F* programs and proofs, it will become important to learn a bit about these heuristics. We'll cover this in a later chapter. If you're impatient, you can also read about in on the F* wiki.

5.2.2 Conjunction, Disjunction, Negation, Implication

In addition to the quantifiers, you can build propositions by combining them with other propositions, using the operators below, in decreasing order of precedence.

Negation

The proposition ~p is valid if the negation of p is valid. This is similar to the boolean operator not, but applies to propositions rather than just booleans.

Conjunction

The proposition p /\ q is valid if both p and q are valid. This is similar to the boolean operator &&, but applies to propositions rather than just booleans.

Disjunction

The proposition p \/ q is valid if at least one of p and q are valid. This is similar to the boolean operator ||, but applies to propositions rather than just booleans.

Implication

The proposition `p ==> q` is valid if whenever `p` is valid, `q` is also valid.

Double Implication

The proposition `p <==> q` is valid if `p` and `q` are equivalent.

Note: This may come as a surprise, but these precedence rules mean that `p /\ q ==> r` is parsed as `(p /\ q) ==> r` rather than `p /\ (q ==> r)`. When in doubt, use parentheses.

5.3 Atomic propositions

We've shown you how to form new propositions by building them from existing propositions using the connectives. But, what about the basic propositions themselves?

5.3.1 Falsehood

The proposition `False` is always invalid.

5.3.2 Truth

The proposition `True` is always valid.

5.3.3 Propositional equality

We learned in the previous chapter about the *two different forms of equality*. The type of propositional equality is

```
val ( == ) (#a:Type) (x:a) (y:a) : prop
```

Unlike decidable equality `(=)`, propositional equality is defined for all types. The result type of `(==)` is `prop`, the type of propositions, meaning that `x == y` is a proof-irrelevant proposition.

Turning a Boolean into a proposition

Propositional equality provides a convenient way to turn a boolean into a proposition. For any boolean `b`, then term `b == true` is a `prop`. One seldom needs to do write this manually (although it does come up occasionally), since F* will automatically insert a `b==true` if you're using a `b:bool` in a context where a `prop` was expected.

5.3.4 Type vs. `prop`

This next bit is quite technical. Don't worry if you didn't understand it at first. It's enough to know at this stage that, just like automatically converting a boolean to *prop*, F* automatically converts any type to `prop`, when needed. So, you can form new atomic propositions out of types.

Every well-typed term in F* has a type. Even types have types, e.g., the type of `int` is `Type`, i.e., `int : Type`, `bool : Type`, and even `prop : Type`. We'll have to leave a full description of this to a later section, but, for now, we'll just remark that another way to form an atomic proposition is to convert a type to a proposition.

For any type `t : Type`, the type `_:unit { t } : prop`. We call this "squashing" a type. This is so common, that F* provides two mechanisms to support this:

1. All the propositional connectives, like `p /\ q` are designed so that both `p` and `q` can be types (i.e., `p,q : Type`), rather than propositions, and they implicitly squash their types.
2. The standard library, `FStar.Squash`, provides several utilities for manipulating squashed types.

5.4 Assertions

Now that we have a way to write down propositions, how can we ask F* to check if those propositions are valid? There are several ways, the most common of which is an *assertion*. Here's an example:

```
let sqr_is_nat (x:int) : unit = assert (x * x >= 0)
```

This defines a function `sqr_is_nat : int -> unit`—meaning it takes a `nat` and always returns `()`. So, it's not very interesting as a function.

However, it's body contains an assertion that `x * x >= 0`. Now, many programming languages support runtime assertions—code to check some property of program when it executes. But, assertions in F* are different—they are checked by the F* compiler *before* your program is executed.

In this case, the `assert` instructs F* to encode the program to SMT and to ask Z3 if `x * x >= 0` is valid for an arbitrary integer `x:int`. If Z3 can confirm this fact (which it can), then F* accepts the program and no trace of the assertion is left in your program when it executes. Otherwise the program is rejected at compile time. For example, if we were to write

```
let sqr_is_pos (x:int) : unit = assert (x * x > 0)
```

Then, F* complains with the following message:

```
Ch2.fst(5,39-5,50): (Error 19) assertion failed; The SMT solver could not prove the␣
↪query, try to spell your proof in more detail or increase fuel/ifuel
```

You can use an assertion with any proposition, as shown below.

```
let max x y = if x > y then x else y
let _ = assert (max 0 1 = 1)
let _ = assert (forall x y. max x y >= x /\
                            max x y >= y /\
                            (max x y = x \/ max x y = y))
```

5.5 Assumptions

The dual of an assertion is an assumption. Rather than asking F* and Z3 to prove a fact, an assumption allows one to tell F* and Z3 to accept that some proposition is valid. You should use assumptions with care—it's easy to make a mistake and assume a fact that isn't actually true.

The syntax of an assumption is similar to an assertion. Here, below, we write `assume (x <> 0)` to tell F* to assume `x` is non-zero in the rest of the function. That allows F* to prove that the assertion that follows is valid.

```
let sqr_is_pos (x:int) = assume (x <> 0); assert (x * x > 0)
```

Of course, the assertion is not valid for all `x`—it's only valid for those `x` that also validate the preceding assumption.

Just like an `assert`, the type of `assume p` is `unit`.

There's a more powerful form of assumption, called an admit. The term admit() can given any type you like. For example,

```
let sqr_is_pos (x:int) : y:nat{y > 0} = admit()
```

Both assume and admit can be helpful when you're working through a proof, but a proof isn't done until it's free of them.

CHAPTER

SIX

INDUCTIVE TYPES AND PATTERN MATCHING

In this chapter, you'll learn how to define new types in F*. These types are called *inductive types*, or, more informally, datatypes. We'll also learn how to define functions over these inductive types by pattern matching and to prove properties about them.

Note: We'll only cover the most basic forms of inductive types here. In particular, the types we show here will not make use of indexing or any other form of dependent types—we'll leave that for a later chapter.

6.1 Enumerations

We've seen that `unit` is the type with just one element `()` and that `bool` is the type with two elements, `true` and `false`.

You can define your own types with an enumeration of elements, like so.

```
type three =
  | One_of_three : three
  | Two_of_three : three
  | Three_of_three : three
```

This introduces a new type `three : Type`, and three *distinct* constants `One_of_three : three`, `Two_of_three : three`, `Three_of_three : three`. These constants are also called "constructors" or "data constructors". The name of a constructor must begin with an uppercase letter.

Note: In this case, it may seem redundant to have to write the type of each constructor repeatedly—of course they're all just constructors of the type `three`. In this case, F* will allow you to just write

```
type three =
  | One_of_three
  | Two_of_three
  | Three_of_three
```

As we start to use indexed types, each constructor can build a different instance of the defined type, so it will be important to have a way to specify the result type of each constructor. For uniformity, throughout this book, we'll always annotate the types of constructors, even when not strictly necessary.

F* can prove that they are distinct and that these are the only terms of type `three`.

```
let distinct = assert (One_of_three <> Two_of_three /\
                       Two_of_three <> Three_of_three /\
                       Three_of_three <> One_of_three)

let exhaustive (x:three) = assert (x = One_of_three \/
                                   x = Two_of_three \/
                                   x = Three_of_three)
```

To write functions that case analyze these new types, one uses the `match` construct. The syntax of `match` in F* is very similar to OCaml or F#. We'll assume that you're familiar with its basics. As we go, we'll learn about more advanced ways to use `match`.

Here are some examples.

```
let is_one (x:three)
  : bool
  = match x with
    | One_of_three -> true
    | _ -> false

let is_two (x:three)
  : bool
  = match x with
    | Two_of_three -> true
    | _ -> false

let is_three (x:three)
  : bool
  = match x with
    | Three_of_three -> true
    | _ -> false
```

6.1.1 Discriminators

These functions test whether `x : three` matches a given constructor, returning a `bool` in each case. Since it's so common to write functions that test whether a value of an inductive type matches one of its constructors, F* automatically generates these functions for you. For example, instead of writing

```
let three_as_int (x:three)
  : int
  = if is_one x then 1
    else if is_two x then 2
    else 3
```

One can write:

```
let three_as_int' (x:three)
  : int
  = if One_of_three? x then 1
    else if Two_of_three? x then 2
    else 3
```

In other words, for every constructor T of an inductive type t, F* generates a function named T? (called a "discriminator") which tests if a v:t matches T.

6.1.2 Exhaustiveness

Of course, an even more direct way of writing three_as_int is

```
let three_as_int'' (x:three)
  : int
  = match x with
    | One_of_three -> 1
    | Two_of_three -> 2
    | Three_of_three -> 3
```

Every time you use a match, F* will make sure to prove that you are handling all possible cases. Try omitting one of the cases in three_as_int above and see what happens.

Exhaustiveness checking in F* is a semantic check and can use the SMT solver to prove that all cases are handled appropriately. For example, you can write this:

```
let only_two_as_int (x:three { not (Three_of_three? x) })
  : int
  = match x with
    | One_of_three -> 1
    | Two_of_three -> 2
```

The refinement on the argument allows F* to prove that the Three_of_three case in the pattern is not required, since that branch would be unreachable anyway.

6.2 Tuples

The next step from enumerations is to define composite types, e.g., types that are made from pairs, triples, quadruples, etc. of other types. Here's how

```
type tup2 (a:Type)  (b:Type) =
  | Tup2 : fst:a -> snd:b -> tup2 a b

type tup3 a b c =
  | Tup3 : fst:a -> snd:b -> thd:c -> tup3 a b c
```

The type definition for tup2 a b states that for any types a : Type and b : Type, Tup2 : a -> b -> tup2 a b. That is, Tup2 is a constructor of tup2, such that given x:a and y:b, Tup2 x y : tup2 a b.

The other types tup3 and tup4 are similar—the type annotations on the bound variables can be inferred.

These are inductive types with just one case—so the discriminators Tup2?, Tup3?, and Tup4? aren't particularly useful. But, we need a way to extract, or *project*, the components of a tuple. You can do that with a match.

```
let tup2_fst #a #b (x:tup2 a b)
  : a
  = match x with
    | Tup2 fst _ -> fst
```

(continues on next page)

(continued from previous page)

```
let tup2_snd #a #b (x:tup2 a b)
  : b
  = match x with
    | Tup2 _ snd -> snd

let tup3_fst #a #b #c (x:tup3 a b c)
  : a
  = match x with
    | Tup3 fst _ _ -> fst

let tup3_snd #a #b #c (x:tup3 a b c)
  : b
  = match x with
    | Tup3 _ snd _ -> snd

let tup3_third #a #b #c (x:tup3 a b c)
  : c
  = match x with
    | Tup3 _ _ thd -> thd
```

6.2.1 Projectors

These projectors are common enough that F* auto-generates them for you. In particular, for any data constructor `T` of type `x1:t1 -> ... -> xn:tn -> t`, F* auto-generates the following function:

- `T?.xi :  y:t{T? y} -> ti`

That is, `T?.xi` is a function which when applied to a `y:t` in case `T? y`, returns the `xi` component of `T x1 ... xn`.

In the case of our `tup2` and `tup3` types, we have

- `Tup2?.fst`, `Tup2?.snd`
- `Tup3?.fst`, `Tup3?.snd`, `Tup3?.thd`

6.2.2 Syntax for tuples

Since tuples are so common, the module `FStar.Pervasives.Native.fst` defines tuple types up to arity 14. So, you shouldn't have to define `tup2` and `tup3` etc. by yourself.

The tuple types in `FStar.Pervasives.Native` come with syntactic sugar.

- You can write `a & b` instead of the `tup2 a b`; `a & b & c` instead of `tup3 a b c`; and so on, up to arity 14.
- You can write `x, y` instead of `Tup2 x y`; `x, y, z` instead of `Tup3 x y z`; an so on, up to arity 14.
- You can write `x._1`, `x._2`, `x._3`, etc. to project the field `i` of a tuple whose arity is at least `i`.

That said, if you're using tuples beyond arity 4 or 5, it's probably a good idea to define a *record*, as we'll see next—since it can be hard to remember what the components of a large tuple represent.

6.2.3 Records

A record is just a tuple with user-chosen names for its fields and with special syntax for constructing then and projecting their fields. Here's an example.

```
type point3D = { x:int; y:int; z:int}

let origin = { y=0; x=0; z=0 }

let dot (p0 p1:point3D) = p0.x * p1.x + p0.y * p1.y + p0.z * p1.z

let translate_X (p:point3D) (shift:int) = { p with x = p.x + shift}

let is_origin (x:point3D) =
  match x with
  | {z=0;y=0;x=0} -> true
  | _ -> false
```

- A record type is defined using curly braces `{}`. See `type point3D`
- A record value is also constructed using curly braces, with an assignment for each field of the record. The fields need not be given in order. See `origin`.
- To access the fields of a record, you can use the dot notation `p.x`; See `dot`, which computes a dot product using dot notation.
- Records also support the `with` notation to construct a new record whose fields are the same as the old record, except for those fields mentioned after the `with`. That is, `translate_X p shift` returns `{ x = p.x + shift; y = p.y; z = p.z}`.
- Records can also be used to pattern match a value. For example, in `is_origin`, we match the fields of the record (in any order) against some patterns.

6.3 Options

Another common type from F*'s standard library is the `option` type, which is useful to represent a possibly missing value.

Consider implementing a function to divide `x / y`, for two integers `x` and `y`. This function cannot be defined when `y` is zero, but it can be defined partially, by excluding the case where `y = 0`, as shown below. (Of course, one can also refine the domain of the function to forbid `y = 0`, but we're just trying to illustrate the `option` type here.)

```
let try_divide (x y:int)
  : option int
  = if y = 0 then None else Some (x / y)

let divide (x:int) (y:int{y<>0}) = x / y
```

Like most other functional languages, F* does not have a `null` value. Whenever a value may possibly be `null`, one typically uses the `option` type, using `None` to signify null and `Some v` for the non-null case.

6.4 Unions, or the `either` type

`FStar.Pervasives` also defines the `either` type, shown below.

```
type either a b =
  | Inl : v: a -> either a b
  | Inr : v: b -> either a b
```

The type `either a b` represents a value that could either be `Inl v` with `v:a`, or `Inr v` with `v:b`. That is, `either a b` is a tagged union of the `a` and `b`. It's easy to write functions to analyze the tag `Inl` (meaning it's "in the left case") or `Inr` ("in the right case") and compute with the underlying values. Here's an example:

```
let same_case #a #b #c #d (x:either a b) (y:either c d)
  : bool
  = match x, y with
    | Inl _, Inl _
    | Inr _, Inr _ -> true
    | _ -> false

let sum (x:either bool int) (y:either bool int{same_case x y})
  : z:either bool int{ Inl? z <==> Inl? x}
  = match x, y with
    | Inl xl, Inl yl -> Inl (xl || yl)
    | Inr xr, Inr yr -> Inr (xr + yr)
```

The `same_case x y` function decides if the two unions are both simultaneously in the left or right case.

Then, in `sum x y`, with a refinement that `x` and `y` are in the same case, we can handle just two cases (when they are both in left, or both in right) and F* can prove that the case analysis is exhaustive. In the left case, the underlying values are boolean, so we combine them with `||`; in the right case, the underlying values are integers, so we combine them with `+`; and return them with the appropriate tag. The type of the result `z:either bool int{ Inl? z <==> Inl? x}` shows that the result has the same case as `x` (and hence also `y`). We could have written the result type as `z:either bool int { same_case z x }`.

6.5 Lists

All the types we've see far have been inductive only in a degenerate sense—the constructors do not refer to the types they construct. Now, for our first truly inductive type, a list.

Here's the definition of `list` from `Prims`:

```
type list a =
  | Nil  : list a
  | Cons : hd:a -> tl:list a -> list a
```

The `list` type is available implicitly in all F* programs and we have special (but standard) syntax for the list constructors:

- `[]` is `Nil`
- `[v1; ...; vn]` is `Cons v1 ... (Cons vn Nil)`
- `hd :: tl` is `Cons hd tl`.

You can always just write out the constructors like *Nil* and *Cons* explicitly, if you find that useful (e.g., to partially apply `Cons hd : list a -> list a`).

6.5.1 Length of a list

Let's write some simple functions on lists, starting with computing the length of a list.

```
let rec length #a (l:list a)
  : nat
  = match l with
    | [] -> 0
    | _ :: tl -> 1 + length tl
```

The `length` function is recursive and implicitly polymorphic in a type `a`. For any list `l : list a`, `length l` returns a `nat`. The definition pattern matches on the list and calls `length` recursively on the tail of list, until the `[]` case is reached.

6.6 Exercises

Click here for the exercise file.

Here's the definition of `append`, a function that concatenates two lists. Can you give it a type that proves it always returns a list whose length is the sum of the lengths of its arguments?

```
let rec append l1 l2
  = match l1 with
    | [] -> l2
    | hd :: tl -> hd :: append tl l2
```

Answer

```
val append (#a:Type) (l1 l2:list a)
  : l:list a { length l = length l1 + length l2 }
```

CHAPTER

SEVEN

PROOFS OF TERMINATION

It's absolutely crucial to the soundness of F*'s core logic that all functions terminate. Otherwise, one could write non-terminating functions like this:

```
let rec loop (x:unit) : False = loop x
```

and show that `loop () :  False`, i.e., we'd have a proof term for `False` and the logic would collapse.

In the previous chapter, we just saw how to define recursive functions to *compute the length of list* and to *append two lists*. We also said *earlier* that all functions in F*'s core are *total*, i.e., they always return in a finite amount of time. So, you may be wondering, what is it that guarantees that recursive function like `length` and `append` actually terminate on all inputs?

The full details of how F* ensures termination of all functions in its core involves several elements, including positivity restrictions on datatype definitions and universe constraints. However, the main thing that you'll need to understand at this stage is that F* includes a termination check that applies to the recursive definitions of total functions. The check is a semantic check, not a syntactic criterion, like in some other dependently typed languages.

We quickly sketch the basic structure of the F* termination check on recursive functions—you'll need to understand a bit of this in order to write more interesting programs.

7.1 A well-founded partial order on terms

In order to prove a function terminating in F* one provides a *measure*: a pure expression depending on the function's arguments. F* checks that this measure strictly decreases on each recursive call. The measure for the arguments of the call is compared to the measure for the previous call according to a well-founded partial order on F* terms. We write *v1* << *v2* when *v1* precedes *v2* in this order.

Note: A relation *R* is a well-founded partial order on a set *S* if, and only if, *R* is a partial order on *S* and there are no infinite descending chains in *S* related by *R*. For example, taking *S* to be *nat*, the set of natural numbers, the integer ordering < is a well-founded partial order (in fact, it is a total order).

Since the measure strictly decreases on each recursive call, and there are no infinite descending chains, this guarantees that the function eventually stops making recursive calls, i.e., it terminates.

7.1.1 The precedes relation

Given two terms `v1:t1` and `v2:t2`, we can prove `v1 << v2` if any of the following are true:

1. **The ordering on integers**:

   `t1 = nat` and `t2 = nat` and `v1 < v2`

   Negative integers are not related by the << relation, which is only a _partial_ order.

2. **The sub-term ordering on inductive types**

   If `v2 = D u1 ... un`, where `D` is a constructor of an inductive type fully applied to arguments `u1` to `un`, then `v1 << v2` if either

   - `v1 = ui` for some `i`, i.e., `v1` is a sub-term of `v2`
   - `v1 = ui x` for some `i` and `x`, i.e., `v1` is the result of applying a sub-term of `v2` to some argument `x`.

7.2 Why `length` terminates

Let's look again at the definition of `length` and see how F* checks that it terminates, i.e.,

```
let rec length #a (l:list a)
  : nat
  = match l with
    | [] -> 0
    | _ :: tl -> 1 + length tl
```

First off, the definition of `length` above makes use of various syntactic shorthands to hide some details. If we were to write it out fully, it would be as shown below:

```
let rec length #a (l:list a)
  : Tot nat (decreases l)
  = match l with
    | [] -> 0
    | _ :: tl -> length tl
```

The main difference is on the second line. As opposed to just writing the result type of `length`, in full detail, we write `Tot nat (decreases l)`. This states two things

- The `Tot nat` part states that `length` is a total function returning a `nat`, just as the `nat` did before.
- The additional `(decreases l)` specifying a *measure*, i.e., the quantity that decreases at each recursive call according the well-founded relation <<.

To check the definition, F* gives the recursively bound name (`length` in this case) a type that's guarded by the measure. I.e., for the body of the function, `length` has the following type:

```
#a:Type -> m:list a{ m << l } -> nat
```

This is to say that when using `length` to make a recursive call, we can only apply it to an argument `m << l`, i.e., the recursive call can only be made on an argument `m` that precedes the current argument `l`. This is enough to ensure that the recursive calls will eventually bottom out, since there are no infinite descending chains related by <<.

In the case of `length`, we need to prove at the recursive call `length tl` that `tl : (m : list a { m << l })`, or, equivalently that `tl << l` is valid. But, from the sub-term ordering on inductive types, `l = Cons _ tl`, so `tl << l` is indeed provable and everything checks out.

7.3 Lexicographic orderings

F* also provides a convenience to enhance the well-founded ordering << to lexicographic combinations of <<. That is, given two lists of terms `v1, ..., vn` and `u1, ..., un`, F* accepts that the following lexicographic ordering:

```
v1 << u1 \/ (v1 == u1 /\ (v2 << u2 \/ (v2 == u2 /\ ( ... vn << un))))
```

is also well-founded. In fact, it is possible to prove in F* that this ordering is well-founded, provided << is itself well-founded.

Lexicographic ordering are common enough that F* provides special support to make it convenient to use them. In particular, the notation:

```
%[v1; v2; ...; vn] << %[u1; u2; ...; un]
```

is shorthand for:

```
v1 << u1 \/ (v1 == u1 /\ (v2 << u2 \/ (v2 == u2 /\ ( ... vn << un))))
```

Let's have a look at lexicographic orderings at work in proving that the classic `ackermann` function terminates on all inputs.

```
let rec ackermann (m n:nat)
  : Tot nat (decreases %[m;n])
  = if m=0 then n + 1
    else if n = 0 then ackermann (m - 1) 1
    else ackermann (m - 1) (ackermann m (n - 1))
```

The `decreases %[m;n]` syntax tells F* to use the lexicographic ordering on the pair of arguments `m, n` as the measure to prove this function terminating.

When defining `ackermann m n`, for each recursive call of the form `ackermann m' n'`, F* checks that `%[m';n'] << %[m;n]`, i.e., F* checks that either

- `m' << m`, or
- `m' = m` and `n' << n`

There are three recursive calls to consider:

1. `ackermann (m - 1) 1`: In this case, since we know that `m > 0`, we have `m - 1 << m`, due to the ordering on natural numbers. Since the ordering is lexicographic, the second argument is irrelevant for termination.
2. `ackermann m (n - 1)`: In this case, the first argument remained the same (i.e., it's still `m`), but we know that `n > 0` so `n - 1 << n` by the natural number ordering.
3. `ackermann (m - 1) (ackermann m (n - 1))`: Again, like in the first case, the first argument `m - 1 << m`, and the second is irrelevant for termination.

7.4 Default measures

As we saw earlier, F* allows you to write the following code, with no decreases clause, and it still accepts it.

```
let rec length #a (l:list a)
  : nat
  = match l with
    | [] -> 0
    | _ :: tl -> 1 + length tl
```

For that matter, you can leave out the decreases clause in ackermann and F* is okay with it.

```
let rec ackermann (m n:nat)
  : nat
  = if m=0 then n + 1
    else if n = 0 then ackermann (m - 1) 1
    else ackermann (m - 1) (ackermann m (n - 1))
```

This is because F* uses a simple heuristic to choose the decreases clause, if the user didn't provide one.

The *default* decreases clause for a total, recursive function is the lexicographic ordering of all the non-function-typed arguments, taken in order from left to right.

That is, the default decreases clause for ackermann is exactly decreases %[m; n]; and the default for length is just decreases %[a; l] (which is equivalent to decreases l). So, you needn't write it.

On the other hand, it you were to flip the order of arguments to ackermann, then the default choice of the measure would not be correct—so, you'll have to write it explicitly, as shown below.

```
let rec ackermann_flip (n m:nat)
  : Tot nat (decreases %[m;n])
  = if m=0 then n + 1
    else if n = 0 then ackermann_flip 1 (m - 1)
    else ackermann_flip (ackermann (n - 1) m) (m - 1)
```

7.5 Mutual recursion

F* also supports mutual recursion and the same check of proving that a measure of the arguments decreases on each (mutually) recursive call applies.

For example, one can write the following code to define a binary tree that stores an integer at each internal node—the keyword and allows defining several types that depend mutually on each other.

To increment all the integers in the tree, we can write the mutually recursive functions, again using and to define incr_tree and incr_node to depend mutually on each other. F* is able to prove that these functions terminate, just by using the default measure as usual.

```
type tree =
  | Terminal : tree
  | Internal : node -> tree

and node = {
  left : tree;
  data : int;
```

(continues on next page)

(continued from previous page)

```
  right : tree
}

let rec incr_tree (x:tree)
  : tree
  = match x with
    | Terminal -> Terminal
    | Internal node -> Internal (incr_node node)

and incr_node (n:node)
  : node
  = {
      left = incr_tree n.left;
      data = n.data + 1;
      right = incr_tree n.right
    }
```

Note: Sometimes, a little trick with lexicographic orderings can help prove mutually recursive functions correct. We include it here as a tip, you can probably skip it on a first read.

```
let rec foo (l:list int)
  : Tot int (decreases %[l;0])
  = match l with
    | [] -> 0
    | x :: xs -> bar xs
and bar (l:list int)
  : Tot int (decreases %[l;1])
  = foo l
```

What's happening here is that when `foo l` calls `bar`, the argument `xs` is legitimately a sub-term of `l`. However, `bar l` simply calls back `foo l`, without decreasing the argument. The reason this terminates, however, is that `bar` can freely call back `foo`, since `foo` will only ever call `bar` again with a smaller argument. You can convince F* of this by writing the decreases clauses shown, i.e., when `bar` calls `foo`, `l` doesn't change, but the second component of the lexicographic ordering does decrease, i.e., `0 << 1`.

7.6 The termination check, precisely

Having seen a few examples at work, we can now describe how the termination check works in general.

Note: We use a slightly more mathematical notation here, so that we can be precise. If it feels unfamiliar, you needn't understand this completely at first. Continue with the examples and refer back to this section, if and when you feel like a precise description would be helpful.

When defining a recursive function

$$\mathsf{f}~(\overline{\mathsf{x} : \mathsf{t}})~~:~~\mathsf{Tot}~\mathsf{r}~(\mathsf{decreases}~\mathsf{m})~~=~~\mathsf{e}$$

i.e., f is a function with several arguments $\mathsf{x1} : \mathsf{t1}, ..., \mathsf{x}_\mathsf{n} : \mathsf{t}_\mathsf{n}$, returning r with measure m, mutually recursively with other functions of several arguments at type:

$$\begin{aligned} \mathsf{f_1}\ (\overline{\mathsf{x_1 : t_1}}) &: \mathsf{Tot}\ \mathsf{r_1}\ (\mathsf{decreases}\ \mathsf{m_1}) \\ &\ldots \\ \mathsf{f_n}\ (\overline{\mathsf{x_n : t_n}}) &: \mathsf{Tot}\ \mathsf{r_n}\ (\mathsf{decreases}\ \mathsf{m_n}) \end{aligned}$$

we check the definition of the function body of f (i.e., e) with all the mutually recursive functions in scope, but at types that restrict their domain, in the following sense:

$$\begin{aligned} \mathsf{f} &: (\overline{\mathsf{y : t}}\{\ \mathsf{m}[\overline{\mathsf{y}}/\overline{\mathsf{x}}]\ << \ \mathsf{m}\ \} \to \mathsf{r}[\overline{\mathsf{y}}/\overline{\mathsf{x}}]) \\ \mathsf{f_1} &: (\overline{\mathsf{x_1 : t_1}}\{\ \mathsf{m_1}\ << \ \mathsf{m}\ \} \to \mathsf{r_1}) \\ &\ldots \\ \mathsf{f_n} &: (\overline{\mathsf{x_n : t_n}}\{\ \mathsf{m_n}\ << \ \mathsf{m}\ \} \to \mathsf{r_n}) \end{aligned}$$

That is, each function in the mutually recursive group can only be applied to arguments that precede the current formal parameters of f according to the annotated measures of each function.

7.7 Exercise: Fibonacci in linear time

Click here for the exercise file.

Here's a function to compute the n-th Fibonacci number.

```
let rec fibonacci (n:nat)
  : nat
  = if n <= 1
    then 1
    else fibonacci (n - 1) + fibonacci (n - 2)
```

Here's a more efficient, tail-recursive, linear-time variant.

```
let rec fib a b n =
   match n with
   | 0 -> a
   | _ -> fib b (a+b) (n-1)

let fibonacci n = fib 1 1 n
```

Add annotations to the functions to get F* to accept them, in particular, proving that fib terminates.

Answer

```
let rec fib (a b n:nat)
  : Tot nat (decreases n)
  = match n with
    | 0 -> a
    | _ -> fib b (a+b) (n-1)

let fibonacci (n:nat) : nat = fib 1 1 n
```

7.8 Exercise: Tail-recursive reversal

Click here for the exercise file.

Here is a function to reverse a list:

```
let rec rev #a (l:list a)
  : list a
  = match l with
    | [] -> []
    | hd::tl -> append (rev tl) hd
```

But, it is not very efficient, since it is not tail recursive and, worse, it is quadratic, it traverses the reversed tail of the list each time to add the first element to the end of it.

This version is more efficient, because it is tail recursive and linear.

```
let rec rev_aux l1 l2 =
  match l2 with
  | []     -> l1
  | hd :: tl -> rev_aux (hd :: l1) tl

let rev l = rev_aux [] l
```

Add type annotations to `rev_aux` and `rev`, proving, in particular, that `rev_aux` terminates.

Answer

```
let rec rev_aux #a (l1 l2:list a)
  : Tot (list a) (decreases l2)
  = match l2 with
    | []     -> l1
    | hd :: tl -> rev_aux (hd :: l1) tl

let rev #a (l:list a) : list a = rev_aux [] l
```

CHAPTER

EIGHT

LEMMAS AND PROOFS BY INDUCTION

Let's say you wrote the `factorial` function and gave it the type `nat -> nat`. Later, you care about some other property about `factorial`, e.g., that if `x > 2` then `factorial x > x`. One option is to revise the type you wrote for `factorial` and get F* to reprove that it has this type. But this isn't always feasible. What if you also wanted to prove that if `x > 3` then `factorial x > 2 * x`. Clearly, polluting the type of `factorial` with all these properties that you may or may not care about is impractical.

You could write assertions to ask F* to check these properties, e.g.,

```
let _ = assert (forall (x:nat). x > 2 ==> factorial x > 2)
```

But, F* complains saying that it couldn't prove this fact. That's not because the fact isn't true—recall, checking the validity of assertions in F* is undecidable. So, there are facts that are true that F* may not be able to prove, at least not without some help.

In this case, proving this property about `factorial` requires a proof by induction. F* and Z3 cannot do proofs by induction automatically—you will have to help F* here by writing a *lemma*.

8.1 Introducing lemmas

A lemma is a function in F* that always returns the `():unit` value. However, the type of lemma carries useful information about which facts are provable.

Here's our first lemma:

```
let rec factorial_is_positive (x:nat)
  : u:unit{factorial x > 0}
  = if x = 0 then ()
    else factorial_is_positive (x - 1)
```

There's a lot of information condensed in that definition. Let's spell it out in detail:

- `factorial_is_positive` is a recursive function with a parameter `x:nat`
- The return type of `factorial_is_positive` is a refinement of unit, namely `u:unit{factorial x > 0}`. That says that the function always returns `()`, but, additionally, when `factorial_is_positive x` returns (which it always does, since it is a total function) it is safe to conclude that `factorial x > 0`.
- The next three lines prove the lemma using a proof by induction on `x`. The basic concept here is that by programming total functions, we can write proofs about other pure expressions. We'll discuss such proofs in detail in the remainder of this section.

8.1.1 Some syntactic shorthands for Lemmas

Lemmas are so common in F* that it's convenient to have special syntax for them. Here's another take at our proof by `factorial x > 0`

```
let rec factorial_is_pos (x:int)
  : Lemma (requires x >= 0)
          (ensures factorial x > 0)
  = if x = 0 then ()
    else factorial_is_pos (x - 1)
```

The type `x:t -> Lemma (requires pre) (ensures post)` is the type of a function

- that can be called with an argument `v:t`
- the argument must satisfies the precondition `pre[v/x]`
- the function always returns a `unit`
- and ensures that the postcondition `post[v/x]` is valid

The type is equivalent to `x:t{pre} -> u:unit{post}`.

When the precondition `pre` is trivial, it can be omitted. One can just write:

```
Lemma (ensures post)
```

or even

```
Lemma post
```

8.1.2 A proof by induction, explained in detail

Let's look at this lemma in detail again—why does it convince F* that `factorial x > 0`?

```
let rec factorial_is_pos (x:int)
  : Lemma (requires x >= 0)
          (ensures factorial x > 0)
  = if x = 0 then ()
    else factorial_is_pos (x - 1)
```

- It is a proof by induction on `x`. Proofs by induction in F* are represented by total recursive functions. The fact that it is total is extremely important—it ensures that the inductive argument is well-founded, i.e., that the induction hypothesis is only applied correctly on strictly smaller arguments.
- The base case of the induction is when `x=0`. In this case, F* + Z3 can easily prove that `factorial 0 > 0`, since this just requires computing `factorial 0` to 1 and checking `1 > 0`.
- What remains is the case where `x > 0`.
- In the inductive case, the type of the recursively bound `factorial_is_pos` represents the induction hypothesis. In this case, its type is

  ```
  y:int {y < x} -> Lemma (requires y >= 0) (ensures factorial y > 0)
  ```

  In other words, the type of recursive function tells us that for all `y` that are smaller than that current argument `x` and non-negative , it is safe to assume that `factorial y > 0`.
- By making a recursive call on `x-1`, F* can conclude that `factorial (x - 1) > 0`.

- Finally, to prove that `factorial x > 0`, the solver figures out that `factorial x = x * factorial (x - 1)`. From the recursive lemma invocation, we know that `factorial (x - 1) > 0`, and since we're in the case where `x > 0`, the solver can prove that the product of two positive numbers must be positive.

8.2 Exercises: Lemmas about integer functions

Click here for the exercise file.

8.2.1 Exercise 1

Try proving the following lemmas about `factorial`:

```
val factorial_is_greater_than_arg (x:int)
  : Lemma (requires x > 2)
          (ensures factorial x > x)
```

Answer

```
let rec factorial_is_greater_than_arg (x:int)
  : Lemma (requires x > 2)
          (ensures factorial x > x)
  = if x = 3 then ()
    else factorial_is_greater_than_arg (x - 1)
```

8.2.2 Exercise 2

Try proving the following lemmas about `fibonacci`:

```
let rec fibonacci (n:nat)
  : nat
  = if n <= 1
    then 1
    else fibonacci (n - 1) + fibonacci (n - 2)

val fibonacci_greater_than_arg (n:nat{n >= 2})
  : Lemma (fibonacci n >= n)
```

Answer (Includes two proofs and detailed explanations)

```
let rec fibonacci_greater_than_arg (n:nat{n >= 2})
  : Lemma (fibonacci n >= n)
  = if n <= 3 then ()
    else (
      fibonacci_greater_than_arg (n - 1);
      fibonacci_greater_than_arg (n - 2)
    )
```

Let's have a look at that proof in some detail. It's much like the proof by induction we discussed in detail earlier, except now we have two uses of the induction hypothesis.

- It's a proof by induction on `n:nat{n >= 2}`, as you can tell from the `let rec`.

- The base cases are when `n = 2` and `n = 3`. In both these cases, the solver can simply compute `fibonacci n` and check that it is greater than `n`.
- Otherwise, in the inductive case, we have `n >= 4` and the induction hypothesis is the type of the recursive function:

```
m:nat{m >= 2 /\ m < n} -> Lemma (fibonacci m >= m)
```

- We call the induction hypothesis twice and get:

```
fibonacci (n - 1) >= n - 1
fibonacci (n - 2) >= n - 2
```

- To conclude, we show:

```
fibonacci n = //by definition
fibonacci (n - 1) + fibonacci (n - 2) >= //from the facts above
(n - 1) + (n - 2) = //rearrange
2*n - 3 >=  //when n >= 4
n
```

As you can see, once you set up the induction, the SMT solver does a lot of the work.

Sometimes, the SMT solver can even find proofs that you might not write yourself. Consider this alternative proof of `fibonacci n >= n`.

```
let rec fib_greater_than_arg (n:nat{n >= 2})
  : Lemma (fibonacci n >= n)
  = if n = 2 then ()
    else (
      fib_greater_than_arg (n - 1)
    )
```

This proof works with just a single use of the induction hypothesis. How come? Let's look at it in detail.

1. It's a proof by induction on `n:nat{n >= 2}`.
2. The base case is when `n=2`. It's easy to compute `fibonacci 2` and check that it's greater than or equal to 2.
3. In the inductive case, we have:

```
n >= 3
```

4. The induction hypothesis is:

```
m:nat{m >= 2 /\ m < n} -> Lemma (fibonacci m >= m)
```

5. We apply the induction hypothesis to `n - 1` and get

```
fibonacci (n - 1) >= n - 1
```

6. We have:

```
fibonacci n = //definition
fibonacci (n - 1) + fibonacci (n - 2) >= //from 5
(n - 1) + fibonacci (n - 2)
```

7. So, our goal is now:

```
(n - 1) + fibonacci (n - 2) >= n
```

8. It suffices if we can show `fibonacci (n - 2) >= 1`
9. From (2) and the definition of `fibonacci` we have:

```
fibonacci (n - 1) = //definition
fibonacci (n - 2) + fibonacci (n - 3) >= //from 5
n - 1 >= // from 3
2
```

10. Now, suppose for contradiction, that `fibonacci (n - 2) = 0`.

    10.1. Then, from step 9, we have `fibonacci (n-3) >= 2`

    10.2 If `n=3`, then `fibonacci 0 = 1`, so we have a contradiction.

    10.3 If `n > 3`, then

        10.3.1. `fibonacci (n-2) = fibonacci (n-3) + fibonacci (n-4)`, by definition

        10.3.2. `fibonacci (n-3) + fibonacci (n-4) >= fibonacci (n-3)`, since `fibonacci (n-4) : nat`.

        10.3.3. `fibonacci (n-2) >= fibonacci (n-3)`, using 10.3.1 and 10.3.2

        10.3.4. `fibonacci (n-2) >= 2`, using 10.1

        10.3.5. But, 10.3.4 contradicts 10; so the proof is complete.

You probably wouldn't have come up with this proof yourself, and indeed, it took us some puzzling to figure out how the SMT solver was able to prove this lemma with just one use of the induction hypothesis. But, there you have it. All of which is to say that the SMT solver is quite powerful!

8.3 Exercise: A lemma about append

Earlier, we saw a definition of `append` with the following type:

```
val append (#a:Type) (l1 l2:list a)
  : l:list a{length l = length l1 + length l2}
```

Now, suppose we were to define *app*`, a version of `append` with a weaker type, as shown below.

```
let rec app #a (l1 l2:list a)
  : list a
  = match l1 with
    | [] -> l2
    | hd :: tl -> hd :: app tl l2
```

Can you prove the following lemma?

```
val app_length (#a:Type) (l1 l2:list a)
  : Lemma (length (app l1 l2) = length l1 + length l2)
```

Answer

```
let rec app_length #a l1 l2
  = match l1 with
    | [] -> ()
    | _ :: tl -> app_length tl l2
```

8.4 Intrinsic vs extrinsic proofs

As the previous exercise illustrates, you can prove properties either by enriching the type of a function or by writing a separate lemma about it—we call these the 'intrinsic' and 'extrinsic' styles, respectively. Which style to prefer is a matter of taste and convenience: generally useful properties are often good candidates for intrinsic specification (e.g, that `length` returns a `nat`); more specific properties are better stated and proven as lemmas. However, in some cases, as in the following example, it may be impossible to prove a property of a function directly in its type—you must resort to a lemma.

```
let rec reverse #a (l:list a)
  : list a
  = match l with
    | [] -> []
    | hd :: tl -> append (reverse tl) [hd]
```

Let's try proving that reversing a list twice is the identity function. It's possible to *specify* this property in the type of `reverse` using a refinement type.

```
val reverse (#a:Type) : f:(list a -> list a){forall l. l == f (f l)}
```

Note: A subtle point: the refinement on `reverse` above uses a *propositional equality*. That's because equality on lists of arbitrary types is not decidable, e.g., consider `list (int -> int)`. All the proofs below will rely on propositional equality.

However, F* refuses to accept this as a valid type for `reverse`: proving this property requires two separate inductions, neither of which F* can perform automatically.

Instead, one can use two lemmas to prove the property we care about. Here it is:

```
(* snoc is "cons" backwards --- it adds an element to the end of a list *)
let snoc l h = append l [h]

let rec snoc_cons #a (l:list a) (h:a)
  : Lemma (reverse (snoc l h) == h :: reverse l)
  = match l with
    | [] -> ()
    | hd :: tl -> snoc_cons tl h

let rec rev_involutive #a (l:list a)
  : Lemma (reverse (reverse l) == l)
  = match l with
    | [] -> ()
    | hd :: tl ->
      // (1) [reverse (reverse tl) == tl]
      rev_involutive tl;
```

(continues on next page)

(continued from previous page)

```
      // (2) [reverse (append (reverse tl) [hd]) == h :: reverse (reverse tl)]
      snoc_cons (reverse tl) hd
      // These two facts are enough for Z3 to prove the lemma:
      //   reverse (reverse (hd :: tl))
      //   =def= reverse (append (reverse tl) [hd])
      //   =(2)= hd :: reverse (reverse tl)
      //   =(1)= hd :: tl
      //   =def= l
```

In the `hd :: tl` case of `rev_involutive` we are explicitly applying not just the induction hypothesis but also the `snoc_cons` auxiliary lemma also proven there.

8.4.1 Exercises: Reverse is injective

Click here for the exercise file.

Prove that reverse is injective, i.e., prove the following lemma.

```
val rev_injective (#a:Type) (l1 l2:list a)
  : Lemma (requires reverse l1 == reverse l2)
          (ensures  l1 == l2)
```

Answer

```
let rec snoc_injective (#a:Type) (l1:list a) (h1:a) (l2:list a) (h2:a)
  : Lemma (requires snoc l1 h1 == snoc l2 h2)
          (ensures l1 == l2 /\ h1 == h2)
  = match l1, l2 with
    | _ :: tl1, _ :: tl2 -> snoc_injective tl1 h1 tl2 h2
    | _ -> ()


let rec rev_injective l1 l2 =
  match l1,l2 with
  | h1::t1, h2::t2 ->
      // assert(reverse (h1::t1) = reverse (h2::t2));
      // assert(snoc (reverse t1) h1  = snoc (reverse t2) h2);
      snoc_injective (reverse t1) h1 (reverse t2) h2;
      // assert(reverse t1 = reverse t2 /\ h1 = h2);
      rev_injective t1 t2
      // assert(t1 = t2 /\ h1::t1 = h2::t2)
  | _, _ -> ()
```

That's quite a tedious proof, isn't it. Here's a simpler proof.

```
let rev_injective_alt (#a:Type) (l1 l2:list a)
  : Lemma (requires reverse l1 == reverse l2)
          (ensures  l1 == l2)
  = rev_involutive l1; rev_involutive l2
```

The `rev_injective_alt` proof is based on the idea that every invertible function is injective. We've already proven that `reverse` is involutive, i.e., it is its own inverse. So, we invoke our lemma, once for `l1` and once for `l2`. This gives to the SMT solver the information that `reverse (reverse l1) = l1` and `reverse (reverse l2) = l2`, which suffices to complete the proof. As usual, when structuring proofs, lemmas are your friends!

8.4.2 Exercise: Optimizing reverse

Earlier, we saw how to implement *a tail-recursive variant* of `reverse`.

```
let rec rev_aux #a (l1 l2:list a)
  : Tot (list a) (decreases l2)
  = match l2 with
    | []     -> l1
    | hd :: tl -> rev_aux (hd :: l1) tl

let rev #a (l:list a) : list a = rev_aux [] l
```

Prove the following lemma to show that it is equivalent to the previous non-tail-recursive implementation, i.e.,

```
val rev_is_ok (#a:_) (l:list a) : Lemma (rev [] l == reverse l)
```

Answer

```
let rec rev_is_ok_aux #a (l1 l2:list a)
  : Lemma (ensures (rev_aux l1 l2 == append (reverse l2) l1))
          (decreases l2)
  = match l2 with
    | [] -> ()
    | hd :: tl  -> rev_is_ok_aux (hd :: l1) tl;
                 append_assoc (reverse tl) [hd] l1

let rev_is_ok #a (l:list a)
  : Lemma (rev l == reverse l)
  = rev_is_ok_aux [] l;
    append_right_unit (reverse l)
```

8.4.3 Exercise: Optimizing Fibonacci

Earlier, we saw how to implement *a tail-recursive variant* of `fibonacci`—we show it again below.

```
let rec fib (a b n:nat)
  : Tot nat (decreases n)
  = match n with
    | 0 -> a
    | _ -> fib b (a+b) (n-1)

let fib_tail (n:nat) : nat = fib 1 1 n
```

Prove the following lemma to show that it is equivalent to the non-tail-recursive implementation, i.e.,

```
val fib_tail_is_ok (n:nat)
  : Lemma (fib_tail n == fibonacci n)
```

Answer

```
let rec fib_is_ok_aux (n: nat) (k: nat)
  : Lemma (fib (fibonacci k)
               (fibonacci (k + 1)) n == fibonacci (k + n))
  = if n = 0 then () else fib_is_ok_aux (n - 1) (k + 1)

let fib_tail_is_ok (n:nat)
  : Lemma (fib_tail n == fibonacci n)
  = fib_is_ok_aux n 0
```

8.5 Higher-order functions

Functions are first-class values—they can be passed to other functions and returned as results. We've already seen some examples in the section on *polymorphism*. Here are some more, starting with the `map` function on lists.

```
let rec map #a #b (f: a -> b) (l:list a)
  : list b
  = match l with
    | [] -> []
    | hd::tl -> f hd :: map f tl
```

It takes a function `f` and a list `l` and it applies `f` to each element in `l` producing a new list. More precisely `map f [v1; ...; vn]` produces the list `[f v1; ...; f vn]`. For example:

```
map (fun x -> x + 1) [0; 1; 2] = [1; 2; 3]
```

8.5.1 Exercise: Finding a list element

Here's a function called `find` that given a boolean function `f` and a list `l` returns the first element in `l` for which `f` holds. If no element is found `find` returns `None`.

```
let rec find f l =
  match l with
  | [] -> None
  | hd :: tl -> if f hd then Some hd else find f tl
```

Prove that if `find` returns `Some x` then `f x = true`. Is it better to do this intrinsically or extrinsically? Do it both ways.

Answer

```
val find (#a:Type) (f: a -> bool) (l:list a)
  : o:option a{ Some? o ==> f (Some?.v o)}
```

```
let rec find_alt f l =
  match l with
  | [] -> None
  | hd :: tl -> if f hd then Some hd else find_alt f tl

let rec find_alt_ok #a (f:a -> bool) (l:list a)
  : Lemma (match find_alt f l with
```

(continues on next page)

(continued from previous page)

```
           | Some x -> f x
           | _ -> true)
  = match l with
    | [] -> ()
    | _ :: tl -> find_alt_ok f tl
```

8.5.2 Exercise: fold_left

Here is a function fold_left, where:

```
fold_left f [b1; ...; bn] a = f (bn, ... (f b2 (f b1 a)))
```

```
let rec fold_left #a #b (f: b -> a -> a) (l: list b) (acc:a)
  : a
  = match l with
    | [] -> acc
    | hd :: tl -> fold_left f tl (f hd acc)
```

Prove the following lemma:

```
val fold_left_Cons_is_rev (#a:Type) (l:list a)
  : Lemma (fold_left Cons l [] == reverse l)
```

Hint: This proof is a level harder from what we've done so far.

You will need to strengthen the induction hypothesis, and possibly to prove that append is associative and that append l [] == l.

Answer

```
let rec append_assoc #a (l1 l2 l3 : list a)
  : Lemma (append l1 (append l2 l3) == append (append l1 l2) l3)
  = match l1 with
    | [] -> ()
    | h1 :: t1 -> append_assoc t1 l2 l3

let rec fold_left_Cons_is_rev_stronger #a (l1 l2: list a)
  : Lemma (fold_left Cons l1 l2 == append (reverse l1) l2)
  = match l1 with
    | [] -> ()
    | h1 :: t1 ->
      // (1) [append (append (reverse t1) [h1]) l2
      //      == append (reverse t1) (append [h1] l2)]
      append_assoc (reverse t1) [h1] l2;
      // (2) [fold_left Cons t1 (h1 :: l2) = append (reverse t1) (h1 :: l2)]
      fold_left_Cons_is_rev_stronger t1 (h1 :: l2)
      // append (reverse l1) l2
      // =def= append (append (reverse t1) [h1]) l2
      // =(1)= append (reverse t1) (append [h1] l2)
      // =def= append (reverse t1) (h1 :: l2)
      // =(2)= fold_left Cons t1 (h1 :: l2)
      // =def= fold_left Cons l1 l2
```

(continues on next page)

(continued from previous page)

```
let rec append_right_unit #a (l1:list a)
  : Lemma (append l1 [] == l1)
  = match l1 with
    | [] -> ()
    | _ :: tl -> append_right_unit tl

let fold_left_Cons_is_rev #a (l:list a)
  : Lemma (fold_left Cons l [] == reverse l)
  = fold_left_Cons_is_rev_stronger l [];
    append_right_unit (reverse l)
```

CHAPTER

NINE

CASE STUDY: QUICKSORT

We'll now put together what we've learned about defining recursive functions and proving lemmas about them to prove the correctness of Quicksort, a classic sorting algorithm.

We'll start with lists of integers and describe some properties that we'd like to hold true of a sorting algorithm, starting with a function `sorted`, which decides when a list of integers is sorted in increasing order, and `mem`, which decides if a given element is in a list. Notice that `mem` uses an `eqtype`, *the type of types that support decidable equality*.

```
let rec sorted (l:list int)
  : bool
  = match l with
    | [] -> true
    | [x] -> true
    | x :: y :: xs -> x <= y && sorted (y :: xs)

let rec mem (#a:eqtype) (i:a) (l:list a)
  : bool
  = match l with
    | [] -> false
    | hd :: tl -> hd = i || mem i tl
```

Given a sorting algorithm `sort`, we would like to prove the following property, meaning that for all input list `l`, the resulting list `sort l` is sorted and has all the elements that `l` does.

```
forall l. sorted (sort l) /\ (forall i. mem i l <==> mem i (sort l))
```

This specification is intentionally a bit weak, e.g., in case there are multiple identical elements in `l`, this specification does not prevent `sort` from retaining only one of them.

We will see how to improve this specification below, as part of an exercise.

If you're unfamiliar with the algorithm, you can read more about it here. We'll describe several slightly different implementations and proofs of Quicksort in detail—you may find it useful to follow along interactively with the entire code development of this sequence.

9.1 Implementing `sort`

Our implementation of Quicksort is pretty simple-minded. It always picks the first element of the list as the pivot; partitions the rest of the list into those elements greater than or equal to the pivot, and the rest; recursively sorts the partitions; and slots the pivot in the middle before returning. Here it is:

```
let rec sort (l:list int)
  : Tot (list int) (decreases (length l))
  = match l with
    | [] -> []
    | pivot :: tl ->
      let hi, lo  = partition ((<=) pivot) tl in
      append (sort lo) (pivot :: sort hi)
```

There are a few points worth discussing in detail:

1. The notation `((<=) pivot)` may require some explanation: it is the *partial application* of the <= operator to just one argument, `pivot`. It is equivalent to `fun x -> pivot <= x`.
2. We have to prove that `sort` terminates. The measure we've provided is `length l`, meaning that at each recursive call, we're claiming that the length of input list is strictly decreasing.
3. Why is this true? Well, informally, the recursive calls `sort lo` and `sort hi` are partitions of the `tl` of the list, which is strictly shorter than `l`, since we've removed the `pivot` element. We'll have to convince F* of this fact by giving `partition` an interesting type that we'll see below.

9.2 Implementing `partition`

Here's an implementation of `partition`. It's a *higher-order function*, where `partition f l` returns a pair of lists $\texttt{l}_1$ and $\texttt{l}_2$, a partitioning of the elements in `l` such that the every element in $\texttt{l}_1$ satisfies `f` and the elements in $\texttt{l}_2$ do not.

```
let rec partition (#a:Type) (f:a -> bool) (l:list a)
  : x:(list a & list a) { length (fst x) + length (snd x) = length l }
  = match l with
    | [] -> [], []
    | hd::tl ->
      let l1, l2 = partition f tl in
      if f hd
      then hd::l1, l2
      else l1, hd::l2
```

The specification we've given `partition` is only partial—we do not say, for instance, that all the elements in $\texttt{l}_1$ satisfy `f`. We only say that the sum of the lengths of the $\texttt{l}_1$ and $\texttt{l}_2$ are equal to the length of `l`. That's because that's the only property we need (so far) about `partition`—this property about the lengths is what we need to prove that on the recursive calls `sort lo` and `sort hi`, the arguments `lo` and `hi` are strictly shorter than the input list.

This style of partial specification should give you a sense of the art of program proof and the design choices between *intrinsic and extrinsic proof*. One tends to specify only what one needs, rather than specifying all properties one can imagine right up front.

9.3 Proving sort correct

Now that we have our definition of sort, we still have to prove it correct. Here's a proof—it requires three auxiliary lemmas and we'll discuss it in detail.

Our first lemma relates partition to mem: it proves what we left out in the intrinsic specification of partition, i.e., that all the elements in l_1 satisfy f, the elements in l_2 do not, and every element in l appears in either l_1 or l_2.

```
let rec partition_mem (#a:eqtype)
                      (f:(a -> bool))
                      (l:list a)
  : Lemma (let l1, l2 = partition f l in
          (forall x. mem x l1 ==> f x) /\
          (forall x. mem x l2 ==> not (f x)) /\
          (forall x. mem x l = (mem x l1 || mem x l2)))
  = match l with
    | [] -> ()
    | hd :: tl -> partition_mem f tl
```

Our next lemma is very specific to Quicksort. If l_1 and l_2 are already sorted, and partitioned by pivot, then slotting pivot in the middle of l_1 and l_2 produces a sorted list. The specification of sorted_concat uses a mixture of refinement types (e.g., l1:list int{sorted l1}) and requires / ensures specifications–this is just a matter of taste.

```
let rec sorted_concat (l1:list int{sorted l1})
                      (l2:list int{sorted l2})
                      (pivot:int)
  : Lemma (requires (forall y. mem y l1 ==> not (pivot <= y)) /\
                    (forall y. mem y l2 ==> pivot <= y))
          (ensures sorted (append l1 (pivot :: l2)))
  = match l1 with
    | [] -> ()
    | hd :: tl -> sorted_concat tl l2 pivot
```

Our third lemma is a simple property about append and mem.

```
let rec append_mem (#t:eqtype)
                   (l1 l2:list t)
  : Lemma (ensures (forall a. mem a (append l1 l2) = (mem a l1 || mem a l2)))
  = match l1 with
    | [] -> ()
    | hd::tl -> append_mem tl l2
```

Finally, we can put the pieces together for our top-level statement about the correctness of sort.

```
let rec sort_correct (l:list int)
  : Lemma (ensures (
          let m = sort l in
          sorted m /\
          (forall i. mem i l = mem i m)))
         (decreases (length l))
  = match l with
    | [] -> ()
    | pivot :: tl ->
```

(continues on next page)

(continued from previous page)

```
      let hi, lo  = partition ((<=) pivot) tl in
      sort_correct hi;
      sort_correct lo;
      partition_mem ((<=) pivot) tl;
      sorted_concat (sort lo) (sort hi) pivot;
      append_mem (sort lo) (pivot :: sort hi)
```

The structure of the lemma is mirrors the structure of `sort` itself.

- In the base case, the proof is automatic.
- In the inductive case, we partition the tail of the list and recursively call the lemma on the the `hi` and `lo` components, just like `sort` itself. The intrinsic type of `partition` is also helpful here, using the `length` measure on the list to prove that the induction here is well-founded.
  - To prove the `ensures` postcondition, we apply our three auxiliary lemmas.
    - `partition_mem ((<=) pivot) tl` gives us the precondition of needed to satisfy the `requires` clause of `sorted_concat`.
    - We also need to prove the `sorted` refinements on `sort lo` and `sort hi` in order to call `sorted_concat`, but the recursive calls of the lemma give us those properties.
    - After calling `sorted_concat`, we have proven that the resulting list is sorted. What's left is to prove that all the elements of the input list are in the result, and `append_mem` does that, using the postcondition of `partition_mem` and the induction hypothesis to relate the elements of `append (sort lo) (pivot :: sort hi)` to the input list `l`.

Here's another version of the `sort_correct` lemma, this time annotated with lots of intermediate assertions.

```
let sort_ok (l:list int) =
    let m = sort l in
    sorted m /\
    (forall i. mem i l = mem i m)

let rec sort_correct_annotated (l:list int)
  : Lemma (ensures sort_ok l)
          (decreases (length l))
  = match l with
    | [] -> ()
    | pivot :: tl ->
      let hi, lo  = partition ((<=) pivot) tl in
      assert (length hi + length lo == length tl);
      sort_correct_annotated hi;
      assert (sort_ok hi);
      sort_correct_annotated lo;
      assert (sort_ok lo);
      partition_mem ((<=) pivot) tl;
      assert (forall i. mem i tl = mem i hi || mem i lo);
      assert (forall i. mem i hi ==> pivot <= i);
      assert (forall i. mem i lo ==> i < pivot);
      assert (sort l == (append (sort lo) (pivot :: sort hi)));
      sorted_concat (sort lo) (sort hi) pivot;
      assert (sorted (sort l));
      append_mem (sort lo) (pivot :: sort hi);
      assert (forall i. mem i l = mem i (sort l))
```

This is an extreme example, annotating with assertions at almost every step of the proof. However, it is indicative of a style that one often uses to interact with F* when doing SMT-assisted proofs. At each point in your program or proof, you can use `assert` to check what the prover "knows" at that point. See what happens if you move the assertions around, e.g., if you move `assert (sort_ok lo)` before calling `sort_correct_annotated lo`, F* will complain that it is not provable.

9.4 Limitations of SMT-based proofs at higher order

You may be wondering why we used `(<=) pivot` instead of `fun x -> pivot <= x` in our code. Arguably, the latter is more readable, particularly to those not already familiar with functional programming languages. Well, the answer is quite technical.

We could indeed have written `sort` like this,

```
let rec sort_alt (l:list int)
  : Tot (list int) (decreases (length l))
  = match l with
    | [] -> []
    | pivot :: tl ->
      let hi, lo  = partition (fun x -> pivot <= x) tl in
      append (sort_alt lo) (pivot :: sort_alt hi)
```

And we could have tried to write our main lemma this way:

```
let sort_alt_ok (l:list int) =
    let m = sort_alt l in
    sorted m /\
    (forall i. mem i l = mem i m)

let rec sort_alt_correct_annotated (l:list int)
  : Lemma (ensures sort_alt_ok l)
          (decreases (length l))
  = match l with
    | [] -> ()
    | pivot :: tl ->
      let hi, lo  = partition (fun x -> pivot <= x) tl in
      assert (length hi + length lo == length tl);
      sort_alt_correct_annotated hi;
      assert (sort_alt_ok hi);
      sort_alt_correct_annotated lo;
      assert (sort_alt_ok lo);
      partition_mem (fun x -> pivot <= x) tl;
      assert (forall i. mem i tl = mem i hi || mem i lo);
      assert (forall i. mem i hi ==> pivot <= i);
      assert (forall i. mem i lo ==> i < pivot);
      //THIS NEXT LINE IS NOT PROVABLE BY SMT ALONE
      assume (sort_alt l == append (sort_alt lo) (pivot :: sort_alt hi));
      sorted_concat (sort_alt lo) (sort_alt hi) pivot;
      assert (sorted (sort_alt l));
      append_mem (sort_alt lo) (pivot :: sort_alt hi);
      assert (forall i. mem i l = mem i (sort_alt l))
```

However, without further assistance, F*+SMT is unable to prove the line at which the `assume` appears. It turns out, this is due to a fundamental limitation in how F* encodes its higher-order logic into the SMT solver's first-order logic. This encoding comes with some loss in precision, particularly for lambda terms. In this case, the SMT solver is unable to prove that the occurrence of `fun x -> pivot <= x` that appears in the proof of `sort_alt_correct_annotated` is identical to the occurrence of the same lambda term in `sort_alt`, and so it cannot conclude that `sort_alt l` is really equal to `append (sort_alt lo) (pivot :: sort_alt hi))`.

This is unfortunate and can lead to some nasty surprises when trying to do proofs about higher order terms. Here are some ways to avoid such pitfalls:

- Try to use named functions at higher order, rather than lambda literals. Named functions do not suffer a loss in precision when encoded to SMT. This is the reason why `(<=) pivot` worked out better than the lambda term here—the `(<=)` is a name that syntactically appears in both the definition of `sort` and the proof of `sort_alt_correct` and the SMT solver can easily see that the two occurrences are identical.
- If you must use lambda terms, sometimes an intrinsic proof style can help, as we'll see below.
- If you must use lambda terms with extrinsic proofs, you can still complete your proof, but you will have to help F* along with tactics or proofs by normalization, more advanced topics that we'll cover in later sections.
- Even more forward looking, recent higher-order variants of SMT solvers are promising and may help address some of these limitations.

9.5 An intrinsic proof of `sort`

As we observed earlier, our proof of `sort_correct` had essentially the same structure as the definition of `sort` itself—it's tempting to fuse the definition of `sort` with `sort_correct`, so that we avoid the duplication and get a proof of correctness of `sort` built-in to its definition.

So, here it is, a more compact proof of `sort`, this time done intrinsically, i.e., by enriching the type of `sort` to capture the properties we want.

```
let rec sort_intrinsic (l:list int)
  : Tot (m:list int {
                sorted m /\
                (forall i. mem i l = mem i m)
         })
   (decreases (length l))
  = match l with
    | [] -> []
    | pivot :: tl ->
      let hi, lo  = partition (fun x -> pivot <= x) tl in
      partition_mem (fun x -> pivot <= x) tl;
      sorted_concat (sort_intrinsic lo) (sort_intrinsic hi) pivot;
      append_mem (sort_intrinsic lo) (pivot :: sort_intrinsic hi);
      append (sort_intrinsic lo) (pivot :: sort_intrinsic hi)
```

We still use the same three auxiliary lemmas to prove the properties we want, but this time the recursive calls to sort the partitioned sub-lists also serve as calls to the induction hypothesis for the correctness property we're after.

Notice also that in this style, the use of a lambda literal isn't problematic—when operating within the same scope, F*'s encoding to SMT is sufficiently smart to treat the multiple occurrences of `fun x -> pivot <= x` as identical functions.

9.5.1 Runtime cost?

You may be concerned that we have just polluted the definition of `sort_intrinsic` with calls to three additional recursive functions–will this introduce any runtime overhead when executing `sort_intrinsic`? Thankfully, the answer to that is “no”.

As we’ll learn in the section on *effects*, F* supports of notion of *erasure*—terms that can be proven to not contribute to the observable behavior of a computation will be erased by the compiler before execution. In this case, the three lemma invocations are total functions returning unit, i.e., these are functions that always return in a finite amount of time with the constant value `()`, with no other observable side effect. So, there is no point in keeping those function calls around—we may as well just optimize them away to their result `()`.

Indeed, if you ask F* to extract the program to OCaml (using `fstar --codegen OCaml`), here’s what you get:

```
let rec (sort_intrinsic : Prims.int Prims.list -> Prims.int Prims.list) =
  fun l ->
    match l with
    | [] -> []
    | pivot::tl ->
       let uu___ = partition (fun x -> pivot <= x) tl in
       (match uu___ with
        | (hi, lo) ->
          append (sort_intrinsic lo) (pivot :: (sort_intrinsic hi)))
```

The calls to the lemmas have disappeared.

9.6 Exercises

9.6.1 Generic sorting

Here’s a file with the scaffolding for this exercise.

The point of this exercise is to define a generic version of `sort` that is parameterized by any total order over the list elements, rather than specializing `sort` to work on integer lists only. Of course, we want to prove our implementations correct. So, let’s do it in two ways, both intrinsically and extrinsically. Your goal is to remove the all the occurrences of `admit` in the development below.

```
module Part1.Quicksort.Generic

//Some auxiliary definitions to make this a standalone example
let rec length #a (l:list a)
  : nat
  = match l with
    | [] -> 0
    | _ :: tl -> 1 + length tl

let rec append #a (l1 l2:list a)
  : list a
  = match l1 with
    | [] -> l2
    | hd :: tl -> hd :: append tl l2

let total_order (#a:Type) (f: (a -> a -> bool)) =
```

(continues on next page)

(continued from previous page)

```
    (forall a. f a a)                                         (* reflexivity   *)
    /\ (forall a1 a2. (f a1 a2 /\ a1=!=a2)  <==> not (f a2 a1))  (* anti-symmetry *)
    /\ (forall a1 a2 a3. f a1 a2 /\ f a2 a3 ==> f a1 a3)        (* transitivity  *)
    /\ (forall a1 a2. f a1 a2 \/ f a2 a1)                       (* totality  *)

let total_order_t (a:Type) = f:(a -> a -> bool) { total_order f }

let rec sorted #a  (f:total_order_t a) (l:list a)
  : bool
  = match l with
    | [] -> true
    | [x] -> true
    | x :: y :: xs -> f x y && sorted f (y :: xs)

let rec mem (#a:eqtype) (i:a) (l:list a)
  : bool
  = match l with
    | [] -> false
    | hd :: tl -> hd = i || mem i tl

let rec sort #a (f:total_order_t a) (l:list a)
  : Tot (list a) (decreases (length l))
  = admit()

let rec sort_correct (#a:eqtype) (f:total_order_t a) (l:list a)
  : Lemma (ensures (
           let m = sort f l in
           sorted f m /\
           (forall i. mem i l = mem i m)))
          (decreases (length l))
  = admit()

let rec sort_intrinsic (#a:eqtype) (f:total_order_t a) (l:list a)
  : Tot (m:list a {
                sorted f m /\
                (forall i. mem i l = mem i m)
         })
   (decreases (length l))
  = admit()
```

Answer

```
module Part1.Quicksort.Generic

//Some auxiliary definitions to make this a standalone example
let rec length #a (l:list a)
  : nat
  = match l with
    | [] -> 0
    | _ :: tl -> 1 + length tl

let rec append #a (l1 l2:list a)
```

(continues on next page)

(continued from previous page)

```
  : list a
  = match l1 with
    | [] -> l2
    | hd :: tl -> hd :: append tl l2

let total_order (#a:Type) (f: (a -> a -> bool)) =
    (forall a. f a a)                                         (* reflexivity   *)
    /\ (forall a1 a2. (f a1 a2 /\ a1=!=a2)  <==> not (f a2 a1))  (* anti-symmetry *)
    /\ (forall a1 a2 a3. f a1 a2 /\ f a2 a3 ==> f a1 a3)        (* transitivity  *)
    /\ (forall a1 a2. f a1 a2 \/ f a2 a1)                       (* totality *)

let total_order_t (a:Type) = f:(a -> a -> bool) { total_order f }

let rec sorted #a  (f:total_order_t a) (l:list a)
  : bool
  = match l with
    | [] -> true
    | [x] -> true
    | x :: y :: xs -> f x y && sorted f (y :: xs)

let rec mem (#a:eqtype) (i:a) (l:list a)
  : bool
  = match l with
    | [] -> false
    | hd :: tl -> hd = i || mem i tl

let rec partition (#a:Type) (f:a -> bool) (l:list a)
  : x:(list a & list a) { length (fst x) + length (snd x) = length l }
  = match l with
    | [] -> [], []
    | hd::tl ->
      let l1, l2 = partition f tl in
      if f hd
      then hd::l1, l2
      else l1, hd::l2

let rec sort #a (f:total_order_t a) (l:list a)
  : Tot (list a) (decreases (length l))
  = match l with
    | [] -> []
    | pivot :: tl ->
      let hi, lo  = partition (f pivot) tl in
      append (sort f lo) (pivot :: sort f hi)


let rec partition_mem (#a:eqtype)
                      (f:(a -> bool))
                      (l:list a)
  : Lemma (let l1, l2 = partition f l in
           (forall x. mem x l1 ==> f x) /\
           (forall x. mem x l2 ==> not (f x)) /\
           (forall x. mem x l = (mem x l1 || mem x l2)))
```

(continues on next page)

(continued from previous page)

```
  = match l with
    | [] -> ()
    | hd :: tl -> partition_mem f tl

let rec sorted_concat (#a:eqtype)
                      (f:total_order_t a)
                      (l1:list a{sorted f l1})
                      (l2:list a{sorted f l2})
                      (pivot:a)
  : Lemma (requires (forall y. mem y l1 ==> not (f pivot y)) /\
                    (forall y. mem y l2 ==> f pivot y))
          (ensures sorted f (append l1 (pivot :: l2)))
  = match l1 with
    | [] -> ()
    | hd :: tl -> sorted_concat f tl l2 pivot

let rec append_mem (#t:eqtype)
                   (l1 l2:list t)
  : Lemma (ensures (forall a. mem a (append l1 l2) = (mem a l1 || mem a l2)))
  = match l1 with
    | [] -> ()
    | hd::tl -> append_mem tl l2

let rec sort_correct (#a:eqtype) (f:total_order_t a) (l:list a)
  : Lemma (ensures (
           let m = sort f l in
           sorted f m /\
           (forall i. mem i l = mem i m)))
          (decreases (length l))
  = match l with
    | [] -> ()
    | pivot :: tl ->
      let hi, lo  = partition (f pivot) tl in
      sort_correct f hi;
      sort_correct f lo;
      partition_mem (f pivot) tl;
      sorted_concat f (sort f lo) (sort f hi) pivot;
      append_mem (sort f lo) (pivot :: sort f hi)

let rec sort_intrinsic (#a:eqtype) (f:total_order_t a) (l:list a)
  : Tot (m:list a {
                sorted f m /\
                (forall i. mem i l = mem i m)
         })
   (decreases (length l))
  = match l with
    | [] -> []
    | pivot :: tl ->
      let hi, lo  = partition (f pivot) tl in
      partition_mem (f pivot) tl;
      sorted_concat f (sort_intrinsic f lo) (sort_intrinsic f hi) pivot;
      append_mem (sort_intrinsic f lo) (pivot :: sort_intrinsic f hi);
```

(continues on next page)

(continued from previous page)

```
      append (sort_intrinsic f lo) (pivot :: sort_intrinsic f hi)
```

9.6.2 Proving that sort is a permutation

We promised at the beginning of this section that we'd eventually give a better specification for sort, one that proves that it doesn't drop duplicate elements in the list. That's the goal of the exercise in this section—we'll prove that our generic Quicksort is returns a permutation of the input list.

Let's start by defining what it means for lists to be permutations of each other—we'll do this using occurrence counts.

```
let rec count (#a:eqtype) (x:a) (l:list a)
  : nat
  = match l with
    | hd::tl -> (if hd = x then 1 else 0) + count x tl
    | [] -> 0

let mem (#a:eqtype) (i:a) (l:list a)
  : bool
  = count i l > 0

let is_permutation (#a:eqtype) (l m:list a) =
  forall x. count x l = count x m

let rec append_count (#t:eqtype)
                     (l1 l2:list t)
  : Lemma (ensures (forall a. count a (append l1 l2) = (count a l1 + count a l2)))
  = match l1 with
    | [] -> ()
    | hd::tl -> append_count tl l2
```

The definitions should be self-explanatory. We include one key lemma append_count to relate occurrence to list concatenations.

The next key lemma to prove is partition_mem_permutation.

```
val partition_mem_permutation (#a:eqtype)
                              (f:(a -> bool))
                              (l:list a)
  : Lemma (let l1, l2 = partition f l in
           (forall x. mem x l1 ==> f x) /\
           (forall x. mem x l2 ==> not (f x)) /\
           (is_permutation l (append l1 l2)))
```

You will also need a lemma similar to the following:

```
val permutation_app_lemma (#a:eqtype) (hd:a) (tl l1 l2:list a)
  : Lemma (requires (is_permutation tl (append l1 l2)))
          (ensures (is_permutation (hd::tl) (append l1 (hd::l2))))
```

Using these, and adaptations of our previous lemmas, prove:

```
val sort_correct (#a:eqtype) (f:total_order_t a) (l:list a)
  : Lemma (ensures
           sorted f (sort f l) /\
           is_permutation l (sort f l))
```

Load the exercise script and give it a try.

Answer

```
module Part1.Quicksort.Permutation
#push-options "--fuel 1 --ifuel 1"

//Some auxiliary definitions to make this a standalone example
let rec length #a (l:list a)
  : nat
  = match l with
    | [] -> 0
    | _ :: tl -> 1 + length tl

let rec append #a (l1 l2:list a)
  : list a
  = match l1 with
    | [] -> l2
    | hd :: tl -> hd :: append tl l2

let total_order (#a:Type) (f: (a -> a -> bool)) =
    (forall a. f a a)                                         (* reflexivity   *)
    /\ (forall a1 a2. (f a1 a2 /\ a1=!=a2)  <==> not (f a2 a1))  (* anti-symmetry *)
    /\ (forall a1 a2 a3. f a1 a2 /\ f a2 a3 ==> f a1 a3)        (* transitivity  *)
    /\ (forall a1 a2. f a1 a2 \/ f a2 a1)                       (* totality *)
let total_order_t (a:Type) = f:(a -> a -> bool) { total_order f }

let rec sorted #a  (f:total_order_t a) (l:list a)
  : bool
  = match l with
    | [] -> true
    | [x] -> true
    | x :: y :: xs -> f x y && sorted f (y :: xs)

//SNIPPET_START: count permutation
let rec count (#a:eqtype) (x:a) (l:list a)
  : nat
  = match l with
    | hd::tl -> (if hd = x then 1 else 0) + count x tl
    | [] -> 0

let mem (#a:eqtype) (i:a) (l:list a)
  : bool
  = count i l > 0

let is_permutation (#a:eqtype) (l m:list a) =
  forall x. count x l = count x m
```

(continues on next page)

(continued from previous page)

```
let rec append_count (#t:eqtype)
                     (l1 l2:list t)
  : Lemma (ensures (forall a. count a (append l1 l2) = (count a l1 + count a l2)))
  = match l1 with
    | [] -> ()
    | hd::tl -> append_count tl l2
//SNIPPET_END: count permutation

let rec partition (#a:Type) (f:a -> bool) (l:list a)
  : x:(list a & list a) { length (fst x) + length (snd x) = length l }
  = match l with
    | [] -> [], []
    | hd::tl ->
      let l1, l2 = partition f tl in
      if f hd
      then hd::l1, l2
      else l1, hd::l2

let rec sort #a (f:total_order_t a) (l:list a)
  : Tot (list a) (decreases (length l))
  = match l with
    | [] -> []
    | pivot :: tl ->
      let hi, lo  = partition (f pivot) tl in
      append (sort f lo) (pivot :: sort f hi)

let rec partition_mem_permutation (#a:eqtype)
                                  (f:(a -> bool))
                                  (l:list a)
  : Lemma (let l1, l2 = partition f l in
           (forall x. mem x l1 ==> f x) /\
           (forall x. mem x l2 ==> not (f x)) /\
           (is_permutation l (append l1 l2)))
  = match l with
    | [] -> ()
    | hd :: tl ->
      partition_mem_permutation f tl;
      let hi, lo = partition f tl in
      append_count hi lo;
      append_count hi (hd::lo);
      append_count (hd :: hi) lo

let rec sorted_concat (#a:eqtype)
                      (f:total_order_t a)
                      (l1:list a{sorted f l1})
                      (l2:list a{sorted f l2})
                      (pivot:a)
  : Lemma (requires (forall y. mem y l1 ==> not (f pivot y)) /\
                    (forall y. mem y l2 ==> f pivot y))
          (ensures sorted f (append l1 (pivot :: l2)))
  = match l1 with
    | [] -> ()
```

(continues on next page)

(continued from previous page)

```
    | hd :: tl -> sorted_concat f tl l2 pivot

let permutation_app_lemma (#a:eqtype) (hd:a) (tl:list a)
                          (l1:list a) (l2:list a)
   : Lemma (requires (is_permutation tl (append l1 l2)))
           (ensures (is_permutation (hd::tl) (append l1 (hd::l2))))
  = append_count l1 l2;
    append_count l1 (hd :: l2)

let rec sort_correct (#a:eqtype) (f:total_order_t a) (l:list a)
  : Lemma (ensures (
           sorted f (sort f l) /\
           is_permutation l (sort f l)))
          (decreases (length l))
  = match l with
    | [] -> ()
    | pivot :: tl ->
      let hi, lo  = partition (f pivot) tl in
      partition_mem_permutation (f pivot) tl;
      append_count lo hi;
      append_count hi lo;
      assert (is_permutation tl (append lo hi));
      sort_correct f hi;
      sort_correct f lo;
      sorted_concat f (sort f lo) (sort f hi) pivot;
      append_count (sort f lo) (sort f hi);
      assert (is_permutation tl (sort f lo `append` sort f hi));
      permutation_app_lemma pivot tl (sort f lo) (sort f hi)


let rec sort_intrinsic (#a:eqtype) (f:total_order_t a) (l:list a)
  : Tot (m:list a {
                sorted f m /\
                is_permutation l m
         })
   (decreases (length l))
  = match l with
    | [] -> []
    | pivot :: tl ->
      let hi, lo  = partition (f pivot) tl in
      partition_mem_permutation (f pivot) tl;
      append_count lo hi;  append_count hi lo;
      sorted_concat f (sort_intrinsic f lo) (sort_intrinsic f hi) pivot;
      append_count (sort_intrinsic f lo) (sort_intrinsic f hi);
      permutation_app_lemma pivot tl (sort_intrinsic f lo) (sort_intrinsic f hi);
      append (sort_intrinsic f lo) (pivot :: sort_intrinsic f hi)
```

CHAPTER

TEN

EXECUTING PROGRAMS

We've been through several chapters already, having learned many core concepts of F*, but we have yet to see how to compile and execute a program, since our focus so far has been on F*'s logic and how to prove properties about programs.

F* offers several choices for executing a program, which we cover briefly here, using Quicksort as a running example.

10.1 Interpreting F* programs

As mentioned in the *capsule summary*, F* includes an engine (called the *normalizer*) that can symbolically reduce F* computations. We'll see many more uses of F*'s normalizer as we go, but one basic usage of it is to use it to interpret programs.

Invoking the interpreter is easy using *fstar-mode.el* in emacs. In emacs, go to the F* menu, then to *Interactive queries*, then choose *Evaluate an expression* (or type C-c C-s C-e): this prompts you to enter an expression that you want to evaluate: type `sort ( <= ) [4;3;2;1]` and then press "Enter". You should see the following output: `sort ( <= ) [4;3;2;1]` $\downarrow\beta\delta\iota\zeta$ `[1; 2; 3; 4] <:  Prims.Tot (list int)`, saying that the input term reduced to `[1; 2; 3; 4]` of type `Tot (list int)`.

The $\downarrow\beta\delta\iota\zeta$ may seem a bit arcane, but it describes the reduction strategy that F* used to interpret the term:

- β means that functions were applied
- δ means that definitions were unfolded
- ι means that patterns were matched
- ζ means that recursive functions were unrolled

We'll revisit what these reduction steps mean in a later chapter, including how to customize them for your needs.

10.2 Compiling to OCaml

The main way to execute F* programs is by compiling, or *extracting*, them to OCaml and then using OCaml's build system and runtime to produce an executable and run it.

Note: The method that we show here works for simple projects with just a few files. For larger projects, F* offers a dependence analysis that can produce dependences for use in a Makefile. F* also offers separate compilation which allows a project to be checked one file at a time, and for the results to be cached and reused. For documentation and examples of how to use these features and structure the build for larger projects see these resources:

- Dealing with dependences

10.2.1 Producing an OCaml library

To extract OCaml code from a F* program use the command-line options, as shown below:

```
fstar --codegen OCaml --extract Part1.Quicksort --odir out Part1.Quicksort.Generic.fst
```

- The `--codegen` option tells F* to produce OCaml code
- The `--extract` option tells F* to only extract all modules in the given namespace, i.e., in this case, all modules in `Part1.Quicksort`
- The `--odir` option tells F* to put all the generated files into the specified directory; in this case `out`
- The last argument is the source file to be checked and extracted

The resulting OCaml code is in the file `Part1_Quicksort_Generic.ml`, where the F* `dot`-separated name is transformed to OCaml's naming convention for modules. The generated code is here

Some points to note about the extracted code:

- F* extracts only those definitions that correspond to executable code. Lemmas and other proof-only aspects are erased. We'll learn more about erasure in a later chapter.
- The F* types are translated to OCaml types. Since F* types are more precise than OCaml types, this translation process necessarily involves a loss in precision. For example, the type of total orders in `Part1.Quicksort.Generic.fst` is:

  ```
  let total_order_t (a:Type) = f:(a -> a -> bool) { total_order f }
  ```

  Whereas in OCaml it becomes

  ```
  type 'a total_order_t = 'a -> 'a -> Prims.bool
  ```

  This means that you need to be careful when calling your extracted F* program from unverified OCaml code, since the OCaml compiler will not complain if you pass in some function that is not a total order where the F* code expects a total order.

10.2.2 Compiling an OCaml library

Our extracted code provides several top-level functions (e.g., `sort`) but not `main`. So, we can only compile it as a library.

For simple uses, one can compile the generated code into a OCaml native code library (a cmxa file) with `ocamlbuild`, as shown below

```
OCAMLPATH=$FSTAR_HOME/lib ocamlbuild -use-ocamlfind -pkg batteries -pkg fstar.lib Part1_
↪Quicksort_Generic.cmxa
```

Some points to note:

- You need to specify the variable `OCAMLPATH` which OCaml uses to find required libraries. For F* projects, the `OCAMLPATH` should include the `bin` directory of the FStar release bundle.
- The `-use-ocamlfind` option enables a utility to find OCaml libraries.

- Extracted F* programs rely on two libraries: `batteries` and `fstar.lib`, which is what the `-pkg` options say.
- Finally, `Part1_Quicksort_Generic.cmxa` references the name of the corresponding `.ml` file, but with the `.cmxa` extension to indicate that we want to compile it as a library.

Your can use the resulting .cmxa file in your other OCaml projects.

10.2.3 Adding a 'main'

We have focused so far on programming and proving *total* functions. Total functions have no side effects, e.g., they cannot read or write state, they cannot print output etc. This makes total functions suitable for use in libraries, but to write a top-level driver program that can print some output (i.e., a `main`), we need to write functions that actually have some effects.

We'll learn a lot more about F*'s support for effectful program in a later section, but for now we'll just provide a glimpse of it by showing (below) a `main` program that calls into our Quicksort library.

```
module Part1.Quicksort.Main
module Q = Part1.Quicksort.Generic

//Printing the elements of an integer list, using the FStar.Printf library
let string_of_int_list l =
  Printf.sprintf "[%s]"
    (String.concat "; " (List.Tot.map (Printf.sprintf "%d") l))

//A main program, which sorts a list and prints it before and after
let main () =
  let orig = [42; 17; 256; 94] in
  let sorted = Q.sort ( <= ) orig in
  let msg =
    Printf.sprintf "Original list = %s\nSorted list = %s\n"
      (string_of_int_list orig)
      (string_of_int_list sorted)
  in
  FStar.IO.print_string msg

//Run ``main ()`` when the module loads
#push-options "--warn_error -272"
let _ = main ()
#pop-options
```

There are few things to note here:

- This time, rather than calling `Q.sort` from unverified OCaml code, we are calling it from F*, which requires us to prove all its preconditions, e.g., that the comparison function `( <= )` that we are passing in really is a total order—F* does that automatically.
- `FStar.IO.print_string` is a library function that prints a string to `stdout`. Its type is `string -> ML unit`, a type that we'll look at in detail when we learn more about effects. For now, keep in mind that functions with the `ML` label in their type may have observable side effects, like IO, raising exceptions, etc.
- The end of the file contains `let _ = main ()`, a top-level term that has a side-effect (printing to `stdout`) when executed. In a scenario where we have multiple modules, the runtime behavior of a program with such top-level side-effects depends on the order in which modules are loaded. When F* detects this, it raises the warning `272`. In this case, we intend for this program to have a top-level effect, so we suppress the warning using the `--warn_error -272` option.

To compile this code to OCaml, along with its dependence on `Part1.Quicksort.Generic`, one can invoke:

```
fstar --codegen OCaml --extract Part1.Quicksort --odir out Part1.Quicksort.Main.fst
```

This time, F* extracts both `Part1.Quicksort.Generic.fst` (as before) and `Part1.Quicksort.Main.fst` to OCaml, producing Part1_Quicksort_Main.ml to OCaml.

You can compile this code in OCaml to a native executable by doing:

```
OCAMLPATH=$FSTAR_HOME/lib ocamlbuild -use-ocamlfind -pkg batteries -pkg fstar.lib Part1_
↪Quicksort_Main.native
```

And, finally, you can execute Part1_Quicksort_Main.native to see the following output:

```
$ ./Part1_Quicksort_Main.native
Original list = [42; 17; 256; 94]
Sorted list = [17; 42; 94; 256]
```

10.3 Compiling to other languages

F* also supports compiling programs to F# and, for a subset of the language, supports compilation to C.

For the F# extraction, use the `--codegen FSharp` option. However, it is more typical to structure an F* project for use with F# using Visual Studio project and solution files. Here are some examples:

- A simple example
- A more advanced example mixing F* and F# code

For extraction to C, please see the tutorial on Low*.

CHAPTER ELEVEN

WRAPPING UP

Congratulations! You've reached the end of an introduction to basic F*.

You should have learned the following main concepts:

- Basic functional programming
- Using types to write precise specifications
- Writing proofs as total functions
- Defining and working with new inductive types
- Lemmas and proofs by induction

Throughout, we saw how F*'s use of an SMT solver can reduce the overhead of producing proofs, and you should know enough now to be productive in small but non-trivial F* developments.

However, it would be wrong to conclude that SMT-backed proofs in F* are all plain sailing. And there's a lot more to F* than SMT proofs—so read on through the rest of this book.

But, if you do plan to forge ahead with mainly SMT-backed proofs, you should keep the following in mind before attempting more challenging projects.

It'll serve you well to learn a bit more about how an SMT solver works and how F* interfaces with it—this is covered in a few upcoming sections, including a section on *classical proofs* and in *understanding how F* uses Z3*. Additionally, if you're interested in doing proofs about arithmetic, particularly nonlinear arithmetic, before diving in, you would do well to read more about the F* library `FStar.Math.Lemmas` and F* arithmetic settings.

Part III

Representing Data, Proofs, and Computations with Inductive Types

Earlier, we learned about *defining new data types* in F*. For example, here's the type of lists parameterized by a type `a` of the list elements.

```
type list a =
  | Nil  : list a
  | Cons : hd:a -> tl:list a -> list a
```

We also saw that it was easy to define basic functions over these types, using pattern matching and recursion. For example, here's a function to compute the length of a list.

```
let rec length #a (l:list a)
  : nat
  = match l with
    | [] -> 0
    | _ :: tl -> 1 + length tl
```

The function `length` defines some property of a `list` (its length) separately from the definition of the `list` type itself. Sometimes, however, it can be convenient to define a property of a type together with the type itself. For example, in some situations, it may be natural to define the length of the list together with the definition of the list type itself, so that every list is structurally equipped with a notion of its length. Here's how:

```
type vec (a:Type) : nat -> Type =
  | Nil : vec a 0
  | Cons : #n:nat -> hd:a -> tl:vec a n -> vec a (n + 1)
```

What we have here is our first indexed type, `vec a n`. One way to understand this definition is that `vec a : nat -> Type` describes a family of types, `vec a 0`, `vec a 1`, ... etc., all representing lists of `a`-typed elements, but where the *index* `n` describes the length of the list. With this definition of `vec`, the function `length` is redundant: given a `v : vec a n` we know that its `length v` is `n`, without having to recompute it.

This style of enriching a type definition with indexes to state properties of the type is reminiscent of what we learned earlier about *intrinsic versus extrinsic proofs*. Rather than defining a single type `list a` for all lists and then separatately (i.e., extrinsically) defining a function `length` to compute the length of a list, with `vec` we've enriched the type of the list intrinsically, so that type of `vec` immediately tells you its length.

Now, you may have seen examples like this length-indexed `vec` type before—it comes up often in tutorials about dependently typed programming. But, indexed types can do a lot more. In this section we learn about indexed inductive types from three related perspectives:

- Representing data: Inductive types allow us to build new data types, includes lists, vectors, trees, etc. in several flavors. We present two case studies: *vectors* and *Merkle trees*, a binary tree data structure equipped with cryptographic proofs.
- Representing proofs: The core logic of F* rests upon several simple inductive type definitions. We revisit the logical connectives we've seen before (including the *propositional connectives* and *equality*) and show how rather than being primitive notions in F*, their definitions arise from a few core constructions involving inductive type. Other core notions in the language, including the handling of *termination proofs*, can also be understood in terms of inductive types that *model well-founded recursion*.
- Representing computations: Inductive type definitions allow embedding other programming languages or computational models within F*. We develop two case studies.
  - We develop a *deep embedding of the simply-typed lambda calculus* with several reduction strategies, and a proof of its syntactic type soundness. The example showcases the use of several inductive types to represent the syntax of a programming language, a relation describing its type system, and another relation describing its operational semantics.

  - We also show how to use *higher-order abstract syntax* to represent well-typed lambda terms, a concise style that illustrates how to use inductive types that store functions.
  - Finally, we look at a *shallow embedding of an imperative programming language with structured concurrency*, representing computations as infinitely branching inductively defined trees. The example introduces modeling computational effects as monads and showcases the use of inductive types at higher order.

This section is somewhat more advanced than the first. It also interleaves some technical material about F*'s core logic with case studies showing some of those core concepts at work. You can certainly work through the material sequentially, but depending on your interests, you may find the following paths through the material to be more accessible.

If you're familiar with dependent types but are new to F* and want a quick tour, the following path might work for you:

- *Length-indexed lists*, F*-specific notations
- *Equality*
- *Logical connectives*
- Any of the case studies, depending on your interest.

If you're unfamiliar with dependent types and are more curious to learn how to use F* by working through examples, following path might work for you:

- *Inductive type definitions*, basic concepts
- *Length-indexed lists*, F*-specific notations in the simplest setting
- *Merkle trees*, a more interesting example, with applications to cryptographic security
- *Logical connectives*, some utilities to manipulate F*'s logical connectives
- Any of the case studies, depending on your interest, with the *Simply Typed Lambda Calculus* perhaps the easiest of them.

But, by the end of this section, through several exercises, we expect the reader to be familiar enough with inductive types to define their own data structures and inductively defined relations, while also gaining a working knowledge of some core parts of F*'s type theory.

CHAPTER

TWELVE

INDUCTIVE TYPE DEFINITIONS

An inductive type definition, sometimes called a *datatype*, has the following general structure.

$$
\begin{gathered}
\mathsf{type}\ T_1\ \overline{(x_1 : p_1)} : \overline{y_1 : q_1} \to \mathsf{Type} = \overline{| D_1 : t_1} \\
\ldots \\
\mathsf{and}\ T_n\ \overline{(x_n : p_n)} : \overline{y_n : q_n} \to \mathsf{Type} = \overline{| D_n : t_n}
\end{gathered}
$$

This defines n mutually inductive types, named $T_1 \ldots T_n$, called the *type constructors*. Each type constructor T_i has a number of *parameters*, the $\overline{x_i : p_i}$, and a number of *indexes*, the $\overline{y_i : q_i}$.

Each type constructor T_i has zero or more *data constructors* $\overline{D_i : t_i}$. For each data constructor D_{ij}, its type t_{ij} must be of the form $\overline{z : s} \to T_i\ \bar{x_i}\ \bar{e}$, i.e., it must be a function type returning an instance of T_i with *the same parameters* $\overline{x_i}$ as in the type constructor's signature, but with any other well-typed terms $\overline{e}$ for the index arguments. This is the main difference between a parameter and an index—a parameter of a type constructor *cannot* vary in the result type of the data constructors, while the indexes can.

Further, in each of the arguments $\overline{z : s}$ of the data constructor, none of the mutually defined type constructors $\overline{T}$ may appear to the left of an arrow. That is, all occurrences of the type constructors must be *strictly positive*. This is to ensure that the inductive definitions are well-founded, as explained below. Without this restriction, it is easy to break soundness by writing non-terminating functions with `Tot` types.

Also related to ensuring logical consistency is the *universe* level of an inductive type definition. We'll return to that later, once we've done a few examples.

12.1 Strictly positive definitions

As a strawman, consider embedding a small dynamically typed programming language within F*. All terms in our language have the same static type `dyn`, although at runtime values could have type `Bool`, or `Int`, or `Function`.

One attempt at representing a language like this using a data type in F* is as follows:

```
noeq
type dyn =
  | Bool : bool -> dyn
  | Int  : int -> dyn
  | Function : (dyn -> dyn) -> dyn
```

The three cases of the data type represent our three kinds of runtime values: `Bool b`, `Int b`, and `Function f`. The `Function` case, however, is problematic: The argument `f` is itself a function from `dyn -> dyn`, and the constructor `Function` allows promoting a `dyn -> dyn` function into the type `dyn` itself, e.g., one can represent the identity function in `dyn` as `Function (fun (x:dyn) -> x)`. However, the `Function` case is problematic: as we will see below, it allows circular definitions that enable constructing instances of `dyn` without actually providing any base case. F* rejects the definition of `dyn`, saying "Inductive type dyn does not satisfy the strict positivity condition".

Consider again the general shape of an inductive type definition:

$$\begin{aligned}\text{type } T_1\ \overline{(x_1 : p_1)} : \overline{y_1 : q_1} \to \text{Type} = \overline{| D_1 : t_1} \\ \ldots \\ \text{and } T_n\ \overline{(x_n : p_n)} : \overline{y_n : q_n} \to \text{Type} = \overline{| D_n : t_n}\end{aligned}$$

This definition is strictly positive when

- for every type constructor $T \in T_1, ..., T_n$,
- and every data constructor $D : t \in \overline{D_1}, ...\overline{D_n}$, where *t* is of the form $x0 : s_0 \to ... \to xn : s_n \to T_i ...$, and $s_0, ..., s_n$ are the types of the fields of D
- and for all instantiations $\overline{v}$ of the type parameters $\overline{p}$ of the type T,
- T does not appear to the left of any arrow in any $s \in (s_0, ..., s_k)[\overline{v}/\overline{p}]$.

Our type `dyn` violates this condition, since the defined typed `dyn` appears to the left of an arrow type in the `dyn -> dyn`-typed field of the `Function` constructor.

To see what goes wrong if F* were to accept this definition, we can suppress the error reported by using the option `__no_positivity` and see what happens.

```
#push-options "--__no_positivity"
noeq
type dyn =
  | Bool : bool -> dyn
  | Int  : int -> dyn
  | Function : (dyn -> dyn) -> dyn
#pop-options
```

Note: F* maintains an internal stack of command line options. The `#push-options` pragma pushes additional options at the top of the stack, while `#pop-options` pops the stack. The pattern used here instructs F* to typecheck `dyn` only with the `__no_positivity` option enabled. As we will see, the `__no_positivity` option can be used to break soundness, so use it only if you really know what you're doing.

Now, having declared that `dyn` is a well-formed inductive type, despite not being strictly positive, we can break the soundness of F*. In particular, we can write terms and claim they are total, when in fact their execution will loop forever.

```
let loop' (f:dyn)
  : dyn
  = match f with
    | Function g -> g f
    | _ -> f

let loop
  : dyn
  = loop' (Function loop')
```

Here, the type of `loop` claims that it is a term that always evaluates in a finite number of steps to a value of type `dyn`. Yet, reducing it produces an infinite chain of calls to `loop' (Function loop')`. Admitting a non-positive definition like `dyn` has allowed us to build a non-terminating loop.

Such loops can also allow one to prove `False` (breaking soundness), as the next example shows.

```
#push-options "--__no_positivity"
noeq
type non_positive =
  | NP : (non_positive -> False) -> non_positive
#pop-options

let almost_false (f:non_positive)
  : False
  = let NP g = f in g f

let ff
  : False
  = almost_false (NP almost_false)
```

This example is very similar to dyn, except NP stores a non-positive function that returns False, which allows use to prove `ff : False`, i.e., in this example, not only does the violation of strict positivity lead to an infinite loop at runtime, it also renders the entire proof system of F* useless, since one can prove False.

Finally, in the example below, although the type also_non_pos does not syntactically appear to the left of an arrow in a field of the ANP constructor, an instantiation of the type parameter f (e.g., with the type f_false) does make it appear to the left of an arrow—so this type too is deemed not strictly positive, and can be used to prove False.

```
#push-options "--__no_positivity"
noeq
type also_non_pos (f:Type -> Type) =
  | ANP : f (also_non_pos f) -> also_non_pos f
#pop-options

let f_false
  : Type -> Type
  = fun a -> (a -> False)

let almost_false_again
  : f_false (also_non_pos f_false)
  = fun x -> let ANP h = x in h x

let ff_again
  : False
  = almost_false_again (ANP almost_false_again)
```

We hope you are convinced that non-strictly positive types should not be admissible in inductive type definitions. In what follows, we will no longer use the __no_positivity option. In a later section, once we've introduced the *effect of divergence*, we will see that non-positive definitions can safely be used in a context where programs are not expected to terminate, allowing one to safely model things like the dyn type, without compromising the soundness of F*.

12.1.1 Strictly Positive Annotations

Sometimes it is useful to parameterize an inductive definition with a type function, without introducing a non-positive definition as we did in `also_non_pos` above.

For example, the definition below introduces a type `free f a`, a form of a tree whose leaf nodes contain `a` values, and whose internal nodes branch according the type function `f`.

```
noeq
type free (f:([@@@ strictly_positive] _:Type -> Type))
          (a:Type)
  : Type =
  | Leaf : a -> free f a
  | Branch : f (free f a) -> free f a
```

We can instantiate this generic `free` to produce various kinds of trees. Note: when instantiating `free list a` in `variable_branching_list` below, we need to explicitly re-define the `list` type with a strict-positivity annotation: F* does not correctly support rechecking type constructors to prove that they are strictly positive when they are used at higher order.

```
let binary_tree (a:Type) = free (fun t -> t & t) a
let list_redef ([@@@strictly_positive] a:Type) = list a
let variable_branching_list a = free list_redef a
let infinite_branching_tree a = free (fun t -> nat -> t) a
```

However, we should only be allowed to instantate `f` with type functions that are strictly positive in their argument, since otherwise we can build a proof of `False`, as we did with `also_non_pos`. The `@@@strictly_positive` attribute on the formal parameter of `f` enforces this.

If we were to try to instantiate `free` with a non-strictly positive type function,

```
let free_bad = free (fun t -> (t -> False)) int
```

then F* raises an error:

```
Binder (t: Type) is marked strictly positive, but its use in the definition is not
```

12.1.2 Unused Annotations

Sometimes one indexes a type by another type, though the index has no semantic meaning. For example, in several F* developments that model mutable state, the a heap reference is just a natural number modeling its address in the heap. However, one might use the type `let ref (a:Type) = nat` to represent the type of a reference, even though the type `a` is not used in the definition. In such cases, it can be useful to mark the parameter as unused, to inform F*'s positivity checker that the type index is actually irrelevant. The snippet below shows an example:

```
irreducible
let ref ([@@@unused] a:Type) = nat
noeq
type linked_list (a:Type) =
  | LL : ref (a & linked_list a) -> linked_list a
noeq
type neg_unused =
  | NU : ref (neg_unused -> bool) -> neg_unused
```

Here, we've marked the parameter of `ref` with the `unused` attribute. We've also marked `ref` as `irreducible` just to ensure for this example that F* does not silently unfold the definition of `ref`.

Now, knowing that the parameter of `ref` is unused, one can define types like `linked_list a`, where although `linked_list a` appears as an argument to the `ref` type, the positivity checker accepts it, since the parameter is unused. This is similar to the use of a `strictly_positive` annotation on a parameter.

However, with the `unused` attribute, one can go further: e.g., the type `neg_unused` shows that even a negative occurrence of the defined type is accepted, so long as it appears only as an instantiation of an unused parameter.

CHAPTER

THIRTEEN

LENGTH-INDEXED LISTS

To make concrete some aspects of the formal definitions above, we'll look at several variants of a parameterized list datatype augmented with indexes that carry information about the list's length.

13.1 Even and Odd-lengthed Lists

Our first example is bit artifical, but helps illustrate a usage of mutually inductive types.

Here, we're defining two types constructors called `even` and `odd`, (i.e, just T_1 and T_2 from our formal definition), both with a single parameter `(a:Type)`, for the type of the lists' elements, and no indexes.

All lists of type `even a` have an even number of elements—zero elements, using its first constructor `ENil`, or using `ECons`, one more than the number of elements in an `odd a`, a list with an odd number of elements. Elements of the type `odd a` are constructed using the constructor `OCons`, which adds a single element to an `even a` list. The types are mutually inductive since their definitions reference each other.

```
type even (a:Type) =
 | ENil : even a
 | ECons : a -> odd a -> even a
and odd (a:Type) =
 | OCons : a -> even a -> odd a
```

Although closely related, the types `even a` and `odd a` are from distinct inductive types. So, to compute, say, the length of one of these lists one generally write a pair of mutually recursive functions, like so:

```
let rec elength #a (e:even a)
  : n:nat { n % 2 == 0}
  = match e with
    | ENil -> 0
    | ECons _ tl -> 1 + olength tl
and olength #a (o:odd a)
  : n:nat { n % 2 == 1 }
  = let OCons _ tl = o in
    1 + elength tl
```

Note, we can prove that the length of an `even a` and `odd a` are really even and odd.

Now, say, you wanted to map a function over an `even a`, you'd have to write a pair of mutually recursive functions to map simultaneoulsy over them both. This can get tedious quickly. Instead of rolling out several mutually inductive but distinct types, one can instead use an *indexed* type to group related types in the same inductive family of types.

The definition of even_or_odd_list below shows an inductive type with one parameter a, for the type of lists elements, and a single boolean index, which indicates whether the list is even or odd. Note how the index varies in the types of the constructors, whereas the parameter stays the same in all instances.

```
type even_or_odd_list (a:Type) : bool -> Type =
 | EONil : even_or_odd_list a true
 | EOCons : a -> #b:bool -> even_or_odd_list a b -> even_or_odd_list a (not b)
```

Now, we have a single family of types for both even and odd lists, and we can write a single function that abstracts over both even and odd lists, just by abstracting over the boolean index. For example, eo_length computes the length of an even_or_odd_list, with its type showing that it returns an even number with b is true and an odd number otherwise.

```
let rec eo_length #a #b (l:even_or_odd_list a b)
  : Tot (n:nat { if b then n % 2 == 0 else n % 2 == 1})
        (decreases l)
  = match l with
    | EONil -> 0
    | EOCons _ tl -> 1 + eo_length tl
```

Note: Note, in eo_length we had to explicitly specify a decreases clause to prove the function terminating. Why? Refer back to the section on *default measures* to recall that by default is the lexicographic ordering of all the arguments in order. So, without the decreases clause, F* will try to prove that the index argument b decreases on the recursive call, which it does not.

This is our first type with with both parameters and indices. But why stop at just indexing to distinguish even and odd-lengthed lists? We can index a list by its length itself.

13.2 Vectors

Let's look again at the definition of the vec type, first shown in *the introduction*.

```
type vec (a:Type) : nat -> Type =
  | Nil : vec a 0
  | Cons : #n:nat -> hd:a -> tl:vec a n -> vec a (n + 1)
```

Here, we're defining just a single type constructor called vec (i.e, just T_1), which a single parameter (a:Type) and one index nat.

vec has two data constructors: Nil builds an instance of vec a 0, the empty vector; and Cons hd tl builds an instance of vec a (n + 1) from a head element hd:a and a tail tl : vec a n. That is, the two constructors build different instances of vec—those instances have the same parameter (a), but different indexes (0 and n + 1).

Note: Datatypes in many languages in the ML family, including OCaml and F#, have parameters but no indexes. So, all the data construtors construct the same instance of the type constructor. Further, all data constructors take at most one argument. If your datatype happens to be simple enough to fit these restrictions, you can use a notation similar to OCaml or F* for those types in F*. For example, here's the option type defined in F* using an OCaml-like notation.

```
type option a =
  | None
  | Some of a
```

This is equivalent to

```
type option a =
  | None : option a
  | Some : a -> option a
```

13.3 Getting an element from a vector

With our length-indexed `vec` type, one can write functions with types that make use of the length information to ensure that they are well-defined. For example, to get the `i` th element of a vector, one can write:

```
let rec get #a #n (i:nat{i < n}) (v:vec a n)
  : a
  = match v with
    | Nil -> false_elim()
    | Cons hd tl ->
      if i = 0 then hd
      else get (i - 1) tl
```

The type of `get i v` says that `i` must be less than `n`, where `n` is the length of `v`, i.e., that `i` is within bounds of the vector, which is enough to prove that `get i v` can always return an element of type `a`. Let's look a bit more closely at how this function is typechecked by F*.

The first key bit is pattern matching `v`:

```
match v with
| Nil -> false_elim()
| Cons hd tl ->
```

In case `v` is `Nil`, we use the library function `Prims.false_elim : squash False -> a` to express that this case is impossible. Intuitively, since the index `i` is a natural number strictly less than the length of the list, we should be able to convince F* that `n <> 0`.

The way this works is that F* typechecks the branch in a context that includes an *equation*, namely that the `v : vec a n` equals the pattern `Nil : vec a 0`. With the assumption that `v == Nil` in the context, F* tries to check that `false_elim` is well-typed, which in turn requires `() : squash False`. This produces an proof obligation sent to the SMT solver, which is able to prove `False` in this case, since from `v = Nil` we must have that `n = 0` which contradicts `i < n`. Put another way, the branch where `v = Nil` is unreachable given the precondition `i < n`.

Note: When a branch is unreachable, F* allows you to just omit the branch altogether, rather than writing it an explicitly calling `false_elim`. For example, it is more common to write:

```
let rec get #a #n (i:nat{i < n}) (v:vec a n)
  : a
  = let Cons hd tl = v in
    if i = 0 then hd
    else get (i - 1) tl
```

where `let Cons hd tl = v` pattern matches `v` against just `Cons hd tl`. F* automatically proves that the other cases of the match are unreachable.

Now, turning to the second case, we have a pattern like this:

```
match v with
| Cons hd tl ->
```

But, recall that `Cons` has an implicit first argument describing the length of `tl`. So, more explicitly, our pattern is of the form below, where `tl : vec a m`.

```
match v with
| Cons #m hd tl ->
```

F* typechecks the branch in a context that includes the equation that `v == Cons #m hd tl`, which lets the solve conclude that `n == m + 1`, from the type of `Cons`.

If `i=0`, we've found the element we want and return it.

Otherwise, we make a recursive call `get (i - 1) tl` and now F* has to:

- Instantiate the implicit argument of `get` to `m`, the length of `tl`. That is, in explicit form, this recursive call is really `get #m (i - 1) tl`. F* does this by relying on a unification algorithm implemented as part of its type inference procedure.
- Prove that `(i - 1) < m`, which follows from `i < n` and `n == m + 1`.
- Prove that the recursive call terminates, by proving that `m << n`, or, equivalently, since `m` and `n` are natural numbers, `m < n`. This is easy, since we have `n == m + 1`.

Let's try a few exercises. The main work is to find a type for the functions in question. Once you do, the rest of the code will "write itself".

13.4 Exercises

13.4.1 Exercise: Concatenating vectors

Click here for the exercise file.

Implement a function to concatenate vectors. It should have the following signature:

```
val append (#a:Type) (#n #m:nat) (v1:vec a n) (v2:vec a m)
  : vec a (n + m)
```

Answer

```
let rec append #a #n #m (v1:vec a n) (v2:vec a m)
  : vec a (n + m)
  = match v1 with
    | Nil -> v2
    | Cons hd tl -> Cons hd (append tl v2)
```

13.4.2 Exercise: Splitting a vector

Write a function called split_at to split a vector v : vec a n at index i into its i -length prefix from position 0 and a suffix starting at i.

Answer

```
let rec split_at #a #n (i:nat{i <= n}) (v:vec a n)
  : vec a i & vec a (n - i)
  = if i = 0
    then Nil, v
    else let Cons hd tl = v in
         let l, r = split_at (i - 1) tl in
         Cons hd l, r
```

Write a tail-recursive version of split_at. You will need a reverse function as a helper.

Answer

```
let rec reverse #a #n (v:vec a n)
  : vec a n
  = match v with
    | Nil -> Nil
    | Cons hd tl -> append (reverse tl) (Cons hd Nil)
```

```
let split_at_tail #a #n (i:nat{i <= n}) (v:vec a n)
  : vec a i & vec a (n - i)
  = let rec aux (j:nat{j <= i})
                (v:vec a (n - (i - j)))
                (out:vec a (i - j))
      : vec a i & vec a (n - i)
      = if j = 0
        then reverse out, v
        else let Cons hd tl = v in
             aux (j - 1) tl (Cons hd out)
    in
    aux i v Nil
```

Bonus: Prove split_at and split_at_tail are equivalent.

13.5 Vectors: Probably not worth it

Many texts about dependent types showcase length-indexed vectors, much as we've done here. Although useful as a simple illustrative example, the vec type we've seen is probably not what you want to use in practice. Especially in F*, where regular lists can easily be used with refinement types, length-indexed vectors are redundant because we simply refine our types using a length function. The code below shows how:

```
module LList

let rec length #a (l:list a)
```

(continues on next page)

(continued from previous page)

```
  : nat
  = match l with
    | [] -> 0
    | _::tl -> 1 + length tl

let rec get #a (i:nat) (v:list a { i < length v })
  = let hd :: tl = v in
    if i = 0 then hd
    else get (i - 1) tl

let rec split_at #a (i:nat) (v:list a { i <= length v })
  : r:(list a & list a){
        length (fst r) == i /\
        length (snd r) == (length v - i)
     }
  = if i = 0
    then [], v
    else let hd :: tl = v in
         let l, r = split_at (i - 1) tl in
         hd::l, r

let rec append #a (v1 v2:list a)
  : v:list a { length v == length v1 + length v2 }
  = match v1 with
    | [] -> v2
    | hd::tl -> hd :: append tl v2
```

In the next few sections, we'll see more useful examples of indexed inductive types than just mere length-indexed vectors.

CHAPTER

FOURTEEN

MERKLE TREES

A Merkle tree is a cryptographic data structure designed by Ralph Merkle in the late 1970s and has grown dramatically in prominence in the last few years, inasmuch as variants of Merkle trees are at the core of most blockchain systems.

A Merkle tree makes use of cryptographic hashes to enable efficient cryptographic proofs of the authenticity of data stored in the tree. In particular, for a Merkle tree containing 2^n data items, it only takes n hash computations to prove that a particular item is in the tree.

In this section, we build a very simple, but canonical, Merkle tree and prove it correct and cryptographically secure. And we'll use several indexed inductive types to do it. Thanks to Aseem Rastogi for this example!

14.1 Setting

Merkle trees have many applications. To motivate our presentation here, consider the following simple scenario.

A content provider (someone like, say, the New York Times) has a large archive of digital artifacts—documents, multimedia files, etc. These artifacts are circulated among users, but when receiving an artifact one may question its authenticity. One way to ensure the authenticity of received artifacts is for the content provider to use a digital signature based on a public-key cryptosystem and for users to verify these signatures upon receiving an artifact. However, signatures can be quite heavyweight for certain applications.

Instead, the content provider can organize their archive into a Merkle tree, a tree of hashes with the artifacts themselves stored at the leaves, such that a single hash associated with the root node of the tree authenticates *all* the artifacts in the tree. By publishing just this root hash, and associating with each artifact a path in the tree from the root to it, a skeptical client can quickly check using a small number of hash computations (logarithmic in the size of the entire archive) whether or not a given artifact is authentic (by recomputing the root hash and checking if it matches the known published root hash).

14.2 Intuitions

Our Merkle tree will be a full binary tree of height n storing 2^n data items and their corresponding hashes at the nodes. The main idea of a Merkle tree is for each internal node to also maintain a *hash of the hashes* stored at each of its children. If the hash algorithm being used is cryptographically secure, in the sense that it is collision resistant (i.e., it is computationally hard to find two strings that hash to the same value), then the hash associated with the root node authenticates the content of the entire tree.

Informally, a Merkle tree is an authenticated data structure in that it is computationally hard to tamper with any of the data items in the tree while still producing the same root hash. Further, to prove that a particular data item `d` is in the tree, it suffices to provide the hashes associated with the nodes in the path from the root to that the leaf containing that item `d`, and one can easily check by comparing hashes that the claimed path is accurate. In fact, we can prove that if a claimed path through the tree attests to the presence of some other item `d' <> d`, then we can construct a collision on the underlying hash algorithm—this property will be our main proof of security.

14.3 Preliminaries

We'll model the resources and the hashes we store in our tree as strings of characters. F* standard library `FStar.String` provides some utilities to work with strings.

In the code listing below, we define the following

- `lstring n`, the type of strings of length `n`. Like the `vec` type, `lstring` is a length-indexed type; unlike `vector` it is defined using a refinement type rather than an indexed inductive type. Defining indexed types using refinements is quite common in F*.
- `concat`, a utility to concatenate strings, with its type proving that the resulting string's length is the sum of the lengths of the input strings.
- `hash_size` and `hash`, a parameter of our development describing the length in characters of a `hash` function. The F* keyword `assume` allows you to assume the existence of a symbol at a given type. Use it with care, since you can trivially prove anything by including an `assume nonsense : False`.
- The type of resources we store in the tree will just be `resource`, an alias for `string`.

```
//Length-indexed strings
let lstring (n:nat) = s:string{String.length s == n}

//Concatenating strings sums their lengths
let concat #n #m (s0:lstring n) (s1:lstring m)
  : lstring (m + n)
  = FStar.String.concat_length s0 s1;
    s0 ^ s1

//A parameter, length of the hash in characters,
//e.g., this would be 32, if a character is 1 byte
//and we're using SHA-256
assume
val hash_size:nat

//The type of a hashed value
let hash_t = lstring hash_size

//The hash function itself
assume
val hash (m:string) : hash_t

//The type of resources stored in the tree
let resource = string
```

14.4 Defining the Merkle tree

The inductive type mtree below defines our Merkle tree. The type has *two* indices, such that mtree n h is the type of a Merkle tree of height n whose root node is associated with the hash h.

Leaves are trees of height 0 and are constructed using L res, where the hash associated with this node is just hash res, the hash of the resource stored at the leaf.

Internal nodes of the tree are constructed using N left right, where both the left and right trees have the same height n, producing a tree of height n + 1. More interestingly, the hash associated with N left right is hash (concat hl hr), the hash of the concatenation of hashes of the left and right subtrees.

```
type mtree: nat -> hash_t -> Type =
  | L:
     res:resource ->
     mtree 0 (hash res)

  | N:
     #n:nat ->
     #hl:hash_t ->
     #hr:hash_t ->
     left:mtree n hl ->
     right:mtree n hr ->
     mtree (n + 1) (hash (concat hl hr))
```

In our previous examples like vectors, the index of the type abstracts, or summarizes, some property of the type, e.g., the length. This is also the case with mtree, where the first index is an abstraction summarizing only the height of the tree; the second index, being a cryptographic hash, summarizes the entire contents of the tree.

14.5 Accessing an element in the tree

A resource identifier resource_id is a path in the tree from the root to the leaf storing that resource. A path is just a list of booleans describing whether to descend left or right from a node.

Just like a regular binary tree, it's easy to access an element in the tree by specifying its resource_id.

14.5.1 Exercise

Implement a function to access an element in a mtree in given a rid : list bool. Figuring out its type, including its decreases clause, is the most interesting part. The function itself is straightforward.

Exercise file

Answer

```
let resource_id = list bool

let rec get #h
            (ri:resource_id)
            (tree:mtree (L.length ri) h)
  : Tot resource (decreases ri)
  = match ri with
    | [] -> L?.res tree
```

(continues on next page)

(continued from previous page)

```
    | b::ri' ->
      if b then
        get ri' (N?.left tree)
      else
        get ri' (N?.right tree)
```

14.6 The Prover

Unlike the ordinary get function, we can define a function get_with_evidence that retrieves a resource from the tree together with some evidence that that resource really is present in the tree. The evidence contains the resource identifier and the hashes of sibling nodes along the path from root to that item.

First, we define resource_with_evidence n, an indexed type that packages a res:resource with its rid:resource_id and hashes:list hash_t—both rid and hashes have the same length, which is the index of the constructed type.

The function get_with_evidence is similar to get, except as it returns from descending into a child node, it adds the hash of the other child node to the list of hashes.

```
type resource_with_evidence : nat -> Type =
  | RES:
      res:resource ->
      ri:resource_id ->
      hashes:list hash_t { L.length ri == L.length hashes } ->
      resource_with_evidence (L.length ri)

/// Retrieves data references by the path, together with the hashes
/// of the sibling nodes along that path
let rec get_with_evidence (#h:_)
                          (rid:resource_id)
                          (tree:mtree (L.length rid) h)
  : Tot (resource_with_evidence (L.length rid))
        (decreases rid)
  = match rid with
    | [] ->
      RES (L?.res tree) [] []

    | bit::rid' ->
      let N #_ #hl #hr left right = tree in
      let p = get_with_evidence rid' left in
      if bit then
        let p = get_with_evidence rid' left in
        RES p.res rid (hr :: p.hashes)
      else
        let p = get_with_evidence rid' right in
        RES p.res rid (hl :: p.hashes)
```

In the cryptographic literature, this function is sometimes called *the prover*. A RES r ri hs is a claimed proof of the membership of r in the tree at the location specified by ri.

Going back to our motivating scenario, artifacts distributed by our content provider would be elements of the type `resource_with_evidence n`, enabling clients to verify that a given artifact is authentic, as shown next.

14.7 The Verifier

Our next step is to build a checker of claimed proofs, sometimes called *a verifier*. The function `verify` below takes a `p:resource_with_evidence n`, re-computes the root hash from the evidence presented, and checks that that hash matches the root hash of a given Merkle tree. Note, the `tree` itself is irrelevant: all that's needed to verify the evidence is *the root hash* of the Merkle tree.

```
let verify #h #n (p:resource_with_evidence n)
                 (tree:mtree n h)
  : bool
  = compute_root_hash p = h
```

The main work is done by `compute_root_hash`, shown below.

- In the first branch, we simply hash the resource itself.
- In the second branch, we recompute the hash from the tail of the path, and then based on which direction was taken, we either concatenate sibling hash on the left or the right, and hash the result.

```
let tail #n (p:resource_with_evidence n { n > 0 })
  : resource_with_evidence (n - 1)
  = RES p.res (L.tail p.ri) (L.tail p.hashes)

let rec compute_root_hash (#n:nat)
                          (p:resource_with_evidence n)
  : hash_t
  = let RES d ri hashes = p in
    match ri with
    | [] -> hash p.res

    | bit::ri' ->
      let h' = compute_root_hash (tail p) in
      if bit then
        hash (concat h' (L.hd hashes))
      else
        hash (concat (L.hd hashes) h')
```

Convince yourself of why this is type-correct—refer back to the description of *vectors*, if needed. For example, why is it safe to call `L.hd` to access the first element of `hashes`?

14.8 Correctness

Now, we can prove our main correctness theorem, namely that `get_with_evidence` returns a resource with verifiable evidence.

```
// Correctness theorem:
//
// Using get_with_evidence's with compute_root_hash correctly
```

(continues on next page)

(continued from previous page)

```
// reconstructs the root hash
let rec correctness (#h:hash_t)
                    (rid:resource_id)
                    (tree:mtree (L.length rid) h)
  : Lemma (ensures (verify (get_with_evidence rid tree) tree))
          (decreases rid)
  = match rid with
    | [] -> ()
    | bit::rid' ->
      let N left right = tree in
      if bit then
        correctness rid' left
      else
        correctness rid' right
```

The proof is a simple proof by induction on the height of the tree, or equivalently, the length of the resource id.

In other words, evidence constructed by a honest prover is accepted by our verifier.

14.9 Security

The main security theorem associated with this construction is the following: if the verifier can be convinced to accept a resource with evidence of the form `RES r rid hs`, and if the resource in the Merkle tree associated with `rid` is *not* `r`, then we can easily construct a collision on the underlying cryptographic hash. Since the hash is meant to be collision resistant, one should conclude that it is at least as hard to convince our verifier to accept incorrect evidence as it is to find collisions on the underlying hash.

We start by defining the type of a `hash_collision`, a pair of distinct strings that hash to the same value.

```
type hash_collision =
  | Collision :
      s1:string ->
      s2:string {hash s1 = hash s2 /\ not (s1 = s2)} ->
      hash_collision
```

The `security` theorem shown below takes a `tree` and `p:resource_with_evidence n`, where the refinement on `p` states that the verifier accepts the evidence (`verify p tree`) although the resource associated with `p.ri` is not `p.res`: in this case, we can build a function, by induction on the height of the tree, that returns a hash collision.

```
(*
 * If [verify] can be tricked into accepting the evidence of [p] when
 * [p.res] is not actually present in the tree at [p.ri], then
 * we can exhibit a hash collision
 *)
let rec security (#n:nat) (#h:hash_t)
                 (tree:mtree n h)
                 (p:resource_with_evidence n {
                   verify p tree /\
                   not (get p.ri tree = p.res)
                 })
  : hash_collision
  = match p.ri with
```

(continues on next page)

(continued from previous page)

```
    | [] -> Collision p.res (L?.res tree)
    | bit::rid' ->
      let N #_ #h1 #h2 left right = tree in
      let h' = compute_root_hash (tail p) in
      let hd :: _ = p.hashes in
      if bit then
        if h' = h1 then
          security left (tail p)
        else (
          String.concat_injective h1 h' h2 hd;
          Collision (concat h1 h2) (concat h' hd)
        )
      else
        if h' = h2 then
          security right (tail p)
        else (
          String.concat_injective h1 hd h2 h';
          Collision (concat h1 h2) (concat hd h')
        )
```

We look at its cases in detail:

- In the base case, it's easy to construct a hash collision directly from the differing resources.
- Otherwise, we recompute the hash associate with the current node from the tail of the evidence presented, and the two cases of the left and right subtrees are symmetric.
  - If the recomputed hash matches the hash of the node, then we can generate a collision just by the induction hypothesis on the left or right subtree.
  - Otherwise, we can build a hash collision, relying on `String.concat_injective`, a lemma from the library stating that the concatenation of two pairs of equal length strings are equal only if their components are. Knowing that `h' <> h1` (or, symmetically, `h' <> h2`) this allows us to prove that the concatenations are unequal, although their hashes are, by assumption, equal.

14.10 Exercise

Implement a function to update an `mtree` at a given `rid:resource_id` with a new resource `res:resource`. The resulting tree will have a new root hash, so you will have to return the new hash along with the updated tree.

Exercise file

Hint

One type of the `update` function could be as follows:

```
type mtree' (n:nat) =
  | MTree : h:hash_t -> mtree n h -> mtree' n

val update_mtree'  (#h:hash_t)
                   (rid:resource_id)
                   (res:resource)
                   (tree:mtree (L.length rid) h)
   : mtree' (L.length rid)
```

Answer

```
let rec update_mtree' #h
                      (rid:resource_id)
                      (res:resource)
                      (tree:mtree (L.length rid) h)
   : Tot (mtree' (L.length rid))
        (decreases rid)
   = match rid with
     | [] -> MTree _ (L res)
     | hd :: rid' ->
       if hd
       then (
         let MTree _ t = update_mtree' rid' res (N?.left tree) in
         MTree _ (N t (N?.right tree))
       )
       else (
         let MTree _ t = update_mtree' rid' res (N?.right tree) in
         MTree _ (N (N?.left tree) t)
       )
```

One interesting part of our solution is that we never explicitly construct the hash of the nodes. Instead, we just use _ and let F* infer the calls to the hash functions.

14.11 Summary and Further Reading

In summary, we've built a simple but powerful authenticated data structure with a proof of its correctness and cryptographic security.

In practice, Merkle trees can be much more sophisticated than our the most basic one shown here. For instance, they can support incremental updates, contain optimizations for different kinds of workloads, including sparse trees, and be implemented using high-performance, mutable structures.

You can read more about various flavors of Merkle trees implemented in F* in the following papers.

- EverCrypt, Section VII (B), describes a high-performance Merkle tree with fast incremental updates.
- FastVer describes the design and use of hybrid authenticated data strutures, including sparse Merkle trees, for applications such as verifiable key-value stores.

CHAPTER

FIFTEEN

EQUALITY TYPES

In an *early section* we learned that F* supports at least two kinds of equality. In this section, we look in detail at definitional equality, propositional equality, extensional equality of functions, and decidable equality. These topics are fairly technical, but are core features of the language and their treatment in F* makes essential use of an indexed inductive type, `equals #t x y`, a proposition asserting the equality of `x:t` and `y:t`.

Depending on your level of comfort with functional programming and dependent types, you may want to skip or just skim this chapter on a first reading, returning to it for reference if something is unclear.

15.1 Definitional Equality

One of the main distinctive feature of a type theory like F* (or Coq, Lean, Agda etc., and in contrast with foundations like set theory) is that *computation* is a primitive notion within the theory, such that lambda terms that are related by reduction are considered identical. For example, there is no way to distinguish within the theory between $(\lambda x.x) 0$ and 0, since the former reduces in a single step of computation to the latter. Terms that are related by reduction are called *definitionally equal*, and this is the most primitive notion of equality in the language. Definitional equality is a congruence, in the sense that within any context $T[]$, $T[n]$ is definitionally equal to $T[m]$, when n and m are definitionally equal.

Since definitionally equal terms are identical, all type theories, including F*, will implicit allow treating a term `v:t` as if it had type `t'`, provided `t` and `t'` are definitionally equal.

Let's look at a few examples, starting again with our type of length-indexed vectors.

```
type vec (a:Type) : nat -> Type =
  | Nil : vec a 0
  | Cons : #n:nat -> hd:a -> tl:vec a n -> vec a (n + 1)
```

As the two examples below show a `v:vec a n` is also has type `vec a m` when `n` and `m` are definitionally equal.

```
let conv_vec_0 (#a:Type) (v:vec a ((fun x -> x) 0))
  : vec a 0
  = v

let conv_vec_1 (#a:Type) (v:vec a ((fun x -> x + 1) 0))
  : vec a 1
  = v
```

In the first case, a single step of computation (a function application, or β-reduction) suffices; while the second case requires a β-reduction followed by a step of integer arithmetic. In fact, any computational step, including unfolding defintions, conditionals, fixpoint reduction etc. are all allowed when deciding if terms are definitionally equivalent—the code below illustrates how F* implicitly reduces the `factorial` function when deciding if two terms are definitionally equal.

```
let rec factorial (n:nat)
  : nat
  = if n = 0 then 1
    else n * factorial (n - 1)

let conv_vec_6 (#a:Type) (v:vec a (factorial 3))
  : vec a 6
  = v
```

Of course, there is nothing particularly special about the `vec` type or its indices. Definitional equality applies everywhere, as illustrated below.

```
let conv_int (x : (fun b -> if b then int else bool) true)
  : int
  = x + 1
```

Here, when adding 1 to `x`, F* implicitly converts the type of `x` to `int` by performing a β-reduction followed by a case analysis.

15.2 Propositional Equality

Definitional equality is so primitive in the language that there is no way to even state within the terms that two terms are definitional equal, i.e., there is no way to state within the logic that two terms are related to each other by reduction. The closest one can get stating that two terms are equal is through a notion called a *provable equality* or propositional equality.

In thinking of propositions as types, we mentioned at the *very start of the book*, that one can think of a type `t` as a proposition, or a statement of a theorem, and `e :  t` as a proof of the theorem `t`. So, one might ask, what type corresponds to the equality proposition and how are proofs of equality represented?

The listing below shows the definition of an inductive type `equals #a x y` representing the equality proposotion between `x:a` and `y:a` . Its single constructor `Reflexivity` is an equality proof.

```
type equals (#a:Type) : a -> a -> Type =
  | Reflexivity : #x:a -> equals x x
```

Its easy to construct some simple equality proofs. In the second case, just as with our vector examples, F* accepts `Reflexivity #_ #6` as having type `equals (factorial 3) 6`, since `equals 6 6` is definitionally equal to `equals (factorial 3) 6`.

```
let z_equals_z
  : equals 0 0
  = Reflexivity

let fact_3_eq_6
  : equals (factorial 3) 6
  = Reflexivity #_ #6
```

Although the only constructor of `equals` is `Reflexivity`, as the the following code shows, `equals` is actually an equivalence relation, satisfying (in addition to reflexivity) the laws of symmetry and transitivity.

```
let reflexivity #a (x:a)
  : equals x x
  = Reflexivity

let symmetry #a (x y : a) (pf:equals x y)
  : equals y x
  = Reflexivity

let transitivity #a (x y z : a) (pf1:equals x y) (pf2:equals y z)
  : equals x z
  = Reflexivity
```

This might seem like magic: how is it is that we can derive symmetry and transitivity from reflexivity alone? The answer lies in how F* interprets inductive type definitions.

In particular, given an inductive type definition of type $T\ \overline{p}$, where $\overline{p}$ is a list of parameters and, F* includes an axiom stating that any value $v : T\ \overline{p}$ must be an application of one of the constructors of T, $D\ \overline{v} : T\ \overline{p'}$, such that $\overline{p} = \overline{p'}$.

In the case of equality proofs, this allows F* to conclude that every equality proof is actually an instance of `Reflexivity`, as shown below.

```
let uip_refl #a (x y:a) (pf:equals x y)
  : equals pf (Reflexivity #a #x)
  = Reflexivity
```

Spend a minute looking at the statement above: the return type is a statement of equality about equality proofs. Write down a version of `uip_refl` making all implicit arguments explicit.

Answer

```
let uip_refl_explicit #a (x y:a) (pf:equals x y)
  : equals #(equals x y) pf (Reflexivity #a #x)
  = Reflexivity #(equals x y) #(Reflexivity #a #x)
```

In fact, from `uip_refl`, a stronger statement showing that all equality proofs are equal is also provable. The property below is known as the *uniqueness of identity proofs* (UIP) and is at the core of what makes F* an extensional type theory.

```
let uip #a (x y:a) (pf0 pf1:equals x y)
  : equals pf0 pf1
  = Reflexivity
```

The F* module `Prims`, the very first module in every program's dependence graph, defines the `equals` type as shown here. The provable equality predicate `(==)` that we've used in several examples already is just a squashed equality proof, as shown below.

```
let ( == ) #a (x y : a) = squash (equals x y)
```

In what follows, we'll mostly use squashed equalities, except where we wish to emphasize the reflexivity proofs.

15.3 Equality Reflection

What makes F* an *extensional* type theory (and unlike the *intensional* type theories implemented by Coq, Lean, Agda, etc.) is a feature known as equality reflection. Whereas intensional type theories treat definitional and provable equalities separate, in F* terms that are provably equal are also considered definitionally equal. That is, if in a given context `x == y` is derivable, the `x` is also definitionally equal to `y`. This has some wide-reaching consequences.

15.3.1 Implicit conversions using provable equalities

Recall from the start of the chapter that `v:vec a ((fun x -> x) 0)` is implicitly convertible to the type `vec a 0`, since the two types are related by congruence and reduction. However, as the examples below show, if `a == b` is derivable in the context, then `v:a` can be implicity converted to the type `b`.

```
let pconv_vec_z (#a:Type) (#n:nat) (_:(n == 0)) (v:vec a n)
  : vec a 0
  = v

let pconv_vec_nm (#a:Type) (#n #m:nat) (_:(n == m)) (v:vec a n)
  : vec a m
  = v

let pconv_int (#a:Type) (_:(a == int)) (x:a)
  : int
  = x + 1

let pconv_ab (#a #b:Type) (_:(a == b)) (v:a)
  : b
  = v
```

We do not require a proof of `a == b` to be literally bound in the context. As the example below shows, the hypothesis `h` is used in conjunction with the control flow of the program to prove that in the `then` branch `aa : int` and in the `else` branch `bb : int`.

```
let pconv_der (#a #b:Type)
              (x y:int)
              (h:((x > 0 ==> a == int) /\
                  (y > 0 ==> b == int) /\
                  (x > 0 \/ y > 0)))
              (aa:a)
              (bb:b)
  : int
  = if x > 0 then aa - 1 else bb + 1
```

In fact, with our understanding of equality proofs, we can better explain how case analysis works in F*. In the code above, the `then`-branch is typechecked in a context including a hypothesis `h_then: squash (equals (x > 0) true)`, while the `else` branch includes the hypothesis `h_else: squash (equals (x > 0) false)`. The presence of these additional control-flow hypotheses, in conjunction with whatever else is in the context (in particular hypothesis `h`) allows us to derive `(a == int)` and `(b == int)` in the respective branches and convert the types of `aa` and `bb` accordingly.

15.3.2 Undecidability and Weak Normalization

Implicit conversions with provable equalities are very convenient—we have relied on it without noticing in nearly all our examples so far, starting from the simplest examples about lists to vectors and Merkle trees, and some might say this is the one key feature which gives F* its programming-oriented flavor.

However, as the previous example hinted, it is, in general, undecidable to determine if `a == b` is derivable in a given context. In practice, however, through the use of an SMT solver, F* can often figure out when terms are provably equal and convert using it. But, it cannot always do this. In such cases, the F* standard library offers the following primitive (in FStar.Pervasives), which allows the user to write `coerce_eq pf x`, to explicitly coerce the type of `x` using the equality proof `pf`.

```
let coerce_eq (#a #b:Type) (_:squash (a == b)) (x:a) : b = x
```

Another consequence of equality reflection is the loss of strong normalization. Intensional type theories enjoy a nice property ensuring that every term will reduce to a canonical normal form, no matter the order of evaluation. F* does not have this property, since some terms, under certain evaluation orders, can reduce infinitely. However, metatheory developed for F* proves that closed terms (terms without free variables) in the `Tot` effect do not reduce infinitely, and as a corollary, there are no closed proofs of `False`.

F* includes various heuristics to avoid getting stuck in an infinite loop when reducing open terms, but one can craft examples to make F*'s reduction macinery loop forever. As such, deciding if possibly open terms have the same normal form is also undecidable in F*.

15.4 Functional Extensionality

Functional extensionality is a principle that asserts the provable equality of functions that are pointwise equal. That is, for functions f and g, $\forall x. fx == gx$ implies $f == g$.

This principle is provable as a theorem in F*, but only for function literals, or, equivalently, η-expanded functions. That is, the following is a theorem in F*.

```
let eta (#a:Type) (#b: a -> Type) (f: (x:a -> b x)) = fun x -> f x
let funext_on_eta (#a : Type) (#b: a -> Type) (f g : (x:a -> b x))
                  (hyp : (x:a -> Lemma (f x == g x)))
  : squash (eta f == eta g)
  = _ by (norm [delta_only [`%eta]];
          pointwise (fun _ ->
             try_with
                     (fun _ -> mapply (quote hyp))
                     (fun _ -> trefl()));
           trefl())
```

Note: Note, the proof of the theorem makes use of tactics, a topic we'll cover in a later chapter. You do not need to understand it in detail, yet. The proof roughly says to descend into every sub-term of the goal and try to rewrite it using the pointwise equality hypothesis hyp, and if it fails to just rewrite the sub-term to itself.

Unfortunately, functional extensionality does not apply to all functions. That is, the following is not provable in F* nor is it sound to assume it as an axiom.

```
let funext =
  #a:Type ->
```

(continues on next page)

(continued from previous page)

```
  #b:(a -> Type) ->
  f:(x:a -> b x) ->
  g:(x:a -> b x) ->
  Lemma (requires (forall (x:a). f x == g x))
        (ensures f == g)
```

The problem is illustrated by the following counterexample, which allows deriving `False` in a context where `funext` is valid.

```
let f (x:nat) : int = 0
let g (x:nat) : int = if x = 0 then 1 else 0
let pos = x:nat{x > 0}
let full_funext_false (ax:funext)
  : False
  = ax #pos f g;
    assert (f == g);
    assert (f 0 == g 0);
    false_elim()
```

The proof works by exploiting the interaction with refinement subtyping. `f` and `g` are clearly not pointwise equal on the entire domain of natural numbers, yet they are pointwise equal on the positive natural numbers. However, from `ax #pos f g` we gain that `f == g`, and in particular that `f 0 == g 0`, which is false.

Note: The trouble arises in part because although `ax:funext` proves `squash (equals #(pos -> int) f g)`, F*'s encoding of the equality to the SMT solver (whose equality is untyped) treats the equality as `squash (equals #(nat -> int) f g)`, which leads to the contradiction.

Further, η-equivalent functions in F* are not considered provably equal. Otherwise, in combination with `funext_on_eta`, an η-equivalence principle leads to the same contradiction as `funext_false`, as shown below.

```
let eta_equiv =
  #a:Type ->
  #b:(a -> Type) ->
  f:(x:a -> b x) ->
  Lemma (f == eta f)

let eta_equiv_false (ax:eta_equiv)
  : False
  = funext_on_eta #pos f g (fun x -> ());
    ax #pos f;
    ax #pos g;
    assert (f == g);
    assert (f 0 == g 0);
    false_elim()
```

The F* standard library module `FStar.FunctionalExtensionality` provides more information and several utilities to work with functional extensionality on η-expanded functions.

Thanks in particular to Aseem Rastogi and Dominique Unruh for many insights and discussions related to functional extensionality.

15.5 Exercise

Leibniz equality leq x y, relates two terms x:a and y:a if for all predicates p:a -> Type, p a implies p b. That is, if no predicate can distinguish x and y, the they must be equal.

Define Leibniz equality and prove that it is an equivalence relation.

Then prove that Leibniz equality and the equality predicate equals x y defined above are isomorphic, in the sense that leq x y -> equals x y and equals x y -> leq x y.

Exercise file

Hint

The section on Leibniz equality here tells you how to do it in Agda.

```
let lbz_eq (#a:Type) (x y:a) = p:(a -> Type) -> p x -> p y

// lbz_eq is an equivalence relation
let lbz_eq_refl #a (x:a)
  : lbz_eq x x
  = fun p px -> px
let lbz_eq_trans #a (x y z:a) (pf1:lbz_eq x y) (pf2:lbz_eq y z)
  : lbz_eq x z
  = fun p px -> pf2 p (pf1 p px)
let lbz_eq_sym #a (x y:a) (pf:lbz_eq x y)
  : lbz_eq y x
  = fun p -> pf (fun (z:a) -> (p z -> p x)) (fun (px: p x) -> px)

// equals and lbz_eq are isomorphic
let equals_lbz_eq (#a:Type) (x y:a) (pf:equals x y)
  : lbz_eq x y
  = fun p px -> px
let lbz_eq_equals (#a:Type) (x y:a) (pf:lbz_eq x y)
  : equals x y
  = pf (fun (z:a) -> equals x z) Reflexivity
```

15.6 Decidable equality and equality qualifiers

To end this chapter, we discuss a third kind of equality in F*, the polymorphic *decidable equality* with the signature shown below taken from the the F* module Prims.

```
val ( = ) (#a:eqtype) (x y:a) : bool
```

On eqtype, i.e., a:Type{hasEq a}, decidable quality (=) and provable equality coincide, as shown below.

```
let dec_equals (#a:eqtype) (x y:a) (_:squash (x = y))
  : equals x y
  = Reflexivity

let equals_dec (#a:eqtype) (x y:a) (_:equals x y)
  : squash (x = y)
  = ()
```

That is, for the class of `eqtype`, `x = y` returns a boolean value that decides equality. Decidable equality and `eqtype` were first covered in *an earlier chapter*, where we mentioned that several primitive types, like `int` and `bool` all validate the `hasEq` predicate and are, hence, instances of `eqtype`.

When introducing a new inductive type definition, F* tries to determine whether or not the type supports decidable equality based on a structural equality of the representation of the values of that type. If so, the type is considered an `eqtype` and uses of the `( = )` operator are compiled at runtime to structural comparison of values provided by the target language chosen, e.g., OCaml, F#, or C.

The criterion used to determine whether or not the type supports equality decidable is the following.

Given an inductive type definition of T with parameters $\overline{p}$ and indexes $\overline{q}$, for each constructor of D with arguments $\overline{v : t_v}$,

1. Assume, or every type parameter $t \in \overline{p}$, $\mathsf{hasEq}\ t$.
2. Assume, for recursive types, for all $\overline{q}$, $\mathsf{hasEq}\ (T\ \overline{p}\ \overline{q})$.
3. For all arguments $\overline{v : t_v}$, prove $\mathsf{hasEq}\ t_v$.

If the proof in step 3 suceeds for all constructors, then F* introduces an axiom $\forall\,\overline{p}\,\overline{q}.(\forall t \in \overline{p}.\mathsf{hasEq}\ t) \Rightarrow \mathsf{hasEq}\ (T\ \overline{p}\ \overline{q})$.

If the check in step 3 fails for any constructor, F* reports an error which the user can address by adding one of two qualifiers to the type.

1. `noeq`: This qualifier instructs F* to consider that the type does not support decidable equality, e.g., if one of the constructors contains a function, as show below.

   ```
   noeq
   type itree (a:Type) =
     | End : itree a
     | Node : hd:nat -> tl:(nat -> itree a) -> itree a
   ```

2. `unopteq`: This qualifier instructs F* to determine whether a given instance of the type supports equality, even when some of its parameters are not themselves instances of `eqtype`. This can be useful in situations such as the following:

   ```
   unopteq
   type t (f: Type -> Type) =
     | T : f bool -> t f

   let _ = assert (hasEq (t list))

   [@@expect_failure]
   let _ = assert (hasEq (fun x -> x -> x))
   ```

This wiki page provides more information about equality qualifiers on inductive types.

CHAPTER

SIXTEEN

CONSTRUCTIVE & CLASSICAL CONNECTIVES

In *an earlier chapter*, we learned about the propositional connectives $\forall, \exists, \Rightarrow, \iff, \wedge, \vee, \neg$, etc. Whereas in other logical frameworks these connectives are primitive, in a type theory like F* these connectives are defined notions, built from inductive type definitions and function types. In this section, we take a closer look at these logical connectives, show their definitions, and present some utilities to manipulate them in proofs.

Every logical connective comes in two flavors. First, in its most primitive form, it is defined as an inductive or arrow type, giving a constructive interpretation to the connective. Second, and more commonly used in F*, is a *squashed*, or proof-irrelevant, variant of the same connective—the squashed variant is classical rather than constructive and its proofs are typically derived by writing partial proof terms with the SMT filling in the missing parts.

Each connective has an *introduction* principle (which describes how to build proofs of that connective) and an *elimination* principle (which describes how to use a proof of that connective to build other proofs). Example uses of introduction and elimination principles for all the connectives can be found in ClassicalSugar.fst

All these types are defined in `Prims`, the very first module in all F* programs.

16.1 Falsehood

The `empty` inductive type is the proposition that has no proofs. The logical consistency of F* depends on there being no closed terms whose type is `empty`.

```
type empty =
```

This definition might look odd at first: it defines an inductive type with *zero* constructors. This is perfectly legal in F*, unlike in languages like OCaml or F#.

The squashed variant of `empty` is called `False` and is defined as shown below:

```
let False = squash empty
```

16.1.1 Introduction

The `False` proposition has no introduction form, since it has no proofs.

16.1.2 Elimination

From a (hypothetical) proof of False, one can build a proof of any other type.

```
let empty_elim (#a:Type) (x:empty) : a = match x with
```

This body of elim_false is a match expression with no branches, which suffices to match all the zero cases of the empty type.

FStar.Pervasives.false_elim provides an analogous elimination rule for False, as shown below, where the termination check for the recursive call succeeds trivially in a context with x:False.

```
let rec false_elim (#a:Type) (x:False) : a = false_elim x
```

16.2 Truth

The trivial inductive type has just a single proof, T.

```
type trivial = T
```

Note: Although isomorphic to the unit type with its single element (), for historic reasons, F* uses the trivial type to represent trivial proofs. In the future, it is likely that trivial will just be replaced by unit.

The squashed form of trivial is written True and is defined as:

```
let True = squash trivial
```

16.2.1 Introduction

The introduction forms for both the constructive and squashed variants are trivial.

```
let _ : trivial = T
let _ : True = ()
```

16.2.2 Elimination

There is no elimination form, since proofs of trivial are vacuous and cannot be used to derive any other proofs.

16.3 Conjunction

A constructive proof of p and q is just a pair containing proofs of p and q, respectively.

```
type pair (p q:Type) = | Pair : _1:p -> _2:q -> pair p q
```

Note: This type is isomorphic to the tuple type `p & q` that we encountered previously *here*. F* currently uses a separate type for pairs used in proofs and those used to pair data, though there is no fundamental reason for this. In the future, it is likely that `pair` will just be replaced by the regular tuple type.

The squashed form of conjunction is written `/\` and is defined as follows:

```
let ( /\ ) (p q:Type) = squash (pair p q)
```

16.3.1 Introduction

Introducing a conjunction simply involves constructing a pair.

```
let and_intro #p #q (pf_p:p) (pf_q:q)
  : pair p q
  = Pair pf_p pf_q
```

To introduce the squashed version, there are two options. One can either rely entirely on the SMT solver to discover a proof of `p /\ q` from proofs of `p` and `q`, which it is usually very capable of doing.

```
let conj_intro #p #q (pf_p:squash p) (pf_q: squash q)
  : Lemma (p /\ q)
  = ()
```

Or, if one needs finer control, F* offers specialized syntax (defined in `FStar.Classical.Sugar`) to manipulate each of the non-trivial logical connectives, as shown below.

```
let conj_intro_sugar #p #q (pf_p:squash p) (pf_q: squash q)
  : squash (p /\ q)
  = introduce p /\ q
    with pf_p
    and  pf_q
```

The sugared introduction form for conjunction is, in general, as follows:

```
introduce p /\ q //Term whose top-level connective is /\
with proof_of_p  //proof_of_p : squash p
and  proof_of_q  //proof_of_q : squash q
```

16.3.2 Elimination

Eliminating a conjunction comes in two forms, corresponding to projecting each component of the pair.

```
let and_elim_1 #p #q (pf_pq:p & q)
  : p
  = pf_pq._1

let and_elim_2 #p #q (pf_pq:p & q)
  : q
  = pf_pq._2
```

For the squashed version, we again have two styles, the first relying on the SMT solver.

```
let conj_elim_1 #p #q (pf_pq:squash (p /\ q))
  : squash p
  = ()

let conj_elim_2 #p #q (pf_pq:squash (p /\ q))
  : squash q
  = ()
```

And a style using syntactic sugar:

```
let conj_elim_sugar_1 #p #q (pf_pq:squash (p /\ q))
  : squash p
  = eliminate p /\ q
    returns p
    with pf_p pf_q. pf_p

let conj_elim_sugar_2 #p #q (pf_pq:squash (p /\ q))
  : squash p
  = eliminate p /\ q
    returns q
    with pf_p pf_q. pf_q
```

16.4 Disjunction

A constructive proof of `p` or `q` is represented by the following inductive type:

```
type sum (p q:Type) =
  | Left : p -> sum p q
  | Right : q -> sum p q
```

The constructors `Left` and `Right` inject proofs of `p` or `q` into a proof of `sum p q`.

Note: Just like before, this type is isomorphic to the type `either p q` from `FStar.Pervasives`.

The classical connective `\/` described previously is just a squashed version of `sum`.

```
let ( \/ ) (p q: Type) = squash (sum p q)
```

16.4.1 Introduction

As with the other connectives, introducing a constructive disjunction is just a matter of using the `Left` or `Right` constructor.

To introduce the squashed version `\/`, one can either rely on the SMT solver, as shown below.

```
let or_intro_left #p #q (pf_p:squash p)
  : squash (p \/ q)
  = ()
```

(continues on next page)

(continued from previous page)

```
let or_intro_right #p #q (pf_q:squash q)
  : squash (p \/ q)
  = ()
```

Or, using the following syntactic sugar, one can specifically provide a proof for either the `Left` or `Right` disjunct.

```
let or_intro_sugar_left #p #q (pf_p:squash p)
  : squash (p \/ q)
  = introduce p \/ q
    with Left pf_p

let or_intro_sugar_right #p #q (pf_q:squash q)
  : squash (p \/ q)
  = introduce p \/ q
    with Right pf_q
```

16.4.2 Elimination

Eliminating a disjunction requires a *motive*, a goal proposition to be derived from a proof of `sum p q` or `p \/ q`.

In constructive style, eliminating `sum p q` amounts to just pattern matching on the cases and constructing a proof of the goal by applying a suitable goal-producing hypothesis.

```
let sum_elim #p #q #r (p_or_q: sum p q)
                       (pr: p -> r)
                       (qr: q -> r)
  : r
  = match p_or_q with
    | Left p -> pr p
    | Right q -> qr q
```

The squashed version is similar, except the case analysis can either be automated by SMT or explicitly handled using the syntactic sugar.

```
let or_elim #p #q #r (pf_p:squash (p \/ q))
                     (pf_pr:squash (p ==> r))
                     (pf_qr:squash (q ==> r))
  : squash r
  = ()
```

```
let or_elim_sugar #p #q #r
                  (pf_p:squash (p \/ q))
                  (pf_pr:unit -> Lemma (requires p) (ensures r))
                  (pf_qr:unit -> Lemma (requires q) (ensures r))
  : squash r
  = eliminate p \/ q
    returns r
    with pf_p. pf_pr () //pf_p : squash p
    and  pf_q. pf_qr () //pf_q : squash q
```

16.5 Implication

One of the elimination principles for disjunction used the implication connective ==>. Its definition is shown below:

```
let ( ==> ) (p q : Type) = squash (p -> q)
```

That is, ==> is just the squashed version of the non-dependent arrow type ->.

Note: In `Prims`, the definition of `p ==> q` is actually `squash (p -> GTot q)`, a **ghost** function from `p` to `q`. We'll learn about this more when we encounter effects.

16.5.1 Introduction

Introducing a constructive arrow `p -> q` just involves constructing a λ-literal of the appropriate type.

One can turn several kinds of arrows into implications, as shown below.

One option is to directly use a function from the `FStar.Classical` library, as shown below:

```
val impl_intro_tot (#p #q: Type) (f: (p -> q)) : (p ==> q)
```

However, this form is seldom used in F*. Instead, one often works with functions between squashed propositions, or Lemmas, turning them into implications when needed. We show a few styles below.

```
let implies_intro_1 #p #q (pq: squash p -> squash q)
  : squash (p ==> q)
  = introduce p ==> q
    with pf_p. pq pf_p

let implies_intro_2 #p #q (pq: unit -> Lemma (requires p) (ensures q))
  : squash (p ==> q)
  = introduce p ==> q
    with pf_p. pq pf_p

let implies_intro_3 #p #q (pq: unit -> Lemma (requires p) (ensures q))
  : Lemma (p ==> q)
  = introduce p ==> q
    with pf_p. pq pf_p
```

Unlike the other connectives, there is no fully automated SMT-enabled way to turn an arrow type into an implication. Of course, the form shown above remains just sugar: it may be instructive to look at its desugaring, shown below.

```
let implies_intro_1 (#p #q:Type) (pq: (squash p -> squash q))
  : squash (p ==> q)
  = FStar.Classical.Sugar.implies_intro
           p
           (fun (_: squash p) -> q)
           (fun (pf_p: squash p) -> pq pf_p)
```

`FStar.Squash` and `FStar.Classical` provide the basic building blocks and the sugar packages it into a more convenient form for use.

16.5.2 Elimination

Of course, the elimination form for a constructive implication, i.e., `p -> q` is just function application.

```
let arrow_elim #p #q (f:p -> q) (x:p) : q = f x
```

The elimination rule for the squashed form is the classical logical rule *modus ponens*, which is usually very well automated by SMT, as shown in `implies_elim` below. We also provide syntactic sugar for it, for completeness, though it is seldom used in practice.

```
let implies_elim #p #q (pq:squash (p ==> q)) (pf_p: squash p)
  : squash q
  = ()

let implies_elim_sugar #p #q (pq:squash (p ==> q)) (pf_p: squash p)
  : squash q
  = eliminate p ==> q
    with pf_p
```

16.6 Negation

Negation is just a special case of implication.

In its constructive form, it corresponds to `p -> empty`.

In `Prims`, we define `~p` as `p ==> False`.

Being just an abbreviation for an implication to `False`, negation has no particular introduction or elimination forms of its own. However, the following forms are easily derivable.

16.6.1 Introduction (Exercise)

Prove the following introduction rule for negation:

Exercise file

```
val neg_intro #p (f:squash p -> squash False)
  : squash (~p)
```

Answer

```
let neg_intro #p (f:squash p -> squash False)
  : squash (~p)
  = introduce p ==> False
    with pf_p. f pf_p
```

16.6.2 Elimination (Exercise)

Prove the following elimination rule for negation using the sugar rather than just SMT only.

```
val neg_elim #p #q (f:squash (~p)) (x:unit -> Lemma p)
  : squash (~q)
```

Exercise file

Answer

```
let neg_elim #p #q (f:squash (~p)) (lem:unit -> Lemma p)
  : squash q
  = eliminate p ==> False
    with lem()
```

16.7 Universal Quantification

Whereas implication is represented by the non-dependent arrow `p -> q`, universal quantification corresponds to the dependent arrow `x:t -> q x`. Its classical form in `forall (x:t). q x`, and is defined in as shown below:

```
let ( forall ) #t (q:t -> Type) = squash (x:t -> q x)
```

Note: As with ==>, in `Prims` uses `x:t -> GTot (q x)`, a ghost arrow, though the difference is not yet significant.

16.7.1 Introduction

Introducing a dependent function type `x:t -> p x` is just like introducing a non-dependent one: use a lambda literal.

For the squashed form, F* provides sugar for use with several styles, where names corresponding to each of the `forall`-bound variables on the `introduce` line are in scope for the proof term on the `with` line.

```
let forall_intro_1 (#t:Type)
                   (#q:t -> Type)
                   (f : (x:t -> squash (q x)))
  : squash (forall (x:t). q x)
  = introduce forall (x:t). q x
    with f x

let forall_intro_2 (#t:Type)
                   (#q:t -> Type)
                   (f : (x:t -> Lemma (q x)))
  : squash (forall (x:t). q x)
  = introduce forall (x:t). q x
    with f x

let forall_intro_3 (#t0:Type)
                   (#t1:t0 -> Type)
```

(continues on next page)

(continued from previous page)

```
                   (#q: (x0:t0 -> x1:t1 x0 -> Type))
                   (f : (x0:t0 -> x1:t1 x0 ->  Lemma (q x0 x1)))
  : squash (forall (x0:t0) (x1:t1 x0). q x0 x1)
  = introduce forall (x0:t0) (x1:t1 x0). q x0 x1
    with f x0 x1
```

Note, as `forall_intro_3` shows, the sugar also works for `forall` quantifiers of arities greater than 1.

16.7.2 Elimination

Eliminating a dependent function corresponds to dependent function application.

```
let dep_arrow_elim #t #q (f:(x:t -> q x)) (x:t) : q x = f x
```

For the squashed version, eliminating a `forall` quantifier amounts to instantiating the quantifier for a given term. Automating proofs that require quantifier instantiation is a large topic in its own right, as we'll cover in a later section—this wiki page provides some hints.

Often, eliminating a universal quantifier is automated by the SMT solver, as shown below, where the SMT solver easily instantiates the quantified hypothesis `f` with `a`.

```
let forall_elim_1 (#t:Type)
                  (#q:t -> Type)
                  (f : squash (forall (x:t). q x))
                  (a:t)
  : squash (q a)
  = ()
```

But, F* also provides syntactic sugar to explicitly trigger quantifier insantiation (as shown below), where the terms provided on the `with` line are instantiations for each of the binders on the `eliminate` line.

```
let forall_elim_2 (#t0:Type)
                  (#t1:t0 -> Type)
                  (#q: (x0:t0 -> x1:t1 x0 -> Type))
                  (f : squash (forall x0 x1. q x0 x1))
                  (v0: t0)
                  (v1: t1 v0)
  : squash (q v0 v1)
  = eliminate forall x0 x1. q x0 x1
    with v0 v1
```

Its desugaring may be illuminating:

```
   FStar.Classical.Sugar.forall_elim
           #(t1 v0)
           #(fun x1 -> q v0 x1)
           v1
           (FStar.Classical.Sugar.forall_elim
              #t0
              #(fun x0 -> forall (x1: t1 x0). q x0 x1)
              v0
              ())
```

16.8 Existential Quantification

Finally, we come to existential quantification. Its constructive form is a dependent pair, a dependent version of the pair used to represent conjunctions. The following inductive type is defined in `Prims`.

```
type dtuple2 (a:Type) (b: a -> Type) =
   | Mkdtuple2 : x:a -> y:b x -> dtuple2 a b
```

As with `tuple2`, F* offers specialized syntax for `dtuple2`:

- Instead of `dtuple2 a (fun (x:a) -> b x)`, one writes `x:a & b x`.
- Instead of writing `Mkdtuple2 x y`, one writes `(| x, y |)`.

The existential quantifier `exists (x:t). p x` is a squashed version of the dependent pair:

```
let ( exists ) (#a:Type) (#b:a -> Type) = squash (x:a & b x)
```

16.8.1 Introduction

Introducing a constructive proof of `x:a & b x` is just a question of using the constructor—we show a concrete instance below.

```
let dtuple2_intro (x:int) (y:int { y > x })
  : (a:int & b:int{b > a})
  = (| x, y |)
```

For the squashed version, introducing an `exists (x:t). p x` automatically using the SMT solver requires finding an instance `a` for the quantifier such that `p a` is derivable—this is the dual problem of quantifier instantiation mentioned with universal

In the first example below, the SMT solver finds the instantiation and proof automatically, while in the latter two, the user picks which instantiation and proof to provide.

```
let exists_intro_1 (#t:Type)
                   (#q:t -> Type)
                   (a:t) (b:t)
                   (f : squash (q a /\ q b))
  : squash (exists x. q x)
  = () //instantiation found by SMT, it chose a or b, unclear/irrelevant which

let exists_intro_2 (#t:Type)
                   (#q:t -> Type)
                   (a:t) (b:t)
                   (f : squash (q a))
                   (g : squash (q b))
  : squash (exists x. q x)
  = introduce exists x. q x
    with a //witness
    and f  //proof term of q applied to witness

let exists_intro_3 (#t:Type)
                   (#q:t -> Type)
                   (a:t) (b:t)
```

(continues on next page)

(continued from previous page)

```
                    (f : squash (q a /\ q b))
  : squash (exists x. q x)
  = introduce exists x. q x
    with a
    and f // f: squash (q a /\ q b) implicitly eliminated to squash (q a) by SMT
```

16.8.2 Elimination

Just as with disjunction and conjunction, eliminating `dtuple2` or `exists` requires a motive, a goal proposition that *does not mention* the bound variable of the quantifier.

For constructive proofs, this is just a pattern match:

```
let dtuple2_elim (#t:Type) (#p:t -> Type) (#q:Type)
                 (pf: (x:t & p x))
                 (k : (x:t -> p x -> q))
  : q
  = let (| x, pf_p |) = pf in
    k x pf_p
```

For the `exists`, the following sugar provides an elimination principle:

```
let exists_elim (#t:Type) (#p:t -> Type) (#q:Type)
                 (pf: squash (exists (x:t). p x))
                 (k : (x:t -> squash (p x) -> squash q))
  : squash q
  = eliminate exists (x:t). p x
    returns q
    with pf_p. k x pf_p

let exists_elim_alt (#t:Type) (#p:t -> Type) (#q:Type)
                    (pf: squash (exists (x:t). p x))
                    (k : (x:t -> Lemma (requires p x)
                                       (ensures q)))
  : Lemma q
  = eliminate exists (x:t). p x
    returns q
    with pf_p. k x
```

Names corresponding to the binders on the `eliminate` line are in scope in the `with` line, which additionally binds a name for a proof term corresponding to the body of the existential formula. That is, in the examples above, `x:t` is implicitly in scope for the proof term, while `pf_p:  squash p`.

16.8.3 Exercise

In a *previous exercise*, we defined a function to insert an element in a Merkle tree and had it return a new root hash and an updated Merkle tree. Our solution had the following signature:

```
type mtree' (n:nat) =
  | MTree : h:hash_t -> mtree n h -> mtree' n

val update_mtree'  (#h:hash_t)
                   (rid:resource_id)
                   (res:resource)
                   (tree:mtree (L.length rid) h)
   : mtree' (L.length rid)
```

Revise the solution so that it instead returns a dependent pair. `dtuple2` is already defined in `Prims`, so you don't have to define it again.

Exercise file

Answer

```
let rec update #h
               (rid:resource_id)
               (res:resource)
               (tree:mtree (L.length rid) h)
   : Tot (h':_ & mtree (L.length rid) h')
         (decreases rid)
   = match rid with
     | [] -> (| _, L res |)
     | hd :: rid' ->
       if hd
       then (
         let (| _, t |) = update rid' res (N?.left tree) in
         (| _, N t (N?.right tree) |)
       )
       else (
         let (| _, t |) = update rid' res (N?.right tree) in
         (| _, N (N?.left tree) t|)
       )
```

CHAPTER

SEVENTEEN

SIMPLY TYPED LAMBDA CALCULUS

In this chapter, we look at how inductively defined types can be used to represent both raw data, inductively defined relations, and proofs relating the two.

By way of illustration, we develop a case study in the simply typed lambda calculus (STLC), a very simple programming language which is often studied in introductory courses on the semantics of programming languages. Its syntax, type system, and runtime behavior can be described in just a few lines. The main result we're interested in proving is the soundness of the type system, i.e., that if a program type checks then it can be executed safely without a certain class of runtime errors.

If you haven't seen the STLC before, there are several good resources for it available on the web, including the Software Foundations book, though we'll try to keep the presentation here as self-contained as possible. Thanks to Simon Forest, Catalin Hritcu, and Simon Schaffer for contributing parts of this case study.

17.1 Syntax

The syntax of programs e is defined by the context-free grammar shown below.

$$e \ ::= \ () \mid x \mid \lambda x : t.e_0 \mid e_0 \; e_1$$

This can be read as follows: a program e is either

- the unit value $()$;
- a variable x;
- a lambda term $\lambda x : t.e_0$ associating a variable x to a type t and a some sub-program e_0;
- or, the application of the sub-program e_0 to another sub-program e_1.

The syntax of the type annotation t is also very simple:

$$t \ ::= \ \mathsf{unit} \mid t_0 \to t_1$$

A type t is either

- the `unit` type constant;
- or, arrow type $t_0 \to t_1$ formed from two smaller types t_0 and t_1

This language is very minimalistic, but it can be easily extended with some other forms, e.g., one could add a type of integers, integer constants, and operators like addition and subtraction. We'll look at that as part of some exercises.

We'll define the syntax of types and programs formally in F* as a pair of simple inductive datatypes `typ` (for types) and `exp` (for programs or expressions) with a constructor for each of the cases above.

The main subtlety is in the representation of variables. For example, ignoring the type annotations, in the term $\lambda x.(\lambda x.x)$ the inner lambda binds *a different* x than the outer one, i.e., the term is equivalent to $\lambda x.(\lambda y.y)$ and our representation of programs must respect this this convention. We'll use a technique called de Bruijn indices, where the names of the variables are no longer significant and instead each variable is represented by a natural number describing the number of λ binders that one must cross when traversing a term from the occurrence of the variable to that variable's λ binder.

For example, the terms $\lambda x.(\lambda x.x)$ and $\lambda x.(\lambda y.y)$ are both represented as $\lambda.(\lambda.0)$, since the inner occurrence of x is associated with the inner λ; while $\lambda x.(\lambda y.(\lambda z.x))$ is represented as $\lambda.(\lambda.(\lambda.2)$, since from the inner occurrence of x one must skip past 2 λ's to reach the λ associated with x. Note, the variable names are no longer significant in de Bruijn's notation.

17.1.1 Representing types

The inductive type `typ` defined below is our representation of types.

```
type typ =
  | TUnit : typ
  | TArr  : typ -> typ -> typ
```

This is entirely straightforward: a constructor for each case in our type grammar, as described above.

17.1.2 Representing programs

The representation of program expressions is shown below:

```
let var = nat
type exp =
  | EUnit : exp
  | EVar  : var -> exp
  | ELam  : typ -> exp -> exp
  | EApp  : exp -> exp -> exp
```

This too is straightforward: a constructor for each case in our program grammar, as described above. We use a `nat` to represent variables `var` and `ELam` represents an annotated lambda term of the form $\lambda{:}t.e$, where the name of the binder is omitted, since we're using de Bruijn's representation.

17.2 Runtime semantics

STLC has just one main computation rule to execute a program—the function application rule or a β reduction, as shown below:

$$(\lambda x : t.e_0)\ e_1 \longrightarrow e_0[x \mapsto e_1]$$

This says that when a λ literal is applied to an argument e_1 the program takes a single step of computation to the body of the lambda literal e_0 with every occurrence of the bound variable x replaced by the argument e_1. The substituion has to be careful to avoid "name capture", i.e., substituting a term in a context that re-binds its free variables. For example, when substituting $y \mapsto x$ in $\lambda x.y$, one must make sure that the resulting term is **not** $\lambda x.x$. Using de Bruijn notation will help us make precise and avoid name capture.

The other computation rules in the language are inductively defined, e.g., $e_0\ e_1$ can take a step to $e_0'\ e_1$ if $e_0 \longrightarrow e_0'$, and similarly for e_1.

By choosing these other rules in different ways one obtains different reduction strategies, e.g., call-by-value or call-by-name etc. We'll leave the choice of reduction strategy non-deterministic and represent the computation rules of the STLC as an indexed inductive type, `step e e'` encoding one or more steps of computation.

17.2.1 Formalizing an Operational Semantics

The inductive type `step` below describes a single step of computation in what is known as a "small-step operational semantics". The type `step e e'` is a relation between an initial program `e` and a program `e'` that results after taking one step of computation on some sub-term of `e`.

```
type step : exp -> exp -> Type =
  | Beta :
     t:typ ->
     e1:exp ->
     e2:exp ->
     step (EApp (ELam t e1) e2) (subst (sub_beta e2) e1)

  | AppLeft :
      #e1:exp ->
      e2:exp ->
      #e1':exp ->
      hst:step e1 e1' ->
      step (EApp e1 e2) (EApp e1' e2)

  | AppRight :
      e1:exp ->
      #e2:exp ->
      #e2':exp ->
      hst:step e2 e2' ->
      step (EApp e1 e2) (EApp e1 e2')
```

- The constructor `Beta` represents the rule for β reduction. The most subtle part of the development is defining `subst` and `sub_beta`—we'll return to that in detail shortly.
- `AppLeft` and `AppRight` allow reducing either the left- or right-subterm of `EApp e1 e2`.

Exercise

Define an inductive relation `steps : exp -> exp -> Type` for the transitive closure of `step`, representing multiple steps of computation.

Use this exercise file for all the exercises that follow.

Answer

```
type steps : exp -> exp -> Type =
  | Single : #e0:exp ->
             #e1:exp ->
             step e0 e1 ->
             steps e0 e1

  | Many  : #e0:exp ->
```

(continues on next page)

(continued from previous page)

```
             #e1:exp ->
             #e2:exp ->
             step e0 e1 ->
             steps e1 e2 ->
             steps e0 e2
```

17.2.2 Substitution: Failed Attempt

Defining substitution is the trickiest part of the system. Our first attempt will convey the main intuitions, but F* will refuse to accept it as well-founded. We'll then enrich our definitions to prove that substitution terminates.

We'll define a substitution as a total function from variables `var` to expressions `exp`.

```
let sub0 = var -> exp
```

These kind of substitutions are sometimes called "parallel substitutions"—the each variable is substituted independently of the others.

When doing a β reduction, we want to substitute the variable associated with de Bruijn index `0` in the body of the function with the argument `e` and then remove the λ binder—`sub_beta0` does just that, replacing variable `0` with `e` and shifting other variables down by 1, since the λ binder of the function is removed.

```
let sub_beta0 (e:exp)
  : sub0
  = fun (y:var) ->
        if y = 0 then e      (* substitute *)
        else EVar (y-1)    (* shift -1 *)
```

The function `subst s e` applies the substitution `s` to `e`:

```
let sub_inc0 : sub0 =  fun y -> EVar (y+1)

[@@expect_failure [19;19]]
let rec subst0 (s:sub0)
               (e:exp)
  : exp
  = match e with
    | EUnit -> EUnit
    | EVar x -> s x
    | EApp e1 e2 -> EApp (subst0 s e1) (subst0 s e2)
    | ELam t e1 -> ELam t (subst0 (sub_elam0 s) e1)

and sub_elam0 (s:sub0)
  : sub0
  =  fun y ->
        if y=0
        then EVar y
        else subst0 sub_inc0 (s (y - 1))
```

- The `EUnit` case is trivial—there are no variables to substitute.
- In the variable case `subst0 s (EVar x)` just applies `s` to `x`.

- In the EApp case, we apply the substitution to each sub-term.
- The ELam case is the most interesting. To apply the substitution s to the body e1, we have to traverse a binder. The mutally recursive function sub_elam0 s adjusts s to account for this new binder, which has de Bruijn index 0 in the body e1 (at least until another binder is encountered).
  - In sub_elam0, if we are applying s to the newly bound variable at index 0, then we leave that variable unchanged, since s cannot affect it.
  - Otherwise, we have a variable with index at least 1, referencing a binder that is bound in an outer scope; so, we shift it down and apply s to it, and then increment all the variables in the resulting term (using sub_inc0) to avoid capture.

This definition of substitution is correct, but F* refuses to accept it since we have not convinced the typechecker that subst0 and sub_elam0 actually terminate. In fact, F* complains in two locations about a failed termination check.

Note: This definition is expected to fail, so the [@@expect_failure [19;19]] attribute on the definition asks F* to check that the definition raises Error 19 twice. We'll look in detail as to why it fails, next.

17.2.3 Substitution, Proven Total

Informally, let's try to convince ourselves why subst0 and sub_elam0 actually terminate.

- The recursive calls in the EApp case are applied to strictly smaller sub-terms (e0 and e1) of the original term e.
- In the ELam case, we apply subst0 to a smaller sub-term e1, but we make a mutally recursive call to sub_elam0 s first—so we need to check that that call terminates. This is the first place where F* complains.
- When calling sub_elam0, it calls back to subst0 on a completely unrelated term s (y - 1), and F* complains that this may not terminate. But, thankfully, this call makes use only of the sub_inc0 substitution, which is just a renaming substitution and which does not make any further recursive calls. Somehow, we have to convince F* that a recursive call with a renaming substitution is fine.

To distinguish renamings from general substitutions, we'll use an indexed type sub r, shown below.

```
let sub (renaming:bool) =
    f:(var -> exp){ renaming <==> (forall x. EVar? (f x)) }
```

- sub true is the type of renamings, substitutions that map variables to variables.
- sub false are substitutions that map at least one variable to a non-variable.

It's easy to prove that sub_inc is a renaming:

```
let sub_inc
  : sub true
  = fun y -> EVar (y+1)
```

The function sub_beta shown below is the analog of sub_beta0, but with a type that tracks whether it is a renaming or not.

```
let sub_beta (e:exp)
  : sub (EVar? e)
  = let f =
      fun (y:var) ->
        if y = 0 then e      (* substitute *)
```

(continues on next page)

(continued from previous page)

```
        else (EVar (y-1))    (* shift -1 *)
    in
    if not (EVar? e)
    then introduce exists (x:var). ~(EVar? (f x))
         with 0 and ();
    f
```

- The type says that `sub_beta e` is a renaming if and only if `e` is itself a variable.
- Proving this type, particularly in the case where `e` is not a variable requires proving an existentially quantified formula, i.e., `exists x. ~(EVar (subst_beta e) x)`. As mentioned *previously*, the SMT solver cannot always automatically instantiate existential quantifiers in the goal. So, we introduce the existential quantifier explicitly, providing the witness `0`, and then the SMT solver can easily prove `~(EVar (subst_beta e) 0)`.

Finally, we show the definitions of `subst` and `sub_elam` below—identical to `subst0` and `sub_elam0`, but enriched with types that allow expressing a termination argument to F* using a 4-ary lexicographic ordering.

```
let bool_order (b:bool) = if b then 0 else 1
let is_var (e:exp) : int = if EVar? e then 0 else 1

let rec subst (#r:bool)
              (s:sub r)
              (e:exp)
  : Tot (e':exp { r ==> (EVar? e <==> EVar? e') })
        (decreases %[bool_order (EVar? e);
                     bool_order r;
                     1;
                     e])
  = match e with
    | EUnit -> EUnit
    | EVar x -> s x
    | EApp e1 e2 -> EApp (subst s e1) (subst s e2)
    | ELam t e1 -> ELam t (subst (sub_elam s) e1)

and sub_elam (#r:bool) (s:sub r)
  : Tot (sub r)
        (decreases %[1;
                     bool_order r;
                     0;
                     EVar 0])
  = let f : var -> exp =
      fun y ->
        if y=0
        then EVar y
        else subst sub_inc (s (y - 1))
    in
    assert (not r ==> (forall x. ~(EVar? (s x)) ==> ~(EVar? (f (x + 1)))));
    f
```

Let's analyze the recursive calls of `subst` and `subst_elam` to see why this order works.

- Cases of `subst`:
  - The `EUnit` and `EVar` cases are trivial, as before.

  - In EApp, e is definitely not a variable, so bool_order (EVar? e) is 1. if e1 (respectively e2) are variables, then this recursive call terminates, the lexicographic tuple (0, _, _, _) << (1, _, _, _), regardles of the other values. Otherwise, the last component of the tuple decreases (since e1 and e2 are proper sub-terms of e), while none of the other components of the tuple change.
  - The call to sub_elam s in ELam terminates because the third component of the tuple decreases from 1 to 0, while the first two do not change.
  - The final recursive call to subst terminates for similar reasons to the recursive calls in the EApp case, since the type of sub_elam guarantees that sub_elam s is renaming if an only of s is (so the r bit does not change).
- Cases of sub_elam, in the recursive call to subst sub_inc (s (y - 1)), we have already proven that sub_inc is a renaming. So, we have two cases to consider:
  - If s (y - 1) is a variable, then bool_order (EVar? e), the first component of the decreases clause of subst is 0, which clearly precedes 1, the first component of the decreases clause of subst_elam.
  - Otherwwise, s (y - 1) is not a variable, so s is definitely not a renaming while sub_inc is. So, the second second component of the decreases clause decreases while the first component is unchanged.

Finally, we need to prove that sub_elam s is a renaming if and only if s is. For this, we need two things:

- First, strengthen the type of subst s to show that it maps variables to variables if and only if r is a renaming,
- Second, we need to instantiate an exisential quantifier in sub_elam, to show that if s is not a renaming, then it must map some x to a non-variable and, hence, sub_elam s (x + 1) is a non-variable too. One way to do this is by asserting this fact, which is a sufficient hint to the SMT solver to find the instantiation needed. Another way is to explicitly introduce the existential, as in the exercise below.

In summary, using indexed types combined with well-founded recursion on lexicographic orderings, we were able to prove our definitions total. That said, coming up with such orderings is non-trivial and requires some ingenuity, but once you do, it allows for relatively compact definitions that handle both substiutions and renamings.

Exercise

Remove the first component of the decreases clause of both definitions and revise the definitions to make F* accept it.

Your solution should have signature

```
let rec subst1 (#r:bool)
               (s:sub r)
               (e:exp)
  : Tot (e':exp { r ==> (EVar? e <==> EVar? e') })
        (decreases %[bool_order r;
                     1;
                     e])
...

and sub_elam1 (#r:bool) (s:sub r)
  : Tot (sub r)
        (decreases %[bool_order r;
                     0;
                     EVar 0])
```

Hint

Inline a case of subst in subst_elam. The answer is included with the next problem below.

Replace the assertion in `subst_elam` with a proof that explicitly introduces the existential quantifier.

Answer

```
let rec subst1 (#r:bool)
              (s:sub r)
              (e:exp)
  : Tot (e':exp { r ==> (EVar? e <==> EVar? e') })
        (decreases %[bool_order r;
                     1;
                     e])
  = match e with
    | EVar x -> s x
    | ELam t e1 -> ELam t (subst1 (sub_elam1 s) e1)
    | EApp e1 e2 -> EApp (subst1 s e1) (subst1 s e2)
    | EUnit -> EUnit

and sub_elam1 (#r:bool) (s:sub r)
  : Tot (sub r)
        (decreases %[bool_order r;
                     0;
                     EVar 0])
  = let f : var -> exp =
      fun y ->
        if y=0
        then EVar y
        else match s (y - 1) with
             | EVar x -> sub_inc x
             | e -> subst1 sub_inc e
    in
    introduce not r ==> (exists x. ~ (EVar? (f x)))
    with not_r.
      eliminate exists y. ~ (EVar? (s y))
      returns _
      with not_evar_sy.
        introduce exists x. ~(EVar? (f x))
        with (y + 1)
        and ()
    ;
    f
```

17.3 Type system

If when running a program, if one ends up with an term like $()\ e$ (i.e., some non-function term like $()$ being used as if it were a function) then a runtime error has occurred and the program crashes. A type system for the simply-typed lambda calculus is designed to prevent this kind of runtime error.

The type system is an inductively defined relation `typing g e t` between a

- typing environment `g:env`, a partial map from variable indexes in a particular scope to their annotated types;
- a program expression `e:exp`;

- and its type `t:typ`.

17.3.1 Environments

The code below shows our representation of typing environments `env`, a total function from variable indexes `var` to `Some t` or `None`.

```
let env = var -> option typ

let empty : env = fun _ -> None

let extend (t:typ) (g:env)
  : env
  = fun y -> if y = 0 then Some t
        else g (y-1)
```

- The `empty` environment maps all variables to `None`.
- Extending an an environment `g` associating a type `t` with a new variable at index `0` involves shifting up the indexes of all other variables in `g` by 1.

17.3.2 Typing Relation

The type system of STLC is defined by the inductively defined relation `typing g e t` shown below. A value of `typing g e t` is a derivation, or a proof, that `e` has type `t` in the environment `g`.

```
noeq
type typing : env -> exp -> typ -> Type =
  | TyUnit :
      #g:env ->
      typing g EUnit TUnit

  | TyVar :
      #g:env ->
      x:var{Some? (g x)} ->
      typing g (EVar x) (Some?.v (g x))

  | TyLam :
      #g :env ->
      t:typ ->
      #e1:exp ->
      #t':typ ->
      hbody:typing (extend t g) e1 t' ->
      typing g (ELam t e1) (TArr t t')

  | TyApp :
      #g:env ->
      #e1:exp ->
      #e2:exp ->
      #t11:typ ->
      #t12:typ ->
      h1:typing g e1 (TArr t11 t12) ->
```

(continues on next page)

(continued from previous page)

```
      h2:typing g e2 t11 ->
      typing g (EApp e1 e2) t12
```

- The type does not support decidable equality, since all its constructors contain a field `g:env`, a function-typed value without decidable equality. So, we mark the inductive with the `noeq` qualifier, *as described previously*.
- `TyUnit` says that the unit value `EUnit` has type `TUnit` in all environments.
- `TyVar` says that a variable `x` is well-typed only in an environment `g` that binds its type to `Some t`, in which case, the program `EVar x` has type `t`. This rule ensures that no out-of-scope variables can be used.
- `TyLam` says that a function literal `ELam t e1` has type `TArr t t'` in environment `g`, when the body of the function `e1` has type `t'` in an environment that extends `g` with a binding for the new variable at type `t` (while shifting and retaining all other ariables).
- Finally, `TyApp` allows applying `e1` to `e2` only when `e1` has an arrow type and the argument `e2` has the type of the formal parameter of `e1`—the entire term has the return type of `e1`.

17.4 Progress

It's relatively easy to prove that a well-typed non-unit or lambda term with no free variables can take a single step of computation. This property is known as *progress*.

17.4.1 Exercise

State and prove progress.

Answer

```
let is_value (e:exp) : bool = ELam? e || EUnit? e

let rec progress (#e:exp {~ (is_value e) })
                 (#t:typ)
                 (h:typing empty e t)
  : (e':exp & step e e')
  = let TyApp #g #e1 #e2 #t11 #t12 h1 h2 = h in
    match e1 with
    | ELam t e1' -> (| subst (sub_beta e2) e1', Beta t e1' e2 |)
    | _          -> let (| e1', h1' |) = progress h1 in
                      (| EApp e1' e2, AppLeft e2 h1'|)
```

17.5 Preservation

Given a well-typed term satisfying `typing g e t` and `steps e e'`, we would like to prove that `e'` has the same type as `e`, i.e., `typing g e' t`. This property is known as *preservation* (or sometimes *subject reduction*). When taken in combination with *progress*, this allows us to show that a well-typed term can keep taking a step until it reaches a value.

The proof below establishes preservation for a single step.

```
let rec preservation_step #e #e' #g #t (ht:typing g e t) (hs:step e e')
  : typing g e' t
  = let TyApp h1 h2 = ht in
    match hs with
    | Beta tx e1' e2' -> substitution_beta h2 (TyLam?.hbody h1)
    | AppLeft e2' hs1   -> TyApp (preservation_step h1 hs1) h2
    | AppRight e1' hs2   -> TyApp h1 (preservation_step h2 hs2)
```

- Since we know the computation takes a step, the typing derivation `ht` must be an instance of `TyApp`.
- In the `AppLeft` and `AppRight` case, we can easily use the induction hypothesis depending on which side actually stepped.
- The `Beta` case is the most interesting and requires a lemma about substitutions preserving typing.

The substitution lemma follows:

```
let subst_typing #r (s:sub r) (g1:env) (g2:env) =
    x:var{Some? (g1 x)} -> typing g2 (s x) (Some?.v (g1 x))

let rec substitution (#g1:env)
                     (#e:exp)
                     (#t:typ)
                     (#r:bool)
                     (s:sub r)
                     (#g2:env)
                     (h1:typing g1 e t)
                     (hs:subst_typing s g1 g2)
   : Tot (typing g2 (subst s e) t)
         (decreases %[bool_order (EVar? e); bool_order r; e])
   = match h1 with
     | TyUnit -> TyUnit
     | TyVar x -> hs x
     | TyApp hfun harg -> TyApp (substitution s hfun hs) (substitution s harg hs)
     | TyLam tlam hbody ->
       let hs'' : subst_typing (sub_inc) g2 (extend tlam g2) =
         fun x -> TyVar (x+1)
       in
       let hs' : subst_typing (sub_elam s) (extend tlam g1) (extend tlam g2) =
         fun y -> if y = 0 then TyVar y
               else substitution sub_inc (hs (y - 1)) hs''
       in
       TyLam tlam (substitution (sub_elam s) hbody hs')
```

It starts with a notion of typability of substitutions, `subst_typing s g1 g2`, which that if a variable `x` has type `g1 x`, then `s x` must have that same type in `g2`.

The substitution lemma lifts this notion to expressions, stating that applying a well-typed substitution `subst_typing` `s g1 g2` to a term well-typed in `g1` produces a term well-typed in `g2` with the same type.

17.5.1 Exercise

Use the substitution lemma to state and prove the `substitution_beta` lemma used in the proof of preservation.

Answer

```
let substitution_beta #e #v #t_x #t #g
                      (h1:typing g v t_x)
                      (h2:typing (extend t_x g) e t)
  : typing g (subst (sub_beta v) e) t
  = let hs : subst_typing (sub_beta v) (extend t_x g) g =
        fun y -> if y = 0 then h1 else TyVar (y-1) in
    substitution (sub_beta v) h2 hs
```

17.5.2 Exercise

Prove a preservation lemma for multiple steps.

Answer

```
let rec preservation #e #e' #g #t (ht:typing g e t) (hs:steps e e')
  : Tot (typing g e' t)
        (decreases hs)
  = match hs with
    | Single s ->
      preservation_step ht s
    | Many s0 s1 ->
      let ht' = preservation_step ht s0 in
      preservation ht' s1
```

17.6 Exercise

Prove a type soundness lemma with the following statement:

```
let soundness #e #e' #t
              (ht:typing empty e t)
  : either (squash (is_value e))
           (e':exp & steps e e' & typing empty e' t)
```

Answer

```
  = if is_value e then Inl ()
    else let (| e', s |) = progress ht in
         let ht' = preservation_step ht s in
         Inr (| e', Single s, ht' |)
```

17.7 Exercise

Add a step for reduction underneath a binder and prove the system sound.

Answer

```
(*
   Copyright 2014-2015
     Simon Forest - Inria and ENS Paris
     Catalin Hritcu - Inria
     Nikhil Swamy - Microsoft Research

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.
*)

module Part2.STLC.Strong
open FStar.Classical.Sugar
(* Constructive-style progress and preservation proof for STLC with
   strong reduction, using deBruijn indices and parallel substitution. *)

type typ =
  | TUnit : typ
  | TArr  : typ -> typ -> typ

let var = nat
type exp =
  | EUnit : exp
  | EVar  : var -> exp
  | ELam  : typ -> exp -> exp
  | EApp  : exp -> exp -> exp

(* Parallel substitution operation `subst` *)
let sub (renaming:bool) =
    f:(var -> exp){ renaming <==> (forall x. EVar? (f x)) }

let bool_order (b:bool) = if b then 0 else 1

let sub_inc
  : sub true
  = fun y -> EVar (y+1)

let is_var (e:exp) : int = if EVar? e then 0 else 1
```

(continues on next page)

(continued from previous page)

```
let rec subst (#r:bool)
              (s:sub r)
              (e:exp)
  : Tot (e':exp { r ==> (EVar? e <==> EVar? e') })
        (decreases %[bool_order (EVar? e);
                     bool_order r;
                     1;
                     e])
  = match e with
    | EVar x -> s x
    | ELam t e1 -> ELam t (subst (sub_elam s) e1)
    | EApp e1 e2 -> EApp (subst s e1) (subst s e2)
    | EUnit -> EUnit

and sub_elam (#r:bool) (s:sub r)
  : Tot (sub r)
        (decreases %[1;
                     bool_order r;
                     0;
                     EVar 0])
  = let f : var -> exp =
      fun y ->
        if y=0
        then EVar y
        else subst sub_inc (s (y - 1))
    in
    introduce not r ==> (exists x. ~ (EVar? (f x)))
    with not_r.
      eliminate exists y. ~ (EVar? (s y))
      returns (exists x. ~ (EVar? (f x)))
      with (not_evar_sy:squash (~(EVar? (s y)))).
        introduce exists x. ~(EVar? (f x))
        with (y + 1)
        and ()
    ;
    f

let sub_beta (e:exp)
  : sub (EVar? e)
  = let f =
      fun (y:var) ->
        if y = 0 then e      (* substitute *)
        else (EVar (y-1))    (* shift -1 *)
    in
    if not (EVar? e)
    then introduce exists (x:var). ~(EVar? (f x))
         with 0 and ();
    f

(* Small-step operational semantics; strong / full-beta reduction is
   non-deterministic, so necessarily as inductive relation *)
```

(continues on next page)

(continued from previous page)

```
type step : exp -> exp -> Type =
  | SBeta : t:typ ->
            e1:exp ->
            e2:exp ->
            step (EApp (ELam t e1) e2) (subst (sub_beta e2) e1)

  | SApp1 : #e1:exp ->
             e2:exp ->
            #e1':exp ->
            hst:step e1 e1' ->
            step (EApp e1 e2) (EApp e1' e2)

  | SApp2 :  e1:exp ->
            #e2:exp ->
            #e2':exp ->
            hst:step e2 e2' ->
            step (EApp e1 e2) (EApp e1 e2')

  | STrans : #e0:exp ->
             #e1:exp ->
             #e2:exp ->
             step e0 e1 ->
             step e1 e2 ->
             step e0 e2

  | SStrong : t:typ ->
              e:exp ->
              e':exp ->
              step e e' ->
              step (ELam t e) (ELam t e')

(* Type system; as inductive relation (not strictly necessary for STLC) *)

type env = var -> option typ

let empty : env = fun _ -> None

(* we only need extend at 0 *)
let extend (t:typ) (g:env)
  : env
  = fun y -> if y = 0 then Some t
          else g (y-1)

noeq
type typing : env -> exp -> typ -> Type =
  | TyUnit : #g:env ->
             typing g EUnit TUnit

  | TyVar :  #g:env ->
             x:var{Some? (g x)} ->
             typing g (EVar x) (Some?.v (g x))
```

(continues on next page)

(continued from previous page)

```
  | TyLam : #g :env ->
            t:typ ->
            #e1:exp ->
            #t':typ ->
            hbody:typing (extend t g) e1 t' ->
            typing g (ELam t e1) (TArr t t')

  | TyApp : #g:env ->
            #e1:exp ->
            #e2:exp ->
            #t11:typ ->
            #t12:typ ->
            h1:typing g e1 (TArr t11 t12) ->
            h2:typing g e2 t11 ->
            typing g (EApp e1 e2) t12


(* Progress *)

let is_value (e:exp) : bool = ELam? e || EUnit? e

let rec progress (#e:exp {~ (is_value e) })
                 (#t:typ)
                 (h:typing empty e t)
  : (e':exp & step e e')
  = let TyApp #g #e1 #e2 #t11 #t12 h1 h2 = h in
    match e1 with
    | ELam t e1' -> (| subst (sub_beta e2) e1', SBeta t e1' e2 |)
    | _          -> let (| e1', h1' |) = progress h1 in
                   (| EApp e1' e2, SApp1 e2 h1'|)

(* Typing of substitutions (very easy, actually) *)
let subst_typing #r (s:sub r) (g1:env) (g2:env) =
    x:var{Some? (g1 x)} -> typing g2 (s x) (Some?.v (g1 x))

(* Substitution preserves typing
   Strongest possible statement; suggested by Steven Schäfer *)
let rec substitution (#g1:env)
                     (#e:exp)
                     (#t:typ)
                     (#r:bool)
                     (s:sub r)
                     (#g2:env)
                     (h1:typing g1 e t)
                     (hs:subst_typing s g1 g2)
   : Tot (typing g2 (subst s e) t)
         (decreases %[bool_order (EVar? e); bool_order r; e])
   = match h1 with
   | TyVar x -> hs x
   | TyApp hfun harg -> TyApp (substitution s hfun hs) (substitution s harg hs)
   | TyLam tlam hbody ->
     let hs'' : subst_typing (sub_inc) g2 (extend tlam g2) =
```

(continues on next page)

(continued from previous page)

```
       fun x -> TyVar (x+1) in
     let hs' : subst_typing (sub_elam s) (extend tlam g1) (extend tlam g2) =
       fun y -> if y = 0 then TyVar y
             else substitution sub_inc (hs (y - 1)) hs''
     in TyLam tlam (substitution (sub_elam s) hbody hs')
   | TyUnit -> TyUnit

(* Substitution for beta reduction
   Now just a special case of substitution lemma *)
let substitution_beta #e #v #t_x #t #g
                      (h1:typing g v t_x)
                      (h2:typing (extend t_x g) e t)
  : typing g (subst (sub_beta v) e) t
  = let hs : subst_typing (sub_beta v) (extend t_x g) g =
        fun y -> if y = 0 then h1 else TyVar (y-1) in
    substitution (sub_beta v) h2 hs

(* Type preservation *)
let rec preservation #e #e' #g #t (ht:typing g e t) (hs:step e e')
  : Tot (typing g e' t)
        (decreases hs)
  = match hs with
    | STrans s0 s1 ->
      let ht' = preservation ht s0 in
      preservation ht' s1
    | _ ->
      match ht with
      | TyApp h1 h2 -> (
        match hs with
        | SBeta tx e1' e2' -> substitution_beta h2 (TyLam?.hbody h1)
        | SApp1 e2' hs1   -> TyApp (preservation h1 hs1) h2
        | SApp2 e1' hs2   -> TyApp h1 (preservation h2 hs2)
      )
      | TyLam t hb ->
        let SStrong t e e' hs' = hs in
        let hb' = preservation hb hs' in
        TyLam t hb'
```

CHAPTER

EIGHTEEN

HIGHER-ORDER ABSTRACT SYNTAX

In the previous chapter, we looked at a *deep embedding* of the simply typed lambda calculus (STLC). The encoding is “deep” in the sense that we used an inductive type to represent the *syntax* of the lambda calculus in F*, and then defined and proved some properties of its semantics represented mathematically in F*.

Another way to embed a language like the STLC in F* is a *shallow embedding*. F* is itself a functional programming language, and it has a type system that is certainly powerful enough to represent simply typed terms, so why not use lambda terms in F* itself to represent STLC, rather than merely encoding STLC’s abstract syntax in F*. This kind of encoding is called a shallow embedding, where we use semantic constructs in the host (or meta) language (F*) to represent analogous features of the embedded (or object) language (STLC, in our example).

In this chapter, we look at a particularly elegant technique for doing this called *higher-order abstract syntax* (or HOAS). For more background about this, a 2008 paper by Adam Chlipala is a good resource, though it develops a more sophisticated parametric version.

Our small case study in HOAS is meant to illustrate the use of inductive types with non-trivial indexes while also containing strictly positive functions as arguments, and also a bit of type-level computation.

18.1 Roadmap

The type `typ` below represents the types we’ll use in our STLC object language, i.e., the base types `Bool` and `Int`, and function types `Arrow t1 t2`.

```
type typ =
  | Bool
  | Int
  | Arrow : typ -> typ -> typ
```

This is analogous to our representation of STLC types in the deep embedding of the previous chapter.

Where things get interesting is in the representation of STLC terms and their semantics. To set the goal posts, we want to

1. Give an interpretation of STLC types into F* types, by defining a function `denote_typ : typ -> Type`
2. Define a type `term t`, to represent well-typed STLC terms whose type is `t:typ`
3. Give an intepretation of STLC terms in to F* terms of the suitable type, i.e., define a function `denote_term (#t:typ) (e:term t) : denote_typ t`, proving that every well-typed STLC term at type `t` can be represented in F* as a function of type `denote_typ t`.

Such a result would encompass the type soundness results we proved in the previous chapter (proving that well-typed programs can always make progress), but would go substantially further to show that the reduction of all such terms always terminates producing F* values that model their semantics.

18.2 Denotation of types

Step 1 in our roadmap is to give an interpreration of STLC types typ into F* types. This is easily done, as shown below.

```
let rec denote_typ (t:typ)
  : Type
  = match t with
    | Bool -> bool
    | Int -> int
    | Arrow t1 t2 -> (denote_typ t1 -> denote_typ t2)
```

We have here a recursive function that computes a *Type* from its argument. This is may seem odd at first, but it's perfectly legal in a dependently typed language like F*.

The function denote_typ The interpretation of Bool and Int are the F* type bool and int, while the interpretation of Arrow is an F* arrow. Note, the function terminates because the two recursive calls are on strict sub-terms of the argument.

18.3 Term representation

The main difficulty in representing a language like STLC (or any language with lambda-like variable-binding structure), is the question of how to represent variables and their binders.

In the deep embedding of the previous chapter, our answer to this questions was very syntactic—variables are de Bruijn indexes, where, at each occurrence, the index used for a variable counts the number of lambdas to traverse to reach the binder for that variable.

The HOAS approach to answering these questions is very different. The idea is to use the binding constructs and variables already available in the host language (i.e., lambda terms in F*) to represent binders and variables in the object language (STLC).

The main type in our representation of terms is the term defined below. There are several clever subtleties here, which we'll try to explain briefly.

```
noeq
type term : typ -> Type =
  | Var : #t:typ -> denote_typ t -> term t
  | TT  : term Bool
  | FF  : term Bool
  | I   : int -> term Int
  | App : #t1:typ -> #t2:typ ->
          term (Arrow t1 t2) ->
          term t1 ->
          term t2
  | Lam : #t1:typ -> #t2:typ ->
          (denote_typ t1 -> term t2) ->
          term (Arrow t1 t2)
```

First, the term type represents both the abstract syntax of the STLC as well as its typing rules. We'll see in detail how this works, but notice already that term is is indexed by a t:typ, which describes the type of encoded STLC term. The indexing structure encodes the typing rules of STLC, ensuring that only well-typed STLC terms can be constructed.

The second interesting part is the use of denote_typ within the syntax—variables and binders at a given STLC type t will be represented by F* variables and binders of the corresponding F* type denote_typ t.

- `Var` : Variables are represented as `Var #t n : term t`, where `n` is a term of type `denote_typ t`.
- `TT` and `FF`: The two boolean constansts are represented by these constructors, both of type `term Bool`, where the index indicates that they have type `Bool`.
- `I`: STLC integers are represented by tagged F* integers.
- `App`: To apply `e1` to `e2` in a well-typed way, we must prove that `e1` has an arrow type `TArrow t1 t2`, while `e2` has type `t1`, and the resulting term `App e1 e2` has type `t2`. Notice how the indexing structure of the `App` constructor precisely captures this typing rule.
- `Lam`: Finally, and crucially, we represent STLC lambda terms using F* functions, i.e., `Lam f` has type `Arrow t1 t2`, when `f` is represented by an F* function from arguments of type `denote_typ t1`, to terms of type `term t`. The `Lam` case includes a function-typed field, but the type of that field, `denote_typ t1 -> term t2` is strictly positive—unlike the the `dyn` type, *shown earlier*.

18.4 Denotation of terms

Finally, we come to Step 3 (below), where we give an interpretation to `term t` as an F* term of type `denote_typ t`. The trickiest part of such a interpretation is to handle functions and variables, but this part is already done in the representation, since these are already represented by the appropriate F* terms.

```
let rec denote_term (#t:typ) (e:term t)
  : Tot (denote_typ t)
        (decreases e)
  = match e with
    | Var x -> x
    | TT -> true
    | FF -> false
    | I i -> i
    | App f a -> (denote_term f) (denote_term a)
    | Lam f -> fun x -> denote_term (f x)
```

Let’s look at each of the cases:

- The `Var` case is easy, since the variable `x` is already interpreted into an element of the appropriate F* type.
- The constants `TT`, `FF`, and `I` are easy too, since we can just interpret them as the suitable boolean or integer constants.
- For the `App #t1 #t2 f a` case, we recursively interpret `f` and `a`. The type indices tell us that `f` must be interpreted into an F* function of type `denote_typ t1 -> denote_typ t2` and that `denote_term a` has type `denote_typ t1`. So, we can simply apply the denotation of `f` to the denotation of `a` to get a term of type `denote_typ t2`. In other words, function application in STLC is represented semantically by function application in F*.
- Finally, in the `Lam #t1 #t2 f` case, we need to produce a term of type `denote_typ t1 -> denote_typ t2`. So, we build an F* lambda term (where the argument `x` has type `denote_typ t1`), and in the body we apply `f x` and recursively call `denote_term` on `f x`.

If that felt a little bit magical, it’s because it almost is! We’ve defined the syntax, typing rules, and an interpreter that doubles as a denotational semantics for the STLC, and proved everything sound in around 30 lines of code and proof. By picking the right representation, everything just follows very smoothly.

18.4.1 Termination

You may be wondering why F* accepts that `denote_term e` terminates. There are three recursive calls to consider

- The two calls in the `App` case are easy: The recursive calls are on strict sub-terms of `App f a`.
- In the `Lam f` case, we have one recursive call `denote_term (f x)`, and F* accepts that `f x` is strictly smaller than `Lam f`. This is an instance of the sub-term ordering on inductive types explained earlier, as part of F*'s `precedes` relation, *explained earlier*.

For a bit of intuition, one way to understand the type `term` is by thinking of it as a tree of finite depth, but possibly infinite width:

- The leaves of the tree are the `Var`, `TT`, and `FF` cases.
- The internal node `App e0 e1` composes two sub-trees, `e0` and `e1`.
- The internal node `Lam #t1 #t2 f` composes a variable number of sub-trees, where the number of sub-trees depends on the parameter `t1`. For example:
  - If `t1 = Unit`, then `f : unit -> term v t`, i.e., there is only one child node, `f()`.
  - If `t1 = Bool`, then `f : bool -> term v t`, i.e., there are two children, `f true` and `f false`.
  - if `t1 = Int`, then `f : int -> term v t`, i.e., there are infinitely many children, `..., f -1, f 0, f 1, ...`.

With this intuition, informally, it is safe to recursively call `denote_term e` on any of the children of `e`, since the depth of the tree will decrease on each recursive call. Hence the call `denote_term (f x)` terminates.

We'll revisit termination arguments for recursive functions more formally in a subsequent chapter on *well-founded recursion*.

18.5 Exercises

Giving a semantics to STLC is just the tip of the iceberg. There's a lot more one can do with HOAS and Chlipala's paper gives lots of examples and sample code in Coq.

For several more advanced exercises, based on the definitions shown below, try reconstructing other examples from Chlipala's paper, including a proof of correctness of a compiler implementing a continuation-passing style (CPS) transformation of STLC.

Exercise file

```
(* This example is transcribed from Adam Chlipala's
   Parametric Higher-order Abstract Syntax (ICFP 2008) paper.

   http://adam.chlipala.net/papers/PhoasICFP08/PhoasICFP08.pdf
*)
module Part2.PHOAS

type typ =
  | Bool
  | Arrow : typ -> typ -> typ

noeq
type term0 (v: typ -> Type) : typ -> Type =
  | Var : #t:typ -> v t -> term0 v t
```

(continues on next page)

(continued from previous page)

```
  | TT  : term0 v Bool
  | FF  : term0 v Bool
  | App : #t1:typ -> #t2:typ ->
          term0 v (Arrow t1 t2) ->
          term0 v t1 ->
          term0 v t2
  | Lam : #t1:typ -> #t2:typ ->
          (v t1 -> term0 v t2) ->
          term0 v (Arrow t1 t2)

let term (t:typ) = v:(typ -> Type) -> term0 v t

let rec denote_typ (t:typ)
  : Type
  = match t with
    | Bool -> bool
    | Arrow t1 t2 -> (denote_typ t1 -> denote_typ t2)

let rec denote_term0 (#t:typ) (e:term0 denote_typ t)
  : Tot (denote_typ t)
        (decreases e)
  = match e with
    | Var x -> x
    | TT -> true
    | FF -> false
    | App f a -> (denote_term0 f) (denote_term0 a)
    | Lam f -> fun x -> denote_term0 (f x)

let denote_term (t:typ) (e:term t)
  : denote_typ t
  = denote_term0 (e _)
```

CHAPTER

NINETEEN

WELL-FOUNDED RELATIONS AND TERMINATION

In an earlier chapter on *proofs of termination*, we learned about how F* checks that recursive functions terminate. In this chapter, we see how the termination check arises from inductive types and structural recursion. Just as with *equality*, termination checking, a core feature of F*'s logic and proof system, finds its foundation in inductive types.

For more technical background on this topic, the following resources may be useful:

- Constructing Recursion Operators in Type Theory, L. Paulson, Journal of Symbolic Computation (1986) 2, 325-355
- Modeling General Recursion in Type Theory, A. Bove & V. Capretta, Mathematical Structures in Computer Science (2005)

Thanks to Aseem Rastogi, Chantal Keller, and Catalin Hritcu, for providing some of the F* libraries presented in this chapter.

- FStar.WellFounded.fst
- FStar.LexicographicOrdering

19.1 Well-founded Relations and Accessibility Predicates

A binary relation R on elements of type T is well-founded if there is no infinite sequence $x_0, x_1, x_2, ...$, such that $x_i \; R \; x_{i+1}$, for all i.

As explained *earlier*, when typechecking a recursive function `f`, F* requires the user to provide a *measure*, some function of the arguments of *f*, and checks that on a recursive call, the measure of the arguments is related to the measure of the formal parameters a built-in well-founded relation on F* terms. Since well-founded relations have no infinite descending chains, every chain of recursive calls related by such a relation must eventually terminate. However, this built-in well-founded relation, written << or `precedes`, is a derived notion.

In its most primitive form, the well-foundedness of a relation can be expressed in terms of an inductive type `acc` (short for "accessible") shown below.

```
let binrel a = a -> a -> Type
noeq
type acc (#a:Type) (r:binrel a) (x0:a) : Type =
  | AccIntro : access_smaller:(x1:a -> r x1 x0 -> acc r x1) -> acc r x0
```

The type `acc` is parameterized by a type `a`; a binary relation `r:  a -> a -> Type` on `a`; and an element of the type `x:a`. Informally, the relation `r y x` is provable when `y` is "smaller" than `x`.

The `acc` type has just one constructor `AccIntro`, whose only argument is a function of type `y:a -> r y x -> acc r y`. Intuitively, this says that in order to build an instance of `acc r x0`, you have to provide a function which can build a proof of `acc r x1` for all `x1:a` smaller than `x0`. The only way to build such a function is one can avoid infinite regress, is if the chain `x0 r x1 r x2 r ...`, eventually terminates in some `xn` such that there are no elements smaller than it according to `r`.

In other words, if one can prove `acc r x` for all `x:a`, then this precisely captures the condition that there are no infinite descending `r`-related chains in `a`, or that `r` is well-founded. This is exactly what the definition below says, where `is_well_founded` is a classical (SMT-automatable) variant of `well_founded`.

```
let well_founded (#a:Type) (r:binrel a) = x:a -> acc r x
let is_well_founded (#a:Type) (r:binrel a) = forall x. squash (acc r x)
```

19.2 Well-founded Recursion

Given a relation `r` and proof of `p:acc r x` , one can define a recursive function on `x` whose termination can be established purely in terms of structural recursion on the proof `p`, even though the function may not itself be structurally recursive on `x`.

The combinator `fix_F` shown below illustrates this at work:

```
let rec fix_F (#aa:Type)
              (#r:binrel aa)
              (#p:(aa -> Type))
              (f: (x:aa -> (y:aa -> r y x -> p y) -> p x))
              (x0:aa)
              (accessible_x0:acc r x0)
  : Tot (p x0) (decreases accessible_x0)
  = f x0 (fun y r_yx -> fix_F f y (accessible_x0.access_smaller y r_yx))
```

If `f` is a function such that every recursive call in the definition of `f x` is on an argument `y`, such that that `y` is smaller than `x` according to some relation `r`; and if starting from some argument `x0`, we have a proof of accessibility `acc r x0` (i.e., no infinite descending `r`-chains starting at `x0`), then the fixpoint of `f` can be defined by structural recursion on the proof of `accessible_x0`.

- `fix_F` is structurally recursive on `accessible_x0` since the recursive call is on an element `h1 y r_yx`, i.e., a child node of the (possibly infinitely branching) tree rooted at `AccIntro h1`.

A slightly simpler version of `fix_F` is derivable if `r` is well-founded, i.e., `r` is accessible for all elements `x:a`.

```
let fix (#aa:Type) (#r:binrel aa) (rwf:well_founded r)
        (p:aa -> Type) (f:(x:aa -> (y:aa -> r y x -> p y) -> p x))
        (x:aa)
  : p x
  = fix_F f x (rwf x)
```

19.3 Some Well-founded Relations

We show how to buid some basic well-founded relations here. For starters, since F* already internalizes that the < ordering on natural numbers as part of its termination check, it is easy to prove that < is well-founded.

```
let lt_nat (x y:nat) : Type = x < y == true
let rec wf_lt_nat (x:nat)
  : acc lt_nat x
  = AccIntro (fun y _ -> wf_lt_nat y)
```

We can also define combinators to derive well-founded relations from other well-founded relations. For example, if a relation `sub_r` is a *sub-relation* of a well-founded relation `r` (meaning we have `r x y` whenever we have `sub_r x y`), then `sub_r` is well-founded too.

```
let subrel_wf (#a:Type) (#r #sub_r:binrel a)
              (sub_w:(x:a -> y:a -> sub_r x y -> r x y))
              (r_wf:well_founded r)
  : well_founded sub_r
  = let rec aux (x:a)
                (acc_r:acc r x)
      : Tot (acc sub_r x) (decreases acc_r)
      = AccIntro
          (fun y sub_r_y_x ->
             aux y (acc_r.access_smaller y (sub_w y x sub_r_y_x)))
    in
    fun x -> aux x (r_wf x)
```

Another useful combinator derives the well-foundedness of a relation `r: binrel b` if it can be defined as the inverse image under some function `f: a -> b` of some other well-founded relation `r: binrel`.

```
let inverse_image (#a #b:Type) (r_b:binrel b) (f:a -> b) : binrel a =
  fun x y -> r_b (f x) (f y)

let inverse_image_wf (#a #b:Type) (#r_b:binrel b)
                     (f:a -> b)
                     (r_b_wf:well_founded r_b)
  : well_founded (inverse_image r_b f)
  = let rec aux (x:a)
                (acc_r_b:acc r_b (f x))
      : Tot (acc (inverse_image r_b f) x)
            (decreases acc_r_b)
      = AccIntro (fun y p -> aux y (acc_r_b.access_smaller (f y) p)) in
    fun x -> aux x (r_b_wf (f x))
```

For example, the > ordering on negative numbers can be proven well-founded by defining it as the inverse image of the < ordering on natural numbers.

```
let neg = x:int { x <= 0 }
let negate (x:neg) : nat = -x
let gt_neg : binrel neg = inverse_image lt_nat negate
let wf_gt_neg = inverse_image_wf negate wf_lt_nat
```

19.4 Termination Checking with Custom Well-founded Relations

In the F* library, `FStar.LexicographicOrdering`, several other relations are proven to be well-founded, including the lexicographic ordering on dependent pairs.

```
noeq
type lexicographic_order (#a:Type)
                         (#b:a -> Type)
                         (r_a:binrel a)
                         (r_b:(x:a -> binrel (b x)))
```

(continues on next page)

(continued from previous page)

```
  : (x:a & b x) -> (x:a & b x) -> Type =
  | Left_lex:
    x1:a -> x2:a ->
    y1:b x1 -> y2:b x2 ->
    r_a x1 x2 ->
    lexicographic_order r_a r_b (| x1, y1 |) (| x2, y2 |)

  | Right_lex:
    x:a ->
    y1:b x -> y2:b x ->
    r_b x y1 y2 ->
    lexicographic_order r_a r_b (| x, y1 |) (| x, y2 |)
```

This order, defined as a `binrel (x:a & b x)`, and is paramterized by a binary relation (`r_a`) on `a` and a family of relations (`r_b`) on `b x`, one for each `x:a`. It has two cases:

- `Left_lex`: The first component of the pair decreases by `r_a`, and the second component is irrelevant.
- `Right_lex`: The first component of the pair is invariant, but the second component decreases by `r_b`.

The proof is a little involved (see `FStar.LexicographicOrdering.fst`), but one can prove that it is well-founded when `r_a` and `r_b` are themselves well-founded, i.e.,

```
val lexicographic_order_wf (#a:Type) (#b:a -> Type)
                           (#r_a:binrel a)
                           (#r_b:(x:a -> binrel (b x)))
                           (wf_a:well_founded r_a)
                           (wf_b:(x:a -> well_founded (r_b x)))
  : well_founded (lexicographic_order r_a r_b)
```

But, with this well-foundedness proof in hand, we can define recursive functions with our own well-founded orders.

To illustrate, let's define the `ackermann` function again (we saw it first *here*), this time using accessibilty and well-founded relations, rather than the built-in `precedes` relation.

```
//A type abbreviation for a pair of nats
//  we're using dependent pairs, even though there's no real dependence here
let nat_pair = (x:nat & nat)

//Making a lexicographic ordering from a pair of nat ordering
let lex_order_nat_pair
  : binrel nat_pair
  = lexicographic_order lt_nat (fun _ -> lt_nat)

// The lex order on nat pairs is well-founded, using our general proof
// of lexicographic composition of well-founded orders
let lex_order_nat_pair_wf
  : well_founded lex_order_nat_pair
  = lexicographic_order_wf wf_lt_nat (fun _ -> wf_lt_nat)

// A utility to introduce lt_nat
let mk_lt_nat (x:nat) (y:nat { x < y })
  : lt_nat x y
  = let _ : equals (x < y) true = Refl in
```

(continues on next page)

(continued from previous page)

```
    ()

// A utility to make a lex ordering of nat pairs
let mk_lex_order_nat_pair (xy0:nat_pair)
                          (xy1:nat_pair {
                            let (|x0, y0|) = xy0 in
                            let (|x1, y1|) = xy1 in
                            x0 < x1 \/ (x0 == x1 /\ y0 < y1)
                          })
  : lex_order_nat_pair xy0 xy1
  = let (|x0, y0|) = xy0 in
    let (|x1, y1|) = xy1 in
    if x0 < x1 then Left_lex x0 x1 y0 y1 (mk_lt_nat x0 x1)
    else Right_lex x0 y0 y1 (mk_lt_nat y0 y1)

// Defining ackermann, where `arec` is called for recursive calls
// on pairs that precede xy, lexicographically
let ackermann' (xy: nat_pair)
               (arec: (xy':nat_pair -> lex_order_nat_pair xy' xy -> nat))
  : nat
  = let (| x, y |) = xy in
    if x = 0 then y + 1
    else if y = 0 then arec (| x - 1, 1 |) (mk_lex_order_nat_pair _ _)
    else arec (| x - 1, arec (| x, y - 1|) (mk_lex_order_nat_pair _ _) |)
              (mk_lex_order_nat_pair _ _)

// Tie the knot with `fix`
let ackermann : nat_pair -> nat = fix lex_order_nat_pair_wf (fun _ -> nat) ackermann'
```

This version is way more verbose than the ackermann we saw earlier—but this version demonstrates that F*'s built-in support for the lexicographic orders over the precedes relation is semantically justified by a more primitive model of well-founded relations

To make user-defined well-founded orderings easier to work with, F* actually provides a variant of the decreases clause to work with well-founded relations. For example, one can use the following syntax to gain from F*'s built-in from SMT automation and termination checking, with the expressiveness of using ones own well-founded relation.

```
module L = FStar.LexicographicOrdering
let rec ackermann_wf (m n:nat)
   : Tot nat
     (decreases
       {:well-founded
         L.lex (coerce_wf wf_lt_nat) (fun _ -> (coerce_wf wf_lt_nat)) (| m, n |)
       })
  = if m = 0 then n + 1
    else if n = 0 then ackermann_wf (m - 1) 1
    else ackermann_wf (m - 1) (ackermann_wf m (n - 1))
```

To explain the syntax:

- decreases {:well-founded p x}: Here, p is meant to be an instance for FStar.LexicographicOrdering.well_founded_relation, applied to some term x built from the formal parameters in scope.

- In this case, we use the combinator L.lex to build a lexicographic ordering from wf_lt_nat (coercing it using a utility coerce_wf to turn the definitions used in our tutorial chapter here to the types expected by the FStar.LexicographicOrdering library).

We show the coercions below for completeness, though one would not necessarily need them outside the context of this tutorial.

```
module W = FStar.WellFounded
let rec coerce #a #r #x (p:acc #a r x)
  : Tot (W.acc r x) (decreases p)
  = W.AccIntro (fun y r -> coerce (p.access_smaller y r))

let coerce_wf #a #r (p: (x:a -> acc r x))
  : x:a -> W.acc r x
  = fun x -> coerce (p x)
```

CHAPTER

TWENTY

A FIRST MODEL OF COMPUTATIONAL EFFECTS

As a final chapter in this section, we show how inductive types can be used model not just data, but also *computations*, including computations with side effects, like mutable state and shared-memory concurrency. This is meant to also give a taste of the next section in this book, which deals with modeling and proving properties of programs with effects and F*'s user-extensible system of indexed effects.

Thanks to Guido Martinez and Danel Ahman, for some of the content in this chapter.

20.1 A First Taste: The State Monad

All the programs we've written so far have been purely functional. However, one can model programs that manipulate mutable state within a purely functional language, and one common but powerful way to do this is with something called a *monad*, an idea that was introduced to functional programmers in the late 1980s and early 90s by Philip Wadler building on semantic foundations developed by Eugenio Moggi. If you've been puzzled about monads before, we'll start from scratch here, and hopefully this time it will all make sense!

Consider modeling a program that manipulates a single piece of mutable state, just a single integer that programs can read and write, and which returns a result of type `a`. One way to do this is to represent such a *stateful computation* as a program whose type is `st a`:

```
let st a = int -> a & int
```

A `(st a)` computation is a function which when given an initial value for the state `s0` returns a pair `(x, s1)` with the result of the computation `x:a` and a final value for the state `s1`.

For example, a computation that reads the state, increments it, and returns the initial value of the state, can be expressed as shown below.

```
let read_and_increment_v0
  : st int
  = fun s0 -> s0, s0 + 1
```

This is pretty straightforward, but writing computations in this style can be quite tedious and error prone. For example, if you wanted to read the state and increment it twice, one would write:

```
let inc_twice_v0
  : st int
  = fun s0 ->
      let x, s1 = read_and_increment_v0 s0 in
      let _, s2 = read_and_increment_v0 s1 in
      x, s2
```

This is quite clumsy, since at each call to `read_and_increment_v0` we had to be careful to pass it the "the most recent version" of the state. For instance, a small typo could easily have caused us to write the program below, where we pass `s0` to the second call of `read_and_increment`, causing the program to only increment the state once.

```
let inc_twice_buggy
  : st int
  = fun s0 ->
      let x, s1 = read_and_increment_v0 s0 in
      let _, s2 = read_and_increment_v0 s0 in
      x, s2
```

The main idea with the state monad is to structure stateful programs by abstracting out all the plumbing related to manipulating the state, eliminating some of the tedium and possibilities for errors.

The way this works is by defining a functions to read and write the state, plus a couple of functions to return a pure value without reading or writing the state (a kind of an identity function that's a noop on the state); and a function to sequentially compose a pair of stateful computations.

- The function `read : st int` below, reads the state and returns it, without modifying the state.

```
let read
  : st int
  = fun s -> s, s
```

- The function `write (s1:int) : st unit` below, sets the state to `s1` and returns `() : unit`.

```
let write (s1:int)
  : st unit
  = fun _ -> (), s1
```

- The function `bind` is perhaps the most interesting. Given a stateful computation `f: st a` and another computation `g : a -> st b` which depends on the result of `f` and then may read or write the state returning a `b`; `bind f g` composes `f` and `g` sequentially, passing the initial state `s0` to `f`, then passing its result `x` and next state `s1` to `g`.

```
let bind #a #b
         (f: st a)
         (g: a -> st b)
  : st b
  = fun s0 ->
      let x, s1 = f s0 in
      g x s1
```

- Finally, `return` promotes a pure value `x:a` into an `st a`, without touching the state.

```
let return #a (x:a)
  : st a
  = fun s -> x, s
```

20.1.1 Some stateful programs

With these combinators in hand, we can write stateful programs in a more compact style, never directly manipulating the underlying integer variable that holds the state directly.

Here's a next attempt at `read_and_increment`:

```
let read_and_increment_v1 : st int =
  bind read (fun x ->
  bind (write (x + 1)) (fun _ ->
  return x))
```

Now, you're probably thinking that this version is even worse than `read_and_increment_v0`! But, the program looks obscure "just" because it's using a convoluted syntax to call `bind`. Many languages, most famously Haskell, provide specialized syntax to simplify writing computations that work with APIs like `bind` and `return`. F* provides some syntactic sugar to handle this too.

20.2 Monadic let bindings

The definition below defines a function with a special name `let!`. Names of this form, the token `let` followed by a sequence of one or more operator characters such as `!`, `?`, `@`, `$`, `<`, and `>` are monadic let-binding operators.

```
let (let!) = bind
```

With `let!` in scope, the following syntactic sugar becomes available:

- Instead of writing `bind f (fun x -> e)` you can write `let! x = f in e`.
- Instead of writing `bind f (fun _ -> e)` you can write `f ;! e`, i.e., a semicolon followed the sequence of operator characters uses in the monadic let-binding operator.
- Instead of writing `bind f (fun x -> match x with ...)`, you can write `match! f with ...`
- Instead of writing `bind f (fun x -> if x then ...)`, you can write `if! f then ...`

See this file MonadicLetBindings.fst for more details an examples of the syntactic sugar.

Using this syntactic sugar, we come to our final version of `read_and_increment`, where now, hopefully, the imperative-looking state updates make the intent of our program clear.

```
let read_and_increment : st int =
  let! x = read in
  write (x + 1) ;!
  return x
```

Having structured our programs with `return` and `bind`, larger `st` computations can be built from smaller ones, without having to worry about how to plumb the state through—that's handled once and for all by our combinators.

```
let inc_twice : st int =
  let! x = read_and_increment in
  read_and_increment ;!
  return x
```

20.2.1 `st` is a monad

It turns out that every API that is structured like our `st` is an instance of a general pattern called a *monad*, an algebraic structure. Specifically, a monad consists of:

- A type operator `m : Type -> Type`
- A function `return (#a:Type) (x:a) : m a`
- A function `bind (#a #b:Type) (f:m a) (g: a -> m b) : m b`

which satisfy the following laws, where ~ is some suitable equivalence relation on `m a`

- Left identity: `bind (return x) f ~ f`
- Right identity: `bind f return ~ f`
- Associativity: `bind f1 (fun x -> bind (f2 x) f3) ~ bind (bind f1 f2) f3`

Its easy to prove that `st`, `return`, and `bind` satisfy these laws in F*, where we pick the equivalence relation to equate functions that take equal arguments to equal results.

```
let feq #a #b (f g : a -> b) = forall x. f x == g x
let left_identity #a #b (x:a) (g: a -> st b)
  : Lemma ((let! v = return x in g v) `feq` g x)
  = ()
let right_identity #a (f:st a)
  : Lemma ((let! x = f in return x) `feq` f)
  = ()
let associativity #a #b #c (f1:st a) (f2:a -> st b) (f3:b -> st c)
  : Lemma ((let! x = f1 in let! y = f2 x in f3 y) `feq`
           (let! y = (let! x = f1 in f2 x) in f3 y))
  = ()
```

These laws are practically useful in that they can catch bugs in our implementations of the combinators. For instance, suppose we were to write `bind_buggy` below, which like in `inc_twice_buggy`, mistakenly re-uses the old state `s0` when calling `g`—in this case, the `right_identity` law below cannot be proved.

```
let bind_buggy #a #b
               (f: st a)
               (g: a -> st b)
  : st b
  = fun s0 ->
      let x, s1 = f s0 in
      g x s0
[@@expect_failure]
let right_identity_fail #a (f:st a)
  : Lemma (bind_buggy f return `feq` f)
  = ()
```

We can also prove laws about how the stateful actions, `read` and `write`, interact with each other in sequential composition.

```
let redundant_read_elim ()
  : Lemma ((read;! read) `feq` read)
  = ()

let redundant_write_elim (x y:int)
```

(continues on next page)

(continued from previous page)

```
  : Lemma ((write x ;! write y) `feq` write y)
  = ()

let read_write_noop ()
  : Lemma ((let! x = read in write x) `feq` return ())
  = ()
```

That completes our tour of our very first monad, the state monad.

20.2.2 Exercise

Make the `st` type generic, so that instead of the state being fixed to a single integer value, it can be used with any type for the state. I.e., define `st (s:Type) (a:Type) : Type`, where `s` is the type of the state.

Adapt the full development seen above to work with `st s`, including proving the various laws.

Exercise file

Answer

```
module Part2.ST

let st s a = s -> a & s

let read #s
  : st s s
  = fun s -> s, s

let write #s (s1:s)
  : st s unit
  = fun _ -> (), s1

let bind #s #a #b
         (f: st s a)
         (g: a -> st s b)
  : st s b
  = fun s0 ->
      let x, s1 = f s0 in
      g x s1

let return #s #a (x:a)
  : st s a
  = fun s -> x, s

let read_and_increment : st int int =
  x <-- read;
  write (x + 1);;
  return x

let inc_twice : st int int =
  x <-- read_and_increment;
  read_and_increment;;
  return x
```

(continues on next page)

(continued from previous page)

```
let feq #a #b (f g : a -> b) = forall x. f x == g x
let left_identity #s #a #b (x:a) (g: a -> st s b)
  : Lemma ((v <-- return x; g v) `feq` g x)
  = ()
let right_identity #s #a (f:st s a)
  : Lemma ((x <-- f; return x) `feq` f)
  = ()
let associativity #s #a #b #c (f1:st s a) (f2:a -> st s b) (f3:b -> st s c)
  : Lemma ((x <-- f1; y <-- f2 x; f3 y) `feq`
           (y <-- (x <-- f1; f2 x); f3 y))
  = ()

let redundant_read_elim ()
  : Lemma ((read;; read) `feq` read)
  = ()

let redundant_write_elim (x y:int)
  : Lemma ((write x ;; write y) `feq` write y)
  = ()

let read_write_noop ()
  : Lemma ((x <-- read;  write x) `feq` return ())
  = ()
```

20.2.3 Exercise

Monads can be used to model many computational effects, not just mutable state. Another common example is to use monads to model computations that may raise runtime errors. Here's an exercise to help you see how.

Prove that the `option` type can be made into a monad, i.e., define `bind` and `return` and prove the monad laws.

Exercise file

20.3 Computation Trees, or Monads Generically

Each time one defines a monad to model a computational effect, one usually thinks first of the effectful *actions* involved (e.g., reading and writing the state, or raising an error), and then finds a way to package those actions into the interface of monad with `return` and `bind`, and then, to keep things honest, proves that the implementation satisfies the monad laws.

However, a lot of this is boilerplate and can be done once and for all by representing effectful computations not just as functions (as we did with `st a = int -> a & s`) but instead as an inductive type that models a *computation tree*, with effectful actions made explicit at each node in the tree. One can prove that this representation, sometimes called a *free monad*, is a monad, and then instantiate it repeatedly for the particular kinds of actions that one may want to use in given program.

Note: In this section, we're scratching the surface of a rich area of research called *algebraic effects*. While what we show here is not a full-blown algebraic effects development (we'll save that for a later chapter), here are some other resources about it.

- Alg.fst: An F* development with algebraic effects (to be covered in detail later).
- Koka, a language with algebraic effects at its core
- A bibliography about effects

We'll start our development of computation trees with a type describing the signature of a language of actions.

```
noeq
type action_class = {
  t : Type;
  input_of : t -> Type;
  output_of : t -> Type;
}
```

This kind of signature is sometimes called a *type class*, a type `act:Type`, together with some operations it supports. In this case, the operations tell us what kind of input and output a given action expects.

Note: F* also provides support for type classes and inference of type class instantiations. This will be described in a later chapter. Meanwhile, you can learn more about type classes in F* from the wiki and from these examples.

For example, if we were interested in just the read/write actions on a mutable integer state (as in our `st a` example), we could build an instance of the `action_class`, as shown below.

```
type rw =
  | Read
  | Write

let input_of : rw -> Type =
  fun a ->
    match a with
    | Read -> unit
    | Write -> int

let output_of : rw -> Type =
  fun a ->
    match a with
    | Read -> int
    | Write -> unit

let rw_action_class = { t = rw; input_of ; output_of }
```

However, we can define a type `tree acts a`, the type of a computation tree whose effectful actions are from the class `acts`, completely generically in the actions themselves.

```
noeq
type tree (acts:action_class) (a:Type) =
  | Return : x:a -> tree acts a
  | DoThen : act:acts.t ->
```

(continues on next page)

(continued from previous page)

```
            input:acts.input_of act ->
            continue: (acts.output_of act -> tree acts a) ->
            tree acts a
```

A `tree act a` has just two cases:

- Either it is a leaf node, `Return x`, modeling a computation that immediately returns the value `x`;
- Or, we have a node `DoThen act input k`, modeling a computation that begins with some action `act`, to which we pass some input, and where `k` represents all the possible "continuations" of the action, represented by a `tree act a` for each possible output returned by the action. That is, `DoThen` represents a node in the tree with a single action and a possibly infinite number of sub-trees.

With this representation we can define `return` and `bind`, and prove the monad laws once and for all.

```
let return #a #acts (x:a)
  : tree acts a
  = Return x

let rec bind #acts #a #b (f: tree acts a) (g: a -> tree acts b)
  : tree acts b
  = match f with
    | Return x -> g x
    | DoThen act i k ->
      DoThen act i (fun x -> bind (k x) g)
```

- The `return` combinator is easy, since we already have a `Return` leaf node in the tree type.
- The `bind` combinator is a little more interesting, involving a structural recursion over the tree, relying (as we did in the previous chapter on well-founded recursion) on the property that all the trees returned by `k` are strictly smaller than the original tree `f`.

To prove the monad laws, we first need to define an equivalence relation on trees—this relation is not quite just `==`, since each continuation in the tree is function which itself returns a tree. So, we define `equiv` blow, relating trees that are both `Returns`, or when they both begin with the same action and have pointwise-equivalent continuations.

```
let rec equiv #acts #a (x y: tree acts a) =
  match x, y with
  | Return vx, Return vy -> vx == vy
  | DoThen actx ix kx, DoThen acty iy ky ->
    actx == acty /\
    ix == iy /\
    (forall o. equiv (kx o) (ky o))
  | _ -> False
```

Note: We are specifically avoiding the use of *functional extensionality* here, a property which allows equating pointwise equal η-expanded functions. We show how one can use functional extensionality in this development as an advanced exercise.

To prove that `equiv` is an equivalence relation, here are lemmas that prove that it is reflexive, symmetric, and transitive—we see here a use of the syntactic sugar for logical connectives, *introduced in a previous chapter*.

```
let rec equiv_refl #acts #a (x:tree acts a)
  : Lemma (equiv x x)
```

(continues on next page)

(continued from previous page)

```
  = match x with
    | Return v -> ()
    | DoThen act i k ->
      introduce forall o. equiv (k o) (k o)
      with (equiv_refl (k o))

let rec equiv_sym #acts #a (x y:tree acts a)
  : Lemma
    (requires equiv x y)
    (ensures equiv y x)
  = match x, y with
    | Return _, Return _ -> ()
    | DoThen act i kx, DoThen _ _ ky ->
      introduce forall o. equiv (ky o) (kx o)
      with equiv_sym (kx o) (ky o)

let rec equiv_trans #acts #a (x y z: tree acts a)
  : Lemma
    (requires equiv x y /\ equiv y z)
    (ensures equiv x z)
  = match x, y, z with
    | Return _, _, _ -> ()
    | DoThen act i kx, DoThen _ _ ky, DoThen _ _ kz ->
      introduce forall o. equiv (kx o) (kz o)
      with equiv_trans (kx o) (ky o) (kz o)
```

Now, we can prove that `tree` satisifies the monad laws with respect to `equiv`.

```
let right_identity #acts #a #b (x:a) (g:a -> tree acts b)
  : Lemma (bind (return x) g `equiv` g x)
  = equiv_refl (g x)

let rec left_identity #acts #a (f:tree acts a)
  : Lemma (bind f return `equiv` f)
  = match f with
    | Return _ -> ()
    | DoThen act i k ->
      introduce forall o. bind (k o) return `equiv` (k o)
      with left_identity (k o)

let rec assoc #acts #a #b #c
              (f1: tree acts a)
              (f2: a -> tree acts b)
              (f3: b -> tree acts c)
  : Lemma (bind f1 (fun x -> bind (f2 x) f3) `equiv`
           bind (bind f1 f2) f3)
  = match f1 with
    | Return v ->
      right_identity v f2;
      right_identity v (fun x -> bind (f2 x) f3)
    | DoThen act i k ->
      introduce forall o. bind (k o) (fun x -> bind (f2 x) f3) `equiv`
```

(continues on next page)

(continued from previous page)

```
                     bind (bind (k o) f2) f3
      with assoc (k o) f2 f3
```

The associativity law, in particular, should make intuitive sense in that a `tree acts a` represents a computation in a canonical fully left-associative form, i.e., a single action followed by its continuation. As such, no matter how you associate computations in `bind`, the underlying representation is always fully left-associated.

Having defined our computation trees generically, we can use them with any actions we like. For example, here's our `read_and_increment` re-built using computation trees.

```
let read : tree rw_action_class int = DoThen Read () Return
let write (x:int) : tree rw_action_class unit = DoThen Write x Return
let read_and_increment
  : tree rw_action_class int
  = x <-- read ;
    write (x + 1);;
    return x
```

Finally, given a computation tree we can "run" it, by interpreting it as a state-passing function.

```
let st a = int -> a & int
let rec interp #a (f: tree rw_action_class a)
  : st a
  = fun s0 ->
     match f with
     | Return x -> x, s0
     | DoThen Read i k ->
       interp (k s0) s0
     | DoThen Write s1 k ->
       interp (k ()) s1
```

Note: A main difference between what we've shown here with `interp` and a general treament of algebraic effects is that rather than "bake-in" the interpretation of the individual actions in `interp`, we can also abstract the semantics of the actions using an idea similar to exception handling, allowing the context to customize the semantics of the actions simply by providing a different handler.

20.3.1 Exercise

Prove that the `interp` function interprets equivalent trees `f` and `g` to pointwise equivalent functions.

Exercise File

Answer

```
let feq #a #b (f g: a -> b) = forall x. f x == g x
let rec interp_equiv #a (f g:tree rw_action_class a)
  : Lemma
    (requires equiv f g)
    (ensures feq (interp f) (interp g))
  = match f, g with
    | Return _, Return _ -> ()
```

(continues on next page)

(continued from previous page)

```
    | DoThen act i kf, DoThen _ _ kg ->
      introduce forall o. feq (interp (kf o)) (interp (kg o))
      with interp_equiv (kf o) (kg o)
```

20.3.2 Exercise

Instead of proving every time that a function like `interp` produces equivalent results when applied to equivalent trees, using functional extensionality, we can prove that equivalent trees are actually provably equal, i.e., `equiv x y ==> x == y`.

This is a little technical, since although functional extensionality is a theorem in F*, it is only true of η-expanded functions.

Try to use `FStar.FunctionalExtensionality.fsti` to adapt the definitions shown above so that we can prove the lemma `equiv x y ==> x == y`.

Answer

```
module Part2.FreeFunExt
open FStar.Classical.Sugar
module F = FStar.FunctionalExtensionality
open FStar.FunctionalExtensionality

noeq
type action_class = {
  t : Type;
  input_of : t -> Type;
  output_of : t -> Type;
}


noeq
type tree (acts:action_class) (a:Type) =
  | Return : x:a -> tree acts a
  | DoThen : act:acts.t ->
             input:acts.input_of act ->
             //We have to restrict continuations to be eta expanded
             //that what `^->` does. Its defined in FStar.FunctionalExtensionality
             continue:(acts.output_of act ^-> tree acts a) ->
             tree acts a

let return #a #acts (x:a)
  : tree acts a
  = Return x

let rec bind #acts #a #b (f: tree acts a) (g: a -> tree acts b)
  : tree acts b
  = match f with
    | Return x -> g x
    | DoThen act i k ->
      //Now, we have to ensure that continuations are instances of
```

(continues on next page)

(continued from previous page)

```
      //F.( ^-> )
      DoThen act i (F.on _ (fun x -> bind (k x) g))

let rec equiv #acts #a (x y: tree acts a) =
  match x, y with
  | Return vx, Return vy -> vx == vy
  | DoThen actx ix kx, DoThen acty iy ky ->
    actx == acty /\
    ix == iy /\
    (forall o. equiv (kx o) (ky o))
  | _ -> False

let rec equiv_refl #acts #a (x:tree acts a)
  : Lemma (equiv x x)
  = match x with
    | Return v -> ()
    | DoThen act i k ->
      introduce forall o. equiv (k o) (k o)
      with (equiv_refl (k o))

let rec equiv_sym #acts #a (x y:tree acts a)
  : Lemma
    (requires equiv x y)
    (ensures equiv y x)
  = match x, y with
    | Return _, Return _ -> ()
    | DoThen act i kx, DoThen _ _ ky ->
      introduce forall o. equiv (ky o) (kx o)
      with equiv_sym (kx o) (ky o)

let rec equiv_trans #acts #a (x y z: tree acts a)
  : Lemma
    (requires equiv x y /\ equiv y z)
    (ensures equiv x z)
  = match x, y, z with
    | Return _, _, _ -> ()
    | DoThen act i kx, DoThen _ _ ky, DoThen _ _ kz ->
      introduce forall o. equiv (kx o) (kz o)
      with equiv_trans (kx o) (ky o) (kz o)

// THIS IS THE MAIN LEMMA OF THE EXERCISE
let rec equiv_is_equal #acts #a (x y: tree acts a)
  : Lemma
    (requires equiv x y)
    (ensures x == y)
  = match x, y with
    | Return _, Return _ -> ()
    | DoThen act i kx, DoThen _ _ ky ->
      introduce forall o. kx o == ky o
      with equiv_is_equal (kx o) (ky o);
      F.extensionality _ _ kx ky
```

(continues on next page)

(continued from previous page)

```
let right_identity #acts #a #b (x:a) (g:a -> tree acts b)
  : Lemma (bind (return x) g `equiv` g x)
  = equiv_refl (g x)

let rec left_identity #acts #a (f:tree acts a)
  : Lemma (bind f return `equiv` f)
  = match f with
    | Return _ -> ()
    | DoThen act i k ->
      introduce forall o. bind (k o) return `equiv` (k o)
      with left_identity (k o)

let rec assoc #acts #a #b #c
              (f1: tree acts a)
              (f2: a -> tree acts b)
              (f3: b -> tree acts c)
  : Lemma (bind f1 (fun x -> bind (f2 x) f3) `equiv`
           bind (bind f1 f2) f3)
  = match f1 with
    | Return v ->
      right_identity v f2;
      right_identity v (fun x -> bind (f2 x) f3)
    | DoThen act i k ->
      introduce forall o. bind (k o) (fun x -> bind (f2 x) f3) `equiv`
                     bind (bind (k o) f2) f3
      with assoc (k o) f2 f3

type rw =
  | Read
  | Write

let input_of : rw -> Type =
  fun a ->
    match a with
    | Read -> unit
    | Write -> int

let output_of : rw -> Type =
  fun a ->
    match a with
    | Read -> int
    | Write -> unit

let rw_action_class = { t = rw; input_of ; output_of }

//Here again the continuation has to be F.( ^-> )
let read : tree rw_action_class int = DoThen Read () (F.on _ Return)
let write (x:int) : tree rw_action_class unit = DoThen Write x (F.on _ Return)
let read_and_increment
  : tree rw_action_class int
  = x <-- read ;
    write (x + 1);;
```

(continues on next page)

(continued from previous page)

```
    return x

let st a = int -> a & int
let rec interp #a (f: tree rw_action_class a)
  : st a
  = fun s0 ->
     match f with
     | Return x -> x, s0
     | DoThen Read i k ->
       interp (k s0) s0
     | DoThen Write s1 k ->
       interp (k ()) s1

// And now since equivalent trees and equal, lemmas such as this
// become trivial
let interp_equiv #a (f g:tree rw_action_class a)
  : Lemma
    (requires equiv f g)
    (ensures feq (interp f) (interp g))
  = equiv_is_equal f g
```

20.4 Manipulating Computation Trees: Nondeterminism and Concurrency

As a final bit, we show that representing computations as trees is not just useful from a perspective of genericity and code re-use. Computation trees expose the structure of a computation in a way that allows us to manipulate it, e.g., interpreting actions in an alternative semantics.

In this section, we enhance our computation trees to support non-deterministic choice, i.e., given pair of computations `l, r:tree acts a`, we can non-deterministically choose to evaluate `l` or `r`. With this capability, we can also express some models of concurrency, e.g., a semantics that interleaves imperative actions from several threads.

Let's start by enhancing our `tree` type to now include an node `Or l r`, to represent non-deterministic choice between `l` and `r`.

```
noeq
type tree (acts:action_class) (a:Type) =
  | Return : x:a -> tree acts a
  | DoThen : act:acts.t ->
             input:acts.input_of act ->
             continue: (acts.output_of act -> tree acts a) ->
             tree acts a
  | Or :  tree acts a -> tree acts a -> tree acts a
```

As before, we can define `return` and `bind`, this time in `bind` we sequence `g` after both choices in `Or`.

```
let return #acts #a (x:a)
  : tree acts a
  = Return x
```

(continues on next page)

(continued from previous page)

```

let rec bind #acts #a #b (f: tree acts a) (g: a -> tree acts b)
  : tree acts b
  = match f with
    | Return x -> g x
    | DoThen act i k ->
      DoThen act i (fun x -> bind (k x) g)
    | Or m0 m1 -> Or (bind m0 g) (bind m1 g)
```

What's more interesting is that, in addition to sequential composition, we can also define parallel composition of a pair of computations using `par f g`, as shown below.

```
let rec l_par #acts #a #b (f:tree acts a) (g:tree acts b)
  : tree acts (a & b)
  = match f with
    | Return v -> x <-- g; return (v, x)
    | DoThen a i k ->
      DoThen a i (fun x -> r_par (k x) g)
    | Or m0 m1 -> Or (l_par m0 g) (l_par m1 g)

and r_par #acts #a #b (f:tree acts a) (g: tree acts b)
  : tree acts (a & b)
  = match g with
    | Return v -> x <-- f; return (x, v)
    | DoThen a i k ->
      DoThen a i (fun x -> l_par f (k x))
    | Or m0 m1 -> Or (r_par f m0) (r_par f m1)

let par #acts #a #b (f: tree acts a) (g: tree acts b)
  : tree acts (a & b)
  = Or (l_par f g) (r_par f g)
```

There's quite a lot going on here, so let's break it down a bit:

- The functions `l_par f g` and `r_par f g` are mutually recursive and define an interleaving semantics of the actions in `f` and `g`.
- `l_par f g` is left-biased: picking an action from `f` to execute first (if any are left); while `r_par f g` is right-biased, picking an action from `g` to execute first.
- Consider the `DoThen` case in `l_par`: if picks the head action `a` from `f` and the recurses in the continuation with `r_par (k x) g`, to prefer executing first an action from `g` rather than `k x`. The `DoThen` case of `r_par` is symmetric.
- For `l_par`, in the non-deterministic choice case (`Or`), we interleave either choice of `f` with `g`, and `r_par` is symmetric.
- Finally, we define parallel composition `par f g` as the non-deterministic choice of either the left-biased or right-biased interleaving of the actions of `f` and `g`. This fixes the semantics of parallel composition to a round-robin scheduling of the actions between the threads, but one could imagine implementing other kinds of schedulers too.

As before, we can now instantiate our tree with read/write actions and write some simple programs, including `par_inc`, a computation that tries to increment the counter twice in parallel.

```
let read : tree rw_actions int = DoThen Read () Return
let write (x:int) : tree rw_actions unit = DoThen Write x Return
let inc
  : tree rw_actions unit
  = x <-- read ;
    write (x + 1)
let par_inc = par inc inc
```

However, there's trouble ahead—because of the interleaving semantics, we don't actually increment the state twice.

To check, let's define an interpretation function to run our computations. Since we need to resolve the non-deterministic choice in the `Or` nodes, we'll parameterize our intepreter by a source of "randomness", an infinite stream of booleans.

```
let randomness = nat -> bool
let par_st a = randomness -> pos:nat -> s0:int -> (a & int & nat)
let rec interp #a (f:tree rw_actions a)
  : par_st a
  = fun rand pos s0 ->
      match f with
      | Return x -> x, s0, pos
      | DoThen Read _ k -> interp (k s0) rand pos s0
      | DoThen Write s1 k -> interp (k ()) rand pos s1
      | Or l r ->
        if rand pos
        then interp l rand (pos + 1) s0
        else interp r rand (pos + 1) s0
let st a = int -> a & int
let interpret #a (f:tree rw_actions a)
  : st a
  = fun s0 ->
      let x, s, _ = interp f (fun n -> n % 2 = 0) 0 s0 in
      x, s
```

This interpreter is very similar to our prior interpreter, except in the `Or` case, we read a boolean from `rand`, our randomness stream, and choose the left- or right-branch accordingly.

We can run our program on this interpreter and check what it returns. One way to do this is to make use of F*'s normalizer, the abstract machine that F* uses to reduce computations during type-checking. The `assert_norm p` feature used below instructs F* to symbolically reduce the term `p` as much as possible and then check that the result is equivalent to `True`.

Note: F*'s emacs mode `fstar-mode.el` provides some utilites to allow reducing terms of F*'s abstract machine and showing the results to the user. F*'s tactics also also allow evaluating terms and viewing the results—we leave further discussion of these features to a future chapter.

```
let test_prog = assert_norm (forall x. snd (interpret par_inc x) == x + 1)
```

In this case, we ask F* to interpret `par_inc` on the interpreter we just defined. And, indeed, F* confirms that in the final state, we have incremented the state only once. Due to the round-robin scheduling of actions, the interpreter has executed both the reads before both the writes, making one of the reads and one of the writes redundant.

20.4.1 Exercise

Define an action class that include an increment operation, in addition to reads and writes. Adapt the interpreter shown above to work with this action class and prove (using `assert_norm`) that a program that contains two parallel atomic increments increments the state twice.

Exercise File

Answer

```
type rwi =
  | R
  | W
  | Inc

let input_of_rwi : rwi -> Type =
  fun a ->
    match a with
    | R -> unit
    | W -> int
    | Inc -> unit

let output_of_rwi : rwi -> Type =
  fun a ->
    match a with
    | R -> int
    | W -> unit
    | Inc -> unit

let rwi_actions = { t = rwi; input_of=input_of_rwi ; output_of=output_of_rwi }

let atomic_inc : tree rwi_actions unit = DoThen Inc () Return

let rec interp_rwi #a (f:tree rwi_actions a)
  : par_st a
  = fun tape pos s0 ->
      match f with
      | Return x -> x, s0, pos
      | DoThen R _ k -> interp_rwi (k s0) tape pos s0
      | DoThen W s1 k -> interp_rwi (k ()) tape pos s1
      | DoThen Inc () k -> interp_rwi (k ()) tape pos (s0 + 1)
      | Or l r ->
        let b = tape pos in
        if b
        then interp_rwi l tape (pos + 1) s0
        else interp_rwi r tape (pos + 1) s0
let interpret_rwi #a (f:tree rwi_actions a)
  : st a
  = fun s0 ->
      let x, s, _ = interp_rwi f (fun n -> n % 2 = 0) 0 s0 in
      x, s

let par_atomic_inc = par atomic_inc atomic_inc
let test_par_atomic_inc = assert_norm (forall x. snd (interpret_rwi par_atomic_inc x) == x + 2)
```

20.5 Looking ahead

Writing correct programs with side-effects is hard, particularly when those effects include features like mutable state and concurrency!

What we've seen here is that although we've been able to model the semantics of these programs, proving that they work correctly is non-trivial. Further, while we have defined interpreters for our programs, these interpreters are far from efficient. In practice, one usually resorts to things like shared-memory concurrency to gain performance and our interpreters, though mathematically precise, are horribly slow.

Addressing these two topics is the main purpose of F*'s user-defined effect system, a big part of the language which we'll cover in a subsequent section. The effect system aims to address two main needs:

- Proofs of effectful programs: The effect system enables developing effectful programs coupled with *program logics* that enable specifying and proving program properties. We'll learn about many different kinds of logics that F* libraries provide, ranging from classical Floyd-Hoare logics for sequential programs, relational logics for program equivalence, weakest precondition calculi, and separation logics for concurrent and distributed programs.
- Effect abstraction: Although programs can be specified and proven against a clean mathematical semantics, for efficient execution, F* provides mechanisms to hide the representation of an effect so that effectful programs can be compiled efficiently to run with native support for effects like state, exceptions, concurrency, and IO.

CHAPTER

TWENTYONE

UNIVERSES

As mentioned *earlier*, `Type` is the type of types. So, one might ask the question, what is the type of `Type` itself? Indeed, one can write the following and it may appear at first that the type of `Type` is `Type`.

```
let ty : Type = Type
```

However, behind the scenes, F* actually has a countably infinite hierarchy of types, `Type u#0`, `Type u#1`, `Type u#2`, . . . , and the type of `Type u#i` is actually `Type u#(i + 1)`. The `u#i` suffixes are called *universe levels* and if you give F* the following option, it will actually show you the universe levels it inferred when it prints a term.

```
#push-options "--print_universes"
```

With this option enabled, in fstar-mode.el, the F* emacs plugin, hovering on the symbol `ty` prints `Type u#(1 + _)`, i.e., the type of `ty` is in a universe that is one greater than some universe metavariable `_`, i.e., `ty` is universe *polymorphic*. But, we're getting a bit ahead of ourselves.

In this chapter, we'll look at universe levels in detail, including why they're necessary to avoid paradoxes, and showing how to manipulate definitions that involve universes. For the most part, F* infers the universe levels of a term and you don't have to think too much about it—in fact, in all that we've seen so far, F* inferred universe levels behind the scenes and we haven't had to mention them. Though, eventually, they do crop up and understanding what they mean and how to work with them becomes necessary.

Other resources to learn about universes:

- The Agda manual has a nice chapter on universes, including universe polymorphism.
- This chapter from Adam Chlipala's Certified Programming with Dependent Types describes universes in Coq. While it also provides useful background, F*'s universe system is more similar to Agda's and Lean's than Coq's.

21.1 Basics

A universe annotation on a term takes the form `u#l`, where `l` is a universe level. The universe levels are terms from the following grammar:

```
k ::= 0 | 1 | 2 | ...     any natural number constant
l ::= k                   universe constant
    | l + k | k + l       constant offset from level l
    | max l1 l2           maximum of two levels
    | a | b | c | ...     level variables
```

Let's revisit our first example, this time using explicit universe annotations to make things clearer.

We've defined, below, instances of `Type` for universe levels `0, 1, 2` and we see that each of them has a type at the next level. The constant `Type u#0` is common enough that F* allows you to write `Type0` instead.

```
let ty0 : Type u#1 = Type u#0
let ty0' : Type u#1 = Type0
let ty1 : Type u#2 = Type u#1
let ty2 : Type u#3 = Type u#2
```

If you try to define `ty_bad` below, F* complains with the following error:

```
let ty_bad : Type u#0 = Type u#0
```

```
Expected expression of type "Type0"; got expression "Type0" of type "Type u#1"
```

The restriction that prevents a `Type` from being an inhabitant of itself if sometimes called *predicativity*. The opposite, *impredicativity*, if not suitably restricted, usually leads to logical inconsistency. F* provides a limited form of impredicativity through the use of `squash` types, which we'll see towards the end of this chapter.

Note: That said, if we didn't turn on the option `--print_universes`, the error message you get may be, sadly, bit baffling:

```
Expected expression of type "Type"; got expression "Type" of type "Type"
```

Turning on `--print_universes` and `--print_implicits` is a good way to make sense of type errors where the expected type and the type that was computed seem identical.

Now, instead of defining several constants like `ty0, ty1, ty2` etc., F* definitions can be *universe polymorphic*. Below, we define `ty_poly` as `Type u#a`, for any universe variable `a`, and so `ty` has type `Type u#(a + 1)`.

```
let ty_poly : Type u#(a + 1) = Type u#a
```

One way to think of `ty_poly` is as a "definition template" parameterized by the by the universe-variable `a`. One can instantiate `ty_poly` with a specific universe level `l` (by writing `ty_poly u#l`) and obtain a copy of its definition specialized to level `l`. F* can prove that instantiation of `ty_poly` are equal to the non-polymorphic definitions we had earlier. As the last example shows, F* can usually infer the universe instantiation, so you often don't need to write it.

```
let _ = assert (ty_poly u#0 == ty0)
let _ = assert (ty_poly u#1 == ty1)
let _ = assert (ty_poly u#2 == ty2)
let _ = assert (ty_poly == ty0)
```

21.2 Universe computations for other types

Every type in F* lives in a particular universe. For example, here are some common types in `Type u#0`.

```
let _ : Type0 = nat
let _ : Type0 = bool
let _ : Type0 = nat -> bool
```

Universe of an arrow type: In general, the universe of an arrow type `x:t -> t'` is the the maximum of the universe of `t` and `t'`.

This means that functions that are type-polymorphic live in higher universes. For example, the polymorphic identity function that we saw in an *earlier section*, is actually also polymorphic in the universe level of its type argument.

```
let id_t : Type u#(i + 1) = a:Type u#i -> a -> a
let id : id_t = fun a x -> x
```

That is, the type of the identity function `id` is `id_t` or `a:Type u#i -> a -> a`—meaning, for all types `a` in universe `Type u#i`, `id a` is function of type `a -> a`.

Now, `id_t` is itself a type in universe `Type u#(i + 1)`, and since the `id` function can be applied to types in any universe, it can be applied to `id_t` too. So, it may look like this allows one to write functions that can be applied to themselves—which would be bad, since that would allow one to create infinite loops and break F*'s logic.

```
let seemingly_self_application : id_t = id _ id
```

However, if we write out the universe levels explicitly, we see that actually we aren't really applying the `id` function to itself. Things are actually stratified, so that we are instead applying an instance of `id` at universe `u#(i + 1)` to the instance of `id` at universes `u#i`.

```
let stratified_application : id_t u#i = id u#(i + 1) (id_t u#i) (id u#i)
```

One intuition for what's happening here is that there are really infinitely many instances of the F* type system nested within each other, with each instance forming a universe. Type-polymorphic functions (like `id`) live in some universe `u#(a + 1)` and are parameterized over all the types in the immediately preceding universe `u#a`. The universe levels ensure that an F* function within universe level `u#a` cannot consume or produce terms that live in some greater universe.

21.2.1 Universe level of an inductive type definition

F* computes a universe level for inductive type definitions. To describe the rules for this in full generality, consider again the general form of an inductive type definition, shown first *here*, but this time with the universe level of each type constructor made explicit, i.e., T_i constructs a type in universe Type u#l$_i$

$$\text{type } T_1 \ \overline{(x_1 : p_i)} : \overline{y_1 : q_1} \to \text{Type } u\#l_1 = \overline{\bar{D}_1 : t_1}$$
$$\text{and } T_n \ \overline{(x_n : p_n)} : \overline{y_n : q_n} \to \text{Type } u\#l_n = \overline{\bar{D}_n : t_n}$$

Recall that each type constructor T_i has zero or more *data constructors* $\overline{D_i : t_i}$ and for each data constructor D_{ij}, its type t_{ij} must be of the form $\overline{z_{ij} : s_{ij}} \to T_i \ \bar{x}_i \ \bar{e}$

In addition to checking, as usual, that the each t_{ij} is well-typed, F* also checks the universe levels according to the following rule:

- Assuming that each T_i has universe level l_i, for every data constructor D_{ij}, and for each of its arguments $z_{ijk} : s_{ijk}$, check s_{ijk} : Type $u\#l_{ijk}$ and $l_{ijk} \leq l_i$.

In other words, the universe level of each type constructor must not be less than the universe of any of the fields of data constructors.

In practice, F* infers the universe levels $l_1, \ldots, l_n$, by collecting level-inequality constraints and solving them using the `max` operator on universe levels, i.e., l_i is set to $max_{jk} \ l_{ijk}$, the maximum of the universe levels of all the fields of the constructors $\overline{D_i : t_i}$. Let's look at some examples.

The list type

The list type below is parameterized by a:Type u#a and constructs a type in the same universe.

```
type list (a:Type u#a) : Type u#a  =
 | Nil : list a
 | Cons : hd:a -> tl:list a -> list a
```

- The Nil constructor has no fields, so it imposes no constraints on the universe level of list a.
- The Cons constructor has two fields. Its first field hd has type a: Type u#a: we have a constraint that u#a $\leq$ u#a; and the second field, by assumption, has type list a : Type u#a, and again we have the constraint u#a $\leq$ u#a.

F* infers the minimum satisfiable universe level assignment, by default. But, there are many solutions to the inequalities, and if needed, one can use annotations to pick another solution. For example, one could write this, though it rarely makes sense to pick a universe for a type higher than necessary (see *this section* for an exception).

```
type list' (a:Type u#a) : Type u#(1 + a)  =
 | Nil' : list' a
 | Cons' : hd:a -> tl:list' a -> list' a
```

Note: Universe level variables are drawn from a different namespace than other variables. So, one often writes a:Type u#a, where a is a regular variable and u#a is the universe level of the type of a.

The pair type

The pair type below is parameterized by a:Type u#a and b:Type u#b and constructs a type in u#(max a b).

```
type pair (a:Type u#a) (b:Type u#b) : Type u#(max a b) =
  | Pair : fst:a -> snd:b -> pair a b
```

- The fst field is in u#a and so we have u#a $\leq$ u#(max a b).
- The snd field is in u#b and so we have u#b $\leq$ u#(max a b).

The top type

The top type below packages a value at any type a:Type u#a with its type.

```
noeq
type top : Type u#(a + 1) =
  | Top : a:Type u#a -> v:a -> top
```

- The a field of Top is in u#(a + 1) while the v field is in u#a. So, top itself is in u#(max (a + 1) a), which simplifies to u#(a + 1).

One intuition for why this is so is that from a value of type t : top one can write a function that computes a value of type Type u#a, i.e., Top?.a t. So, if instead we have top: Type u#a and t:top, then Top?.a : top -> Type u#a, which would break the stratification of universes, since from a value top in universe u#a, we would be able to project out a value in Type u#(a + 1), which leads to a paradox, as we'll see next.

What follows is quite technical and you only need to know that the universe system exists to avoid paradoxes, not how such paradoxes are constructed.

21.3 Russell's Paradox

Type theory has its roots in Bertrand Russell's The Principles of Mathematics, which explores the logical foundations of mathematics and set theory. In this work, Russell proposed the paradoxical set Δ whose elements are exactly all the sets that don't contain themselves and considered whether or not Δ contained itself. This self-referential construction is paradoxical since:

- If $\Delta \in \Delta$, then since the only sets in Δ are the sets that don't contain themselves, we can conclude that $\Delta \notin \Delta$.
- On the other hand, if $\Delta \notin \Delta$, then since Δ contains all sets that don't contain themselves, we can conclude that $\Delta \in \Delta$.

To avoid such paradoxes, Russell formulated a stratified system of types to prevent nonsensical constructions that rely on self-reference. The universe levels of modern type theories serve much the same purpose.

In fact, as the construction below shows, if it were possible to break the stratification of universes in F*, then one can code up Russell's Δ set and prove `False`. This construction is derived from Thorsten Altenkirch's Agda code. Liam O'Connor also provides some useful context and comparison. Whereas the Agda code uses a special compiler pragma to enable unsound impredicativity, in F* we show how a user-introduced axiom can simulate impredicativity and enable the same paradox.

21.3.1 Breaking the Universe System

Consider the following axioms that intentionally break the stratification of F*'s universe system. We'll need the following ingredients:

1. A strictly positive type constructor `lower` that takes a type in any universe `a:Type u#a`, and returns a type in `u#0`. Note, we covered *strictly positive type functions, previously*.

```
assume
val lower ([@@@strictly_positive] a:Type u#a) : Type u#0
```

2. Assume there is a function called `inject`, which takes value of type `x:a` and returns value of type `lower a`.

```
assume
val inject (#a:Type u#a) (x:a) : lower a
```

3. `lower` and `inject` on their own are benign (e.g., `let lower _ = unit` and `let inject _ = ()`). But, now if we assume we have a function `project` that is the inverse of `inject`, then we've opened the door to paradoxes.

```
assume
val project (#a:Type u#a) (x:lower a) : a

assume
val inj_proj (#a:Type u#a) (x:a)
  : Lemma (project (inject x) == x)
```

21.3.2 Encoding Russell's Paradox

To show the paradox, we'll define a notion of `set` in terms of a form of set comprehensions `f:  x -> set`, where `x:Type u#0` is the domain of the comprehension, supposedly bounding the cardinality of the set. We'll subvert the universe system by treating `set` as living in universe `u#0`, even though its constructor `Set` has a field `x:Type u#0` that has universe level `u#1`

```
noeq
type _set : Type u#1 =
  | Set : x:Type0 -> f:(x -> set) -> _set
and set : Type0 = lower _set
```

This construction allows us to define many useful sets. For example, the empty set `zero` uses the empty type `False` as the domain of its comprehension and so has no elements; or the singleton set `one` whose only element is the empty set; or the set `two` that contains the empty set `zero` and the singleton set `one`.

```
let zero : set = inject (Set False (fun _ -> false_elim()))
let one : set = inject (Set True (fun _ -> zero))
let two : set = inject (Set bool (fun b -> if b then zero else one))
```

One can also define set membership: A set `a` is a member of a set `b`, if one can exhibit an element `v` of the domain type of `b` (i.e., `(project b).x`), such that `b`'s comprehension `(project b).f` applied to `v` is `a`.

For example, one can prove `mem zero two` by picking `true` for `v` and `mem one two` by picking `false` for `v`. Non-membership is just the negation of membership.

```
let mem (a:set) (b:set) : Type0 =
  (v:(project b).x & (a == (project b).f v))

let not_mem (a:set) (b:set) : Type0 = mem a b -> False
```

Now, we are ready to define Russell's paradoxical set Δ. First, we define `delta_big` in a larger universe and then use `inject` to turn it into a `set :  Type u#0`. The encoding of `delta_big` is fairly direct: Its domain type is the type of sets `s` paired with a proof that `s` does not contain itself; and its comprehension function just returns `s` itself.

```
let delta_big = Set (s:set & not_mem s s) dfst
let delta : set = inject delta_big
```

We can now prove both ``delta `mem` delta`` and ``delta `not_mem` delta``, using the unsound `inj_proj` axiom that breaks the universe system, and derive `False`.

```
let x_in_delta_x_not_in_delta (x:set) (mem_x_delta:mem x delta)
  : not_mem x x
  = let (| v, r |) = mem_x_delta in // mem proofs are pairs
    let v : (project delta).x = v in // whose first component is an element of delta's␣
↪comprehension domain
    let r : (x == (project delta).f v) = r in //and whose second component proves that x␣
↪is equal to an element in delta
    inj_proj delta_big; // we use the unsound axiom to conclude that `v` is actually the␣
↪domain of delta_big
    let v : (s:set & not_mem s s) = v in //and now we can retype `v` this way
    let (| s, pf |) = v in //and unpack it into its components
    let r : (x == s) = r in //and the axiom also allows us to retype `r` this way
    let pf : not_mem x x = pf in //which lets us convert pf from `not_mem s s` to the goal
    pf //not_mem x x
```

(continues on next page)

(continued from previous page)

```
let delta_not_in_delta
  : not_mem delta delta
  = fun (mem_delta_delta:mem delta delta) ->
      x_in_delta_x_not_in_delta
          delta
          mem_delta_delta
          mem_delta_delta

let x_not_mem_x_x_mem_delta (x:set) (x_not_mem_x:x `not_mem` x)
  : x `mem` delta
  = let v : (s:set & not_mem s s) = (| x, x_not_mem_x |) in //an element of the domain␣
↪set of delta_big
    inj_proj delta_big; // the unsound axiom now lets us relate it to delta
    let s : (x == (project delta).f v) = //and prove that projecting delta's␣
↪comprehension and applying to v return x`
        FStar.Squash.return_squash Refl
    in
    (| v,  s |)

let delta_in_delta
  : mem delta delta
  = x_not_mem_x_x_mem_delta delta delta_not_in_delta

let ff : False = delta_not_in_delta delta_in_delta
```

The proofs are more detailed than they need to be, and if you're curious, maybe you can follow along by reading the comments.

The upshot, however, is that without the stratification of universes, F* would be unsound.

21.4 Refinement types, FStar.Squash, `prop`, and Impredicativity

We've seen how universes levels are computed for arrow types and inductive type definitions. The other way in which types can be formed in F* is with refinement types: `x:t{p}`. As we've seen previously, a value `v` of type `x:t{p}` is just a `v:t` where `p[v/x]` is derivable in the current scope in F*'s SMT-assisted classical logic—there is no way to extract a proof of `p` from a proof of `x:t{p}`, i.e., refinement types are F*'s mechanism for proof irrelevance.

Universe of a refinement type: The universe of a refinement type `x:t{p}` is the universe of `t`.

Since the universe of a refinement type does not depend on `p`, it enables a limited form of impredicativity, and we can define the following type (summarized here from the F* standard library `FStar.Squash`):

```
let squash (p:Type u#p) : Type u#0 = _:unit { p }
let return_squash (p:Type u#p) (x:p) : squash p = ()
```

This is a lot like the `lower` and `inject` assumptions that we saw in the previous section, but, importantly, there is no `project` operation to invert an `inject`. In fact, `FStar.Squash` proves that `squash p` is proof irrelevant, meaning that all proofs of `squash p` are equal.

```
val proof_irrelevance (p: Type u#p) (x y: squash p) : squash (x == y)
```

FStar.Squash does provide a limited way to manipulate a proof of p given a squash p, using the combinator bind_squash shown below, which states that if f can build a proof squash b from any proof of a, then it can do so from the one and only proof of a that is witnessed by x:squash a.

```
val bind_squash (#a: Type u#a) (#b: Type u#b) (x: squash a) (f: (a -> squash b)) :␣
↪squash b
```

It is important that bind_squash return a squash b, maintaining the proof-irrelevance of the squash type. Otherwise, if one could extract a proof of a from squash a, we would be perilously close to the unsound project axiom which enables paradoxes.

This restriction is similar to Coq's restriction on its Prop type, forbidding functions that match on Prop to return results outside Prop.

The F* type prop (which we saw first *here*) is defined primitively as type of all squashed types, i.e., the only types in prop are types of the form squash p; or, equivalently, every type t : prop, is a subtype of unit. Being the type of a class of types, prop in F* lives in u#1

```
let _ : Type u#1 = prop
```

However, prop still offers a form of impredicativity, e.g., you can quantify over all prop while remaining in prop.

```
let _ : Type u#1 = a:prop -> a
let _ : Type u#0 = squash (a:prop -> a)
let _ : prop = forall (a:prop). a
let _ : prop = exists (a:prop). a
```

- The first line above shows that, as usual, an arrow type is in a universe that is the maximum of the universes of its argument and result types. In this case, since it has an argument prop : Type u#1 the arrow itself is in u#1.
- The second line shows that by squashing the arrow type, we can bring it back to u#0
- The third line shows the more customary way of doing this in F*, where forall (a:prop). a is just syntactic sugar for squash (a:prop -> a). Since this is a squash type, not only does it live in Type u#0, it is itself a prop.
- The fourth line shows that the same is true for exists.

21.5 Raising universes and the lack of cumulativity

In some type theories, notably in Coq, the universe system is *cumulative*, meaning that Type u#i : Type u#(max (i + i) j); or, that Type u#i inhabits all universes greater than i. In contrast, in F*, as in Agda and Lean, Type u#i : Type u#(i + 1), i.e., a type resides only in the universe immediately above it.

Cumulativity is a form of subtyping on universe levels, and it can be quite useful, enabling definitions at higher universes to be re-used for all lower universes. However, systems that mix universe polymorphism with cumulativity are quite tricky, and indeed, it was only recently that Coq offered both universe polymorphism and cumulativity.

Lacking cumulativity, F* provides a library FStar.Universe that enables lifting a term from one universe to a higher one. We summarize it here:

```
val raise_t ([@@@ strictly_positive] t : Type u#a) : Type u#(max a b)

val raise_val (#a:Type u#a) (x:a) : raise_t u#a u#b a

```

(continues on next page)

(continued from previous page)

```
val downgrade_val (#a:Type u#a) (x:raise_t u#a u#b a) : a

val downgrade_val_raise_val (#a: Type u#a) (x: a)
  : Lemma (downgrade_val u#a u#b (raise_val x) == x)

val raise_val_downgrade_val (#a: Type u#a) (x: raise_t u#a u#b a)
  : Lemma (raise_val (downgrade_val x) == x)
```

The type `raise_t t` is strictly positive in `t` and raises `t` from `u#a` to `u#(max a b)`. `raise_val` and `downgrade_val` are mutually inverse functions between `t` and `raise_t t`.

This signature is similar in structure to the unsound signature for `lower, inject, project` that we use to exhibit Russell's paradox. However, crucially, the universe levels in `raise_t` ensure that the universe levels *increase*, preventing any violation of universe stratification.

In fact, this signature is readily implemented in F*, as shown below, where the universe annotation on `raise_t` explicitly defines the type in a higher universe `u#(max a b)` rather than in its minimum universe `u#a`.

```
noeq
type raise_t (a : Type u#a) : Type u#(max a b) =
  | Ret : a -> raise_t a

let raise_val #a x = Ret x
let downgrade_val #a x = match x with Ret x0 -> x0
let downgrade_val_raise_val #a x = ()
let raise_val_downgrade_val #a x = ()
```

21.6 Tips for working with universes

Whenever you write `Type` in F*, you are implicitly writing `Type u#?x`, where `?x` is a universe *metavariable* left for F* to infer. When left implicit, this means that F* may sometimes infer universes for your definition that are not what you expect—they may be too general or not general enough. We conclude this section with a few tips to detect and fix such problems.

- If you see puzzling error messages, enable the following pragma:

  ```
  #push-options "--print_implicits --print_universes"
  ```

  This will cause F* to print larger terms in error messages, which you usually do not want, except when you are confronted with error messages of the form "expected type t; got type t".

- Aside from the built-in constants `Type u#a`, the `->` type constructor, and the refinement type former, the only universe polymorphic F* terms are top-level definitions. That is, while you can define `i` at the top-level and use it polymorphically:

  ```
  let i (#a:Type) (x:a) = x
  let _ = i u#0 0, i u#1 nat, i u#2 (Type u#0)
  ```

  You cannot do the same in a non-top-level scope:

  ```
  let no_universe_poly_locally () =
    let i (#a:Type) (x:a) = x in
  ```

(continues on next page)

(continued from previous page)

```
  let _ = i u#0 0, i u#1 nat, i u#2 (Type u#0) in
  ()
```

Of course, non-universe-polymorphic definitions work at all scopes, e.g., here, the `i` is polymorphic in all types at universe `u#0`.

```
let type_poly_locally () =
  let i (#a:Type) (x:a) = x in
  let _ = i #unit (), i #bool true, i #nat 0 in
  ()
```

- If you write a `val f : t` declaration for `f`, F* will compute the most general universe for the type `t` independently of the `let f = e` or `type f =` definition.

  A simple example of this behavior is the following. Say, you declare `tup2` as below.

  ```
  val tup2 (a:Type) (b:Type) : Type
  ```

  Seeing this declaration F* infers `val tup2 (a:Type u#a) (b:Type u#b) : Type u#c`, computing the most general type for `tup2`.

  If you now try to define `tup2`,

  ```
  let tup2 a b = a & b
  ```

  F* complains with the following error (with `--print_universes` on):

  ```
  Type u#(max uu___43588 uu___43589) is not a subtype of the expected type Type u#uu__
  ↪_43590
  ```

  Meaning that the inferred type for the definition of `tup2 a b` is `Type u#(max a b)`, which is of course not the same as `Type u#c`, and, sadly, the auto-generated fresh names in the error message don't make your life any easier.

  The reason for this is that one can write a `val f : t` in a context where a definition for `f` may never appear, in which case F* has to compute some universes for `t`—it chooses the most general universe, though if you do try to implement `f` you may find that the most general universe is too general.

  A good rule of thumb is the following:

  - Do not write a `val` declaration for a term, unless you are writing an *interface*. Instead, directly write a `let` or `type` definition and annotate it with the type you expect it to have—this will lead to fewer surprises. For example, instead of separating the `val tup2` from `let tup2` just write them together, as shown below, and F* infers the correct universes.

    ```
    let tuple2 (a:Type) (b:Type) : Type = a & b
    ```

  - If you must write a `val f : t`, because, say, the type `t` is huge, or because you are writing an interface, it's a good idea to be explicit about universes, so that when defining `f`, you know exactly how general you have to be in terms of universes; and, conversely, users of `f` know exactly how much universe polymorphism they are getting. For example:

    ```
    val tup2_again (a:Type u#a) (b:Type u#b) : Type u#(max a b)
    let tup2_again a b = a & b
    ```

- When defining an inductive type, prefer using parameters over indexes, since usually type parameters lead to types in lower universes. For example, one might think to define lists this way:

```
noeq
type list_alt : Type u#a -> Type u#(a + 1) =
  | NilAlt: a:Type -> list_alt a
  | ConsAlt: a:Type -> hd:a -> tl:list_alt a -> list_alt a
```

Although semantically equivalent to the standard list

```
type list (a:Type u#a) : Type u#a  =
 | Nil : list a
 | Cons : hd:a -> tl:list a -> list a
```

`list_alt` produces a type in `u#(a + 1)`, since both `NilAlt` and `ConsAlt` have fields of type `a:Type u#a`. So, unless the index of your type varies among the constructors, use a parameter instead of an index.

That said, recall that it's the fields of the constructors of the inductive type that count. You can index your type by a type in any universe and it doesn't influence the result type. Here's an artificial example.

```
type t : Type u#100 -> Type u#0 =
  | T : unit -> t (FStar.Universe.raise_t unit)
```

Part IV

Modularity With Interfaces and Typeclasses

In this section, we'll learn about two abstraction techniques used to structure larger F* developments: *interfaces* and *typeclasses*.

Interfaces: An F* module `M` (in a file `M.fst`) can be paired with an interface (in a file `M.fsti`). A module's interface is a subset of the module's declarations and definitions. Another module `Client` that uses `M` can only make use of the part of `M` revealed in its interface—the rest of `M` is invisible to `Client`. As such, interfaces provide an abstraction mechanism, enabling the development of `Client` to be independent of any interface-respecting implementation of `M`.

Unlike module systems in other ML-like languages (which provide more advanced features like signatures, functors, and first-class modules), F*'s module system is relatively simple.

- A module can have at most one interface.
- An interface can have at most one implementation.
- A module lacking an interface reveals all its details to clients.
- An interface lacking an implementation can be seen as an assumption or an axiom.

Typeclasses: Typeclasses cater to more complex abstraction patterns, e.g., where an interface may have several implementations. Many other languages, ranging from Haskell to Rust, support typeclasses that are similar in spirit to what F* also provides.

Typeclasses in F* are actually defined mostly by a "user-space" metaprogam (relying on general support for metaprogramming in Meta-F*), making them very flexible (e.g., multi-parameter classes, overlapping instances, etc. are easily supported).

That said, typeclasses are a relatively recent addition to the language and most of F*'s standard library does not yet use typeclasses. As such, they are somewhat less mature than interfaces and some features require encodings (e.g., typeclass inheritance), rather than being supported with built-in syntax.

Thanks especially to Guido Martinez, who designed and implemented most of F*'s typeclass system.

CHAPTER

TWENTYTWO

INTERFACES

Look through the F* standard libary (in the ulib folder) and you'll find many files with .fsti extensions. Each of these is an interface file that pairs with a module implementation in a corresponding .fst file.

An interface (.fsti) is very similar to a module implementation (.fst): it can contain all the elements that a module can, including inductive type definitions; let and let rec definitions; val declarations; etc. However, unlike module implementations, an interface is allowed to declare a symbol val f : t without any corresponding definition of f. This makes f abstract to the rest of the interface and all client modules, i.e., f is simply assumed to have type t without any definition. The definition of f is provided in the .fst file and checked to have type t, ensuring that a client module's assumption of f:t is backed by a suitable definition.

To see how interfaces work, we'll look at the design of the **bounded integer** modules FStar.UInt32, FStar.UInt64, and the like, building our own simplified versions by way of illustration.

22.1 Bounded Integers

The F* primitive type int is an unbounded mathematical integer. When compiling a program to, say, OCaml, int is compiled to a "big integer", implemented by OCaml's ZArith package. However, the int type can be inefficient and in some scenarios (e.g., when compiling F* to C) one may want to work with bounded integers that can always be represented in machine word. FStar.UInt32.t and FStar.UInt64.t are types from the standard library whose values can always be represented as 32- and 64-bit unsigned integers, respectively.

Arithmetic operations on 32-bit unsigned integers (like addition) are interpreted modulo 2^32. However, for many applications, one wants to program in a discipline that ensures that there is no unintentional arithmetic overflow, i.e., we'd like to use bounded integers for efficiency, and by proving that their operations don't overflow we can reason about bounded integer terms without using modular arithmetic.

Note: Although we don't discuss them here, F*'s libraries also provide signed integer types that can be compiled to the corresponding signed integters in C. Avoiding overflow on signed integer arithmetic is not just a matter of ease of reasoning, since signed integer overflow is undefined behavior in C.

22.1.1 Interface: UInt32.fsti

The interface UInt32 begins like any module with the module's name. Although this could be inferred from the name of the file (UInt32.fsti, in this case), F* requires the name to be explicit.

```
module UInt32

val t : eqtype
```

UInt32 provides one abstract type val t : eqtype, the type of our bounded integer. Its type says that it supports decidable equality, but no definition of t is revealed in the interface.

The operations on t are specified in terms of a logical model that relates t to bounded mathematical integers, in particular u32_nat, a natural number less than pow2 32.

```
let n = 32
let min : nat = 0
let max : nat = pow2 n - 1
let u32_nat = n:nat{ n <= max }
```

Note: Unlike interfaces in languages like OCaml, interfaces in F* *can* include let and let rec definitions. As we see in UInt32, these definitions are often useful for giving precise specifications to the other operations whose signatures appear in the interface.

To relate our abstract type t to u32_nat, the interface provides two coercions v and u that go back and forth between t and u32_nat. The lemma signatures vu_inv and uv_inv require v and u to be mutually inverse, meaning that t and u32_nat are in 1-1 correspondence.

```
val v (x:t) : u32_nat
val u (x:u32_nat) : t

val uv_inv (x : t) : Lemma (u (v x) == x)
val vu_inv (x : u32_nat) : Lemma (v (u x) == x)
```

Modular addition and subtraction

Addition and subtraction on t values are defined modulo pow2 32. This is specified by the signatures of add_mod and sub_mod below.

```
(** Addition modulo [2^n]

    Unsigned machine integers can always be added, but the postcondition is now
    in terms of addition modulo [2^n] on mathematical integers *)
val add_mod (a:t) (b:t)
  : y:t { v y = (v a + v b) % pow2 n }
```

```
(** Subtraction modulo [2^n]

    Unsigned machine integers can always be subtracted, but the postcondition is now
    in terms of subtraction modulo [2^n] on mathematical integers *)
val sub_mod (a:t) (b:t)
  : y:t { v y = (v a - v b) % pow2 n }
```

Bounds-checked addition and subtraction

Although precise, the types of add_mod and sub_mod aren't always easy to reason with. For example, proving that add_mod (u 2) (u 3) == u 5 requires reasoning about modular arithmetic—for constants like 2, 3, and 5 this is easy enough, but proofs about modular arithmetic over symbolic values will, in general, involve reasoning about non-linear arithmetic, which is difficult to automate even with an SMT solver. Besides, in many safety critical software systems, one often prefers to avoid integer overflow altogether.

So, the UInt32 interface also provides two additional operations, add and sub, whose specification enables two t values to be added (resp. subtracted) only if there is no overflow (or underflow).

First, we define an auxiliary predicate fits to state when an operation does not overflow or underflow.

```
let fits (op: int -> int -> int)
         (x y : t)
  = min <= op (v x) (v y) /\
    op (v x) (v y) <= max
```

Then, we use fits to restrict the domain of add and sub and the type ensures that the result is the sum (resp. difference) of the arguments, without need for any modular arithmetic.

```
(** Bounds-respecting addition

    The precondition enforces that the sum does not overflow,
    expressing the bound as an addition on mathematical integers *)
val add (a:t) (b:t { fits (+) a b })
  : y:t{ v y == v a + v b }
```

```
(** Bounds-respecting subtraction

    The precondition enforces that the difference does not underflow,
    expressing the bound as an subtraction on mathematical integers *)
val sub (a:t) (b:t { fits (fun x y -> x - y) a b })
  : y:t{ v y == v a - v b }
```

Note: Although the addition operator can be used as a first-class function with the notation (+), the same does not work for subtraction, since (-) resolves to unary integer negation, rather than subtraction—so, we write fun x y -> x - y.

Comparison

Finally, the interface also provides a comparison operator lt, as specified below:

```
(** Less than *)
val lt (a:t) (b:t)
  : r:bool { r <==> v a < v b }
```

22.1.2 Implementation: UInt32.fst

An implementation of UInt32 must provide definitions for all the val declarations in the UInt32 interface, starting with a representation for the abstract type t.

There are multiple possible representations for t, however the point of the interface is to isolate client modules from these implementation choices.

Perhaps the easiest implementation is to represent t as a u32_nat itself. This makes proving the correspondence between t and its logical model almost trivial.

```
module UInt32

let t = n:nat { n <= pow2 32 - 1}

let v (x:t) = x
let u (x:u32_nat) = x

let uv_inv x = ()
let vu_inv x = ()

let add_mod a b = (a + b) % pow2 32
let sub_mod a b = (a - b) % pow2 32

let add a b = a + b
let sub a b = a - b

let lt (a:t) (b:t) = v a < v b
```

Another choice may be to represent t as a 32-bit vector. This is a bit harder and proving that it is correct with respect to the interface requires handling some interactions between Z3's theory of bit vectors and uninterpreted functions, which we handle with a tactic. This is quite advanced, and we have yet to cover F*'s support for tactics, but we show the code below for reference.

```
open FStar.BV
open FStar.Tactics

let t = bv_t 32
let v (x:t) = bv2int x
let u x = int2bv x

let sym (#a:Type) (x y:a)
  : Lemma (requires x == y)
          (ensures y == x)
  = ()

let dec_eq (#a:eqtype) (x y:a)
  : Lemma (requires x = y)
          (ensures x == y)
  = ()

let uv_inv (x:t)
  = assert (u (v x) == x)
        by (mapply (`sym);
            mapply (`dec_eq);
```

(continues on next page)

(continued from previous page)

```
            mapply (`inverse_vec_lemma))

let ty (x y:u32_nat)
  : Lemma (requires eq2 #(FStar.UInt.uint_t 32) x y)
          (ensures x == y)
  = ()

let vu_inv (x:u32_nat)
  = assert (v (u x) == x)
        by (mapply (`ty);
            mapply (`sym);
            mapply (`dec_eq);
            mapply (`inverse_num_lemma))

let add_mod a b =
  let y = bvadd #32 a b in
  assert (y == u (FStar.UInt.add_mod #32 (v a) (v b)))
     by  (mapply (`sym);
          mapply (`int2bv_add);
          pointwise
              (fun _ -> try mapply (`uv_inv) with | _ -> trefl());
          trefl());
  vu_inv ((FStar.UInt.add_mod #32 (v a) (v b)));
  y

let sub_mod a b =
  let y = bvsub #32 a b in
  assert (y == u (FStar.UInt.sub_mod #32 (v a) (v b)))
     by
         (mapply (`sym);
          mapply (`int2bv_sub);
          pointwise
              (fun _ -> try mapply (`uv_inv) with | _ -> trefl());
          trefl());
  vu_inv ((FStar.UInt.sub_mod #32 (v a) (v b)));
  y

let add a b =
  let y = add_mod a b in
  FStar.Math.Lemmas.modulo_lemma (v a + v b) (pow2 32);
  y

let sub a b =
  let y = sub_mod a b in
  FStar.Math.Lemmas.modulo_lemma (v a - v b) (pow2 32);
  y

let lt (a:t) (b:t) = v a < v b
```

Although both implementations correctly satisfy the `UInt32` interface, F* requires the user to pick one. Unlike module systems in some other ML-like languages, where interfaces are first-class entities which many modules can implement, in F* an interface can have at most one implementation. For interfaces with multiple implementations, one must use typeclasses.

22.1.3 Interleaving: A Quirk

The current F* implementation views an interface and its implementation as two partially implemented halves of a module. When checking that an implementation is a correct implementation of an interface, F* attempts to combine the two halves into a complete module before typechecking it. It does this by trying to *interleave* the top-level elements of the interface and implementation, preserving their relative order.

This implementation strategy is far from optimal in various ways and a relic from a time when F*'s implementation did not support separate compilation. This implementation strategy is likely to change in the future (see this issue for details).

Meanwhile, the main thing to keep in mind when implementing interfaces is the following:

- The order of definitions in an implementation much match the order of `val` declarations in the interface. E.g., if the interface contains `val f : tf` followed by `val g : tg`, then the implementation of `f` must precede the implementation of `g`.

Also, remember that if you are writing `val` declarations in an interface, it is a good idea to be explicit about universe levels. See *here for more discussion*.

Other issues with interleaving that may help in debugging compiler errors with interfaces:

- Issue 2020
- Issue 1770
- Issue 959

22.1.4 Comparison with machine integers in the F* library

F*'s standard library includes `FStar.UInt32`, whose interface is similar, though more extensive than the `UInt32` shown in this chapter. For example, `FStar.UInt32` also includes multiplication, division, modulus, bitwise logical operations, etc.

The implementation of `FStar.UInt32` chooses a representation for `FStar.UInt32.t` that is similar to `u32_nat`, though the F* compiler has special knowledge of this module and treats `FStar.UInt32.t` as a primitive type and compiles it and its operations in a platform-specific way to machine integers. The implementation of `FStar.UInt32` serves only to prove that its interface is logically consistent by providing a model in terms of bounded natural numbers.

The library also provides several other unsigned machine integer types in addition to `FStar.UInt32`, including `FStar.UInt8`, `FStar.UInt16`, and `FStar.UInt64`. F* also has several signed machine integer types.

All of these modules are very similar, but not being first-class entities in the language, there is no way to define a general interface that is instantiated by all these modules. In fact, all these variants are generated by a script from a common template.

Although interfaces are well-suited to simple patterns of information hiding and modular structure, as we'll learn next, typeclasses are more powerful and enable more generic solutions, though sometimes requiring the use of higher-order code.

CHAPTER

TWENTYTHREE

TYPECLASSES

Consider writing a program using bounded unsigned integers while being generic in the actual bounded integer type, E.g., a function that sums a list of bounded integers while checking for overflow, applicable to both `UInt32` and `UInt64`. Since F*'s interfaces are not first-class, one can't easily write a program that abstracts over those interfaces. Typeclasses can help.

Some background reading on typeclasses:

- Phil Wadler and Stephen Blott introduced the idea in 1989 in a paper titled "How to make ad hoc polymorphism less ad hoc." Their work, with many extensions over the years, is the basis of Haskell's typeclasses.
- A tutorial on typeclasses in the Coq proof assistant is available here.
- Typeclasses are used heavily in the Lean proof assistant to structure its math library.

23.1 Printable

A typeclass associates a set of *methods* to a tuple of types, corresponding to the operations that can be performed using those types.

For instance, some types may support an operation that enables them to be printed as strings. A typeclass `printable (a:Type)` represent the class of all types that support a `to_string :  a -> string` operation.

```
class printable (a:Type) =
{
  to_string : a -> string
}
```

The keyword `class` introduces a new typeclass, defined as a *record type* with each method represented as a field of the record.

To define instances of a typeclass, one uses the `instance` keyword, as shown below.

```
instance printable_bool : printable bool =
{
  to_string = Prims.string_of_bool
}

instance printable_int : printable int =
{
  to_string = Prims.string_of_int
}
```

The notation `instance printable_bool : printable bool = e` states that the value `e` is a record value of type `printable bool`, and just as with a `let`-binding, the term `e` is bound to the top-level name `printable_bool`.

The convenience of typeclasses is that having defined a class, the typeclass method is automatically overloaded for all instances of the class, and the type inference algorithm finds the suitable instance to use. This is the original motivation of typeclasses—to provide a principled approach to operator overloading.

For instance, we can now write `printb` and `printi` and use `to_string` to print both booleans and integers, since we shown that they are instances of the class `printable`.

```
let printb (x:bool) = to_string x
let printi (x:int) = to_string x
```

Instances need not be only for base types. For example, all lists are printable so long as their elements are, and this is captured by what follows.

```
instance printable_list (#a:Type) (x:printable a) : printable (list a) =
{
  to_string = (fun l -> "[" ^ FStar.String.concat "; " (List.Tot.map to_string l) ^ "]")
}
```

That is, `printable_list` constructs a `to_string` method of type `list a -> string` by mapping the `to_string` method of the `printable a` instance over the list. And now we can use `to_string` with lists of booleans and integers too.

```
let printis (l:list int) = to_string l
let printbs (l:list bool) = to_string l
```

There's nothing particularly specific about the ground instances `printable bool` and `printable int`. It's possible to write programs that are polymorphic in printable types. For example, here's a function `print_any_list` that is explicitly parameterized by a `printable a`—one can call it by passing in the instance that one wishes to use explicitly:

```
let print_any_list_explicit #a ( _ : printable a ) (l:list a) = to_string l
let _ = print_any_list_explicit printable_int [1;2;3]
```

However, we can do better and have the compiler figure out which instance we intend to use by using a bit of special syntax for a typeclass parameter, as shown below.

```
let print_any_list #a {| _ : printable a |} (l:list a) = to_string l
let _ex1 = print_any_list [[1;2;3]]
let _ex2 = print_any_list #_ #(printable_list printable_int) [[1;2;3]]
```

The parameter `{| _ : printable a |}` indicates an implicit argument that, at each call site, is to be computed by the compiler by finding a suitable typeclass instance derivable from the instances in scope. In the first example above, F* figures out that the instance needed is `printable_list printable_int : printable (list int)`. Note, you can always pass the typeclass instance you want explicitly, if you really want to, as the second example `_ex2` above shows.

In many cases, the implicit typeclass argument need not be named, in which case one can just omit the name and write:

```
let print_any_list_alt #a {| printable a |} (l:list a) = to_string l
```

23.1.1 Under the hood

When defining a `class`, F* automatically generates generic functions corresponding to the methods of the class. For instance, in the case of `printable`, F* generates:

```
let to_string #a {| i : printable a |} (x:a) = i.to_string x
```

Having this in scope overloads `to_string` for all instance of the `printable` class. In the implementation of `to_string`, we use the instance `i` (just a record, sometimes called a dictionary in the typeclass literature) and project its `to_string` field and apply it to `x`.

Defining an `instance p x1..xn : t = e` is just like an ordinary let binding `let p x1..xn : t = e`, however the `instance` keyword instructs F*'s type inference algorithm to consider using `p` when trying to instantiate implicit arguments for typeclass instances.

For example, at the call site `to_string (x:bool)`, having unified the implicit type arguments `a` with `bool`, what remains is to find an instance of `printable bool`. F* looks through the current context for all variable bindings in the local scope, and `instance` declarations in the top-level scope, for a instance of `printable bool`, taking the first one it is able to construct.

The resolution procedure for `to_string [[1;2;3]]` is a bit more interesting, since we need to find an instance `printable (list int)`, although no such ground instance exists. However, the typeclass resolution procedure finds the `printable_list` instance function, whose result type `printable (list a)` matches the goal `printable (list int)`, provided `a = int`. The resolution procedure then spawns a sub-goal `printable int`, which it solves easily and completes the derivation of `printable (list int)`.

This backwards, goal-directed search for typeclass resolution is a kind of logic programming. An interesting implementation detail is that most of the typeclass machinery is defined as a metaprogran in `FStar.Tactics.Typeclasses`, outside of the core of F*'s compiler. As such, the behavior of typeclass resolution is entirely user-customizable, simply by revising the metaprogram in use. Some details about how this works can be found in a paper on Meta F*.

23.1.2 Exercises

Define instances of `printable` for `string`, `a & b`, `option a`, and `either a b`. Check that you can write `to_string [Inl (0, 1); Inr (Inl (Some true)); Inr (Inr "hello") ]` and have F* infer the typeclass instance needed.

Also write the typeclass instances you need explicitly, just to check that you understand how things work. This is exercise should also convey that typeclasses do not increase the expressive power in any way—whatever is expressible with typeclasses, is also expressible by explicitly passing records that contain the operations needed on specific type parameters. However, expliciting passing this operations can quickly become overwhelming—typeclass inference keeps this complexity in check and makes it possible to build programs in an generic, abstract style without too much pain.

This exercise file provides the definitions you need.

Answer

```
instance printable_string : printable string =
{
  to_string = fun x -> "\"" ^ x ^ "\""
}

instance printable_pair #a #b {| printable a |} {| printable b |} : printable (a & b) =
{
  to_string = (fun (x, y) -> "(" ^ to_string x ^ ", " ^ to_string y ^ ")")
}
```

(continues on next page)

(continued from previous page)

```
instance printable_option #a {| printable a |} : printable (option a) =
{
  to_string = (function None -> "None" | Some x -> "(Some " ^ to_string x ^ ")")
}

instance printable_either #a #b {| printable a |} {| printable b |} : printable (either␣
↪a b) =
{
  to_string = (function Inl x -> "(Inl " ^ to_string x ^ ")" | Inr x -> "(Inr " ^ to_
↪string x ^ ")")
}

let _ = to_string [Inl (0, 1); Inr (Inl (Some true)); Inr (Inr "hello") ]

//You can always pass the typeclass instance you want explicitly, if you really want to
//typeclass resolution really saves you from LOT of typing!
let _ = to_string #_
             #(printable_list
                (printable_either #_ #_ #(printable_pair #_ #_ #printable_int #printable_
↪int)
                                          #(printable_either #_ #_ #(printable_option #_
↪#printable_bool)
                                                                   #(printable_string))))
             [Inl (0, 1); Inr (Inl (Some true)); Inr (Inr "hello")]
```

23.2 Bounded Unsigned Integers

The printable typeclass is fairly standard and can be defined in almost any language that supports typeclasses. We now turn to a typeclass that leverages F*'s dependent types by generalizing the interface of bounded unsigned integers that we developed in a *previous chapter*.

A type a is in the class bounded_unsigned_int, when it admits:

- An element bound : a, representing the maximum value
- A pair of functions from_nat and to_nat that form a bijection between a and natural numbers less than to_nat bound

This is captured by the class below:

```
class bounded_unsigned_int (a:Type) = {
   bound      : a;
   as_nat     : a -> nat;
   from_nat   : (x:nat { x <= as_nat bound }) -> a;
   [@@@FStar.Tactics.Typeclasses.no_method]
   properties : squash (
     (forall (x:a). as_nat x <= as_nat bound) /\ // the bound is really an upper bound
     (forall (x:a). from_nat (as_nat x) == x) /\ //from_nat/as_nat form a bijection
     (forall (x:nat{ x <= as_nat bound}). as_nat (from_nat x) == x)
```

(continues on next page)

(continued from previous page)

```
   )
}
```

Note: The attribute `FStar.Tactics.Typeclasses.no_method` on the `properties` field instructs F* to not generate a typeclass method for this field. This is useful here, since we don't really want to overload the name `properties` as an operator over all instances bound the class. It's often convenient to simply `open FStar.Tactics.Typeclasses` when using typeclasses, or to use a module abbreviation like `module TC = FStar.Tactics.Typeclasses` so that you don't have to use a fully qualified name for `no_method`.

For all `bounded_unsigned_ints`, one can define a generic `fits` predicate, corresponding to the bounds check condition that we introduced in the `UInt32` interface.

```
let fits #a {| bounded_unsigned_int a |}
            (op: int -> int -> int)
            (x y:a)
  : prop
  = 0 <= op (as_nat x) (as_nat y) /\
    op (as_nat x) (as_nat y) <= as_nat #a bound
```

Likewise, the predicate `related_ops` defines when an operation `bop` on bounded integers is equivalent to an operation `iop` on mathematical integers.

```
let related_ops #a {| bounded_unsigned_int a |}
                (iop: int -> int -> int)
                (bop: (x:a -> y:a { fits iop x y } -> a))
  = forall (x y:a).  fits iop x y ==> as_nat (bop x y) = as_nat x `iop` as_nat y
```

23.2.1 Typeclass Inheritance

Our `bounded_unsigned_int` a class just showed that `a` is in a bijection with natural numbers below some bound. Now, we can define a separate class, extending `bounded_unsigned_int` with the operations we want, like addition, subtraction, etc.

```
class bounded_unsigned_int_ops (a:Type) = {
   [@@@TC.no_method]
   base       : bounded_unsigned_int a;
   add        : (x:a -> y:a { fits ( + ) x y } -> a);
   sub        : (x:a -> y:a { fits op_Subtraction x y } -> a);
   lt         : (a -> a -> bool);
   [@@@TC.no_method]
   properties : squash (
     related_ops ( + ) add /\
     related_ops op_Subtraction sub /\
     (forall (x y:a). lt x y <==> as_nat x < as_nat y) /\ // lt is related to <
     (forall (x:a). fits op_Subtraction bound x) //subtracting from the maximum element␣
↪never triggers underflow
   )
}
```

The class above makes use of *typeclass inheritance*. The `base` field stores an instance of the base class `bounded_unsigned_int`, while the remaining fields extend it with:

- add: a bounded addition operation
- sub: a bounded subtraction operation
- lt : a comparison function
- properties, which show that
  - add is related to integer addition +
  - sub is related to integer subtraction -
  - lt is related to <
  - and that sub bound x is always safe

Typeclass inheritance in the form of additional fields like base is completely flexible, e.g., multiple inheritance is permissible (though, as we'll see below, should be used with care, to prevent surprises).

Treating an instance of a class as an instance of one its base classes is easily coded as instance-generating function. The code below says that an instance from bounded_unsigned_int a can be derived from an instance of d : bounded_unsigned_int_ops a just by projecting its base field.

```
instance ops_base #a {| d : bounded_unsigned_int_ops a |}
  : bounded_unsigned_int a
  = d.base
```

23.2.2 Infix Operators

F* does not allows the fields of a record to be named using infix operator symbols. This will likely change in the future. For now, to use custom operations with infix notation for typeclass methods, one has to define them by hand:

```
let ( +^ ) #a {| bounded_unsigned_int_ops a |}
           (x : a)
           (y : a { fits ( + ) x y })
  : a
  = add x y

let ( -^ ) #a {| bounded_unsigned_int_ops a |}
           (x : a)
           (y : a { fits op_Subtraction x y })
  : a
  = sub x y

let ( <^ ) #a {| bounded_unsigned_int_ops a |}
           (x : a)
           (y : a)
  : bool
  = lt x y
```

23.2.3 Derived Instances

We've already seen how typeclass inheritance allows a class to induce an instance of its base class(es). However, not all derived instances are due to explicit inheritance—some instances can be *computed* from others.

For example, here's a class eq for types that support decidable equality.

```
class eq (a:Type) = {
  eq_op: a -> a -> bool;

  [@@@TC.no_method]
  properties : squash (
    forall x y. eq_op x y <==> x == y
  )
}

let ( =?= ) #a {| eq a |} (x y: a) = eq_op x y
```

We'll write x =?= y for an equality comparison method from this class, to not confuse it with F*'s built-in decidable equality (=) on eqtype.

Now, from an instance of bounded_unsigned_int_ops a we can compute an instance of eq a, since we have <^, a strict comparison operator that we know is equivalent to < on natural numbers. F*, from all the properties we have on bounded_unsigned_int_ops and its base class bounded_unsigned_int, can automatically prove that not (x <^ y) && not (y <^ x) is valid if and only if x == y. This instance of eq now also lets us easily implement a non-strict comparison operation on bounded unsigned ints.

```
instance bounded_unsigned_int_ops_eq #a {| bounded_unsigned_int_ops a |}
  : eq a
  = {
      eq_op = (fun x y -> not (x <^ y) && not (y <^ x));
      properties = ()
    }

let ( <=^ ) #a {| bounded_unsigned_int_ops a |} (x y : a)
  : bool
  = x <^ y || x =?= y
```

23.2.4 Ground Instances

We can easily provide ground instances of bounded_unsigned_int_ops for all the F* bounded unsigned int types—we show instances for FStar.UInt32.t and FStar.UInt64.t, where the proof of all the properties needed to construct the instances is automated.

```
module U32 = FStar.UInt32
module U64 = FStar.UInt64
instance u32_instance_base : bounded_unsigned_int U32.t =
  let open U32 in
  {
    bound    = 0xfffffffful;
    as_nat   = v;
    from_nat = uint_to_t;
    properties = ()
```

(continues on next page)

(continued from previous page)

```
}

instance u32_instance_ops : bounded_unsigned_int_ops U32.t =
  let open U32 in
  {
    base = u32_instance_base;
    add  = (fun x y -> add x y);
    sub  = (fun x y -> sub x y);
    lt   = (fun x y -> lt x y);
    properties = ()
  }


instance u64_instance_base : bounded_unsigned_int U64.t =
  let open U64 in
  {
    bound    = 0xffffffffffffffffuL;
    as_nat   = v;
    from_nat = uint_to_t;
    properties = ()
}

instance u64_instance_ops : bounded_unsigned_int_ops U64.t =
  let open U64 in
  {
    base = u64_instance_base;
    add  = (fun x y -> add x y);
    sub  = (fun x y -> sub x y);
    lt   = (fun x y -> lt x y);
    properties = ()
  }
```

And one can check that typeclass resolution works well on those ground instances.

```
let test32 (x:U32.t)
           (y:U32.t)
  = if x <=^ 0xfffffffful &&
       y <=^ 0xfffffffful
    then Some (x +^ y)
    else None

let test64 (x y:U64.t)
  = if x <=^ 0xfffffffful &&
       y <=^ 0xfffffffful
    then Some (x +^ y)
    else None
```

Finally, as promised at the start, we can write functions that are generic over all bounded unsigned integers, something we couldn't do with interfaces alone.

```
module L = FStar.List.Tot
let sum #a {| bounded_unsigned_int_ops a |}
```

(continues on next page)

(continued from previous page)

```
        (l:list a) (acc:a)
  : option a
  = L.fold_right
     (fun (x:a) (acc:option a) ->
       match acc with
       | None -> None
       | Some y ->
         if x <=^ bound -^ y
         then Some (x +^ y)
         else None)
     l
     (Some acc)

let testsum32 : U32.t = Some?.v (sum [0x01ul; 0x02ul; 0x03ul] 0x00ul)
let testsum64 : U64.t = Some?.v (sum [0x01uL; 0x02uL; 0x03uL] 0x00uL)
```

F* can prove that the bounds check in `sum` is sufficient to prove that the addition does not overflow, and further, that the two tests return `Some _` without failing due to overflow.

However, the proving that `Some? (sum [0x01ul; 0x02ul; 0x03ul] 0x00ul)` using the SMT solver alone can be expensive, since it requires repeated unfolding of the recursive function `sum`–such proofs are often more easily done using F*'s normalizer, as shown below—we saw the `assert_norm` construct in a *previous section*.

```
let testsum32' : U32.t =
  let x =
    sum #U32.t
        [0x01ul; 0x02ul; 0x03ul;
         0x01ul; 0x02ul; 0x03ul;
         0x01ul; 0x02ul; 0x03ul]
        0x00ul
  in
  assert_norm (Some? x /\ as_nat (Some?.v x) == 18);
  Some?.v x
```

Note: That said, by using dependently typed generic programming (which we saw a bit of *earlier*), it is possible to write programs that abstract over all machine integer types without using typeclasses. The F* library `FStar.Integers` shows how that works. Though, the typeclass approach shown here is more broadly applicable and extensible.

23.3 Dealing with Diamonds

One may be tempted to factor our `bounded_unsigned_int_ops` typeclass further, separating out each operation into a separate class. After all, it may be the case that some instances of `bounded_unsigned_int` types support only addition while others support only subtraction. However, when designing typeclass hierarchies one needs to be careful to not introduce coherence problems that result from various forms of multiple inheritance.

Here's a typeclass that captures only the subtraction operation, inheriting from a base class.

```
class subtractable_bounded_unsigned_int (a:Type) = {
   [@@@no_method]
   base   : bounded_unsigned_int a;
```

(continues on next page)

(continued from previous page)

```
   sub    : (x:a -> y:a { fits op_Subtraction x y } -> a);

   [@@@no_method]
   properties : squash (
     related_ops op_Subtraction sub /\
     (forall (x:a). fits op_Subtraction bound x)
   )
}

instance subtractable_base {| d : subtractable_bounded_unsigned_int 'a |}
  : bounded_unsigned_int 'a
  = d.base
```

And here's another typeclass that provides only the comparison operation, also inheriting from the base class.

```
class comparable_bounded_unsigned_int (a:Type) = {
   [@@@no_method]
   base   : bounded_unsigned_int a;
   comp   : a -> a -> bool;

   [@@@no_method]
   properties : squash (
     (forall (x y:a).{:pattern comp x y} comp x y <==> as_nat x < as_nat y)
   )
}

instance comparable_base {| d : comparable_bounded_unsigned_int 'a |}
  : bounded_unsigned_int 'a
  = d.base
```

However, now when writing programs that expect both subtractable and comparable integers, we end up with a coherence problem.

The sub operation fails to verify, with F* complaining that it cannot prove `fits op_Subtraction bound acc`, i.e., this sub may underflow.

```
[@@expect_failure [19]]
let try_sub #a {| s: subtractable_bounded_unsigned_int a|}
               {| c: comparable_bounded_unsigned_int a |}
            (acc:a)
  = bound `sub` acc
```

At first, one may be surprised, since the `s : subtractable_bounded_unsigned_int a` instance tells us that subtracting from the bound is always safe. However, the term bound is an overloaded (nullary) operator and there are two ways to resolve it: `s.base.bound` or `c.base.bound` and these two choices are not equivalent. In particular, from `s : subtractable_bounded_unsigned_int a`, we only know that ``s.base.bound `sub` acc`` is safe, not that ``c.base.bound `sub` acc`` is safe.

Slicing type typeclass hierarchy too finely can lead to such coherence problems that can be hard to diagnose. It's better to avoid them by construction, if at all possible. Alternatively, if such problems do arise, one can sometimes add additional preconditions to ensure that the multiple choices are actually equivalent. There are many ways to do this, ranging from indexing typeclasses by their base classes, to adding equality hypotheses—the equality hypothesis below is sufficient.

```
let try_sub {| s: subtractable_bounded_unsigned_int 'a |}
            {| c: comparable_bounded_unsigned_int 'a |}
            (acc:'a { s.base == c.base } )
  = bound `sub` acc
```

23.4 Overloading Monadic Syntax

We now look at some examples of typeclasses for *type functions*, in particular, typeclasses for functors and monads.

Note: If you're not familiar with monads, referring back to *A First Model of Computational Effects* may help.

In *a previous chapter*, we introduced syntactic sugar for monadic computations. In particular, F*'s syntax supports the following:

- Instead of writing `bind f (fun x -> e)` you can define a custom `let!`-operator and write `let! x = f in e`.
- And, instead of writing `bind f (fun _ -> e)` you can write `f ;! e`.

Now, if we can overload the symbol `bind` to work with any monad, then the syntactic sugar described above would work for all of them. This is accomplished as follows.

We define a typeclass `monad`, with two methods `return` and `( let! )`.

```
class monad (m:Type -> Type) =
{
   return : (#a:Type -> a -> m a);
   ( let! )  : (#a:Type -> #b:Type -> (f:m a) -> (g:(a -> m b)) -> m b);
}
```

Doing so introduces `return` and `( let! )` into scope at the following types:

```
let return #m {| d : monad m |} #a (x:a) : m a = d.return x
let ( let! ) #m {| d : monad m |} #a #b (f:m a) (g: a -> m b) : m b = d.bind f g
```

That is, we now have `( let! )` in scope at a type general enough to use with any monad instance.

Note: There is nothing specific about `let!`; F* allows you to add a suffix of operator characters to the `let`-token. See this file for more examples of monadic let operators

The type `st s` is a state monad parameterized by the state `s`, and `st s` is an instance of a `monad`.

```
let st (s:Type) (a:Type) = s -> a & s

instance st_monad s : monad (st s) =
{
   return = (fun #a (x:a) -> (fun s -> x, s));
   ( let! ) = (fun #a #b (f: st s a) (g: a -> st s b) (s0:s) ->
               let x, s1 = f s0 in
               g x s1);
}
```

With some basic actions `get` and `put` to read and write the state, we can implement `st` computations with a syntax similar to normal, direct-style code.

```
let get #s
  : st s s
  = fun s -> s, s

let put #s (x:s)
  : st s unit
  = fun _ -> (), x

let get_inc =
  let! x = get in
  return (x + 1)
```

Of course, we can also do proofs about our `st` computations: here's a simple proof that `get_put` is `noop`.

```
let get_put #s =
  let! x = get #s in
  put x

let noop #s : st s unit = return ()

let get_put_identity (s:Type)
  : Lemma (get_put #s `FStar.FunctionalExtensionality.feq` noop #s)
  = ()
```

Now, the nice thing is that since `( let! )` is monad polymorphic, we can define other monad instances and still use the syntactic sugar to build computations in those monads. Here's an example with the `option` monad, for computations that may fail.

```
instance opt_monad : monad option =
{
   return = (fun #a (x:a) -> Some x);
   ( let! ) = (fun #a #b (x:option a) (y: a -> option b) ->
             match x with
             | None -> None
             | Some a -> y a)
}

let raise #a : option a = None

let div (n m:int) =
  if m = 0 then raise
  else return (n / m)

let test_opt_monad (i j k:nat) =
  let! x = div i j in
  let! y = div i k in
  return (x + y)
```

23.4.1 Exercise

Define a typeclass for functors, type functions `m: Type -> Type` which support the operations `fmap : (a -> b) -> m a -> m b`.

Build instances of `functor` for a few basic types, e.g., `list`.

Derive an instance for functors from a monad, i.e., prove

`instance monad_functor #m {| monad m |} : functor m = admit()`

This file provides the definitions you need.

Answer

```
class functor (m:Type -> Type) =
{
  fmap: (#a:Type -> #b:Type -> (a -> b) -> m a -> m b);
}

let id (a:Type) = a

instance id_functor : functor id =
{
  fmap = (fun #a #b f -> f);
}

let test_id (a:Type) (f:a -> a) (x:id a) = fmap f x

instance option_functor : functor option =
{
  fmap = (fun #a #b (f:a -> b) (x:option a) ->
            match x with
            | None -> None
            | Some y -> Some (f y));
}

let test_option (f:int -> bool) (x:option int) = fmap f x

instance monad_functor #m (d:monad m) : functor m =
{
  fmap = (fun #a #b (f:a -> b) (c:m a) -> let! x = c in return (f x))
}
```

23.5 Beyond Monads with Let Operators

Many monad-like structures have been proposed to structure effectful computations. Each of these structures can be captured as a typeclass and used with F*'s syntactic sugar for let operators.

As an example, we look at *graded monads*, a construction studied by Shin-Ya Katsumata and others, in several papers. This example illustrates the flexibility of typeclasses, including typeclasses for types that themselves are indexed by other typeclasses.

The main idea of a graded monad is to index a monad with a monoid, where the monoid index characterizes some property of interest of the monadic computation.

A monoid is a typeclass for an algebraic structure with a single associative binary operation and a unit element for that operation. A simple instance of a monoid is the natural numbers with addition and the unit being `0`.

```
class monoid (a:Type) =
{
   op   : a -> a -> a;
   one  : a;
   properties: squash (
     (forall (x:a). op one x == x /\ op x one == x) /\
     (forall (x y z:a). op x (op y z) == op (op x y) z)
   );
}

instance monoid_nat_plus : monoid nat =
{
  op = (fun (x y:nat) -> x + y);
  one = 0;
  properties = ()
}
```

A graded monad is a type constructor `m` indexed by a monoid as described by the class below. In other words, `m` is equipped with two operations:

- a `return`, similar to the `return` of a monad, but whose index is the unit element of the monoid
- a `( let+ )`, similar to the `( let! )` of a monad, but whose action on the indexes corresponds to the binary operator of the indexing monoid.

```
class graded_monad (#index:Type) {| monoid index |}
                   (m : index -> Type -> Type) =
{
  return : #a:Type -> x:a -> m one a;

   ( let+ )   : #a:Type -> #b:Type -> #ia:index -> #ib:index ->
           m ia a ->
           (a -> m ib b) ->
           m (op ia ib) b

}
```

With this class, we have overloaded `( let+ )` to work with all graded monads. For instance, here's a graded state monad, `count_st` whose index counts the number of `put` operations.

```
let count_st (s:Type) (count:nat) (a:Type) = s -> a & s & z:nat{z==count}

let count_return (#s:Type) (#a:Type) (x:a) : count_st s one a = fun s -> x, s, one #nat

let count_bind (#s:Type) (#a:Type) (#b:Type) (#ia:nat) (#ib:nat)
               (f:count_st s ia a)
               (g:(a -> count_st s ib b))
  : count_st s (op ia ib) b
  = fun s -> let x, s, n = f s in
```

(continues on next page)

(continued from previous page)

```
          let y, s', m = g x s in
          y, s', op #nat n m

instance count_st_graded (s:Type) : graded_monad (count_st s) =
{
  return = count_return #s;
  ( let+ ) = count_bind #s;
}

// A write-counting grade monad
let get #s : count_st s 0 s = fun s -> s, s, 0
let put #s (x:s) : count_st s 1 unit = fun _ -> (), x, 1
```

We can build computations in our graded `count_st` monad relatively easily.

```
let test #s =
  let+ x = get #s in
  put x

//F* + SMT automatically proves that the index simplifies to 2
let test2 #s : count_st s 2 unit =
  let+ x = get in
  put x ;+
  put x
```

F* infers the typeclass instantiations and the type of `test` to be `count_st s (op #monoid_nat_plus 0 1) unit`.

In `test2`, F* infers the type `count_st s (op #monoid_nat_plus 0 (op #monoid_nat_plus 1 1)) unit`, and then automatically proves that this type is equivalent to the user annotation `count_st s 2 unit`, using the definition of `monoid_nat_plus`. Note, when one defines `let+`, one can also use `e1 ;+ e2` to sequence computations when the result type of `e1` is `unit`.

23.6 Summary

Typeclasses are a flexible way to structure programs in an abstract and generic style. Not only can this make program construction more modular, it can also make proofs and reasoning more abstract, particularly when typeclasses contain not just methods but also properties characterizing how those methods ought to behave. Reasoning abstractly can make proofs simpler: for example, if the monoid-ness of natural number addition is the only property needed for a proof, it may be simpler to do a proof generically for all monoids, rather than reasoning specifically about integer arithmetic.

That latter part of this chapter presented typeclasses for computational structures like monads and functors. Perhaps conspicuous in these examples were the lack of algebraic laws that characterize these structures. Indeed, we focused primarily on programming with monads and graded monads, rather than reasoning about them. Enhancing these typeclasses with algebraic laws is a useful, if challenging exercise. This also leads naturally to F*'s effect system in the next section of this book, which is specifically concerned with doing proofs about programs built using monad-like structures.

Part V

Computational Effects

All the programs we've considered so far have been *pure*, which is to say that only thing that can be observed about a program by its context is the value that the program returns—the inner workings of the program (e.g., whether it multiplies two numbers by repeated addition or by using a primitive operation for multiplication) cannot influence the behavior of its context. That is, pure terms can be reasoned about like pure mathematical functions.[1]

However, many practical programs exhibit behaviors that are beyond just their output. For example, they may mutate some global state, or they may read and write files, or receive or send messages over the network—such behaviors are often called *side effects*, *computational effects*, or just *effects*.

In this section, we look at F*'s effect system which allows users to

- Model the semantics of various kinds of effects, e.g., mutable state, input/output, concurrency, etc.
- Develop reasoning principles that enable building proofs of various properties of effectful programs
- Simplify effectful program construction by encapsulating the semantic model and reasoning principles within an abstraction that allows users to write programs in F*'s native syntax while behind the scenes, for reasoning and execution purposes, programs are elaborated into the semantic model

Aside from user-defined effects, F*'s also supports the following *primitive* effects:

- **Ghost**: An effect which describes which parts of a program have no observable behavior at all, and do not even influence the result returned by the program. This allows optimizing a program by erasing the parts of a program that are computationally irrelevant.
- **Divergence**: An effect that encapsulates computations which may run forever. Potentially divergent computations cannot be used as proofs (see *termination*) and the effect system ensures that this is so.
- **Partiality**: Partial functions are only defined over a subset of their domain. F* provides primitive support for partial functions as an effect. Although currently primitive, in the future, we hope to remove the special status of partial functions and make partial functions a user-defined notion too.

[1] Although pure F* programs are mathematical functions in the ideal sense, when executing these programs on a computer, they do exhibit various side effects, including consuming resources like time, power, and memory. Although these side effects are clearly observable to an external observer of a running F* program, the resourse-usage side effects of one component of a pure F* program are not visible to another component of the same program.

CHAPTER

TWENTYFOUR

COMPUTATION TYPES TO TRACK DEPENDENCES

A main goal for F*'s effect system is to *track dependences* among the various parts of a program. For example, the effect system needs to ensures that the total part of a program that is proven to always terminate never calls a function in the divergent fragment (since that function call may loop forever). Or, that the runtime behavior of a compiled program does not depend on ghost computations that get erased by the compiler.

In a paper from 1999 called A Core Calculus of Dependency, Abadi et al. present DCC, a language with a very generic way to track dependences. DCC's type system includes an indexed, monadic type T_l, where the index l ranges over the elements of a lattice, i.e., the indexes are arranged in a partial order. DCC's type system ensures that a computation with type T_l can depend on another computation with type T_m only if $m \leq l$ in the lattice's partial order. F*'s effect system is inspired by DCC, and builds on a 2011 paper by Swamy et al. called Lightweight Monadic Programming in ML which develops a DCC-like system for an ML-like programming language.

At its core, F*'s effect system includes the following three elements:

Computation Types: F*'s type system includes a notion of *computation type*, a type of the form `M t` where `M` is an *effect label* and `t` is the return type of the computation. A term `e` can be given the computation type `M t` when executing `e` exhibits *at-most* the effect `M` and (possibly) returns a value of type `t`. We will refine this intuition as we go along. In contrast with computation types, the types that we have seen so far (`unit`, `bool`, `int`, `list int`, other inductive types, refinement types, and arrows) are called *value types*.

Partially Ordered Effect Labels: The effect label of a computation type is drawn from an open-ended, user-extensible set of labels, where the labels are organized in a user-chosen partial order. For example, under certain conditions, one can define the label `M` to be a sub-effect of `N`, i.e., `M < N`. For any pair of labels `M` and `N`, a partial function `lub M N` (for least upper bound) computes the least label greater than both `M` and `N`, if any.

Typing Rules to Track Dependences: The key part of the effect system is a rule for composing computations sequentially using `let x = e1 in e2`. Suppose `e1 :  M t1`, and suppose `e2 :  N t2` assuming `x:t1`, then the composition `let x = e1 in e2` has type `L t2`, where `L = lub M N`—if `lub M N` is not defined, then the `let`-binding is rejected. Further, a computation with type `M t` can be implicitly given the type `N t`, when `M < N`, i.e., moving up the effect hierarchy is always permitted. The resulting typing discipline enforces the same dependence-tracking property as DCC: a computation `M t` can only depend on `N t` when `lub M N = M`.

In full generality, F*'s computation types are more complex than just an effect label `M` and a result type (i.e., more than just `M t`), and relying on F*'s dependent types, computation types do more than just track dependences, e.g., a computation type in F* can also provide full, functional correctness specifications. The papers referenced below provide some context and we discuss various elements of these papers throughout this part of the book.

- Verifying Higher-order Programs with the Dijkstra Monad, introduces the idea of a Dijkstra monad, a construction to structure the inference of weakest preconditions of effectful computations.
- This 2016 paper, Dependent Types and Multi-Monadic Effects in F*, has become the canonical reference for F*. It shows how to combine multiple Dijkstra monads with a DCC-like system.
- Dijkstra Monads for Free presents an algorithm to construct Dijkstra monads automatically for a class of simple monads.

- Dijkstra Monads for All generalizes the construction to apply to relate any computational monad to a specificational counterpart, so long as the two are related by a monad morphism.
- Programming and Proving with Indexed Effects describes F*'s user-defined effect system in its most general form, allowing it to be applied to any indexed effects, including Dijkstra monads, but several other constructions as well.

CHAPTER

TWENTYFIVE

THE EFFECT OF TOTAL COMPUTATIONS

At the very bottom of the effect label hierarchy is `Tot`, used to describe pure, total functions. Since they are at the bottom of the hierarchy, `Tot` computations can only depend on other `Tot` computations, ensuring that F*'s logical core remains total.

Every term in F* is typechecked to have a computation type. This includes the total terms we have been working with. Such terms are classified in the default effect called `Tot`, the effect of total computations that do not have any observable effect, aside from the value they return. Any meaning or intuition that we have ascribed to typing `e:t` extends to `e:Tot t`. For example, if `e:Tot t`, then at runtime, `e` terminates and produces a value of type `t`. In addition, since its effect label is `Tot`, it has no side-effect.

In fact, as we have already *seen*, notationally `x:t0 -> t1` is a shorthand for `x:t0 -> Tot t1` (where `t1` could itself be an arrow type). More generally, arrow types in F* take the form `x:t -> C`, representing a function with argument type `t0` and body computation type `C` (which may depend on `x`).

Similarly, the return type annotation that we have seen in the `let` definitions is also a shorthand, e.g., the *id function*

```
let id (a:Type) (x:a) : a = x
```

is a shorthand for

```
let id (a:Type) (x:a) : Tot a = x  //the return type annotation is a computation type
```

and the type of `id` is `a:Type -> a -> Tot a`. More generally, the return type annotations on `let` definitions are computation types `C`.

The *explicit annotation syntax* `e <:  t` behaves a little differently. F* allows writing `e <:  C`, and checks that `e` indeed has computation type `C`. But when the effect label is omitted, `e <:  t`, it is interpreted as `e <:  _ t`, where the omitted effect label is inferred by F* and does not default to `Tot`.

25.1 Evaluation order

For pure functions, the evaluation order is irrelevant.[1] F* provides abstract machines to interpret pure terms using either a lazy evaluation strategy or a call-by-value strategy (see a *forthcoming chapter on F*'s normalizers*). Further, compiling pure programs to OCaml, F* inherits the OCaml's call-by-value semantics for pure terms.

When evaluating function calls with effectful arguments, the arguments are reduced to values first, exhibiting their effects, if any, prior to the function call. That is, where `e1` and `e2` may be effectful, the application `e1 e2` is analogous to `bind f = e1 in bind x = e2 in f x`: in fact, internally this is what F* elaborates `e1 e2` to, when either of

[1] Since F* is an extensional type theory, pure F* terms are only *weakly* normalizing. That is, some evaluation strategies (e.g., repeatedly reducing a recursive function deep in an infeasible code path) need not terminate. However, for every closed, pure term, there is a reduction strategy that will reduce it fully. As such, the evaluation order for pure functions is irrelevant, except that some choices of evaluation order may lead to non-termination.

them may have non-trivial effects. As such, for effectful terms, F* enforces a left-to-right, call-by-value semantics for effectful terms.

Since only the value returned by a computation is passed as an argument to a function, function argument types in F* are always value types. That is, they always have the form `t -> C`. To pass a computation as an argument to another function, you must encapsulate the computation in a function, e.g., in place of `C -> C`, one can write `(unit -> C) -> C'`. Since functions are first-class values in F*, including functions whose body may have non-trivial effects, one can always do this.

CHAPTER

TWENTYSIX

ERASURE AND THE GHOST EFFECT

When writing proof-oriented programs, inevitably, some parts of the program serve only to state and prove properties about the code that actually executes. Our first non-trivial effect separates the computationally relevant parts of the program from the computationally irrelevant (i.e., specificational or *ghost*) parts of a program. This separation enables the F* compiler to guarantee that all the ghost parts of a program are optimized away entirely.

For a glimpse of what all of this is about, let's take a look again at length-indexed vectors—we saw them first *here*.

```
type vec (a:Type) : nat -> Type =
  | Nil : vec a 0
  | Cons : #n:nat -> hd:a -> tl:vec a n -> vec a (n + 1)
```

and a function to concatenate two vectors:

```
let rec append #a #n #m (v1:vec a n) (v2:vec a m)
  : vec a (n + m)
  = match v1 with
    | Nil -> v2
    | Cons hd tl -> Cons hd (append tl v2)
```

Compare this with concatenating two lists:

```
let rec list_append #a (l1 l2:list a) =
    match l1 with
    | [] -> []
    | hd::tl -> hd :: list_append tl l2
```

Superficially, because of the implicit arguments, it may look like concatenating vectors with `append` is just as efficient as a concatenating lists—the length indexes seem to impose no overhead. But, let's look at the code that F* extracts to OCaml for length-indexed vectors.

First, in the definition of the `vec` type, since OCaml is not dependently typed, the `nat`-index of the F* `vec` is replaced by a `'dummy` type argument—that's fine. But, notice that the `Cons` constructor contains three fields: a `Prims.nat` for the length of the tail of the list, the head of the list, and then then tail, i.e., the length of the tail of the list is stored at every `Cons` cell, so the `vec` type is actually less space efficient than an ordinary `list`.

```
type ('a, 'dummy) vec =
  | Nil
  | Cons of Prims.nat * 'a * ('a, unit) vec
```

Next, in the OCaml definition of `append`, we see that it receives additional arguments `n` and `m` for the lengths of the vectors, and worse, in the last case, it incurs an addition to sum `n' + m` when building the result vector. So, `append` is also less time-efficient than `List.append`.

```
let rec append :
  'a .
    Prims.nat ->
      Prims.nat -> ('a, unit) vec -> ('a, unit) vec -> ('a, unit) vec
  =
  fun n ->
    fun m ->
      fun v1 ->
        fun v2 ->
          match v1 with
          | Nil -> v2
          | Cons (n', hd, tl) ->
              Cons ((n' + m), hd, (append n' m tl v2))
```

This is particularly unfortunate, since the computational behavior of append doesn't actually depend on the length indexes of the input vectors. What we need is a principled way to indicate to the F* compiler that some parts of a computation are actually only there for specification or proof purposes and that they can be removed when compiling the code, without changing the observable result computed by the program. This is what *erasure* is about—removing the computationally irrelevant parts of a term for compilation.

Here's a revised version of vectors, making use of the erased type from the FStar.Ghost library to indicate to F* which parts must be erased by the compiler.

```
module VecErased
open FStar.Ghost

noeq
type vec a : nat -> Type =
  | Nil : vec a 0
  | Cons : #n:erased nat -> hd:a -> tl:vec a n -> vec a (n + 1)

let rec append #a (#n #m:erased nat) (v0:vec a n) (v1:vec a m)
  : vec a (n + m)
  = match v0 with
    | Nil -> v1
    | Cons hd tl -> Cons hd (append tl v1)
```

We'll look into this in much more detail in what follows, but notice for now that:

1. The first argument of Cons now has type erased nat.
2. The implicit arguments of append corresponding to the indexes of the input vectors have type erased nat.

If we extract this code to OCaml, here's what we get:

```
type ('a, 'dummy) vec =
  | Nil
  | Cons of unit * 'a * ('a, unit) vec
```

```
let rec append :
  'a . unit -> unit -> ('a, unit) vec -> ('a, unit) vec -> ('a, unit) vec =
  fun n ->
    fun m ->
      fun v0 ->
        fun v1 ->
```

(continues on next page)

(continued from previous page)

```
        match v0 with
        | Nil -> v1
        | Cons (uu___, hd, tl) -> Cons ((), hd, (append () () tl v1))
```

Notice that the erased arguments have all been turned into the unit value `()`, and the needless addition in `append` is gone too.

Of course, the code would be cleaner if F* were to have entirely removed the argument instead of leaving behind a unit term, but we leave it to the downstream compiler, e.g., OCaml itself, to remove these needless units. Further, if we're compiling the ML code extracted from F* to C, then KaRaMeL does remove these additional units in the C code it produces.

26.1 Ghost: A Primitive Effect

The second, primitive effect in F*'s effect system is the effect of *ghost* computations, i.e., computation types whose effect label is `GTot`.[1] The label `GTot` is strictly above `Tot` in the effect hierarchy, i.e., `Tot < GTot`. This means that a term with computation type `GTot t` cannot influence the behavior of a term whose type is `Tot s`. Conversely, every `Tot` computation can be implicitly promoted to a `GTot` computation.

Ghost computations are just as well-behaved as pure, total terms—they always terminate on all inputs and exhibit no observable effects, except for the value they return. As such, F*'s logical core really includes both `Tot` and `GTot` computations. The distinction between `Tot` and `GTot` is only relevant when considering how programs are compiled. Ghost computations are guaranteed to be erased by the the compiler, while `Tot` computations are retained.

Since `Tot` terms are implicitly promoted to `GTot`, it is easy to designate that some piece of code should be erased just by annotating it with a `GTot` effect label. For example, here is an ghost version of the factorial function:

```
let rec factorial (n:nat)
  : GTot nat
  = if n = 0 then 1 else n * factorial (n - 1)
```

Its definition is identical to the corresponding total function that we saw earlier, except that we have annotated the return computation type of the function as `GTot nat`. This indicates to F* that `factorial` is to be erased during compilation, and the F* type-and-effect checker ensures that `Tot` computation cannot depend on an application of `factorial n`.

26.2 Ghost Computations as Specifications

A ghost function like `factorial` can be used in specifications, e.g., in a proof that a tail recursion optimization `factorial_tail` is equivalent to `factorial`.

```
let rec factorial_tail (n:nat) (out:nat)
  : Tot (r:nat { r == out * factorial n })
  = if n = 0 then out
    else factorial_tail (n - 1) (n * out)

let fact (n:nat)
  : Tot (r:nat { r == factorial n })
  = factorial_tail n 1
```

[1] The name `GTot` is meant to stand for "Ghost and Total" computations, and is pronounced "gee tote". However, it's a poor name and is far from self-explanatory. We plan to change the name of this effect in the future (e.g., to something like `Spec`, `Ghost`, or `Erased`), though this is a breaking change to a large amount of existing F* code.

This type allows a client to use the more efficient `fact`, but for reasoning purposes, one can use the more canonical `factorial`, proven equivalent to `fact`.

In contrast, if we were to try to implement the same specification by directly using the factorial ghost function, F* complains with a effect incompatibility error.

```
[@@expect_failure]
let factorial_bad (n:nat) (out:nat)
  : Tot (r:nat { r == out * factorial n })
  = out * factorial n
```

The error is:

```
Computed type "r: nat{r == out * factorial n}" and
effect "GTot" is not compatible with the annotated
type "r: nat{r == out * factorial n}" effect "Tot"
```

So, while F* forbids using ghost computations in `Tot` contexts, it seems to be fine with accepting a use of factorial in specifications, e.g., in the type `r:nat { r == out * factorial n }`. We'll see in a moment why this is permitted.

26.3 Erasable and Non-informative Types

In addition to using the `GTot` effect to classifies computations that must be erased, F* also provides a way to mark certain *value types* as erasable.

Consider introducing an inductive type definition that is meant to describe a proof term only and for that proof term to introduce no runtime overhead. In a system like Coq, the type of Coq propositions `Prop` serves this purpose, but `prop` in F* is quite different. Instead, F* allows an inductive type definition to be marked as `erasable`.

For example, when we looked at the *simply typed lambda calculus (STLC)*, we introduced the inductive type below, to represent a typing derivation for an STLC term. One could define a typechecker for STLC and give it the type shown below to prove it correct:

```
val check (g:env) (e:exp) : (t : typ & typing g e t)
```

However, this function returns both the type `t:typ` computed for `e`, we well as the typing derivation. Although the typing derivation may be useful in some cases, often returning the whole derivation is unnecessary. By marking the definition of the `typing` inductive as shown below (and keeping the rest of the definition the same), F* guarantees that the compiler will extract `typing g e t` to the `unit` type and correspondinly, all values of `typing g e t` will be erased to the unit value `()`

```
[@@erasable]
noeq
type typing : env -> exp -> typ -> Type = ...
```

Marking a type with the `erasable` attribute and having it be erased to `unit` is safe because F* restricts how `erasable` types can be used. In particular, no `Tot` computations should be able to extract information from a value of an erasable type.

Closely related to erasable types are a class of types that are called *non-informative*, defined inductively as follows:

1. The type `Type` is non-informative
2. The type `prop` is non-informative (i.e., unit and all its subtypes)
3. An erasable type is non-informative

4. A function type `x:t -> Tot s` is non-informative, if `s` is non-informative
5. A ghost function type `x:t -> GTot s` is non-informative
6. A function type `x:t -> C`, with user-defined computation type `C`, is non-informative if the effect label of `C` has the erasable attribute.

Intuitively, a non-informative type is a type that cannot be case-analyzed in a `Tot` context.

With this notion of non-informative types, we can now define the restrictions on an `erasable` type:

1. Any computation that pattern matches on an erasable type must return a non-informative type.
2. Inductive types with the `erasable` attribute do not support built-in decidable equality and must also be marked `noeq`.

26.4 The *erased* type, *reveal*, and *hide*

The `erasable` attribute can only be added to new inductive type definitions and every instance of that type becomes erasable. If you have a type like `nat`, which is not erasable, but some occurrences of it (e.g., in the arguments to `Vector.append`) need to be erased, the F* standard library `FStar.Ghost.fsti` offers the following:

```
(** [erased t] is the computationally irrelevant counterpart of [t] *)
[@@ erasable]
val erased (t:Type u#a) : Type u#a
```

`FStar.Ghost` also offers a pair of functions, `reveal` and `hide`, that form a bijection between `a` and `erased a`.

```
val reveal (#a: Type u#a) (v:erased a) : GTot a

val hide (#a: Type u#a) (v:a) : Tot (erased a)

val hide_reveal (#a: Type) (x: erased a)
  : Lemma (ensures (hide (reveal x) == x))
          [SMTPat (reveal x)]

val reveal_hide (#a: Type) (x: a)
  : Lemma (ensures (reveal (hide x) == x))
          [SMTPat (hide x)]
```

Importantly, `reveal v` breaks the abstraction of `v:erased a` returning just an `a`, but doing so incurs a `GTot` effect—so, `reveal` cannot be used in an arbitrary `Tot` contexts.

Dually, `hide v` can be used to erase `v:a`, since a `Tot` context cannot depend on the value of an `erased a`.

The SMT patterns on the two lemmas allow F* and Z3 to automatically instantiate the lemmas to relate a value and its hidden counterpart—*this chapter* provides more details on how SMT patterns work.

Implicit coercions

`FStar.Ghost.erased`, `reveal`, and `hide` are so commonly used in F* that the compiler provides some special support for it. In particular, when a term `v:t` is used in a context that expects an `erased t`, F* implictly coerces `v` to `hide v`. Likewise, when the context expects a `t` where `v:erased t` is provided, F* implicitly coerces `v` to `reveal v`.

The following examples illustrates a few usages and limitations. You can ask F* to print the code with implicits enabled by using `--dump_module RevealHideCoercions --print_implicits`.

```
module RevealHideCoercions
open FStar.Ghost

let auto_hide (x:nat) : erased nat = x
let auto_reveal (x:erased nat) : GTot nat = x

[@@expect_failure] //Expect GTot
let auto_reveal_2 (x:erased nat) : Tot nat = x

let incr (x:nat) : nat = x + 1
let incr_e (x:erased nat) : erased nat = incr x

let incr' (x:nat) : GTot nat = incr_e x

[@@expect_failure]
let poly (x:nat) (y:erased nat) = x == y
```

A few comments on these examples:

- The first two functions illustrate how a `nat` is coerced implicitly to `erased nat`. Note, the effect of `auto_reveal` is `GTot`
- `auto_reveal_2` fails, since the the annotation claims, incorrectly, that the effect label is `Tot`
- `incr` is just a `nat -> nat` function.
- `incr_e` is interesting because it calls `incr` with an `erased nat` and the annotation expects an `erased nat` too. The body of `incr_e` is implicitly coerced to `hide (incr (reveal x))`
- `incr'` is interesting, since it calls `incr_e`: its body is implicitly coerced to `reveal (incr_e (hide x))`
- Finally, `poly` shows the limitations of implicit coercion: F* only inserts coercions when the expected type of the term in a context and the type of the term differ by an `erased` constructor. In `poly`, since == is polymorphic, the expected type of the context is just an unresolved unification variable and, so, no coercion is inserted. Instead, F* complains that `y` has type `erased nat` when the type `nat` was expected.

26.5 Using Ghost Computations in Total Contexts

We have already noted that `Tot < GTot`, enabling `Tot` computations to be re-used in `GTot` contexts. For erasure to be sound, it is crucial that `GTot` terms cannot be used in `Tot` contexts, and indeed, F* forbids this in general. However, there are two exceptions.

26.5.1 Effect Promotion for Non-informative Types

Consider a term `f` with type `GTot s`, where `s` is a non-informative type. Since `s` is non-informative, no total context can extract any information from `f`. As such, F* allows implicitly promoting `GTot s` to `Tot s`, when `s` is a non-informative type.

For instance, the following is derivable, `hide (factorial 0) : Tot (erased nat)`: let's work through it in detail.

1. We know that that `factorial n : GTot nat`

2. Recall from the discussion on *evaluation order* and the application of functions to effectful arguments, hide (factorial 0) is equivalent to let x = factorial 0 in hide x, where x:nat and hide x : Tot (erased nat).

3. From the rule for sequential composition of effectful terms, the type of let x = factorial 0 in hide x should be GTot (erased nat), since GTot = lub GTot Tot.

4. Since erased nat is a non-informative type, GTot (erased nat) is promoted to Tot (erased nat), which is then the type of hide (factorial 0).

Effect promotion for ghost functions returning non-informative types is very useful. It allows one to mix ghost computations with total computations, so long as the result of the ghost sub-computation is hidden with an erased type. For instance, in the code below, we use hide (factorial (n - 1)) and use the result f_n_1 in an assertion to or some other proof step, all within a function that is in the Tot effect.

```
let rec factorial_tail_alt (n:nat) (out:nat)
  : Tot (r:nat { r == out * factorial n })
  = if n = 0 then out
    else (
      let f_n_1 = hide (factorial (n - 1)) in
      let result = factorial_tail_alt (n - 1) (n * out) in
      assert (result == (n * out) * f_n_1);
      result
    )
```

26.5.2 FStar.Ghost.Pull.pull

Consider applying map factorial [0;1;2], where map : (a -> Tot b) -> list a -> list b. F* refuses to typecheck this written, since factorial : nat -> GTot nat, which is not a subtype of nat -> Tot nat.

Using effect promotion, one could write map (fun x -> hide (factorial x)) [0;1;2], but this has type list (erased nat), which is not the same as list nat.

This is unfortunate since it seems that basic libraries designed for use with with higher-order total functions cannot be reused with ghost functions.[2]

The library function FStar.Ghost.Pull.pull can help in such situatinos. It has the following signature:

```
val pull (#a:Type) (#b:a -> Type) (f: (x:a -> GTot (b x)))
  : GTot (x:a -> b x)

val pull_equiv (#a:Type) (#b:a -> Type) (f: (x:a -> GTot (b x))) (x:a)
  : Lemma (ensures pull f x == f x)
          [SMTPat (pull f x)]
```

This type states that for any ghost function f, we can exhibit a total function g with the same type as f, while the lemma pull_equiv states that g is pointwise equal to f.[3] However, it may not be possible, in general, to compute g in a way that enables it to be compiled by F*. So, pull f itself has ghost effect, indicating that applications of pull cannot be used in compilable code.

Using pull, one can write map (pull factorial) [0;1;2] : GTot (list nat), thereby reusing ghost functions where total functions are expected.

[2] F* does not have any built-in support for effect-label polymorphism. That is, one cannot write a type like this: map #a #b #e (f: a -> e b) (l:list a) : e (list b), where e is an effect variable. However, using F*'s dependent types, one can code up various ad hoc forms of effect polymorphism. Some other languages with effect systems, notably Koka, do support effect polymorphism primitively.

[3] The SMTPat annotation on the pull_equiv lemma instructs F* to allow Z3 to instantiate this lemma automatically. You can learn more *about SMT patterns and quantifier instantiation in a later chapter*.

26.6 Revisiting Vector Concatenation

We now have all the ingredients to understand how the vector append example shown at the start of this chapter works. Here, below, is a version of the same code with all the implicit arguments and reveal/hide operations made explicit.

```
module VecErasedExplicit
open FStar.Ghost

noeq
type vec a : nat -> Type =
  | Nil : vec a 0
  | Cons : #n:erased nat -> hd:a -> tl:vec a (reveal n) -> vec a (reveal n + 1)

let rec append #a (#n #m:erased nat) (v0:vec a (reveal n)) (v1:vec a (reveal m))
  : Tot (vec a (reveal n + reveal m))
        (decreases (reveal n))
  = match v0 with
    | Nil -> v1
    | Cons #_ #n_tl hd tl ->
      Cons #a
           #(hide (reveal n_tl + reveal m))
           hd
           (append #a
                   #n_tl
                   #m
                   tl
                   v1)
```

Definition of vec

In the definition of the inductive type `vec a`, we have two occurrences of `reveal`. Consider `vec a (reveal n)`, the type of the `tl` of the vector. `reveal n` is a ghost computation of type `GTot nat`, so `vec a (reveal n) : GTot Type`. But, since `Type` is non-informative, `GTot Type` is promoted to `Tot Type`. The promotion from `GTot Type` to `Tot Type` is pervasive in F* and enables ghost computations to be freely used in types and other specifications.

The `vec a (reveal n + 1)` in the result type of `Cons` is similar. Here `reveal n + 1` has type `GTot nat`, but applying it to `vec a` produces a `GTot Type`, which is promoted to `Tot Type`.

Type of append

The type of `append` has four occurrences of `reveal`. Three of them, in the type of `v0`, `v1`, and the return type behave the same as the typing the fields of `Cons`: the `GTot Type` is promoted to `Tot Type`.

One additional wrinkle is in the decreases clause, where we have an explicit `reveal n`, since what decreases on each recursive call is the `nat` that's in bijection with the parameter `n`, rather than `n` itself. When F* infers a decreases clause for a function, any erased terms in the clause are automatically revealed.

Definition of append

The recursive call instantiates the index parameters to `n_tl` and `m`, which are both erased.

When constructing the `Cons` node, its index argument is instantiated to `hide (reveal n_tl + reveal m)`. The needless addition is marked with a `hide` enabling that F* compiler to erase it. As we saw before in `factorial_tail_alt`, using `hide` allows one to mingle ghost computations (like `(reveal n - 1)`) with total computations, as needed for specifications and proofs.

All of this is painfully explicit, but the implicit reveal/hide coercions inserted by F* go a long way towards make things relatively smooth.

CHAPTER

TWENTYSEVEN

DIVERGENCE, OR NON-TERMINATION

Most dependently typed languages are not Turing complete. This is because, as explained *earlier*, it is crucial to the soundness of a type theory to have all functions terminate. This means that you cannot program, say, an interpreter for a general-purpose programming language in a language like Coq, since such an interpreter would not be able to handle programs that intentionally loop forever.[1]

F*'s logical core of total (and ghost) functions can only express terminating computations. However, F*'s also allows expressing non-terminating or *divergent* computations, relying on the effect system to isolate divergent computations from the logical core. In particular, the computation type `Dv t` describes a computation that may loop forever, but if it completes, it returns a value of type `t`.

Relying on the effect system as a dependency tracking mechanism, F* ensures that `Tot` computations cannot rely on `Dv` computations by placing `Dv` above `Tot` in the effect hierarchy, while, conversely, a total computation `Tot t` can be silently promoted to `Dv t`, the type of computations that may not terminate, i.e., `Tot < Dv` in the effect partial order.

Recursive functions that return computations in the `Dv` effect are not checked for termination. As such, using the `Dv` effect, one can write programs such as the one below, which computes Collatz sequences—whether or not this program terminates for all inputs is an open problem.

```
(* You can program a function to compute Collatz sequences
   ... though no one knows if it actually terminates for all n *)
let rec collatz (n:pos)
  : Dv (list pos)
  = if n = 1 then [n]
    else if n % 2 = 0
    then n::collatz (n / 2)
    else n::collatz (3 * n + 1)
```

In this chapter, we'll look in detail at the `Dv` effect and how it interacts with other features of the language, including the other effects, recursive type definitions, and the styles of programming and proving it enables.

[1] In place of general recursion and potential non-termination, other dependently typed languages like Coq and Agda offer features like corecursion and coinduction. Coinduction can be used to express a class of *productive* non-terminating programs. For instance, using coinduction, one could program a web server that loops forever to handle an infinite stream of requests, while producing a response for each request in a finite amount of time. Even the `collatz` function can be given a corecursive definition that computes a potentially infinite stream of numbers. However, not all non-terminating computations can be implemented with coinduction/corecursion. F* does not yet support coinduction.

27.1 The Dv effect

The effect Dv (for divergence) is a primitive effect in F*. Computations in Dv may not terminate, even with infinite resources. In other words, computations in the Dv effect have the observational behavior of non-termination. For example, the following loop function has type unit -> Dv unit and it always diverges when called:

```
let rec loop (): Dv unit = loop ()
```

If we remove the Dv effect label annotation, then F* treats the function as total and will try to prove that the recursive call terminates, according to its usual termination checking rules, i.e., F* will attempt to prove () << () which fails, as expected.

Since the Dv effect admits divergence, F* essentially turns-off the termination checker when typechecking Dv computations. So the recursive loop () call does not require a decreasing termination metric.

27.2 Partial correctness semantics of Dv

The Tot effect in F* has a *total correctness* semantics. That is, if a term has type e:Tot t, then e terminates terminates and produces a value of type t.

Terms with type Dv t have a *partial correctness* semantics. That is, a term e:Dv t, e may either run forever, but if it terminates then the resulting value has type t.

Another perspective is that aside from disabling the termination checking features of F*, all other type-checking constraints are enforced on Dv term. This means that one can still give interesting sound, specifications to Dv programs, e.g., the type below proves that if the Collatz function terminates, then the last element of the sequence is 1.

```
let rec collatz_ends_in_one (n:pos)
  : Dv (l:list pos { Cons? l /\ last l == 1 })
  = if n = 1 then [n]
    else if n % 2 = 0
    then n::collatz_ends_in_one (n / 2)
    else n::collatz_ends_in_one (3 * n + 1)
```

If, for example, in the base case we were to return the empty list [] rather than [n], then F* would refuse to accept the program, since the program could terminate while returning a value that is not an element of the annotated return type.

27.3 Isolating Dv from the logical core

Since Dv terms need not terminate, a program that always loops forever can be given any return type. For instance, the program below has return type False:

```
let rec dv_false () : Dv False = dv_false()
```

Importantly, a term of type Dv False should not be confused as a *proof* of False, since that would lead immediately to unsoundness of F*'s logical core. In particular, it should be impossible to turn a e:Dv t into a term of type Tot t. This is achieved by F*'s effect system, which treats Tot as a sub-effect of Dv, i.e., Tot < Dv, in the effect order. As explained in *earlier*, this ensures that no Tot term can depend on a Dv term, maintaining soundness of the total correctness interpretation of Tot.

As an example, the following attempt to "cast" dv_false to Tot fails, as does trying to use dv_false to produce incorrect proofs of other types.

```
[@@expect_failure]
let tot_false : Tot False = dv_false()
[@@ expect_failure]
let bad_zero : Tot (y:int{y == 0}) = dv_false (); 1
```

While F* does not allow Tot computations to depend on Dv computations, going the other way is perfectly fine. Intuitively, always terminating computations are potentially non-terminating. We can think of it like a *weakening* of the specification:

```
let add_one (x:int) : int = x + 1
let add_one_div (x:int) : Dv int = add_one x
```

The effect system of F* automatically *lifts* Tot computations into Dv, meaning that Tot functions can be seamlessly used in Dv functions.

The weakening of Tot terms to other effects is so pervasive in F* that one hardly even thinks about it, e.g., in the collatz program, sub-terms like n / 2 are in Tot but are easily used within a computation in the Dv effect.

27.4 No extrinsic proofs for Dv computations

One important consequence of any effectful code, including Dv, being outside the logical core of F* is that it is not possible to do *extrinsic proofs* about effectful code. One cannot even state properties of Dv computations in specifications, since even specifications must be total. For example, even stating the following lemma is illegal:

```
[@@expect_failure]
val collatz_property (n:pos)
  : Lemma (Cons? (collatz n) /\ last (collatz n) = 1)
```

This is nonsensical in F* since writing Cons? (collatz n) supposes that collatz n is *defined*, whereas it might actually just loop forever.

The only way to state properties about divergent programs is to encode the property intrinsically in the computation type, as we saw above.

```
let last #a (l:list a { Cons? l }) : a = L.index l (L.length l - 1)
val collatz_ends_in_one (n:pos)
  : Dv (l:list pos { Cons? l /\ last l == 1 })
```

27.4.1 Exercise

Define a predicate collatz_spec (n:pos) (l:list pos) : bool that decides if l is a valid Collatz sequence starting at n.

Implement val collatz' (n:pos) : Dv (l:list pos { collatz_spec n l }).

What does this type mean? Are there other ways to implement collatz' with the same type?

Answer

```
let rec collatz_spec (n:pos) (l:list pos)
  : Tot bool (decreases l)
  = match l with
    | [] -> false
```

(continues on next page)

(continued from previous page)

```
    | hd :: tl ->
      hd = n && (
        if hd = 1 then tl = []
        else if n%2 = 0 then collatz_spec (n/2) tl
        else collatz_spec (3*n + 1) tl
      )
// collatz' may loop forever on some inputs
// but if it completes it always returns a valid
// Collatz sequence
let rec collatz' (n:pos)
  : Dv (l:list pos { collatz_spec n l } )
  = if n = 1 then [n]
    else if n % 2 = 0
    then n::collatz' (n / 2)
    else n::collatz' (3 * n + 1)

// here's another bogus implementation that always loops
let rec collatz'' (n:pos)
  : Dv (l:list pos { collatz_spec n l } )
  = collatz'' n
```

27.5 General Recursive Types and Impredicativity with Dv

Aside from disabling the decreases metric on recursive functions in Dv, F* also disables two other forms of termination checking on Dv computations.

Recall from a *previous chapter* that inductive type definitions are subject to the *strict positivity* condition, since non-positive definitions allow the definition of recursive types and non-terminating computations. However, since computations in the Dv effect are already allowed to loop forever, the strict positivity condition can be relaxed when Dv types are involved. For example, one can define this:

```
noeq
type nonpos =
  | NonPos : (nonpos -> Dv False) -> nonpos

let loop_nonpos' (f:nonpos) : Dv False =
   let NonPos g = f in g f

let loop_nonpos () : Dv False  = loop_nonpos' (NonPos loop_nonpos')
```

The type `nonpos` is not strictly positive, since it appears to the left of an arrow in a field of one of its constructors. Indeed, usingn `nonpos` it is possible to define (without using `let rec`) an infinitely looping program `loop_nonpos()`—however, the type `Dv False` tells us that this program may loop forever, and the infinite loop is safely isolated from F*'s logical core.

The other place in F*'s type system where termination checking comes into play is in the *universe levels*. As we learned previously, the logical core of F* is organized into an infinite hierarchy with copies of the F* type system arranged in a tower of universes. This stratification is necessary to prevent inconsistencies within the logical core. However, terms in the Dv effect are outside the logical core and, as such, restrictions on the universe levels no longer apply. As the snippet below shows a total function returning a type in universe `u#a` resides in universe `u#(a + 1)`. However, a Dv function returning a type in u#a is just in universe 0, since the only way to obtain the type dv_type returns is by incurring a Dv effect and moving outside F*'s logical core.

```
let tot_type : Type u#(a + 1) = unit -> Tot (Type u#a)
let dv_type : Type0 = unit -> Dv (Type u#a)
```

27.6 Top-level Effects

A top-level F* term is not meant to be effectful. If one defines the following term, F* accepts the term but raises a warning saying "Top-level let bindings must be total—this term may have effects".

```
let inconsistent : False = loop_nonpos()
```

Top-level effects can be problematic for a few reasons:

1. The order of evaluation of the effects in top-level terms is undefined for programs with multiple modules—it depends on the order in which modules are loaded at runtime.
2. Top-level effects, particularly when divergence is involved, can render F*'s typechecking context inconsistent. For example, once inconsistent is defined, then any other assertion can be proven.

   ```
   let _  = let _ = FStar.Squash.return_squash inconsistent in
            assert false
   ```

Nevertheless, when used carefully, top-level effects can be useful, e.g., to initialize the state of a module, or to start the main function of a program. So, pay attention to the warning F* raises when you have a top-level effect and make sure you really know what you're doing.

27.7 Example: Untyped Lambda Calculus

In this section, we put together the various things we've learned about Dv computations to define several variants of an untyped lambda calculus.

You can refer back to our prior development of the *simply typed lambda calculus* if you need some basic background on the lambda calculus.

27.7.1 Interpreting Deeply Embedded Lambda Terms

We start by defining the syntax of untyped lambda terms, below. The variables use the de Bruijn convention, where a index of a variable counts the number of lambda-binders to traverse to reach its binding occurrence. The Lam case just has the body of the lambda term, with no type annotation on the binder, and no explicit name for the variable.

```
let var = nat
type term =
  | Var  : var -> term
  | Int  : int -> term
  | Lam  : term -> term
  | App  : term -> term -> term
```

As usual, we can define what it means to substitute a variable x with a (closed) term v in t—this is just a regular Tot function.

```
let rec subst (x:var) (v:term) (t:term)
  : Tot term  (decreases t) =
  match t with
  | Var y -> if x = y then v else t
  | Int _ -> t
  | Lam t -> Lam (subst (x + 1) v t)
  | App t0 t1 -> App (subst x v t0) (subst x v t1)
```

Finally, we can define an interpreter for `term`, which can (intentionally) loop infinitely, as is clear from the `Dv` type annotation.

```
(* This interpreter can (intentionally) loop infinitely *)
let rec interpret (t:term)
  : Dv (option term)
  = match t with
    | Var _
    | Int _
    | Lam _ -> Some t
    | App t0 t1 ->
      let head = interpret t0 in
      match head with
      | None -> None
      | Some (Lam body) -> interpret (subst 0 t1 body)
      | _ -> None //type error, expected a function

(* (\x. x x) (\x. x x) *)
let loops () : Dv _ = interpret (App (Lam (App (Var 0) (Var 0)))
                                     (Lam (App (Var 0) (Var 0))))
```

Exercise

This exercise is designed to show how you can prove non-trivial properties of `Dv` computations by giving them interesting dependent types.

The substitution function defined here is only sound when the term being substituted is closed, otherwise, any free variables it has can be captured when substituted beneath a lambda.

A term is closed if it satisfies this definition:

```
let rec closed' (t:term) (offset:int)
  : bool
  = match t with
    | Int _ -> true
    | Var i -> i <= offset
    | Lam t -> closed' t (offset + 1)
    | App t0 t1 -> closed' t0 offset && closed' t1 offset
let closed t = closed' t (-1)
```

Restrict the type of `subst` so that its argument is `v :  term { closed v }`—you will have to also revise the type of its other argument for the proof to work.

Next, give the following type to the interpreter itself, proving that interpreting closed terms produces closed terms, or loops forever.

```
let rec interpret (t:term { closed t })
  : Dv (option (t:term { closed t }))
  = match t with
    | Int _
    | Lam _ -> Some t
    | App t0 t1 ->
      let head = interpret t0 in
      match head with
      | None -> None
      | Some (Lam body) -> interpret (subst 0 t1 body)
      | _ -> None //type error, expected a function
```

Answer

```
module Part4.UTLCEx1

let var = nat
type term =
  | Var  : var -> term
  | Int  : int -> term
  | Lam  : term -> term
  | App  : term -> term -> term

//SNIPPET_START: closed$
let rec closed' (t:term) (offset:int)
  : bool
  = match t with
    | Int _ -> true
    | Var i -> i <= offset
    | Lam t -> closed' t (offset + 1)
    | App t0 t1 -> closed' t0 offset && closed' t1 offset
let closed t = closed' t (-1)
//SNIPPET_END: closed$

let rec closed'_weaken (t:term) (offset offset':int)
  : Lemma
    (requires closed' t offset /\
              offset <= offset')
    (ensures closed' t offset')
  = match t with
    | Int _ -> ()
    | Var _ -> ()
    | Lam t -> closed'_weaken t (offset + 1) (offset' + 1)
    | App t0 t1 ->
      closed'_weaken t0 offset offset';
      closed'_weaken t1 offset offset'

let rec subst (x:var)
              (v:term { closed v })
              (t:term { closed' t x })
  : Tot (t1:term { closed' t1 (x - 1) }) (decreases t) =
  match t with
  | Var y -> if x = y then (closed'_weaken v (-1) (x - 1); v) else t
```

(continues on next page)

(continued from previous page)

```
  | Int _ -> t
  | Lam t -> Lam (subst (x + 1) v t)
  | App t0 t1 -> App (subst x v t0) (subst x v t1)

//SNIPPET_START: interpret$
let rec interpret (t:term { closed t })
  : Dv (option (t:term { closed t }))
  = match t with
    | Int _
    | Lam _ -> Some t
    | App t0 t1 ->
      let head = interpret t0 in
      match head with
      | None -> None
      | Some (Lam body) -> interpret (subst 0 t1 body)
      | _ -> None //type error, expected a function
//SNIPPET_END: interpret$
```

27.7.2 Denoting Lambda Terms into an F* Recursive Type

We now look at a variation on the interpreter above to illustrate how (non-positive) recursive types using Dv can also be used to give a semantics to untyped lambda terms.

Consider the type dyn shown below—it has a non-positive constructor DFun. We can use this type to interpret untyped lambda terms into dynamically typed, potentially divergent, F* terms, showing, in a way, that untyped lambda calculus is no more expressive than F* with the Dv effect.

```
noeq
type dyn =
  | DErr  : string -> dyn
  | DInt  : int -> dyn
  | DFun  : (dyn -> Dv dyn) -> dyn
```

The program denote shown below gives a semantics to term using dyn. It is parameterized by a ctx : ctx_t, which interprets the free variables of the term into dyn.

```
let ctx_t = nat -> dyn

let shift (ctx:ctx_t) (v:dyn)
  : ctx_t
  = fun n -> if n = 0 then v else ctx (n - 1)

let rec denote (t:term) (ctx:ctx_t)
  : Dv dyn
  = match t with
    | Var v -> ctx v
    | Int i -> DInt i
    | Lam t -> DFun (fun v -> denote t (shift ctx v))
    | App t0 t1 ->
      match denote t0 ctx with
```

(continues on next page)

(continued from previous page)

```
      | DFun f -> f (denote t1 ctx)
      | DErr msg -> DErr msg
      | DInt _ -> DErr "Cannot apply an integer"
```

We look at the cases in detail:

- In the Var case, the intepretation just refers to the context.
- Integers constants in term are directly interpreted to integers in dyn.
- The case of Lam is the most interesting: An lambda abstraction in term is interpreted as an F* function dyn -> Dv dyn, recursively calling the denotation function on the body when the function is applied. Here's where we see the non-positivity of DFun at play—it allows us to inject the function into the dyn type.
- Finally, in the application case, we interpret a syntactic application in term as function application in F* (unless the head is not a function, in which case we have a type error).

Exercise

This exercise is similar in spirit to the previous one and designed to show that you can prove some simple properties of denote by enriching its type.

Can you prove that a closed term can be interpreted in an empty context?

First, let's refine the type of contexts so that it only provides an interpretation to only some variables:

```
let ctx_t (i:int) = x:nat{x <= i} -> dyn
```

Next, let's define free t to compute the greatest index of a free variable in a term.

```
let max (x y:int) : int = if x < y then y else x
let rec free (t:term)
  : int
  = match t with
    | Var x -> x
    | Int _ -> -1
    | Lam t -> free t - 1
    | App t0 t1 -> max (free t0) (free t1)
```

Can you give the same denote function shown earlier the following type?

```
val denote (t:term) (ctx:ctx_t (free t))
  : Dv dyn
```

Next, define the empty context as shown below:

```
let empty_context : ctx_t (-1) = fun _ -> false_elim ()
```

Given a closed term t : term { closed t }, where closed t = (free t = -1), can you use denote to give an interpretation to closed terms in the empty context?

Answer

```
module Part4.UTLCEx2
let var = nat
```

(continues on next page)

(continued from previous page)

```
type term =
  | Var  : var -> term
  | Int  : int -> term
  | Lam  : term -> term
  | App  : term -> term -> term

//SNIPPET_START: free$
let max (x y:int) : int = if x < y then y else x
let rec free (t:term)
  : int
  = match t with
    | Var x -> x
    | Int _ -> -1
    | Lam t -> free t - 1
    | App t0 t1 -> max (free t0) (free t1)
//SNIPPET_END: free$

noeq
type dyn =
  | DErr  : string -> dyn
  | DInt  : int -> dyn
  | DFun  : (dyn -> Dv dyn) -> dyn

//SNIPPET_START: ctx_t$
let ctx_t (i:int) = x:nat{x <= i} -> dyn
//SNIPPET_END: ctx_t$

let shift #i (ctx:ctx_t i) (v:dyn)
  : ctx_t (i + 1)
  = fun n -> if n = 0 then v else ctx (n - 1)

(* This is similar to the interpreter, but
   "interprets" terms into the F* type dyn
    rather than just reducing syntax to syntax *)
let rec denote (t:term)
               (ctx:ctx_t (free t))
  : Dv dyn
  = match t with
    | Var v -> ctx v
    | Int i -> DInt i
    | Lam t -> DFun (fun v -> denote t (shift ctx v))
    | App t0 t1 ->
      match denote t0 ctx with
      | DFun f -> f (denote t1 ctx)
      | DErr msg -> DErr msg
      | DInt _ -> DErr "Cannot apply an integer"

//SNIPPET_START: empty_context$
let empty_context : ctx_t (-1) = fun _ -> false_elim ()
//SNIPPET_END: empty_context$

let closed t = free t = -1
```

(continues on next page)

(continued from previous page)

```
let denote_closed (t:term { closed t })
  : Dv dyn
  = denote t empty_context
```

27.7.3 Shallowly Embedded Dynamically Typed Programming

In the previous example, we saw how the syntax of untyped lambda terms can be interpreted into the F* type dyn. In this example, rather than going via the indirection of the syntax of lambda terms, we show how the type dyn can be used directly to embed within F* a small Turing complete, dynamically typed programming language.

We can start by lifting the F* operations on integers and functions to (possibly failing) operations on dyn.

```
(* Lifting operations on integers to operations on dyn *)
let lift (op: int -> int -> int) (n m:dyn) : dyn
  = match n, m with
    | DInt i, DInt j -> DInt (op i j)
    | _ -> DErr "Expected integers"
let mul = lift op_Multiply
let sub = lift op_Subtraction
let add = lift op_Addition
let div (n m:dyn)
  = match n, m with
    | DInt i, DInt j ->
      if j = 0 then DErr "Division by zero"
      else DInt (i / j)
    | _ -> DErr "Expected integers"
let mod (n m:dyn)
  = match n, m with
    | DInt i, DInt j ->
      if j = 0 then DErr "Division by zero"
      else DInt (i % j)
    | _ -> DErr "Expected integers"
```

We also encode provide operations to compare dyn-typed integers and to branch on them, treating 0 as false.

```
(* Branching *)
let if_ (d:dyn) (then_ else_:dyn) =
  match d with
  | DInt b ->
    if b<>0 then then_ else else_
  | _ -> DErr "Can only branch on integers"

(* comparison *)
let eq_ (d:dyn) (d':dyn)
  : dyn
  = match d, d' with
    | DInt i, DInt j -> DInt (if i = j then 1 else 0)
    | _ -> DErr "Can only compare integers"
```

For functions, we can provide combinators to apply functions and, importantly, a combinator fix that provides general recursion.

```
(* Dynamically typed application *)
let app (f:dyn) (x:dyn)
  : Dv dyn
  = match f with
    | DFun f -> f x
    | _ -> DErr "Can only apply a function"
(* general recursion *)
let rec fix (f: (dyn -> Dv dyn) -> dyn -> Dv dyn) (n:dyn)
  : Dv dyn
  = f (fix f) n
```

An aside on the arity of recursive functions: You may wonder why `fix` is defined as shown, rather than `fix_alt` below, which removes a needless additional abstraction. The reason is that with `fix_alt`, to instruct F* to disable the termination checker on the recursive definition, we need an additional `Dv` annotation: indeed, evaluating `fixalt f` in a call-by-value semantics would result, unconditionally, in an infinite loop, whereas `fix f` would immediately return the lambda term `fun n -> f (fix f) n`. In other words, eta reduction (or removing redundant function applications) does not preserve semantics in the presence of divergence.

```
let rec fix_alt (f: (dyn -> Dv dyn) -> dyn -> Dv dyn)
  : Dv (dyn -> Dv dyn)
  = f (fix_alt f)
```

With that, we can program non-trivial dynamically typed, general recursive programs within F* itself, as seen below.

```
(* shorthands *)
let i (i:int) : dyn = DInt i
let lam (f:dyn -> Dv dyn) : dyn = DFun f
(* a dynamically typed analog of collatz *)
let collatz_dyn
  : dyn
  = lam (fix (fun collatz n ->
                if_ (eq_ n (i 1))
                    (i 1)
                    (if_ (eq_ (mod n (i 2)) (i 0))
                         (collatz (div n (i 2)))
                         (collatz (add (mul n (i 3)) (i 1))))))
```

All of which is to illustrate that with general recursion and non-positive datatypes using `Dv`, F* is a general-purpose programming language like ML, Haskell, Lisp, or Scheme, or other functional languages you may be familiar with.

CHAPTER

TWENTYEIGHT

PRIMITIVE EFFECT REFINEMENTS

Note: This chapter provides some background on Floyd-Hoare logic and weakest-precondition-based verification condition generation. This is necessary if you want to understand a bit about how F* infers the logical constraints needed to prove the correctness of a program. It is also useful background for more advanced material in subsequent chapters about defining custom effects in F*, e.g., effects to model state, exceptions, or concurrency.

Refinement types `x:t{p}` *refine* value types `t` and allow us to make more precise assertions about the values in the program. For example, when we have `v : x:int{x >= 0}`, then not only we know that `v` is an `int`, but also that `v >= 0`.

In a similar manner, F* allows refining computation types with specifications that describe some aspects of a program's computational behavior. These *effect refinements* can, in general, be defined by the user in a reasoning system of their choosing, e.g., the refinements may use separation logic, or they may count computation steps.

However, F* has built-in support for refining the specification of pure programs with effect refinements that encode the standard reasoning principles of Floyd-Hoare Logic and weakest precondition-based calculi. Foreshadowing what's about to come in this chapter, we can write the following specification for the `factorial` function:

```
open FStar.Mul
let rec factorial (x:int)
  : Pure int (requires x >= 0) (ensures fun r -> r >= 1)
  = if x = 0
    then 1
    else x * factorial (x - 1)
```

Intuitively, this type states that `factorial x` is a computation defined only when `x >= 0` and always terminates returning a value `r >= 1`. In a way, this type is closely related to other, more familiar, types we have given to `factorial` so far, e.g., `nat -> pos`, and, indeed, `factorial` can be used at this type.

```
let fact (x:nat) : pos = factorial x
```

Actually, in all the code we've seen so far, what's happening under the covers is that F* infers a type for a pure program similar to `Pure t pre post` and then checks that that type can be subsumed to a user-provided specification of the form `Tot t'`.

In this chapter, we look into how these `Pure` specifications work, starting with a primer on Floyd-Hoare Logic and weakest precondition calculi. If the reader is familiar with these, they may safely skip the next subsections, though even if you are an expert, if may be of interest to see how such program logics can be formalized in F*.

28.1 A Primer on Floyd-Hoare Logic and Weakest Preconditions

Floyd-Hoare Logic is a system of specifications and rules to reason about the logical properties of programs, introduced by Robert Floyd in a paper titled Assigning Meaning to Programs and by Tony Hoare in An axiomatic basis for computer programming. The notation used in most modern presentations (called Hoare triples) is due to Hoare. An algorithm to compute Hoare triples was developed by Edsger Dijkstra presented first in this paper , using a technique called *weakest preconditions*. All of them received Turing Awards for their work on these and other related topics.

For an introduction to these ideas, we'll develop a small imperative language with global variables, presenting

- An operational semantics for the language, formalized as an interpreter.
- A Floyd-Hoare program logic proven sound with respect to the operational semantics.
- And, finally, an algorithm to compute weakest preconditions proved sound against the Floyd-Hoare logic.

Our language has the following abstract syntax:

```
let var = nat

type expr =
  | EConst : int -> expr                          // constants: ..., -1, 0, 1, ...
  | EVar   : var -> expr                          // global variables
  | EAdd   : expr -> expr -> expr                 // arithmetic: e1 + e2

type program =
  | Assign : var -> expr -> program               // x := e
  | Seq    : program -> program -> program        // p1; p2
  | If     : expr -> program -> program -> program // if e then p1 else p2
  | Repeat : expr -> program -> program           // repeat n { p }
```

Expressions includes integer constants, global variables (represented just as natural numbers), and some other forms, e.g., arithmetic expressions like addition.

A program includes:

- Assignments, `EAssign x e`, representing the assignment of the result of an expression `e` to a global variable `x`, i.e., `x := e`
- `Seq`, to compose programs sequentially
- `If` to compose programs conditionally
- And `Repeat n p`, which represents a construct similar to a `for`-loop (or primitive recursion), where the program `p` is repeated `n` times, where `n` evaluates to a non-negative integer.

Our language does not have `while` loops, whose semantics are a bit more subtle to develop. We will look at a semantics for `while` in a subsequent chapter.

28.1.1 Operational Semantics

Our first step in giving a semantics to programs is to define an interpreter for it to run a program while transforming a memory that stores the values of the global variables.

To model this memory, we use the type `state` shown below:

```
// The state of a program is a map from global variable to integers
let state = var -> int
let read (s:state) (i:var) : int = s i
let write (s:state) (i:var) (v:int)
  : state
  = fun j -> if i=j then v else read s j
```

Writing a small evaluator for expressions is easy:

```
let rec eval_expr (e:expr) (s:state)
  : int
  = match e with
    | EConst v ->
      v

    | EVar x ->
      read s x

    | EAdd e1 e2 ->
      eval_expr e1 s + eval_expr e2 s
```

The interpreter for programs itself takes a bit more work, since programs can both read and write the state. To structure our interpreter, we'll introduce a simple state monad `st a`. We've seen this construction before in *a previous chapter*—so, look there if the state monad is unfamiliar to you. Recall that F* has support for monadic let operators: the `let!` provides syntactic sugar to convenient compose `st` terms.

```
let st (a:Type) = state -> (a & state)

let get : st state = fun s -> (s, s)

let put (s:state) : st unit = fun _ -> ((), s)

let (let!) #a #b (f:st a) (g: a -> st b) : st b =
  fun s -> let v, s' = f s in g v s'

let return #a (x:a) : st a = fun s -> (x, s)
```

Now, the interpreter itself is a total, recursive function `run` which interprets a program `p` as a state-passing function of type `st unit`, or `state -> unit & state`.

```
let rec run (p:program)
  : Tot (st unit)
        (decreases %[p;0])
  = match p with
    | Assign x e ->
      let! s = get in
      let v = eval_expr e s in
```

(continues on next page)

(continued from previous page)

```
      put (write s x v)

    | Seq p1 p2 ->
      run p1;!
      run p2

    | If e p1 p2 ->
      let! s = get in
      let n = eval_expr e s in
      if n <> 0 then run p1 else run p2

    | Repeat e p ->
      let! s = get in
      let n = eval_expr e s in
      if n <= 0 then return ()
      else run_repeat p n

and run_repeat (p:program) (n:nat)
  : Tot (st unit)
        (decreases %[p; 1; n])
  = if n = 0 then return ()
    else ( run p ;! run_repeat p (n - 1) )
```

Let's look at its definition in detail:

- `Assign x e`: Evaluate `e` in the current state and then update the state with a new value of `x`.
- `Seq p1 p2`: Simply run `p1` and then run `p2`, where `;!` is syntactic sugar for `let! _ = run p1 in run p2`.
- `If e p1 p2`: Evaluate `e` in the current state, branch on its result and run either `p1` or `p2`
- `Repeat e p`: Evaluate `e` to `n`, and if `n` is greater than zero, call the mutually recursive `run_repeat n p`. Most of the subtlety here is in convincing F* that this mutually recursive function terminates, but this is fairly straightforward once you know how—we discussed *termination proofs for mutually recursive functions earlier*.

These operational semantics are the ground truth for our programming language—it defines how programs execute. Now that we have that settled, we can look at how a Floyd-Hoare logic makes it possible to reason about programs in a structured way.

28.1.2 Floyd-Hoare Logic

The goal of a Floyd-Hoare logic is to provide a way to reason about a program based on the structure of its syntax, rather than reasoning directly about its operational semantics. The unit of reasoning is called a *Hoare triple*, a predicate of the form `{P} c {Q}`, where `P` and `Q` are predicates about the state of the program, and `c` is the program itself.

We can *define* Hoare triples for our language by interpreting them as an assertion about the operational semantics, where `triple p c q` represents, formally, the Hoare triple `{ p } c { q }`.

```
let triple (pre:state -> prop)
           (c:program)
           (post:state -> prop)
  = forall s0. pre s0 ==> post (snd (run c s0))
```

The predicate `triple p c q` is valid, if when executing `c` in a state that satisfies `p` results in a state that satisfies `q`. The predicates `p` and `q` are also called precondition and postcondition of `c`, respectively.

For each syntactic construct of our language, we can prove a lemma that shows how to build an instance of the `triple` predicate for that construct. Then, to build a proof of program, one stitches together these lemmas to obtain a `triple p main q`, a statement of correctess of the `main` program.

Assignment

Our first rule is for reasoning about variable assignment:

```
let assignment (x:var) (e:expr) (post:state -> prop)
  : Lemma (triple (fun s0 -> post (write s0 x (eval_expr e s0)))
                  (Assign x e)
                  post)
  = ()
```

This lemma says that `post` holds after executing `x := e` in the initial state `s0`, if `post` holds on the initial state updated at `x` with the value of `e`.

For example, to prove that after executing `z := y + 1` in `s0`, if we expect the value of `z` to be greater than zero`, then the assignment rule says that `read s0 y + 1 > 0` should hold before the assignment, which is what we would expect.

Sequence

Our next lemma about triples stitches together triples for two programs that are sequentially composed:

```
let sequence (p1 p2:program)
             (pre pre_mid post:state -> prop)
  : Lemma
    (requires
      triple pre p1 pre_mid /\
      triple pre_mid p2 post)
    (ensures
      triple pre (Seq p1 p2) post)
  = ()
```

The lemma says that if we can derive the Hoare triples of the two statements such that postcondition of `p1` matches the precondition of `p2`, then we can compose them.

Conditional

The lemma for conditionals is next:

```
let conditional (e:expr)
                (p1 p2:program)
                (pre post:state -> prop)
  : Lemma
    (requires
      triple (fun s -> pre s /\ eval_expr e s =!= 0) p1 post /\
      triple (fun s -> pre s /\ eval_expr e s == 0)  p2 post)
    (ensures
      triple pre (If e p1 p2) post)
  = ()
```

It says that to derive the postcondition `post` from the `If e p1 p2`, we should be able to derive it from each of the branches with the same precondition `pre`. In addition, since we know that `p1` executes only when `e` is non-zero, we can add these facts to the preconditions of each branch.

Repeat

In all the cases so far, these lemmas are proved automated by F* and Z3. In the case of repeats, however, we need to do a little more work, since an inductive argument is involved.

The rule for `repeat` requires a *loop invariant* `inv`. The loop invariant is an assertion that holds before the loop starts, is maintained by each iteration of the loop, and is provided as the postcondition of the loop.

The lemma below states that if we can prove that `triple inv p inv`, then we can also prove `triple inv (Repeat e p) inv`.

```
// We need an auxilary lemma to prove it by induction for repeat_n
let rec repeat_n (p:program) (inv:state -> prop) (n:nat)
  : Lemma
    (requires triple inv p inv)
    (ensures (forall s0. inv s0 ==> inv (snd (run_repeat p n s0))))
 = if n = 0 then ()
   else repeat_n p inv (n - 1)

// Then, we use that auxiliary lemma to prove the main
// rule for reasoning about repeat using an invariant
let repeat (e:expr) (p:program) (inv:state -> prop)
  : Lemma
    (requires triple inv p inv)
    (ensures triple inv (Repeat e p) inv)
  = introduce forall s0. inv s0 ==> inv (snd (run (Repeat e p) s0)) with
    introduce _ ==> _  with
      inv_s0 . (
        let n = eval_expr e s0 in
        if n <= 0 then ()
        else repeat_n p inv n
    )
```

The auxiliary lemma `repeat_n` proves that `run_repeat p n` preserves `inv`, if `p` preserves `inv`.

To call this lemma from the main `repeat` lemma, we need to "get our hands on" the initial state `s0`, and the *syntactic sugar to manipulate logical connectives* makes this possible.

Consequence

The final lemma about our Hoare triples is called the rule of consequence. It allows strengthening the precondition and weakening the postcondition of a triple.

```
let consequence (p:program) (pre pre' post post':state -> prop)
  : Lemma
    (requires
      triple pre p post /\
      (forall s. pre' s ==> pre s) /\
      (forall s. post s ==> post' s))
```

(continues on next page)

(continued from previous page)

```
    (ensures
      triple pre' p post')
  = ()
```

A precondition of a program is an obligation before the statement is executed. So if p requires pre, we can always strengthen the precondition to pre', provided pre' ==> pre, i.e. it is logically valid to require more than necessary in the precondition. Similarly, postcondition is what a statement guarantees. So if p guarantees post, we can always weaken it to guarantee less, i.e. some post' where post ==> post'.

28.1.3 Weakest Preconditions

The rules of Floyd-Hoare logic provide an abstract way to reason about programs. However, the rules of the logic are presented declaratively. For example, to apply the sequence rule, one has derive triples for each component in a way that they prove exactly the same assertion (pre_mid) about the intermediate state. There may be many ways to do this, e.g., one could apply the rule of consequence to weaken the postcondition of the first component, or to strengthen the precondition of the second component.

Dijkstra's system of weakest preconditions eliminates such ambiguity and provides an *algorithm* for computing valid Hoare triples, provided the invariants of all loops are given. This makes weakest preconditions the basis of many program proof tools, since given a program annotated with loop invariants, one can simply compute a logical formula (called a verification condition) whose validity implies the correctness of the program.

At the core of the approach is a function WP (c, Q), which computes a unique, weakest precondition P for the program c and postcondition Q. The semantics of WP is that WP (c, Q) is the weakest precondition that should hold before executing c for the postcondition Q to be valid after executing c. Thus, the function WP assigns meaning to programs as a transformer of postconditions Q to preconditions WP (s, Q).

The wp function for our small imperative language is shown below:

```
let rec wp (c:program) (post: state -> prop)
  : state -> prop
  = match c with
    | Assign x e ->
      fun s0 -> post (write s0 x (eval_expr e s0))

    | Seq p1 p2 ->
      wp p1 (wp p2 post)

    | If e p1 p2 ->
      fun s0 ->
        (eval_expr e s0 =!= 0 ==> wp p1 post s0) /\
        (eval_expr e s0 == 0  ==> wp p2 post s0)

    | Repeat e p ->
      fun s0 ->
        (exists (inv:state -> prop).
          inv s0 /\
          (forall s. inv s ==> post s) /\
          (forall s. inv s ==> wp p inv s))
```

- The case of Assign is identical to the assignment lemma shown earlier.
- The case of Seq sequentially composes the wp's. That is, to prove the post after running p1 ;; p2 we need to prove wp p2 post after running p1. It may be helpful to read this case as the equivalent form fun s0 -> wp

p1 (fun s1 -> wp p2 post s1) s0, where s0 is the initial state and s1 is the state that results after running just p1.

- The If case computes the WPs for each branch and requires them to be proven under the suitable branch condition.
- The Repeat case is most interesting: it involves an existentially quantified invariant inv, which is the loop invariant. That is, to reason about Repeat n p, one has to somehow find an invariant inv that is true initially, and implies both the WP of the loop body as well as the final postcondition.

The wp function is sound in the sense that it computes a sufficient precondition, as proven by the following lemma.

```
let rec wp_soundness (p:program) (post:state -> prop)
  : Lemma (triple (wp p post) p post)
  = match p with
    | Assign x e -> ()
    | Seq p1 p2 ->
      wp_soundness p2 post;
      wp_soundness p1 (wp p2 post)
    | If e p1 p2 ->
      wp_soundness p1 post;
      wp_soundness p2 post
    | Repeat e p ->
      introduce forall s0. wp (Repeat e p) post s0 ==>
                      post (snd (run (Repeat e p) s0)) with
      introduce _ ==> _ with
        _ . (
          eliminate exists (inv:state -> prop).
                          inv s0 /\
                          (forall s. inv s ==> post s) /\
                          (forall s. inv s ==> wp p inv s)
          returns _
          with inv_props. (
            wp_soundness p inv;
            repeat e p inv
          )
      )
```

One could also prove that wp computes the weakest precondition, i.e., if triple p c q then forall s. p s ==> wp c q s, though we do not prove that formally here.

28.1.4 A Sample Program Proof

We now illustrate some sample proofs using our Hoare triples and wp function. To emphasize that Hoare triples provide an *abstract* way of reasoning about the execution of programs, we define the hoare p c q an alias for triple p c q marked with an attribute to ensure that F* and Z3 cannot reason directly about the underlying definition of triple—that would allow Z3 to find proofs by reasoning about the operational semantics directly, which we want to avoid , since it would not scale to larger programs. For more about the opaque_to_smt and reveal_opaque construct, please see *this section on opaque definitions*.

```
[@@"opaque_to_smt"]
let hoare p c q = triple p c q

let wp_hoare (p:program) (post:state -> prop)
```

(continues on next page)

(continued from previous page)

```
  : squash (hoare (wp p post) p post)
  = reveal_opaque (`%hoare) hoare;
    wp_soundness p post

let hoare_consequence (#p:program)
                      (#pre #post:state -> prop)
                      (pre_annot:state -> prop)
                      (_: squash (hoare pre p post))
                      (_: squash (forall s. pre_annot s ==> pre s))
  : squash (hoare pre_annot p post)
  = reveal_opaque (`%hoare) hoare
```

The lemmas above are just restatements of the `wp_soundness` and `consequence` lemmas that we've already proven. Now, these are the only two lemmas we have to reason about the `hoare p c q` predicate.

Next, we define some notation to make it a bit more convenient to write programs in our small language.

```
let ( := ) (x:var) (y:expr) = Assign x y

let add (e1 e2:expr) = EAdd e1 e2

let c (i:int) : expr = EConst i

let v (x:var) : expr = EVar x

let rec prog (p:list program { Cons? p })
  : program
  = match p with
    | [c] -> c
    | c::cs -> Seq c (prog cs)

let x = 0
let y = 1
let z = 2
```

Finally, we can build proofs of some simple, loop-free programs automatically by computing their `wp` using `wp_hoare` and applying `hoare_consequence` to get F* and Z3 to prove that the inferred WP is implied by the annotated precondition.

```
let swap = prog [ z := v x; x := v y; y := v z]
let proof_swap (lx ly:int)
  : squash (hoare (fun s -> read s x = lx /\ read s y = ly) swap
                  (fun s -> read s x = ly /\ read s y = lx))
  = hoare_consequence _ (wp_hoare swap _) ()
```

This recipe of computing verification conditions using WPs and then checking the computed WP against the annotated specification using a solver like Z3 is a very common and powerful pattern. In fact, as we'll see below, the methodology that we've developed here for our small imperative language is exactly what the F* typechecker does (at a larger scale and for the whole F* language) when checking an F* program.

28.2 The PURE Effect: A Dijkstra Monad for Pure Computations

F* provides a weakest precondition calculus for reasoning about pure computations. The calculus is based on a *Dijkstra Monad*, a construction first introduced in this paper. In this chapter, we will learn about Dijkstra Monad and its usage in specifying and proving pure programs in F*.

The first main difference in adapting the Hoare triples and weakest precondition computations that we saw earlier to the setting of F*'s functional language is that there are no global variables or mutable state (we'll see about how model mutable state in F*'s effect system later). Instead, each pure expression in F*'s *returns* a value and the postconditions that we will manipulate are predicates about these values, rather than state predicates.

To illustrate, we sketch the definition of pure WPs below.

```
WP c Q                      = Q c
WP (let x = e1 in e2) Q     = WP e1 (fun x -> WP e2 Q)
WP (if e then e1 else e2) Q = (e ==> WP e1 Q) /\ (~e ==> WP e2 Q)
```

- The WP of a constant `c` is just the postcondition `Q` applied to `c`.
- The WP of a `let` binding is a sequential composition of WPs, applied to the *values* returned by each subexpression
- The WP of a condition is the WP of each branch, weakened by the suitable branch condition, as before.

The F* type system internalizes and generalizes this WP construction to apply it to all F* terms. The form this takes is as a computation type in F*, `PURE a wp`, where in `prims.fst`, `PURE` is defined as an F* primitive effect with a signature as shown below–we'll see much more of the `new_effect` syntax as we look at user-defined effects in subsequent chapters; for now, just see it as a reserved syntax in F* to introduce a computation type constructor.

```
new_effect PURE (a:Type) (w:wp a) { ... }
```

where

```
let pre = Type0
let post (a:Type) = a -> Type0
let wp (a:Type) = post a -> pre
```

A program `e` of type `PURE a wp` is a computation which

- Is defined only when `wp (fun _ -> True)` is valid
- If `wp post` is valid, then `e` terminates without any side effects and returns a value `v:a` satisfying `post v`.

Notice that `wp a` is the type of a function transforming postconditions (`a -> Type0`) to preconditions (`Type0`).[1] The `wp` argument is also called an *index* of the `PURE` effect.[2]

The return operator for `wp a` is shown below: it is analogous to the `WP c Q` and `WP x Q` rules for variables and constants that we showed earlier:

[1] It is also possible to define `post a = a -> prop` and `pre = prop`. However, the F* libraries for pure WPs using `Type0` instead of `prop`, so we remain faithful to that here.

[2] Dijkstra monads are also related to the continuation monad. Continuation monad models Continuation Passing Style programming, where the control is passed to the callee explicitly in the form of a continuation. For a result type `r`, the continuation monad is defined as follows:

```
let cont (r:Type) (a:Type) = (a -> r) -> r  // (a -> r) is the continuation
let return #r (#a:Type) (x:a) : cont r a = fun k -> k x
let bind #r (#a #b:Type) (f:cont r a) (g:a -> cont r b)
  : cont r b
  =  fun k -> f (fun x -> g x k)
```

If we squint a bit, we can see that the `wp` monad we defined earlier, is nothing but a continuation into `Type0`, i.e., `wp a = cont Type0 a` (or `cont prop a`, if one prefers to use `prop`).

```
let return_wp (#a:Type) (x:a) : wp a = fun post -> post x
```

The bind operator for `wp a` is analogous to the rule for sequencing WPs, i.e., the rule for `WP (let x = e1 in e2) Q` above:

```
let bind_wp (#a #b:Type) (wp1:wp a) (wp2:a -> wp b)
  : wp b
  = fun post -> wp1 (fun x -> wp2 x post)
```

Finally, analogous to the WP rule for conditionals, one can write a combinator for composing `wp a` in a branch:

```
let if_then_else_wp (#a:Type) (b:bool) (wp1 wp2:wp a)
  : wp a
  = fun post -> if b then wp1 post else wp2 post
```

This is the essence of the Dijkstra monad construction for pure programs: the rule for computing weakest preconditions for a computation *returning* a value `x` is `return_wp`; the rule for computing the WP of the sequential composition of terms is the sequential composition of WPs using `bind_wp`; the rule for computing the WP of a conditional term is the conditional composition of WPs using `if_then_else_wp`.

If fact, if one thinks of pure computations as the identity monad, `tot a` as shown below:

```
let tot (a:Type) = a
let return_tot (#a:Type) (x:a) : tot a = x
let bind_tot (#a #b:Type) (x:tot a) (y:a -> tot b)
  : tot b
  = let v = x in y v
```

then the parallel between the `tot` monad and `wp` becomes even clearer—the WP analog of `return_tot` is `return_wp` and of `bind_tot` is `bind_wp`.

It turns out that `wp a` (for monotonic weakest preconditions) is itself a monad, as shown below by a proof of the monad laws:

```
(* A monotonic WP maps stronger postconditions to stronger preconditions *)
let monotonic (#a:Type) (wp:wp a) =
  forall (p q:post a). (forall x. p x ==> q x) ==> (wp p ==> wp q)

let mwp (a:Type) = w:wp a { monotonic w }

(* An equivalence relation on WPs *)
let ( =~= ) (#a:Type) (wp1 wp2:wp a)
  : prop
  = forall post. wp1 post <==> wp2 post

(* The three monad laws *)
let left_identity (a b:Type) (x:a) (wp:a -> mwp a)
  : Lemma (bind_wp (return_wp x) wp =~= wp x)
  = ()

let right_identity (a b:Type) (wp:mwp a)
  : Lemma (wp =~= (bind_wp wp return_wp))
  = ()
```

(continues on next page)

(continued from previous page)

```
let associativity (a b c:Type) (wp1:mwp a) (wp2:a -> mwp b) (wp3:b -> mwp c)
  : Lemma (bind_wp wp1 (fun x -> bind_wp (wp2 x) wp3) =~=
           bind_wp (bind_wp wp1 wp2) wp3)
  = ()
```

28.3 PURE and Tot

When typechecking a program, F* computes a weakest precondition which characterizes a necessary condition for the program to satisfy all its typing constraints. This computed weakest precondition is usually hidden from the programmer, but if you annotate your program suitably, you can get access to it, as shown in the code snippet below:

```
let square (n:int)
  : PURE nat (as_pure_wp #nat (fun q -> n*n >= 0 /\ q (n * n)))
  = n * n
```

The type says that `square n` is a pure function, which for any postcondition `q:nat -> prop`,

- Is defined only when `n * n >= 0` and when `q (n * n)` is valid
- And returns a value `m:nat` satisfying `q m`

Let's look at another example:

```
let maybe_incr (b:bool) (x:int)
  : PURE int (as_pure_wp (if_then_else_wp b
                                        (bind_wp (return_wp (x + 1)) (fun y -> return_wp␣
↪y))
                                        (return_wp x)))
  = if b
    then let y = x + 1 in y
    else x
```

Notice how the `wp` index of `PURE` mirrors the structure of the computation itself—it starts with an `if_then_else_wp`, then in the first branch, uses a `bind_wp` followed by a return; and in the else branch it returns `x`.

As such, the wp-index simply "lifts" the computation into a specification in a form amenable to logical reasoning, e.g., using the SMT solver. For pure programs this may seem like overkill, since the pure term itself can be reasoned about directly, but when the term contains non-trivial typing constraints, e.g., such as those that arise from refinement type checking, lifting the entire program into a single constraint structures and simplifies logical reasoning.

Of course, one often writes specifications that are more abstract than the full logical lifting of the program, as in the example below, which says that to prove `post` of the return value, the precondition is to prove `post` on all `y >= x`. This is a valid, although weaker, characterization of the function's return value.

```
let maybe_incr2 (b:bool) (x:int)
  : PURE int (as_pure_wp (fun post -> forall (y:int). y >= x ==> post y))
  = if b
    then let y = x + 1 in y
    else x
```

The `PURE` computation type comes with a built-in weakening rule. In particular, if a term is computed to have type `PURE a wp_a` and it is annotated to have type `PURE b wp_b`, then F* does the following:

1. It computes a constraint `p :  a -> Type0`, which is sufficient to prove that `a` is a subtype of `b`, e.g., is `a = int` and `b = nat`, the constraint `p` is `fun (x:int) -> x >= 0`.
2. Next, it strengthens `wp_a` to assert that the returned value validates the subtyping constraints `p x`, i.e., it builds `assert_wp wp_a p`, where

   ```
   let assert_wp (#a:Type) (w:wp a) (p: a -> Type0)
     : wp (x:a{ p x })
     = fun post -> w (fun (x:a) -> p x /\ post x)
   ```

3. Finally, it produces the verification condition `stronger_wp #b wp_b (assert_wp wp_a p)`, where `stronger_wp` is defined as shown below:

   ```
   let stronger_wp (#a:Type) (wp1 wp2:wp a) : prop =
      forall post. wp1 post ==> wp2 post
   ```

   That is, for any postcondition `post`, the precondition `wp_b post` implies the original precondition `wp_a post` as well as the subtyping constraint `p x`. This matches the intuition about preconditions that we built earlier: it is always sound to require more in the precondition.

Thus, when we have `e:PURE a wp` in F*, the `wp` is *a* predicate transformer for `e`, not necessarily the weakest one.

Of course, even `maybe_incr2` is not particularly idiomatic in F*. One would usually annotate a program with a refinement type, such as the one below:

```
let maybe_incr_tot (b:bool) (x:int)
  : Tot (y:int { y >= x })
  = if b
    then let y = x + 1 in y
    else x
```

Internally to the compiler, F* treats `Tot t` as the following instance of `PURE`:

```
Tot t = PURE t (fun post -> forall (x:t). post x)
```

Once `Tot t` is viewed as just an instance of `PURE`, checking if a user annotation `Tot t` is stronger than the inferred specification of a term `PURE a wp` is just as explained before.

28.3.1 `Pure`: Hoare Triples for `PURE`

Although specification are easier to *compute* using WPs, they are more natural to read and write when presented as Hoare triples, with a clear separation between precondition and postconditions. Further, when specifications written as Hoare triples naturally induce monotonic WPs.

F* provides an effect abbreviation called `Pure` for writing and typechecking Hoare-style specifications for pure programs, and is defined as shown below in `prims.fst`:

```
effect Pure (a:Type) (req:Type0) (ens:a -> Type0) =
       PURE a (fun post -> req /\ (forall x. ens x ==> post x))
```

The signature of `Pure` is `Pure a req ens`, where `req` is the precondition and `ens:a -> Type0` is the postcondition. Using `Pure`, we can write the `factorial` function we saw at the top of this chapter—F* infers a `PURE a wp` type for it, and relates it to the annotated `Pure int req ens` type, proving that the latter has a stronger precondition and weaker postcondition.

One may wonder when one should write specifications using the notation `x:a -> Pure b req ens` versus `x:a{req} -> Tot (y:b { ens y })`. The two styles are closely related and choosing between them is mostly a matter of taste.

As you have seen, until this point in the book, we have not used Pure a req ens at all. However, when a function has many pre and postconditions, it is sometimes more convenient to use the Pure a req ens notation, rather than stuffing all the constraints in refinement types.

28.4 GHOST and DIV

Just as PURE is an wp-indexed refinement of Tot, F* provides two more primitive wp-indexed effects:

- GHOST (a:Type) (w:wp a) is a refinement of GTot a
- DIV (a:Type) (w:wp a) is a refinement of Dv a

That is, F* uses the GHOST effect to infer total correctness WPs for ghost computations, where, internally, GTot a is equivalent to GHOST a (fun post -> forall x. post x)

Likewise, F* uses the DIV effect to infer *partial correctness* WPs for potentially non-terminating computations, where, internally, Dv a is equivalent to DIV a (fun post -> forall x. post x).

As with Tot and PURE, F* automatically relates GTot and GHOST computations, and Dv and DIV computations. Further, the effect ordering Tot < Dv and Tot < GTot extends to PURE < DIV and PURE < GHOST as well.

The prims.fst library also provides Hoare-triple style abbreviations for GHOST and DIV, i.e.,

```
effect Ghost a req ens = GHOST a (fun post -> req /\ (forall x. ens x /\ post x))
effect Div a req ens = DIV a (fun post -> req /\ (forall x. ens x /\ post x))
```

These Hoare-style abbreviations are more convenient to use than their more primitive WP-based counterparts.

The tradeoffs of using Ghost vs. GTot or Div vs. Dv are similar to those for Pure vs Tot—it's mostly a matter of taste. In fact, there are relatively few occurrences of Pure, Ghost, and Div in most F* codebases. However, there is one important exception: Lemma.

28.5 The Lemma abbreviation

We can finally unveil the definition of the Lemma syntax, which we introduced as a syntactic shorthand in *an early chapter*. In fact, Lemma is defined in prims.fst as follows:

```
effect Lemma (a: eqtype_u)
             (pre: Type)
             (post: (squash pre -> Type))
             (smt_pats: list pattern) =
       Pure a pre (fun r -> post ())
```

That is, Lemma is an instance of the Hoare-style refinement of pure computations Pure a req ens. So, when you write a proof term and annotate it as e : Lemma (requires pre) (ensures post), F* infers a specification for e : PURE a wp, and then, as with all PURE computations, F* tries to check that the annotated Lemma specification has a stronger WP-specification than the computed weakest precondition.

Of course, F* still includes syntactic sugar for Lemma, e.g., Lemma (requires pre) (ensures post) is desugared to Lemma unit pre (fun _ -> post) []. The last argument of a lemma, the smt_pats are used to introduce lemmas to the SMT solver for proof automation—a *later chapter* covers that in detail.

Finally, notice the type of the post, which assumes squash pre as an argument–this is what allows the ensures clause of a Lemma to assume that what was specified in the `requires clause.

Part VI

Tactics and Metaprogramming with Meta-F*

This part of the book is still heavily under construction

So far, we have mostly relied on the SMT solver to do proofs in F*. This works rather well: we got this far, after all! However, sometimes, the SMT solver is really not able to complete our proof, or takes too long to do so, or is not *robust* (i.e. works or fails due to seemingly insignificant changes).

This is what Meta-F* was originally designed for. It provides the programmer with more control on how to break down a proof and guide the SMT solver towards finding a proof by using *tactics*. Moreover, a proof can be fully completed within Meta-F* without using the SMT solver at all. This is the usual approach taken in other proof assistants (such as Lean, Coq, or Agda), but it's not the preferred route in F*: we will use the SMT for the things it can do well, and mostly write tactics to "preprocess" obligations and make them easier for the solver, thus reducing manual effort.

Meta-F* also allows for *metaprogramming*, i.e. generating programs (or types, or proofs, ...) automatically. This should not be surprising to anyone already familiar with proof assistants and the Curry-Howard correspondence. There are however some slight differences between tactic-based proofs and metaprogramming, and more so in F*, so we will first look at automating proofs (i.e. tactics), and then turn to metaprogramming (though we use the generic name "metaprogram" for tactics as well).

In summary, when the SMT solver "just works" we usually do not bother writing tactics, but, if not, we still have the ability to roll up our sleeves and write explicit proofs.

Speaking of rolling up our sleeves, let us do just that, and get our first taste of tactics.

CHAPTER

TWENTYNINE

AN OVERVIEW OF TACTICS

In this chapter, we quickly introduce several of the main concepts underlying Meta-F* and its use in writing tactics for proof automation. The goal is to get you quickly up to speed on basic uses of tactics. Subsequent chapters will revisit the concepts covered here in more detail, introduce more advanced aspects of Meta-F*, and show them at use in several case studies.

29.1 Decorating assertions with tactics

As you know already, F* verifies programs by computing verification conditions (VCs) and calling an SMT solver (Z3) to prove them. Most simple proof obligations are handled completely automatically by Z3, and for more complex statements we can help the solver find a proof via lemma calls and intermediate assertions. Even when using lemma calls and assertions, the VC for a definition is sent to Z3 in one single piece (though *SMT queries can be split via an option*.). This "monolithic" style of proof can become unwieldy rapidly, particularly when the solver is being pushed to its limits.

The first ability Meta-F* provides is allowing to attach specific tactics to assertions. These tactics operate on the "goal" that we want to prove, and can "massage" the assertion by simplifying it, splitting it into several sub-goals, tweaking particular SMT options, etc.

For instance, let us take the the following example, where we want to guarantee that `pow2 x` is less than one million given that `x` is at most `19`. One way of going about this proof is by noting that `pow2` is an increasing function, and that `pow2 19` is less than one million, so we try to write something like this:

```
let pow2_bound_19 (x:nat{x <= 19}) : Lemma (pow2 x < 1000000) =
  assert (forall (x y : nat). x <= y ==> pow2 x <= pow2 y);
  assert (pow2 19 ==  524288);
  assert (pow2 x < 1000000);
  ()
```

Sadly, this doesn't work. First of all, Z3 cannot automatically prove that `pow2` is increasing, but that is to be expected. We could prove this by a straightforward induction. However, we only need this fact for `x` and `19`, so we can simply call `FStar.Math.Lemmas.pow2_le_compat` from the library:

```
let pow2_bound_19' (x:nat{x <= 19}) : Lemma (pow2 x < 1000000) =
  FStar.Math.Lemmas.pow2_le_compat 19 x;
  assert (pow2 19 == 524288);
  assert (pow2 x < 1000000);
  ()
```

Now, the second assertion fails. Z3 will not, with the default fuel limits, unfold pow2 enough times to compute `pow2 19` precisely. (You can read more about how F* *uses "fuel" to control the SMT solver's ability to unfold recursive* *definitions*.) Here we will use our first call into Meta-F*: via the by keyword, we can attach a tactic to an assertion. In this case, we'll ask Meta-F* to `compute()` over the goal, simplifying as much as it can via F*'s normalizer, like this:

```
let pow2_bound_19'' (x:nat{x <= 19}) : Lemma (pow2 x < 1000000) =
  FStar.Math.Lemmas.pow2_le_compat 19 x;
  assert (pow2 19 == 524288) by compute ();
  assert (pow2 x < 1000000);
  ()
```

Now the lemma verifies! Meta-F* reduced the proof obligation into a trivial equality. Crucially, however, the `pow2 19 == 524288` shape is kept as-is in the postcondition of the assertion, so we can make use of it! If we were just to rewrite the assertion into `524288 == 524288` that would not be useful at all.

How can we know what Meta-F* is doing? We can use the `dump` tactic to print the state of the proof after the call to `compute()`.

```
let pow2_bound_19''' (x:nat{x <= 19}) : Lemma (pow2 x < 1000000) =
  FStar.Math.Lemmas.pow2_le_compat 19 x;
  assert (pow2 19 == 524288) by (compute (); dump "after compute");
  assert (pow2 x < 1000000);
  ()
```

With this version, you should see something like:

```
Goal 1/1
x: x: nat{x < 20}
p: pure_post unit
uu___: forall (pure_result: unit). pow2 x < 1000000 ==> p pure_result
pure_result: unit
uu___'0: pow2 x <= pow2 19
--------------------------------------------------------------------------------
squash (524288 == 524288)
(*?u144*) _
```

as output from F* (or in the goals buffer if you are using emacs with fstar-mode.el). The `print` primitive can also be useful.

A "goal" is some proof obligation that is yet to be solved. Meta-F* allows you to capture goals (e.g. via `assert..by`), modify them (such as with `compute`), and even to completely solve them. In this case, we can solve the goal (without Z3) by calling `trivial()`, a helper tactic that discharges trivial goals (such as trivial equalities).

```
let pow2_bound_19'''' (x:nat{x <= 19}) : Lemma (pow2 x < 1000000) =
  FStar.Math.Lemmas.pow2_le_compat 19 x;
  assert (pow2 19 == 524288) by (
    compute ();
    trivial ();
    qed ()
   );
  assert (pow2 x < 1000000);
  ()
```

If you `dump` the state just after the `trivial()` call, you should see no more goals remain (this is what `qed()` checks).

Note: Meta-F* does not yet allow a fully interactive style of proof, and hence we need to re-check the entire proof after every edit. We hope to improve this soon.

There is still the "rest" of the proof, namely that `pow2 x < 1000000` given the hypothesis and the fact that the assertion holds. We call this *skeleton* of the proof, and it is (by default) not handled by Meta-F*. In general, we only use tactics on those assertions that are particularly hard for the SMT solver, but leave all the rest to it.

29.2 The `Tac` effect

Note: Although we have seen a bit about *monads and computational effects in a previous chapter*, we have yet to fully describe F*'s effect system. So, some of what follows may be a bit confusing. However, you don't need to fully understand how the `Tac` effect is implemented to use tactics. Feel free to skip ahead, if this section doesn't make much sense to you.

What, concretely, are tactics? So far we've written a few simple ones, without too much attention to their structure.

Tactics and metaprograms in F* are really just F* terms, but *in a particular effect*, namely `Tac`. To construct interesting metaprograms, we have to use the set of *primitives* provided by Meta-F*. Their full list is in the `FStar.Tactics.Builtins` module. So far, we have actually not used any primitive directly, but only *derived* metaprograms present in the standard library.

Internally, `Tac` is implemented via a combination of 1) a state monad, over a `proofstate`, 2) exceptions and 3) divergence or non-termination. The state monad is used to implicitly carry the proofstate, without us manually having to handle all goals explicitly. Exceptions are a useful way of doing error handling. Any declared exception can be `raise`'d within a metaprogram, and the `try..with` construct works exactly as for normal programs. There are also `fail`, `catch` and `recover` primitives.

Metaprograms cannot be run directly. This is needed to retain the soundness of pure computations, in the same way that stateful and exception-raising computations are isolated from the `Pure` fragment (and from each other). Metaprograms can only be used where F* expects them , such as in an `assert..by` construct. Here, F* will run the metaprogram on an initial proofstate consisting (usually) of a single goal, and allow the metaprogram to modify it.

To guarantee soundness, i.e., that metaprograms do not prove false things, all of the primitives are designed to perform small and correct modifications of the goals. Any metaprogram constructed from them cannot do anything to the proofstate (which is abstract) except modifying it via the primitives.

Having divergence as part of the `Tac` effect may seem a bit odd, since allowing for diverging terms usually implies that one can form a proof of false, via a non-well-founded recursion. However, we should note that this possible divergence happens at the *meta* level. If we call a divergent tactic, F* will loop forever waiting for it to finish, never actually accepting the assertion being checked.

As you know, F* already has exceptions and divergence. All `Dv` and `Ex` functions can readily be used in Meta-F* metaprograms, as well as all `Tot` and `Pure` functions. For instance, you can use all of the `FStar.List.Tot` module if your metaprogram uses lists.

29.3 Goals

Essentially, a Meta-F* tactic manipulates a *proofstate*, which is essentially a set of *goals*. Tactic primitives usually work on the goals, for example by simplifying (like `compute()`) or by breaking them down into smaller *sub*-goals.

When proving assertions, all of our goals will be of the shape `squash phi`, where `phi` is some logical formula we must prove. One way to break down a goal into subparts is by using the `mapply` tactic, which attempts to prove the goal by instantiating the given lemma or function, perhaps adding subgoals for the hypothesis and arguments of the lemma. This "working backwards" style is very common in tactics frameworks.

For instance, we could have proved the assertion that `pow2 x <= pow2 19` in the following way:

```
assert (pow2 x <= pow2 19) by (mapply (`FStar.Math.Lemmas.pow2_le_compat));
```

This reduces the proof of `pow2 x <= pow2 19` to `x <= 19` (the precondition of the lemma), which is trivially provably by Z3 in this context. Note that we do not have to provide the arguments to the lemma: they are inferred by F* through *unification*. In a nutshell, this means F* finds there is an obvious instantiation of the arguments to make the postcondition of the lemma and the current assertion coincide. When some argument is *not* found via unification, Meta-F* will present a new goal for it.

This style of proof is more *surgical* than the one above, since the proof that `pow2 x <= pow2 19` does not "leak" into the rest of the function. If the proof of this assertion required several auxiliary lemmas, or a tweak to the solver's options, etc, this kind of style can pay off in robustness.

Most tactics works on the *current* goal, which is the first one in the proofstate. When a tactic reduces a goal `g` into `g1,...,gn`, the new `g1,..,gn` will (usually) be added to the beginning of the list of goals.

In the following simplified example, we are looking to prove `s` from `p` given some lemmas. The first thing we do is apply the `qr_s` lemma, which gives us two subgoals, for `q` and `r` respectively. We then need to proceed to solve the first goal for `q`. In order to isolate the proofs of both goals, we can `focus` on the current goal making all others temporarily invisible. To prove `q`, we then just use the `p_r` lemma and obtain a subgoal for `p`. This one we will just just leave to the SMT solver, hence we call `smt()` to move it to the list of SMT goals. We prove `r` similarly, using `p_r`.

```
assume val p : prop
assume val q : prop
assume val r : prop
assume val s : prop

assume val p_q : unit -> Lemma (requires p) (ensures q)
assume val p_r : squash p -> Lemma r
assume val qr_s : unit -> Lemma (q ==> r ==> s)

let test () : Lemma (requires p) (ensures s) =
  assert s by (
    mapply (`qr_s);
    focus (fun () ->
      mapply (`p_q);
      smt());
    focus (fun () ->
      mapply (`p_r);
      smt());
    ()
  )
```

Once this tactic runs, we are left with SMT goals to prove `p`, which Z3 discharges immediately.

Note that `mapply` works with lemmas that ensure an implication, or that have a precondition (`requires`/`ensures`), and even those that take a squashed proof as argument. Internally, `mapply` is implemented via the `apply_lemma` and `apply` primitives, but ideally you should not need to use them directly.

Note, also, that the proofs of each part are completely isolated from each other. It is also possible to prove the `p_gives_s` lemma by calling the sublemmas directly, and/or adding SMT patterns. While that style of proof works, it can quickly become unwieldy.

29.4 Quotations

In the last few examples, you might have noted the backticks, such as in ``(`FStar.Math.Lemmas.pow2_le_compat)``. This is a *quotation*: it represents the *syntax* for this lemma instead of the lemma itself. It is called a quotation since the idea is analogous to the word “sun” being syntax representing the sun.

A quotation always has type `term`, an abstract type representing the AST of F*.

Meta-F* also provides *antiquotations*, which are a convenient way of modifying an existing term. For instance, if `t` is a term, we can write ``` `(1 + `#t) ``` to form the syntax of “adding 1” to `t`. The part inside the antiquotation (`` `# ``) can be anything of type `term`.

Many metaprogramming primitives, however, do take a `term` as an argument to use it in proof, like `apply_lemma` does. In this case, the primitives will typecheck the term in order to use it in proofs (to make sure that the syntax actually corresponds to a meaningful well-typed F* term), though other primitives, such as `term_to_string`, won’t typecheck anything.

We will see ahead that quotations are just a convenient way of constructing syntax, instead of doing it step by step via `pack`.

29.5 Basic logic

Meta-F* provides some predefined tactics to handle “logical” goals.

For instance, to prove an implication `p ==> q`, we can “introduce” the hypothesis via `implies_intro` to obtain instead a goal for `q` in a context that assumes `p`.

For experts in Coq and other provers, this tactic is simply called `intro` and creates a lambda abstraction. In F* this is slightly more contrived due to squashed types, hence the need for an `implies_intro` different from the `intro`, explained ahead, that introduces a binder.

Other basic logical tactics include:

- `forall_intro`: for a goal `forall x. p`, introduce a fresh `x` into the context and present a goal for `p`.
- `l_intros`: introduce both implications and foralls as much as possible.
- `split`: split a conjunction (`p /\ q`) into two goals
- `left`/`right`: prove a disjunction `p \/ q` by proving `p` or `q`
- `assumption`: prove the goal from a hypothesis in the context.
- `pose_lemma`: given a term `t` representing a lemma call, add its postcondition to the context. If the lemma has a precondition, it is presented as a separate goal.

See the FStar.Tactics.Logic module for more.

29.6 Normalizing and unfolding

We have previously seen `compute()`, which blasts a goal with F*'s normalizer to reduce it into a *normal form*. We sometimes need a bit more control than that, and hence there are several tactics to normalize goals in different ways. Most of them are implemented via a few configurable primitives (you can look up their definitions in the standard library)

- `compute()`: calls the normalizer with almost all steps enabled
- `simpl()`: simplifies logical operations (e.g. reduces `p /\ True` to `p`).
- `whnf()` (short for "weak head normal form"): reduces the goal until its "head" is evident.
- `` unfold_def `t ``: unfolds the definition of the name `t` in the goal, fully normalizing its body.
- `trivial()`: if the goal is trivial after normalization and simplification, solve it.

The `norm` primitive provides fine-grained control. Its type is `list norm_step -> Tac unit`. The full list of `norm_step` s can be found in the `FStar.Pervasives` module, and it is the same one available for the `norm` marker in `Pervasives` (beware of the name clash between `Tactics.norm` and `Pervasives.norm`!).

29.7 Inspecting and building syntax

As part of automating proofs, we often need to inspect the syntax of the goal and the hypotheses in the context to decide what to do. For instance, instead of blindly trying to apply the `split` tactic (and recovering if it fails), we could instead look at the *shape* of the goal and apply `split` only if the goal has the shape `p1 /\ p2`.

Note: inspecting syntax is, perhaps obviously, not something we can just do everywhere. If a function was allowed to inspect the syntax of its argument, it could behave differently on `1+2` and `3`, which is bad, since `1+2 == 3` in F*, and functions are expected to map equal arguments to the same result. So, for the most part, we cannot simply turn a value of type `a` into its syntax. Hence, quotations are *static*, they simply represent the syntax of a term and one cannot turn values into terms. There is a more powerful mechanism of *dynamic quotations* that will be explained later, but suffice it to say for now that this can only be done in the `Tac` effect.

As an example, the `cur_goal()` tactic will return a value of type `typ` (an alias for `term` indicating that the term is really the representation of an F* type) representing the syntax of the current goal.

The `term` type is *abstract*: it has no observable structure itself. Think of it as an opaque "box" containing a term inside. A priori, all that can be done with a `term` is pass it to primitives that expect one, such as `tc` to type-check it or `norm_term` to normalize it. But none of those give us full, programatic access to the structure of the term.

That's where the `term_view` comes in: following a classic idea introduced by Phil Wadler, there is function called `inspect` that turns a `term` into a `term_view`. The `term_view` type resembles an AST, but crucially it is not recursive: its subterms have type `term` rather than `term_view`.

Listing 1: Part of the `term_view` type.

```
noeq
type term_view =
  | Tv_FVar   : v:fv -> term_view
  | Tv_App    : hd:term -> a:argv -> term_view
  | Tv_Abs    : bv:binder -> body:term -> term_view
  | Tv_Arrow  : bv:binder -> c:comp -> term_view
  ...
```

The `inspect` primitves "peels away" one level of the abstraction layer, giving access to the top-level shape of the term.

The Tv_FVar node above represents (an ocurrence of) a global name. The fv type is also abstract, and can be viewed as a name (which is just list string) via inspect_fv.

For instance, if we were to inspect `qr_s (which we used above) we would obtain a Tv_FVar v, where inspect_fv v is something like ["Path"; "To"; "Module"; "qr_s"], that is, an "exploded" representation of the fully-qualified name Path.To.Module.qr-s.

Every syntactic construct (terms, free variables, bound variables, binders, computation types, etc) is modeled abstractly like term and fv, and has a corresponding inspection functions. A list can be found in FStar.Reflection.Builtins.

If the inspected term is an application, inspect will return a Tv_App f a node. Here f is a term, so if we want to know its structure we must recursively call inspect on it. The a part is an *argument*, consisting of a term and an argument qualifier (aqualv). The qualifier specifies if the application is implicit or explicit.

Of course, in the case of a nested application such as f x y, this is nested as (f x) y, so inspecting it would return a Tv_App node containing f x and y (with a Q_Explicit qualifier). There are some helper functions defined to make inspecting applications easier, like collect_app, which decompose a term into its "head" and all of the arguments the head is applied to.

Now, knowing this, we would then like a function to check if the goal is a conjunction. Naively, we need to inspect the goal to check that it is of the shape squash ((/\) a1 a2), that is, an application with two arguments where the head is the symbol for a conjunction, i.e. (/\). This can already be done with the term_view, but is quite inconvenient due to there being *too much* information in it.

Meta-F* therefore provides another type, formula, to represent logical formulas more directly. Hence it suffices for us to call term_as_formula and match on the result, like so:

```
(* Check if a given term is a conjunction, via term_as_formula. *)
let isconj_t (t:term) : Tac bool =
  match term_as_formula t with
  | And _ _ -> true
  | _ -> false

(* Check if the goal is a conjunction. *)
let isconj () : Tac bool = isconj_t (cur_goal ())
```

The term_as_formula function, and all others that work on syntax, are defined in "userspace" (that is, as library tactics/metaprograms) by using inspect.

Listing 2: Part of the formula type.

```
noeq
type formula =
  | True_  : formula
  | False_ : formula
  | And    : term -> term -> formula
  | Or     : term -> term -> formula
  | Not    : term -> formula
  | Implies: term -> term -> formula
  | Forall : bv -> term -> formula
  ...
```

Note: For experts: F* terms are (internally) represented with a locally-nameless representation, meaning that variables do not have a name under binders, but a de Bruijn index instead. While this has many advantages, it is likely to be counterproductive when doing tactics and metaprogramming, hence inspect *opens* variables when it traverses a binder, transforming the term into a fully-named representation. This is why inspect is effectul: it requires freshness to avoid name clashes. If you prefer to work with a locally-nameless representation, and avoid the effect label, you can use `inspect_ln` instead (which will return `Tv_BVar` nodes instead of `Tv_Var` ones).

Dually, a `term_view` can be transformed into a `term` via the `pack` primitive, in order to build the syntax of any term. However, it is usually more comfortable to use antiquotations (see above) for building terms.

29.8 Usual gotchas

- The `smt` tactic does *not* immediately call the SMT solver. It merely places the current goal into the "SMT Goal" list, all of which are sent to the solver when the tactic invocation finishes. If any of these fail, there is currently no way to "try again".
- If a tactic is natively compiled and loaded as a plugin, editing its source file may not have any effect (it depends on the build system). You should recompile the tactic, just delete its object file, or use the F* option `--no_plugins` to run it via the interpreter temporarily,
- When proving a lemma, we cannot just use `_ by ...` since the expected type is just `unit`. Workaround: assert the postcondition again, or start without any binder.

29.9 Coming soon

- Metaprogramming
- Meta arguments and typeclasses
- Plugins (efficient tactics and metaprograms, `--codegen Plugin` and `--load`)
- Tweaking the SMT options
- Automated coercions inspect/pack
- `e <:  C by ...`
- Tactics can be used as steps of calc proofs.
- Solving implicits (Steel)

Part VII

Pulse: Proof-oriented Programming in Concurrent Separation Logic

Many F* projects involve building domain-specific languages with specialized programming and proving support. For example, Vale supports program proofs for a structured assembly language; Low* provides effectful programming in F* with a C-like memory model; EverParse is a DSL for writing low-level parsers and serializers. Recently, F* has gained new features for building DSLs embedded in F* with customized syntax, type checker plugins, extraction support, etc., with *Pulse* as a showcase example of such a DSL.

Pulse is a new programming language embedded in F*, inheriting many of its features (notably, it is higher order and has dependent types), but with built-in support for programming with mutable state and concurrency, with specifications and proofs in Concurrent Separation Logic.

As a first taste of Pulse, here's a function to increment a mutable integer reference.

```
fn incr (x:ref int)
requires pts_to x 'i
ensures pts_to x ('i + 1)
{
    let v = !x;
    x := v + 1;
}
```

And here's a function to increment two references in parallel.

```
fn par_incr (x y:ref int)
requires pts_to x 'i ** pts_to y 'j
ensures pts_to x ('i + 1) ** pts_to y ('j + 1)
{
   par (fun _ -> incr x)
       (fun _ -> incr y)
}
```

You may not have heard about separation logic before—but perhaps these specifications already make intuitive sense to you. The type of `incr` says that if "x points to 'i" initially, then when `incr` returns, "x points to 'i + 1"; while `par_incr` increments the contents of `x` and `y` in parallel by using the `par` combinator.

Concurrent separation logic is an active research area and there are many such logics out there, all with different tradeoffs. Pulse's logic is based on a logic called PulseCore, formalized entirely within F*—you can find the formalization here. Proofs of programs in Pulse's surface language correspond to proofs of correctness in the PulseCore program logic. But, you should not need to know much about how the logic is formalized to use Pulse effectively. We'll start from the basics and explain what you need to know about concurrent separation logic to start programming and proving in Pulse. Additionally, Pulse is an extension of F*, so all you've learned about F*, lemmas, dependent types, refinement types, etc. will be of use again.

Note: Why is it called Pulse? Because it grew from a prior logic called Steel, and one of the authors and his daughter are big fans of a classic reggae band called Steel Pulse. We wanted a name that was softer than Steel, and, well, a bit playful. So, Pulse!

CHAPTER

THIRTY

GETTING UP AND RUNNING WITH CODESPACES

There are three main ways of running Pulse, roughly sorted in increasing order of difficulty.

The easiest way of using Pulse is with Github Codespaces. With a single click, you can get a full-fledged IDE (VS Code) running in your browser already configured with F* and Pulse.

You can also run Pulse inside a container locally, for a similar 1-click setup that is independent of Github.

Finally, you can also extract a Pulse release tarball and run the binaries directly in your system.

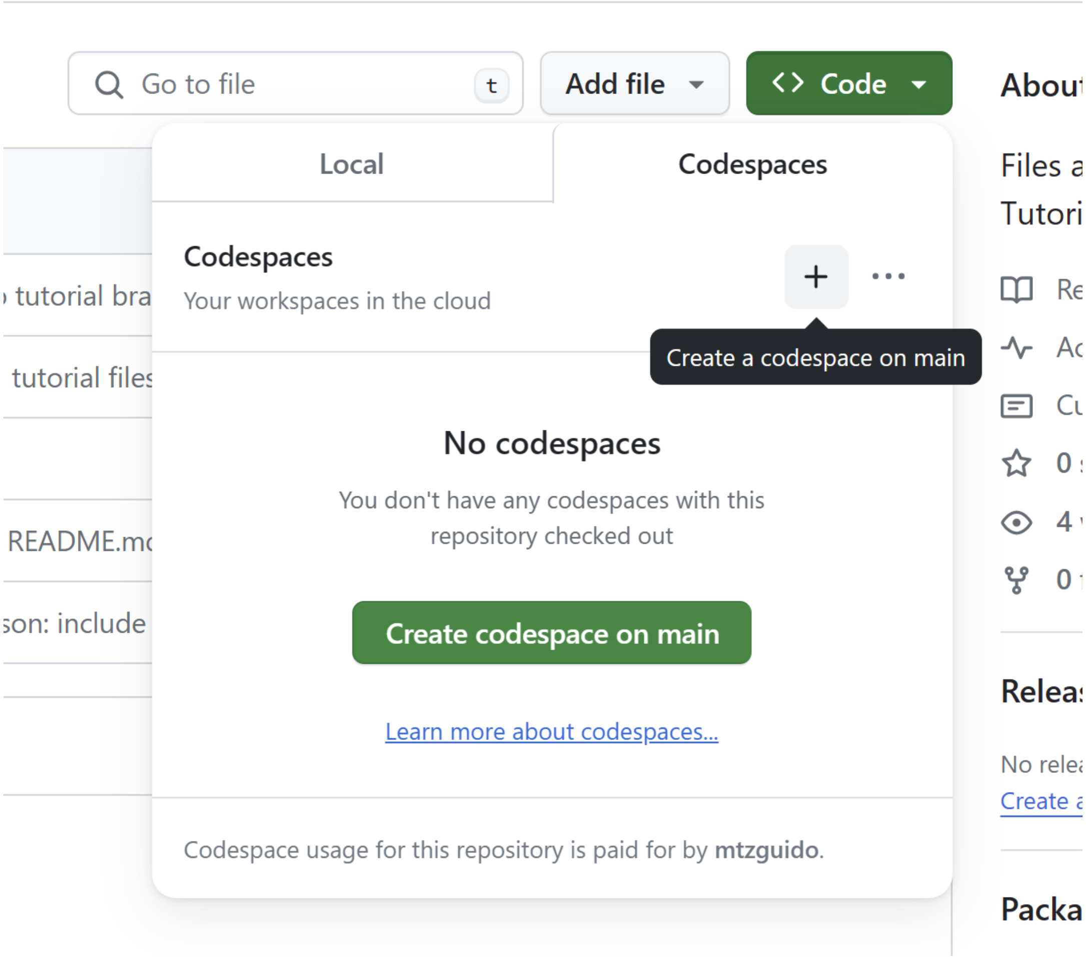
(Building from source is not well-documented yet.)

Note: Unlike the pure F* parts of this tutorial, Pulse code does not yet work in the online playground. Use one of the methods described below to try the examples in this part of the book.

You can find all the source files associated with each chapter in this folder, in files named `PulseTutorial.*.fst`.

30.1 Creating a Github Codespace

To do so, go to the this repository and click on the '<>Code' button, then select 'Create codespace on main'. This will use the Dev Container definition in the *.devcontainer* directory to set up container where F* and Pulse can run in a reproducible manner.



Codespaces

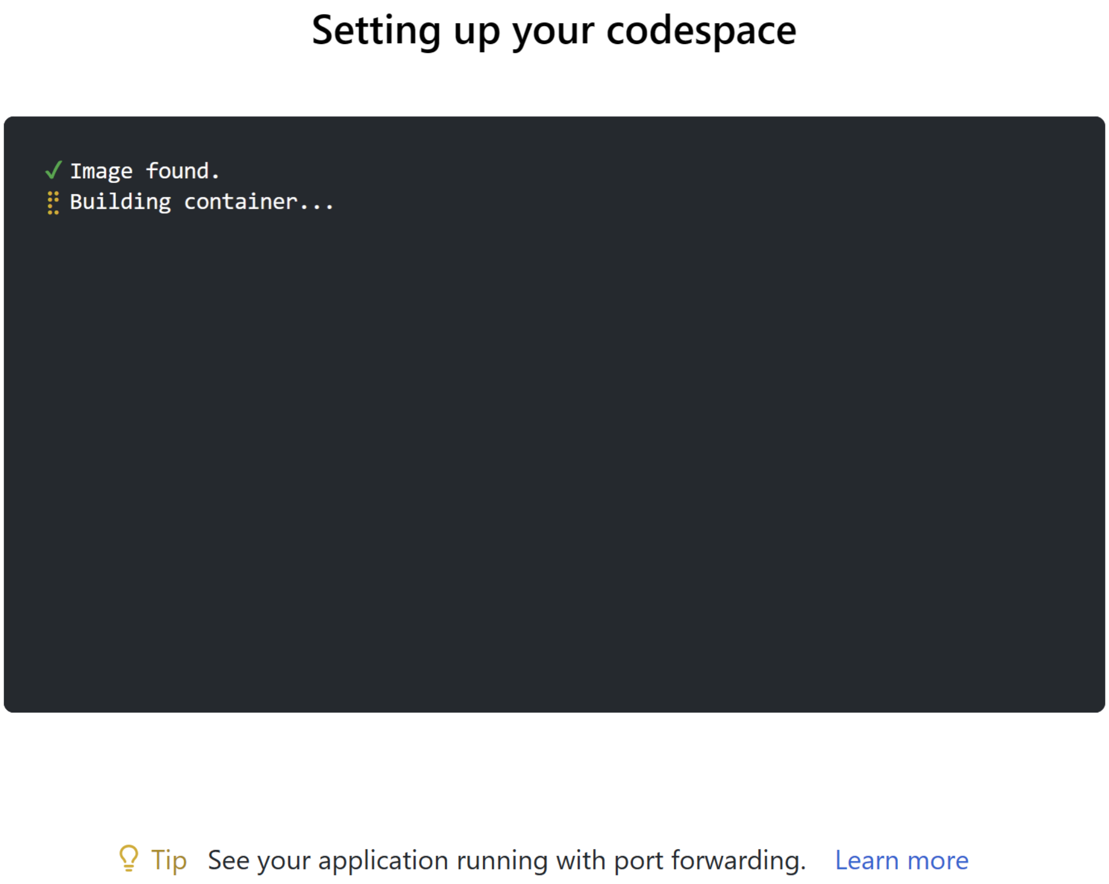
Your workspaces in the cloud

No codespaces

You don't have any codespaces with this repository checked out

Codespace usage for this repository is paid for by **mtzguido**.

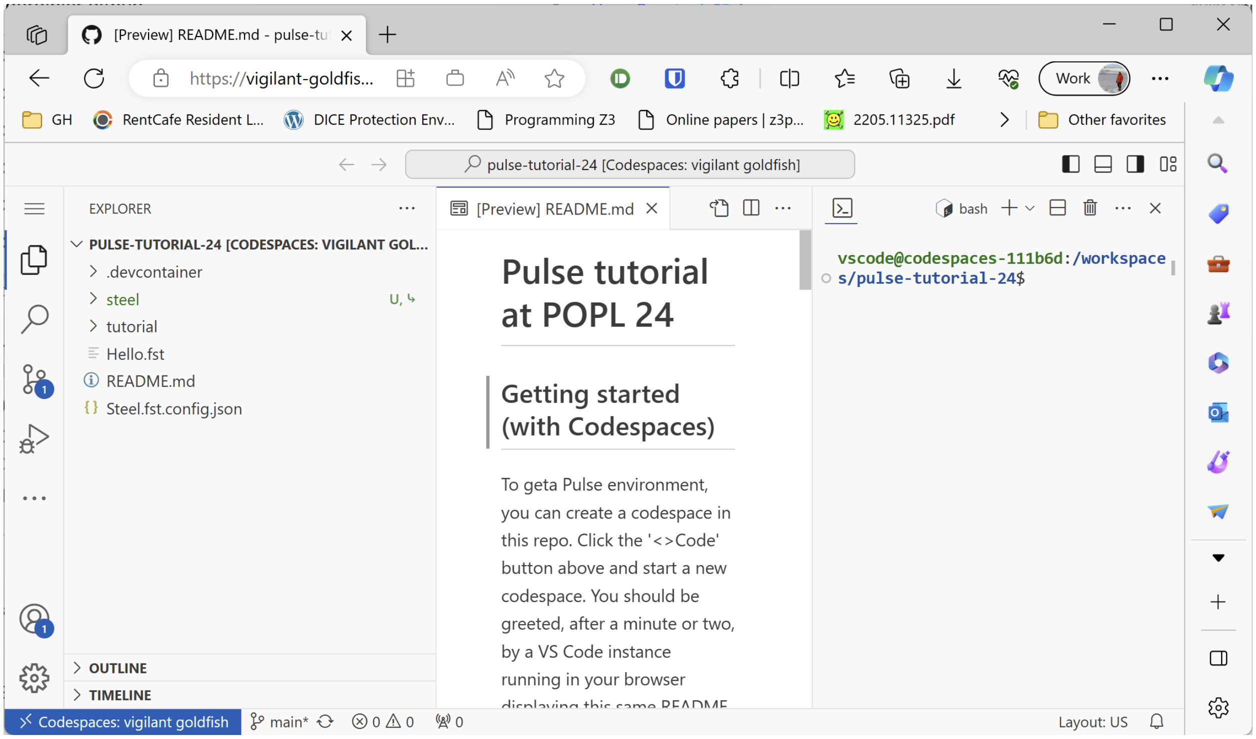
You should be greeted, after a minute or two, by a VS Code instance running in your browser displaying this same README.



Setting up your codespace

```
✓ Image found.
⠿ Building container...
```

💡 Tip See your application running with port forwarding. Learn more



```
vscode@codespaces-111b6d:/workspaces/pulse-tutorial-24$
```

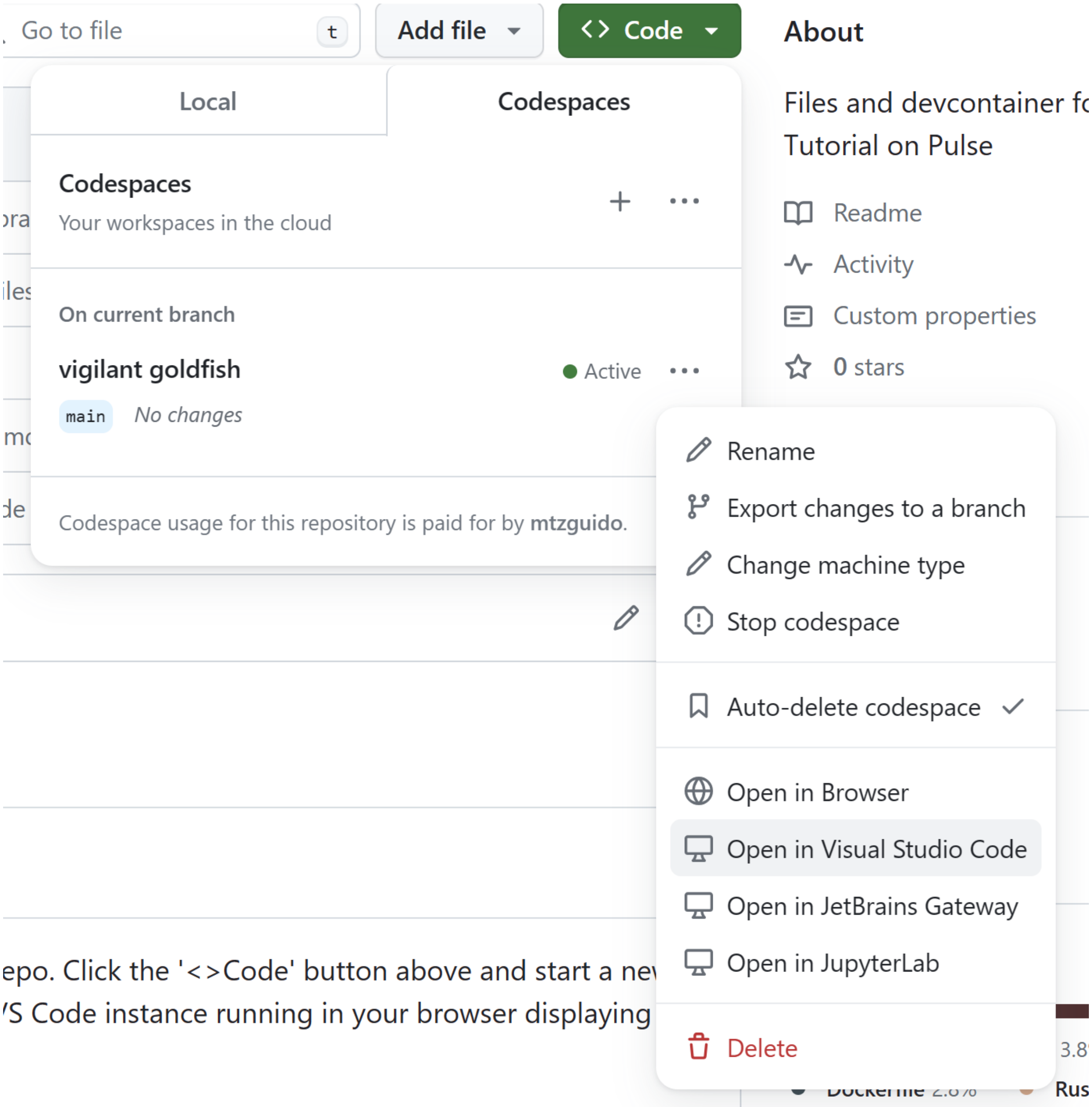
Pulse tutorial at POPL 24

Getting started (with Codespaces)

To geta Pulse environment, you can create a codespace in this repo. Click the '<>Code' button above and start a new codespace. You should be greeted, after a minute or two, by a VS Code instance running in your browser displaying this same README

All the usual F* navigation commands should work on Pulse files.

If you prefer a local UI instead of a browser tab, you can “open” the Codespace from your local VS Code installation like so:



Codespace usage for this repository is paid for by **mtzguido**.

epo. Click the '<>Code' button above and start a nev
'S Code instance running in your browser displaying

F* and Pulse are still running on Github’s servers, so the usage is still computed, but you may find the UI more comfortable.

30.2 Running the Dev Container locally

The Dev Container configuration contains all that is needed to run Pulse in an isolated, reproducible manner. If you would like to avoid Codespaces and just run locally, VS Code can set up the Dev Container locally for you very easily.

Simply open the repository in VS Code. You should see a popup claiming that the project has a Dev Container. Choose 'Reopen in Dev Container' to trigger a build of the container. VS Code will spawn a new window to download the base Docker image, set up the extension in it, and open the repository again.

This new window should now work as usual.

30.3 Using a Pulse release

A release of Pulse, including related F* tools, is available here. Uncompress the archive and add follow the instructions in the README.md, notably setting the recommended environment variables.

We also recommend installing VS Code and the fstar-vscode-assistant, from the VS Code marketplace. This should pick up the F* and Pulse installation from your path.

CHAPTER

THIRTYONE

PULSE BASICS

A Pulse program is embedded in an F* program, where the Pulse parts are delimited by a syntax extension marker “```pulse … ```”, as shown in the program below.

~~~
let fstar_five : int = 5

```pulse
fn five ()
  requires emp
  returns n:int
  ensures pure (n == 5)
{
  fstar_five
}
```

let pulse_five_in_fstar = five ()
~~~

This program starts with a bit of regular F* defining `fstar_five` followed by an a Pulse function `five` that references that F* definition and proves that it always returns the constant `5`. Finally, we have a bit of regular F* referencing the `five` defined in Pulse. This is a really simple program, but it already illustrates how Pulse and F* interact in both directions.

In what follows, unless we really want to emphasize that a fragment of code is Pulse embedded in a larger F* context, when showing Pulse code, we’ll just show the Pulse part, omitting the “```pulse … ```” delimiters.

31.1 A Separation Logic Primer

Separation Logic was invented by John Reynolds, Peter O’Hearn, and others in the late 1990s as a way to reason about imperative programs that use shared, mutable data structures, e.g., linked lists and graphs—see this paper for an introduction to separation logic. In the subsequent decades, several innovations were added to separation logic by many people, generalizing it beyond just sequential heap-manipulating programs to distributed programs, concurrent programs, asynchronous programs, etc. that manipulate abstract resources of various kinds, including time and space, messages sent over communication channels, etc.

Much like other Hoare Logics, which we reviewed in *an earlier section*, separation logic comes in two parts.

Separation Logic Propositions First, we have a language of propositions that describe properties about program resources, e.g., the heap. These propositions have the type `vprop` in Pulse,[1] and, under the covers in the PulselCore

[1] They are called `vprop` for mostly historical reasons. A version of the Steel separation logic from which Pulse grew is based allows associating a *value* with a separation logic proposition, so these came to be known as “value” propositions. However, Pulse does not make use of this feature—perhaps a better name in Pulse would be `slprop`, for separation logic proposition.

semantics of Pulse, a `vprop = state -> prop`, where `state` represents the state of a program, e.g., the contents of memory. It is useful (at least at first) to think of a `vprop` as a memory property, though we will eventually treat it more abstractly and use it to model many other kinds of resources.

Separation Logic Hoare Triples To connect `vprop`'s to programs, separation logics use Hoare triples to describe the action of a program on its state. For example, the Hoare triple `{ p } c { n. q }` describes a program `c` which when run in an initial state `s0` satisfying `p s0` (i.e., `p` is a precondition); `c` returns a value `n` while transforming the state to `s1` satisfying `q n s1` (i.e., `q` is a postcondition). Pulse's program logic is a partial-correctness logic, meaning that `c` may also loop forever, deadlock with other threads, etc.

Some simple vprops and triples: Here are two of the simplest
`vprops` (defined in `Pulse.Lib.Pervasives`):

- `emp`, the trivial proposition (equivalent to `fun s -> True`).
- `pure p`, heap-independent predicate `fun s -> p`. `emp` is equivalent to `pure True`.

The type of the program `five` illustrates how these `vprop`'s are used in program specifications:

- It is a function with a single unit argument—Pulse functions use the keyword `fn`.
- The precondition is just `emp`, the trivial assertion in separation logic, i.e., `five` can be called in any initial state.
- The return value is an integer `n:int`
- The postcondition may refer to the name of the return value (`n` in this case) and here claims that the final state satisfies the `pure` proposition, `n == 5`.

In other words, the type signature in Pulse is a convenient way to write the Hoare triple `{ emp } five () { n:int. pure (n == 5) }`.

Ownership At this point you may wonder if the postcondition of `five` is actually strong enough. We've only said that the return value `n == 5` but have not said anything about the state that results from calling `five ()`. Perhaps this specification allows `five` to arbitrarily change any memory location in the state, since `pure (5 == 5)` is true of any state.[2] If you're familiar with Low*, Dafny, or other languages based on Hoare logic for heaps, you may be wondering about how come we haven't specified a `modifies`-clause, describing exactly which part of the state a function may have changed. The nice thing in separation logic is that there is no need to describe what parts of the state you may have modified. This is because a central idea in logic is the concept of *ownership*. To a first approximation, a computation can only access those resources that it is explicitly granted access to in its precondition or those that it creates itself.[3] In this case, with a precondition of `emp`, the function `five` does not have permission to access *any* resources, and so `five` simply cannot modify the state in any observable way.

Separating Conjunction and the Frame Rule Let's go back to `incr` and `par_incr` that we saw in the previous section and look at their types closely. We'll need to introduce two more common `vprop`'s, starting with the "points-to" predicate:

- `pts_to x v` asserts that the reference `x` points to a cell in the current state that holds the value `v`.

`vprop`'s can also be combined in various ways, the most common one being the "separating conjunction", written `**` in Pulse.[4]

- `p ** q`, means that the state can be split into two *disjoint* fragments satisfying `p` and `q`, respectively. Alternatively, one could read `p ** q` as meaning that one holds the permissions associated with both `p` and `q` separately in a given state. The `**` operator satisfies the following laws:
  - Commutativity: `p ** q` is equivalent to `q ** p`
  - Associativity: `p ** (q ** r)` is equivalent to `(p ** q) ** r`

[2] For experts, Pulse's separation logic is *affine*.

[3] When we get to things like invariants and locks, we'll see how permissions can be acquired by other means.

[4] In the separation logic literature, separating conjunction is written `p * q`, with just a single star. We use two stars `**` to avoid a clash with multiplication.

  - Left and right unit: `p ** emp` is equivalent to `p`. Since `**` is commutative, this also means that `emp ** p` is equivalent to `p`

Now, perhaps the defining characteristic of separation logic is how the `**` operator works in the program logic, via a key rule known as the *frame* rule. The rule says that if you can prove the Hoare triple `{ p } c { n. q }`, then, for any other `f : vprop`, you can also prove `{ p ** f } c { n. q ** f }`—`f` is often called the “frame”. It might take some time to appreciate, but the frame rule captures the essence of local, modular reasoning. Roughly, it states that if a program is correct when it only has permission `p` on the input state, then it remains correct when run in a larger state and is guaranteed to preserve any property (`f`) on the part of the state that it doesn’t touch.

With this in mind, let’s look again at the type of `incr`, which requires permission only to `x` and increments it:

```
fn incr (x:ref int)
requires pts_to x 'i
ensures pts_to x ('i + 1)
{
    let v = !x;
    x := v + 1;
}
```

Because of the frame rule, we can also call `incr` in a context like `incr_frame` below, and we can prove without any additional work that `y` is unchanged.

```
fn incr_frame (x y:ref int)
requires pts_to x 'i ** pts_to y 'j
ensures pts_to x ('i + 1) ** pts_to y 'j
{
   incr x;
}
```

In fact, Pulse lets us use the frame rule with any `f:vprop`, and we get, for free, that `incr x` does not disturb `f`.

```
fn incr_frame_any (x:ref int) (f:vprop)
requires pts_to x 'i ** f
ensures pts_to x ('i + 1) ** f
{
   incr x;
}
```

A point about the notation: The variable `'i` is an implicitly bound logical variable, representing the value held in the ref-cell `x` in the initial state. In this case, `'i` has type `FStar.Ghost.erased int`—we learned about *erased types in a previous section*. One can also bind logical variables explicitly, e.g., this is equivalent:

```
fn incr_explicit_i (x:ref int) (i:erased int)
requires pts_to x i
ensures pts_to x (i + 1)
{
    let v = !x;
    x := v + 1;
}
```

Other vprop connectives In addition the separating conjunction, Pulse, like other separation logics, provides other ways to combine `vprops`. We’ll look at these in detail in the subsequent chapters, but we list the most common other connectives below just to give you a taste of the logic.

- `exists* (x1:t1) ... (xn:tn). p`: Existential quantification is used extensively in the Pulse libraries, and the language provides many tools to make existentials convenient to use. `exists x. p` is valid in a state `s` if there is a witness `w` such that `p [w/x]` is valid in `s`. For experts, existential quantification is impredicative, in the sense that one can quantify over `vprops` themselves, i.e., `exists* (p:vprop). q` is allowed.

- `forall* (x1:t1) ... (xn:tn). p`: Universal quantification is also supported, though less commonly used. `forall (x:t). p` is valid in `s` if `p[w/x]` is valid for all values `w:t`. Like existential quantification, it is also impredicative.
- `p @==> q` is a form of separating implication similar to an operator called a *magic wand* or a *view shift* in other separation logics.

Pulse does not yet provide libraries for conjunction or disjunction. However, since Pulse is embedded in F*, new vprops can also be defined by the user and it is common to do so, e.g., recursively defined predicates, or variants of the connectives described above.

CHAPTER

THIRTYTWO

MUTABLE REFERENCES

Pulse aims to support programming with explicit control over memory management and without need for a garbage collector, similar to languages like C or Rust, but, of course, in a proof-oriented style. Towards that end, one of the main features it offers (especially in comparison to purely functional F*) is support for references to mutable memory that can be both allocated and reclaimed.

In this chapter, we'll learn about three kinds of mutable references: stack references, heap references (or boxes), and ghost references. Stack references point to memory allocated in the stack frame of the current function (in which case the memory is reclaimed when the function returns). Heap references, or boxes, point to memory locations in the heap, and heap memory is explicitly reclaimed by calling `drop` or `free`. Ghost references are for specification and proof purposes only and point to memory locations that do not really exist at runtime.

32.1 `ref t`: Stack or Heap References

Most of the operations on mutable references are agnostic to whether the memory referenced resides on the stack or the heap—the main difference is that stack references are allocated in a scope and implicitly reclaimed when they go out of scope; whereas heap references are explicitly allocated and deallocated.

The type `Pulse.Lib.Reference.ref t` is the basic type of a mutable reference. We have already seen `ref t` used in the `incr` function of the previous section. We show below another common function to swap the contents of two references:

```
fn swap #a (r0 r1:ref a)
requires pts_to r0 'v0 ** pts_to r1 'v1
ensures pts_to r0 'v1 ** pts_to r1 'v0
{
    let v0 = !r0;
    let v1 = !r1;
    r0 := v1;
    r1 := v0;
}
```

32.1.1 Reading a reference

Let’s start by taking a closer look at how dereferencing works in the function `value_of` below:

```
fn value_of (#a:Type) (r:ref a)
requires pts_to r 'v
returns v:a
ensures pts_to r 'v ** pure (v == 'v)
{
    !r;
}
```

Its slightly more explicit form is shown below, where `w:erased a` is an erased value witnessing the current contents referenced by `r`.

```
fn value_of_explicit (#a:Type) (r:ref a) (#w:erased a)
requires pts_to r w
returns v:a
ensures pts_to r w ** pure (v == reveal w)
{
    !r;
}
```

Notice how the precondition requires `pts_to r w` while the postcondition retains `pts_to r w`, along with the property that `v == reveal w`, i.e., the type proves that if we read the reference the value we get is equal to the logical witness provided.

32.1.2 Erased values are for specification and proof only

The logical witness is an erased value, so one cannot directly use it in a non-ghost computation. For example, if instead of reading the reference, we attempt to just return `reveal w`, the code fails to check with the error shown below.

```
fn value_of_explicit_fail (#a:Type) (r:ref a) (#w:erased a)
requires pts_to r w
returns v:a
ensures pts_to r w ** pure (v == reveal w)
{
    reveal w
}
```

```
Expected a Total computation, but got Ghost
```

32.1.3 Writing through a reference

The function `assign` below shows how to mutate the contents of a reference—the specification shows that when the function returns, `r` points to the assigned value `v`.

```
fn assign (#a:Type) (r:ref a) (v:a)
requires pts_to r 'v
ensures pts_to r v
{
```

(continues on next page)

(continued from previous page)

```
    r := v;
}
```

32.1.4 Dereferencing is explicit

Unlike languages like C or Rust which make a distinction between l-values and r-values and implicitly read the content of references, in Pulse (like in OCaml), references are explicitly dereferenced.[1] As the program below illustrates, references themselves can be passed to other functions (e.g., as in/out-parameters) while their current values must be passed explicitly.

The function `add` takes both a reference `r:ref int` and a value `n:int` as arguments:

```
fn add (r:ref int) (n:int)
requires pts_to r 'v
ensures pts_to r ('v + n)
{
    let v = !r;
    r := v + n;
}
```

Meanwhile, the function `quadruple` calls `add` twice to double the value stored in `r` each time.

```
fn quadruple (r:ref int)
requires pts_to r 'v
ensures pts_to r (4 * 'v)
{
    let v1 = !r;
    add r v1;
    let v2 = !r;
    add r v2;
}
```

32.1.5 Inspecting the proof state

A Pulse program is checked one stateful operation at a time, "pushing through" the `vprop` assertions starting with the precondition, until the end of function's body. The inferred `vprop` at the exit of a function must match the annotated postcondition. Along the way, the Pulse checker will make several calls to the SMT solver to prove that, say, `pts_to x (v + v)` is equal to `pts_to x (2 * v)`.

At each point in the program, the Pulse checker maintains a proof state, which has two components:

- A typing environment, binding variables in scope to their types, including some refinement types that reflect properties about those variables in scope, e.g., `x:int; y:erased int; _:squash (x == reveal y)`.
- A separation logic context, called just "the context", or sometimes "the `vprop` context". The context contains all known facts about the current state of the program.

Pulse provides a command called `show_proof_state` that allows the user to inspect the proof state at a particular program point, aborting the Pulse checker at that point. It's quite common when developing a Pulse program to repeatedly inspect the proof state and to advance it by a single or just a few steps at a time. This makes the experience

[1] We are considering switching this convention and the Pulse option `--ext 'pulse:rvalues'` can be enabled to add implicit dereferencing; however, it is not yet recommended for use.

of developing a Pulse program quite interactive, similar perhaps to writing tactics in F* or other languages. Except, in Pulse, one incrementally writes an imperative program together with its proof of correctness.

Here below is the `quadruple` program again, with the proof states annotated at each point, and a `show_proof_state` command in the middle.

```
fn quadruple (r:ref int)
requires pts_to r 'v
ensures pts_to r (4 * 'v)
{
    let v1 = !r; // Env=v1:int; _:squash (v1 == 'v)       Ctxt= pts_to r v1
    add r v1;    // ...                                    Ctxt= pts_to r (v1 + v1)
    show_proof_state;
    let v2 = !r; // Env=...; v2:int; _:squash(v2==v1+v1)  Ctxt= pts_to r v2
    add r v2;    // Env=...                                Ctxt= pts_to r (v2 + v2)
                 // ..                                     Ctxt= pts_to r (4 * 'v)
}
```

The output from `show_proof_state` is shown below:

```
- Current context:
    pts_to r (reveal (hide v1) + v1) **
    emp
- In typing environment:
    [_#5 : unit,
    _#4 : squash (reveal 'v == v1),
    v1#3 : int,
    'v#2 : erased int,
    r#1 : ref int]
```

The comments show how the proof state evolves after each command.

- Pulse typechecks each step of a program by checking the current assumptions in the proof state are sufficient to prove the precondition of that step, ensuring that all unused permissions are retained in the context—using the frame rule, discussed in the previous section. Given a context that is equivalent to `p ** q`, if `p` is sufficient to prove `goal`, then `p` is called *the support* for `goal`, while `q` is the *frame*.
- Like F*, Pulse tries to instantiate implicit arguments automatically, e.g., at the second call to `add`, Pulse automatically instantiates `'v` to `v2`.
- Pulse automatically moves any `pure p` property in the `vprop` context to a `squash p` hypothesis in the typing environment. Pulse also proves `pure` properties automatically, by sending queries to the SMT solver, which can make use of the hypothesis in the typing environment only.
- Pulse also uses the SMT solver to convert `pts_to r (v2 + v2)` to `pts_to r (4 * 'v)`.

32.1.6 Stateful commands are explicitly sequenced

Pulse expects the results of all stateful operations to be explicitly `let`-bound. For example, the following code fails to type check:

```
fn quad_fail (r:ref int)
requires pts_to r 'v
ensures pts_to r (4 * 'v)
{
```

(continues on next page)

(continued from previous page)

```
    add r (!r);
    add r (!r);
}
```

```
- Expected type "int"; but "!r" has type
  "stt int
       (pts_to r (reveal (*?u93*) _))
       (fun x -> pts_to r x ** pure (reveal (*?u93*) _ == x))"
```

The error points to the first occurrence of `(!r)`. The message is admittedly cryptic and should be improved. As we'll see in a later chapter, the type `stt _ _ _` is the type of an unevaluated Pulse computation—this error complains that `add` expected an argument of type `int` but instead got an unevaluated computation.

32.1.7 Fractional Permissions

Pulse distinguishes read-only references from read/write references. As in languages like Rust, Pulse ensures that there can be at most one thread that holds read/write permission to a reference, although many threads can share read-only references. This ensures that Pulse programs are free of data races. At a more abstract level, Pulse's permission system ensures that one can reason locally about the contents of memory, since if one holds read/write permission to a reference, one can be sure that its contents cannot be changed by some part of the program.

To implement this permission discipline, Pulse uses a system of fractional permissions, an idea due to John Boyland. In particular, the `pts_to` predicate that we have been using actually has an additional implicit arguments that describes how much permission one holds on a reference.

The full type of the `pts_to` predicate is shown below:

```
val pts_to (#a:Type u#0) (r:ref a) (#p:perm) (v:a) : vprop
```

We have so far been writing `pts_to r v` instead of `pts_to #a r #p v`. Usually, one does not need to write the first argument `#a` since it is computed by type inference; the `#p:perm` argument is more interesting—when omitted, it defaults to the value `1.0R`. The type `perm` (defined in `PulseCore.FractionalPermission`) is a real number strictly greater than `0.0R` and less than or equal to `1.0R`.

The `pts_to r #1.0R v` represents exclusive, read/write permission on a reference. Revisiting the `assign` function from previously, we can write down the permissions explicitly.

```
fn assign_full_perm (#a:Type) (r:ref a) (v:a)
requires pts_to r #1.0R 'v
ensures pts_to r #1.0R v
{
    r := v;
}
```

In contrast, when reading a reference, any permission `p` will do, as shown below:

```
fn value_of_perm #a #p (r:ref a)
requires pts_to r #p 'v
returns v:a
ensures pts_to r #p 'v ** pure (v == 'v)
{
    !r;
}
```

If we try to write to a reference without holding full permission on it, Pulse rejects the program, as shown below.

~~~
#push-options "--print_implicits"
[@@expect_failure]
```pulse
fn assign_perm #a #p (r:ref a) (v:a) (#w:erased a)
requires pts_to r #p w
ensures pts_to r #p w
{
    r := v;
}
```
#pop-options
~~~

```
- Cannot prove:
    pts_to #a r #1.0R (reveal #a _)
- In the context:
    pts_to #a r #p (reveal #a w)
```

The full error message requires the F* option `--print_implicits`.

The functions `share` and `gather` allow one to divide and combine permissions on references, as shown below.

```
fn share_ref #a #p (r:ref a)
requires pts_to r #p 'v
ensures pts_to r #(p /. 2.0R) 'v ** pts_to r #(p /. 2.0R) 'v
{
    share r;
}
```

```
fn gather_ref #a (#p:perm) (r:ref a)
requires
    pts_to r #(p /. 2.0R) 'v0 **
    pts_to r #(p /. 2.0R) 'v1
ensures
    pts_to r #p 'v0 **
    pure ('v0 == 'v1)
{
    gather r
}
```

The type of `gather_ref` has an additional interesting element: its postcondition proves that `'v0 == 'v1`. That is, since `x` can point to at most one value, given two separate points-to assertions about `x`, allows one to conclude that the pointed-to witnesses are identical.

32.2 Stack references

32.2.1 let mut creates a new stack ref

To create a new `ref t`, one uses the `let mut` construct of Pulse, as shown below.

```
fn one ()
requires emp
returns v:int
ensures pure (v == 1)
{
                   //     .     |- emp
    let mut i = 0; // i:ref int |- pts_to i 0
    incr i;        // i:ref int |- pts_to i (0 + 1)
    !i             //      .    |- v:int. emp ** pure (v == 1)

}
```

The body of the program is annotated to show program assertions that are true after each command.

- Initially, only the precondition `emp` is valid.
- After `let mut i = 0`, we have `i :  ref int` and `pts_to i 0`, meaning that `i` points to a stack slot that holds the value `0`.
- After calling `incr i`, we have `pts_to i (0 + 1)`
- Finally, we dereference `i` using `!i` and return `v:int` the current value of `i`.
- At the point where the scope of a `let mut x` ends, the Pulse checker requires that the context contains `pts_to x #1.0R _v` for some value `_v`. This ensures that the code cannot squirrel away a permission to the soon-to-be out-of-scope reference in some other permission. Once the scope ends, and the memory it points to is reclaimed, and the `pts_to x #1.0R _v` is consumed.

A few additional points to note here:

- Pulse proves `pure` properties automatically, by sending queries to the SMT solver.
- Pulse simplifies `vprop` implicitly, e.g., Pulse will automatically rewrite `emp ** p` to `p`.
- Like F*, Pulse tries to instantiate implicit arguments automatically, e.g., at the call to `incr`, Pulse automatically instantiates `'v` to `0` (actually, to `hide 0`).

32.2.2 Stack references are scoped and implicitly reclaimed

To emphasize that stack references allocated with `let mut` are scoped, let's look at the program below that Pulse refuses to check:

```
fn refs_are_scoped ()
requires emp
returns s:ref int
ensures pts_to s 0
{
    let mut s = 0;
    s
}
```

The error points to the location of `s` with the message below, meaning that the current assertion on the heap is only `emp`, while the goal to be proven for the postcondition is `pts_to s 0`. In other words, we no longer have ownership on `s` once it goes out of scope.

```
- Cannot prove:
    pts_to s 0
- In the context:
    emp
```

32.3 Heap references

The type `Pulse.Lib.Box.box t` is the type of heap references—the name is meant to evoke Rust's type of heap references, `Box<T>`. We use the module alias `Box` in what follows:

```
module Box = Pulse.Lib.Box
```

The `Box` module provides most of the same predicates and functions that we have with regular references, including `pts_to`, `(!)`, `(:=)`, `share`, and `gather`. Additionally, heap references are explicitly allocated using `alloc` and deallocated using `free`, as shown below.

```
fn new_heap_ref (#a:Type) (v:a)
requires emp
returns r:Box.box a
ensures Box.pts_to r v
{
    Box.alloc v
}
```

Note, we can return a freshly allocated heap reference from a function, unlike a `let mut` scoped, stack-allocated reference.

In the following example, we use `open Box;` to open the namespace `Box` in the following scope.

```
fn last_value_of #a (r:Box.box a)
requires Box.pts_to r 'v
returns v:a
ensures pure (v == 'v)
{
    open Box;
    let v = !r;
    free r;
    v
}
```

`box t` references can be demoted to regular `ref t` references for code reuse. For example, in the code below, we increment the contents of `r:box int` by first calling `Box.to_ref_pts_to` to convert `Box.pts_to r 'v` to a regular `pts_to (box_to_ref r) 'v`; then calling `incr (box_to_ref r)`; and then converting back to a `Box.pts_to`.

```
fn incr_box (r:Box.box int)
requires Box.pts_to r 'v
ensures Box.pts_to r ('v + 1)
{
    Box.to_ref_pts_to r;     //Box.pts_to (box_to_ref r) 'v
```

(continues on next page)

(continued from previous page)

```
    incr (Box.box_to_ref r); //pts_to (box_to_ref r) ('v + 1)
    Box.to_box_pts_to r      //Box.pts_to r ('v + 1)
}
```

Finally, unlike Rust's `Box<T>` type, which is always treated linearly (i.e., in Rust, one always holds exclusive read/write permission on a ` `Box<T>`), in Pulse, `Box.pts_to r #p v` has an implicit fractional permission as with regular references.

32.4 Ghost references

CHAPTER

THIRTYTHREE

EXISTENTIAL QUANTIFICATION

A very common specification style in Pulse involves the use of the existential quantifier. Before we can start to write interesting examples, let's take a brief look at how existential quantification works.

As mentioned in the *introduction to Pulse*, one of the connectives of Pulse's separation logic is the existential quantifier. Its syntax is similar to F*'s existential quantifier, except it is written `exists*` instead of just `exists`, and its body is a `vprop`, as in the examples shown below.

```
exists* (v:nat). pts_to x v

exists* v. pts_to x v

exists* v1 v2. pts_to x v1 ** pts_to y v2

...
```

33.1 Some simple examples

Looking back to the `assign` example from the previous chapter (shown below), you may have wondered why we bothered to bind a logical variable `'v` in precondition of the specification, since it is never actually used in any other predicate.

```
fn assign (#a:Type) (r:ref a) (v:a)
requires pts_to r 'v
ensures pts_to r v
{
    r := v;
}
```

And indeed, another way to write the specification of `assign`, without the logical variable argument, is shown below.

```
fn assign #a (x:ref a) (v:a)
requires
  exists* w. pts_to x w
ensures
  pts_to x v
{
  x := v
}
```

This time, in the precondition, we use an existential quantifier to say that `assign` is callable in a context where `x` points to any value `w`.

Usually, however, the postcondition of a function *relates* the initial state prior to the call to the state after the call and existential variables are only in scope as far to the right as possible of the enclosing `vprop`. So, existential quantifiers in the precondition of a function are not so common.

To illustrate, the following attempted specification of `incr` does not work, since the existentially bound `w0` is not in scope for the postcondition.

~~~
[@@expect_failure]
```pulse
fn incr #a (x:ref int)
requires
  exists* w0. pts_to x w
ensures
  pts_to x (w0 + 1) //w0 is not in scope here
{
  let w = !x
  x := w + 1
}
```
~~~

However, existential quantification often appears in postconditions, e.g., in order to abstract the behavior of function by underspecifying it. To illustrate, consider the function `make_even` below. It's type states that it sets the contents of `x` to some even number `w1`, without specifying `w1` exactly. It also uses an existential quantification in its precondition, since its postcondition does not depend on the initial value of `x`.

```
fn make_even (x:ref int)
requires
  exists* w0. pts_to x w0
ensures
  exists* w1. pts_to x w1 ** pure (w1 % 2 == 0)
{
  let v = !x;
  x := v + v;
}
```

33.2 Manipulating existentials

In a previous chapter on *handling classical connectives*, we saw how F* provides various constructs for introducing and eliminating logical connectives, including the existential quantifier. Pulse also provides constructs for working explicitly with existential quantifiers, though, usually, Pulse automation takes care of introducing and eliminating existentials behind the scenes. However, the explicit operations are sometimes useful, and we show a first example of how they work below

```
fn make_even_explicit (x:ref int)
requires
  exists* w0. pts_to x w0
ensures
  exists* w1. pts_to x w1 ** pure (w1 % 2 == 0)
{
  with w0. assert (pts_to x w0);
```

(continues on next page)

(continued from previous page)

```
  let v = !x; // v:int; _:squash(v==w0); Ctxt=pts_to x v
  x := v + v; //                          ... pts_to x (v + v)
  introduce
  exists* w1. pts_to x w1 ** pure (w1 % 2 == 0)
  with (v + v);
}
```

33.2.1 Eliminating existentials

The form `with w0...wn. assert p; rest` is often used as an eliminator for an existential. When the context contains `exists* x0...xn. p`, the `with` construct binds `w0 ... wn` to the existentially bound variables in the remainder of the scope `rest`.

A `show_proof_state` immediately after the `with w0. assert (pts_to x w0)` prints the following:

```
- Current context:
    pts_to x (reveal w0) **
    emp
- In typing environment:
    [w0#2 : erased int,
    x#1 : ref int]
```

That is, we have `w0:erased int` in scope, and `pts_to x (reveal w0)` in context.

Here is another example usage of `with`, this time with multiple binders.

```
fn make_even_explicit_alt (x y:ref int)
requires
  exists* wx wy. pts_to x wx ** pts_to y wy ** pure (wx % 2 == wy % 2)
ensures
  exists* wx' wy'. pts_to x wx' ** pts_to y wy' ** pure (wx' % 2 == 0)
{
  with wx wy. _;
  let vx = !x;
  let vy = !y;
  x := vx + vy;
  introduce exists* wx' wy'. pts_to x wx' ** pts_to y wy' ** pure (wx' % 2 == 0)
  with (vx + vy) vy;
}
```

When there is a single existential formula in the context, one can write `with x1..xn. _` to “open” the formula, binding its witnesses in scope. A `show_proof_state` after the first line prints:

```
- Current context:
    pts_to x (reveal wx) **
    pts_to y (reveal wy) **
    pure (eq2 (op_Modulus (reveal wx) 2) (op_Modulus (reveal wy) 2)) **
    emp
- In typing environment:
    [_#5 : squash (eq2 (op_Modulus (reveal wx) 2) (op_Modulus (reveal wy) 2)),
    wy#4 : erased int,
    wx#3 : erased int,
```

(continues on next page)

(continued from previous page)

```
    y#2 : ref int,
    x#1 : ref int]
```

33.2.2 Introducing existentials

The Pulse checker will automatically introduce existential formulas by introduces new unification variables for each existentially bound variable, and then trying to find solutions for those variables by matching `vprops` in the goal with those in the context.

However, one can also introduce existential formulas explicitly, using the `introduce exists*` syntax, as seen in the two examples above. In general, one can write

```
introduce exists* x1 .. xn. p
with w1...wn
```

explicitly providing witnesses `w1..wn` for each of the existentially bound variables `x1..xn`.

CHAPTER

THIRTYFOUR

USER-DEFINED PREDICATES

In addition to the vprop predicates and connectives that the Pulse libraries provide, users very commonly define their own vprops. We show a few simple examples here—subsequent examples will make heavy use of user-defined predicates. For example, see this section for *recursively defined predicates*.

34.1 Fold and Unfold with Diagonal Pairs

A simple example of a user-defined abstraction is show below.

```
let pts_to_diag
        #a
        (r:ref (a & a))
        (v:a)
: vprop
= pts_to r (v, v)
```

pts_to_diag r v is a vprop defined in F* representing a reference to a pair whose components are equal.

We can use this abstraction in a Pulse program, though we have to be explicit about folding and unfolding the predicate.

```
fn double (r:ref (int & int))
requires pts_to_diag r 'v
ensures pts_to_diag r (2 * 'v)
{
  unfold (pts_to_diag r 'v);
  let v = !r;
  let v2 = fst v + snd v;
  r := (v2, v2);
  fold (pts_to_diag r v2);
}
```

The unfold p command checks that p is provable in the current context by some term q, and then rewrites the context by replacing that occurrence of q with the term that results from unfolding the head symbol of p. A show_proof_state after the unfold shows that we have a pts_to r (reveal 'v, reveal 'v) in the context, exposing the abstraction of the pts_to_diag predicate.

At the end of function, we use the fold p command: this checks that the unfolding of p is provable in the context by some term q and then replaces q in the context with p.

fold and unfold is currently very manual in Pulse. While in the general case, including with recursively defined predicates, automating the placement of folds and unfolds is challenging, many common cases (such as the ones here) can be easily automated. We are currently investigating adding support for this.

Some initial support for this is already available, inasmuch as Pulse can sometimes figure out the arguments to the vprops that need to be folded/unfolded. For instance, in the code below, we just mention the name of the predicate to be unfolded/folded, without needing to provide all the arguments.

```
fn double_alt (r:ref (int & int))
requires pts_to_diag r 'v
ensures pts_to_diag r (2 * 'v)
{
  unfold pts_to_diag;
  let v = !r;
  let v2 = fst v + snd v;
  r := (v2, v2);
  fold pts_to_diag;
}
```

34.2 Mutable Points

As a second, perhaps more realistic example of a user-defined abstraction, we look at defining a simple mutable data structure: a structure with two mutable integer fields, representing a 2-dimensional point.

```
noeq
type point = {
    x:ref int;
    y:ref int;
}

let is_point (p:point) (xy: int & int) =
    pts_to p.x (fst xy) **
    pts_to p.y (snd xy)
```

A `point` is just an F* record containing two references. Additionally, we define `is_point`, a `vprop`, sometimes called a "representation predicate", for a `point`. `is_point p xy` says that `p` is a representation of the logical point `xy`, where `xy` is pure, mathematical pair.

We can define a function `move`, which translates a point by some offset `dx, dy`.

```
fn move (p:point) (dx:int) (dy:int)
requires is_point p 'xy
ensures is_point p (fst 'xy + dx, snd 'xy + dy)
{
  unfold is_point;
  let x = !p.x;
  let y = !p.y;
  p.x := x + dx;
  p.y := y + dy;
  fold (is_point p (x + dx, y + dy));
}
```

Implementing `move` is straightforward, but like before, we have to `unfold` the `is_point` predicate first, and then fold it back up before returning.

Unfortunately, Pulse cannot infer the instantiation of `is_point` when folding it. A `show_proof_state` prior to the fold should help us see why:

- We have pts_to p.x (x + dx) ** pts_to p.y (y + dy)
- For fold (is_point p.x ?w) to succeed, we rely on F*'s type inference to find a solution for the unsolved witness ?w such that fst ?w == (x + dx) and snd ?w == (y + dy). This requires an eta-expansion rule for pairs to solve ?w := (x + dx, y + dy), but F*'s type inference does not support such a rule for pairs.

So, sadly, we have to provide the full instantiation is_point p (x + dx, y + dy) to complete the proof.

This pattern is a common problem when working with representation predicates that are indexed by complex values, e.g., pairs or records. It's common enough that it is usually more convenient to define a helper function to fold the predicate, as shown below.

```
ghost
fn fold_is_point (p:point)
requires pts_to p.x 'x ** pts_to p.y 'y
ensures is_point p (reveal 'x, reveal 'y)
{
  fold (is_point p (reveal 'x, reveal 'y))
}
```

Note: We've marked this helper function ghost. We'll look into ghost functions in much more detail in a later chapter.

This allows type inference to work better, as shown below.

```
fn move_alt (p:point) (dx:int) (dy:int)
requires is_point p 'xy
ensures is_point p (fst 'xy + dx, snd 'xy + dy)
{
  unfold is_point;
  let x = !p.x;
  let y = !p.y;
  p.x := x + dx;
  p.y := y + dy;
  fold_is_point p;
}
```

34.3 Rewriting

In addition to fold and unfold, one also often uses the rewrite command when working with defined predicates. Its general form is:

```
with x1 ... xn. rewrite p as q;
rest
```

Its behavior is to find a substitution subst that instantiates the x1 ... xn as v1 ... vn, such that subst(p) is supported by c in the context, Pulse aims to prove that subst(p) == subst(q) and replaces c in the context by subst(q) and proceeds to check subst(rest).

To illustrate this at work, consider the program below:

```
fn create_and_move ()
requires emp
ensures emp
{
    let mut x = 0;
    let mut y = 0;
    let p = { x; y };
    //pts_to x 0 ** pts_to y 0
    with _v. rewrite pts_to x _v as pts_to p.x _v;
    with _v. rewrite pts_to y _v as pts_to p.y _v;
    //pts_to p.x 0 ** pts_to p.y 0
    fold_is_point p;
    move p 1 1;
    assert (is_point p (1, 1));
    unfold is_point;
    //pts_to p.x (fst (1, 1)) ** pts_to p.y (snd (1, 1))
    with _v. rewrite pts_to p.x _v as pts_to x _v;
    with _v. rewrite pts_to p.y _v as pts_to y _v;
    //pts_to x (fst (1, 1)) ** pts_to y (snd (1, 1))
}
```

We allocate two references and put them in the structure `p`. Now, to call `fold_is_point`, we need `pts_to p.x _` and `pts_to p.y _`, but the context only contains `pts_to x _` and `pts_to y _`. The `rewrite` command transforms the context as needed.

At the end of the function, we need to prove that `pts_to x _` and `pts_to y _` as we exit the scope of `y` and `x`, so that they can be reclaimed. Using `rewrite` in the other direction accomplishes this.

This is quite verbose. As with `fold` and `unfold`, fully automated `rewrite` in the general case is hard, but many common cases are easy and we expect to add support for that to the Pulse checker.

In the meantime, Pulse provides a shorthand to make some common rewrites easier.

The `rewrite each` command has the most general form:

```
with x1 ... xn. rewrite each e1 as e1', ..., en as en' in goal
```

This is equivalent to:

```
with x1 ... xn. assert goal;
rewrite each e1 as e1', ..., en as en' in goal
```

```
rewrite each e1 as e1', ..., en as en' in goal
```

is equivalent to

```
rewrite goal as goal'
```

where `goal'` is computed by rewriting, in parallel, every occurrence of `ei` as `ei'` in `goal`.

Finally, one can also write:

```
rewrite each e1 as e1', ..., en as en'
```

omitting the `goal` term. In this case, the `goal` is taken to be the entire current `vprop` context.

Using `rewrite each ...` makes the code somewhat shorter:

```
fn create_and_move_alt ()
requires emp
ensures emp
{
    let mut x = 0;
    let mut y = 0;
    let p = { x; y };
    rewrite each x as p.x, y as p.y;
    fold_is_point p;
    move p 1 1;
    assert (is_point p (1, 1));
    unfold is_point;
    rewrite each p.x as x, p.y as y;
}
```

CHAPTER

THIRTYFIVE

CONDITIONALS

To start writing interesting programs, we need a few control constructs. In this chapter, we'll write some programs with branches of two kinds: `if/else` and `match`.

35.1 A Simple Branching Program: Max

Here's a simple program that returns the maximum value stored in two references.

~~~
let max_spec x y = if x < y then y else x

```pulse
fn max #p #q (x y:ref int)
requires pts_to x #p 'vx ** pts_to y #q 'vy
returns n:int
ensures pts_to x #p 'vx ** pts_to y #q 'vy
        ** pure (n == max_spec 'vx 'vy)
{
    let vx = !x;
    let vy = !y;
    if (vx > vy)
    {
        vx
    }
    else
    {
        vy
    }
}
```
~~~

This program illustrates a very common specification style.

- We have a pure, F* function `max_spec`
- And a Pulse function working on mutable references, with a specification that relates it to the pure F* spec. In this case, we prove that `max` behaves like `max_spec` on the logical values that witness the contents of the two references.

The implementation of `max` uses a Pulse conditional statement. Its syntax is different from the F* `if-then-else` expression: Pulse uses a more imperative syntax with curly braces, which should be familiar from languages like C.

35.1.1 Limitation: Non-tail Conditionals

Pulse's inference machinery does not yet support conditionals that appear in non-tail position. For example, this variant of `max` fails, with the error message shown below.

```
fn max_alt #p #q (x y:ref int)
requires pts_to x #p 'vx ** pts_to y #q 'vy
returns n:int
ensures pts_to x #p 'vx ** pts_to y #q 'vy
        ** pure (n == max_spec 'vx 'vy)
{
    let mut result = 0;
    let vx = !x;
    let vy = !y;
    if (vx > vy)
    {
        result := vx;
    }
    else
    {
        result := vy;
    };
    !result;
}
```

```
Pulse cannot yet infer a postcondition for a non-tail conditional statement;
Either annotate this `if` with `returns` clause; or rewrite your code to use a tail␣
↪conditional
```

Here's an annotated version of `max_alt` that succeeds.

```
fn max_alt #p #q (x y:ref int)
requires pts_to x #p 'vx ** pts_to y #q 'vy
returns n:int
ensures pts_to x #p 'vx ** pts_to y #q 'vy
        ** pure (n == max_spec 'vx 'vy)
{
    let mut result = 0;
    let vx = !x;
    let vy = !y;
    if (vx > vy)
    ensures
     exists* r.
       pts_to x #p 'vx **
       pts_to y #q 'vy **
       pts_to result r **
       pure (r == max_spec 'vx 'vy)
    {
        result := vx;
    }
    else
    {
        result := vy;
```

(continues on next page)

(continued from previous page)

```
    };
    !result;
}
```

We are working on adding inference for non-tail conditionals.

35.2 Pattern matching with nullable references

To illustrate the use of pattern matching, consider the following representation of a possibly null reference.

```
let nullable_ref a = option (ref a)

let pts_to_or_null #a
        (x:nullable_ref a)
        (#[default_arg (`1.0R)] p:perm) //implicit argument with a default
        (v:option a)
: vprop
= match x with
  | None -> pure (v == None)
  | Some x -> exists* w. pts_to x #p w ** pure (v == Some w)
```

35.2.1 Representation predicate

We can represent a nullable ref as just an `option (ref a)` coupled with a representation predicate, `pts_to_or_null`. A few points to note:

- The notation ``(#[default_arg (\`1.0R)] p:perm)`` is F* syntax for an implicit argument which when omitted defaults to `1.0R`—this is exactly how predicates like `Pulse.Lib.Reference.pts_to` are defined.
- The definition is by cases: if the reference `x` is `None`, then the logical witness is `None` too.
- Otherwise, the underlying reference points to some value `w` and the logical witness `v == Some w` agrees with that value.

Note, one might consider defining it this way:

```
let pts_to_or_null #a
     (x:nullable_ref a)
     (#[default_arg (`1.0R)] p:perm)
     (v:option a)
: vprop
= match x with
  | None -> pure (v == None)
  | Some x -> pure (Some? v) ** pts_to x #p (Some?.v v)
```

However, unlike F*'s conjunction `p /\ q` where the well-typedness of `q` can rely on `p`, the `**` operator is not left-biased; so `(Some?.v v)` cannot be proven in this context and the definition is rejected.

Another style might be as follows:

```
let pts_to_or_null #a
     (x:nullable_ref a)
```

(continues on next page)

(continued from previous page)

```
     (#[default_arg (`1.0R)] p:perm)
     (v:option a)
: vprop
= match x, v with
  | None, None -> emp
  | Some x, Some w -> pts_to x #p w
  | _ -> pure False
```

This could also work, though it would require handling an additional (impossible) case.

35.2.2 Reading a nullable ref

Let’s try our first pattern match in Pulse:

```
fn read_nullable #a #p (r:nullable_ref a)
requires pts_to_or_null r #p 'v
returns o:option a
ensures pts_to_or_null r #p 'v
        ** pure ('v == o)
{
    match r {
     Some x -> {
        rewrite each r as (Some x);
        unfold (pts_to_or_null (Some x) #p 'v);
        let o = !x;
        fold (pts_to_or_null (Some x) #p 'v);
        rewrite each (Some x) as r;
        Some o
     }
     None -> {
        rewrite each r as None;
        unfold (pts_to_or_null None #p 'v);
        fold (pts_to_or_null None #p 'v);
        rewrite each (None #(ref a)) as r;
        None #a
     }
    }
}
```

The syntax of pattern matching in Pulse is more imperative and Rust-like than what F* uses.

- The entire body of match is enclosed within braces
- Each branch is also enclosed within braces.
- Pulse (for now) only supports simple patterns with a single top-level constructor applied to variables, or variable patterns: e.g., you cannot write `Some (Some x)` as a pattern.

The type of `read_nullable` promises to return a value equal to the logical witness of its representation predicate.

The code is a little tedious—we’ll see how to clean it up a bit shortly.

A `show_proof_state` in the `Some x` branch prints the following:

```
- Current context:
    pts_to_or_null r (reveal 'v)
- In typing environment:
    [branch equality#684 : squash (eq2 r (Some x)),
     ...
```

The interesting part is the `branch equality` hypothesis, meaning that in this branch, we can assume that `(r == Some x)`. So, the first thing we do is to rewrite `r`; then we `unfold` the representation predicate; read the value `o` out of `x`; fold the predicate back; rewrite in the other direction; and return `Some o`. The `None` case is similar.

Another difference between Pulse and F* matches is that Pulse does not provide any negated path conditions. For example, in the example below, the assertion fails, since the pattern is only a wildcard and the Pulse checker does not prove `not (Some? x)` as the path condition hypothesis for the preceding branches not taken.

~~~
[@@expect_failure]
```pulse
fn read_nullable_alt #a #p (r:nullable_ref a)
requires pts_to_or_null r #p 'v
returns o:option a
ensures emp
{
    match r {
     Some x -> { admit () }
     _ -> {
        // we only have `r == _` in scope
        // not the negation of the prior branch conditions
        // i.e., unlike F*, we don't have not (Some? r)
        // so the assertion below fails
        assert (pure (r == None));
        admit() }
    }
}
```
~~~

We plan to enhance the Pulse checker to also provide these negated path conditions.

35.2.3 Helpers

When a `vprop` is defined by cases (like `pts_to_or_null`) it is very common to have to reason according to those cases when pattern matching. Instead of rewriting, unfolding, folding, and rewriting every time, one can define helper functions to handle these cases.

~~~
```pulse
ghost
fn elim_pts_to_or_null_none #a #p (r:nullable_ref a)
requires pts_to_or_null r #p 'v ** pure (r == None)
ensures pts_to_or_null r #p 'v ** pure ('v == None)
{
    rewrite each r as None;
    unfold (pts_to_or_null None #p 'v);
    fold (pts_to_or_null None #p 'v);
    rewrite each (None #(ref a)) as r;
}
~~~

(continues on next page)

(continued from previous page)

~~~
```

```pulse
ghost
fn intro_pts_to_or_null_none #a #p (r:nullable_ref a)
requires pure (r == None)
ensures pts_to_or_null r #p None
{
    fold (pts_to_or_null #a None #p None);
    rewrite each (None #(ref a)) as r;
}
```

```pulse
ghost
fn elim_pts_to_or_null_some #a #p (r:nullable_ref a) (x:ref a)
requires pts_to_or_null r #p 'v ** pure (r == Some x)
ensures exists* w. pts_to x #p w ** pure ('v == Some w)
{
    rewrite each r as (Some x);
    unfold (pts_to_or_null (Some x) #p 'v);
}
```

```pulse
ghost
fn intro_pts_to_or_null_some #a #p (r:nullable_ref a) (x:ref a)
requires pts_to x #p 'v ** pure (r == Some x)
ensures pts_to_or_null r #p (Some 'v)
{
    fold (pts_to_or_null (Some x) #p (Some 'v));
    rewrite each (Some x) as r;
}
```
~~~

These functions are all marked `ghost`, indicating that they are purely for proof purposes only.

Writing these helpers is often quite mechanical: One could imagine that the Pulse checker could automatically generate them from the definition of `pts_to_or_null`. Using F*'s metaprogramming support, a user could also auto-generate them in a custom way. For now, we write them by hand.

Using the helpers, case analyzing a nullable reference is somewhat easier:

```
fn read_nullable_alt #a #p (r:nullable_ref a)
requires pts_to_or_null r #p 'v
returns o:option a
ensures pts_to_or_null r #p 'v
        ** pure ('v == o)
{
    match r {
     Some x -> {
        elim_pts_to_or_null_some r x;
        let o = !x;
```

(continues on next page)

(continued from previous page)

```
        intro_pts_to_or_null_some r x;
        Some o
     }
     None -> {
        elim_pts_to_or_null_none r;
        None #a
     }
    }
}
```

35.2.4 Writing a nullable reference

Having defined our helpers, we can use them repeatedly. For example, here is a function to write a nullable reference.

```
fn write_nullable #a (r:nullable_ref a) (v:a)
requires pts_to_or_null r 'v
ensures exists* w. pts_to_or_null r w ** pure (Some? r ==> w == Some v)
{
    match r {
     None -> { () }
     Some x -> {
        elim_pts_to_or_null_some r x;
        x := v;
        intro_pts_to_or_null_some r x;
     }
    }
}
```

CHAPTER

THIRTYSIX

LOOPS & RECURSION

In this chapter, we'll see how various looping constructs work in Pulse, starting with `while` and also recursive functions.

By default, Pulse's logic is designed for partial correctness. This means that programs are allowed to loop forever. When we say that program returns `v:t` satisfying a postcondition `p`, this should be understood to mean that the program could loop forever, but if it does return, it is guaranteed to return a `v:t` where the state satisfies `p v`.

36.1 While loops: General form

The form of a while loop is:

```
while ( guard )
invariant (b:bool). p
{ body }
```

Where

- `guard` is a Pulse program that returns a `b:bool`
- `body` is a Pulse program that returns `unit`
- `invariant (b:bool). p` is an invariant where
  - `exists* b. p` must be provable before the loop is entered and as a postcondition of `body`.
  - `exists* b. p` is the precondition of the guard, and `p b` is its postcondition, i.e., the `guard` must satisfy:

    ```
    requires exists* b. p
    returns b:bool
    ensures p
    ```

  - the postcondition of the entire loop is `invariant false`.

One way to understand the invariant is that it describes program assertions at three different program points.

- When `b==true`, the invariant describes the program state at the start of the loop body;
- when `b==false`, the invariant describes the state at the end of the loop;
- when `b` is undetermined, the invariant describes the property of the program state just before the guard is (re)executed, i.e., at the entry to the loop and at the end of loop body.

Coming up with an invariant to describe a loop often requires some careful thinking. We'll see many examples in the remaining chapters, starting with some simple loops here.

36.1.1 Countdown

Here's our first Pulse program with a loop: `count_down` repeatedly decrements a reference until it reaches `0`.

```
fn count_down (x:ref nat)
requires pts_to x 'v
ensures pts_to x 0
{
    let mut keep_going = true;
    while (
        !keep_going
    )
    invariant b.
      exists* v.
        pts_to keep_going b **
        pts_to x v **
        pure (b == false ==> v == 0)
    {
        let n = !x;
        if (n = 0)
        {
            keep_going := false;
        }
        else
        {
            x := n - 1;
        }
    }
}
```

While loops in Pulse are perhaps a bit more general than in other languages. The `guard` is an arbitrary Pulse program, not just a program that reads some local variables. For example, here's another version of `count_down` where the `guard` does all the work and the loop body is empty, and we don't need an auxiliary `keep_going` variable.

```
fn count_down2 (x:ref nat)
requires pts_to x 'v
ensures pts_to x 0
{
    while (
        let n = !x;
        if (n = 0)
        {
            false
        }
        else
        {
            x := n - 1;
            true
        }
    )
    invariant b.
      exists* v.
        pts_to x v **
        pure (b == false ==> v == 0)
```

(continues on next page)

(continued from previous page)

```
    { () }
}
```

36.1.2 Partial correctness

The partial correctness interpretation means that the following infinitely looping variant of our program is also accepted:

```
fn count_down_loopy (x:ref nat)
requires pts_to x 'v
ensures pts_to x 0
{
    while (
        let n = !x;
        if (n = 0)
        {
            false
        }
        else
        {
            x := n + 1;
            true
        }
    )
    invariant b.
      exists* v.
        pts_to x v **
        pure (b == false ==> v == 0)
    { () }
}
```

This program increments instead of decrement `x`, but it still satisfies the same invariant as before, since it always loops forever.

We do have a fragment of the Pulse logic, notably the logic of `ghost` and `atomic` computations that is guaranteed to always terminate. We plan to also support a version of the Pulse logic for general purpose sequential programs (i.e., no concurrency) that is also terminating.

36.1.3 Multiply by repeated addition

Our next program with a loop multiplies two natural numbers `x, y` by repeatedly adding `y` to an accumulator `x` times. This program has a bit of history: A 1949 paper by Alan Turing titled "Checking a Large Routine" is often cited as the first paper about proving the correctness of a computer program. The program that Turing describes is one that implements multiplication by repeated addition.

```
fn multiply_by_repeated_addition (x y:nat)
    requires emp
    returns z:nat
    ensures pure (z == x * y)
{
    let mut ctr : nat = 0;
    let mut acc : nat = 0;
```

(continues on next page)

(continued from previous page)

```
    while (
        let c = !ctr;
        (c < x)
    )
    invariant b.
    exists* c a.
        pts_to ctr c **
        pts_to acc a **
        pure (c <= x /\
              a == (c * y) /\
              b == (c < x))
    {
        let a = !acc;
        acc := a + y;
        let c = !ctr;
        ctr := c + 1;
    };
    !acc
}
```

A few noteworthy points:

- Both the counter `ctr` and the accumulator `acc` are declared `nat`, which implicitly, by refinement typing, provides an invariant that they are both always at least `0`. This illustrates how Pulse provides a separation logic on top of F*'s existing dependent type system.
- The invariant says that the counter never exceeds `x`; the accumulator is always the product of counter and `y`; and the loop continues so long as the counter is strictly less than `x`.

36.1.4 Summing the first `N` numbers

This next example shows a Pulse program that sums the first `n` natural numbers. It illustrates how Pulse programs can developed along with pure F* specifications and lemmas.

We start with a specification of `sum`, a simple recursive function in F* along with a lemma that proves the well-known identity about this sum.

```
let rec sum (n:nat)
: nat
= if n = 0 then 0 else n + sum (n - 1)

#push-options "--z3rlimit 20"
noextract
let rec sum_lemma (n:nat)
: Lemma (sum n == n * (n + 1) / 2)
= if n = 0 then ()
  else sum_lemma (n - 1)
#pop-options
```

Now, let's say we want to implement `isum`, an iterative version of `sum`, and prove that it satisfies the identity proven by `sum_lemma`.

~~~
#push-options "--z3cliopt 'smt.arith.nl=false'"
noextract
```pulse
fn isum (n:nat)
requires emp
returns z:nat
ensures pure ((n * (n + 1) / 2) == z)
{
    let mut acc : nat = 0;
    let mut ctr : nat = 0;
    while (
        let c = !ctr;
        (c < n)
    )
    invariant b.
    exists* c a.
        pts_to ctr c **
        pts_to acc a **
        pure (c <= n /\
              a == sum c /\
              b == (c < n))
    {
        let a = !acc;
        let c = !ctr;
        ctr := c + 1;
        acc := a + c + 1;
    };
    sum_lemma n; //call an F* lemma inside Pulse
    !acc;
}
```
#pop-options
~~~

This program is quite similar to `multiply_by_repeated_addition`, but with a couple of differences:

- The invariant says that the current value of the accumulator holds the sum of the the first `c` numbers, i.e., we prove that the loop refines the recursive implementation of `sum`, without relying on any properties of non-linear arithmetic—notice, we have disabled non-linear arithmetic in Z3 with a pragma.
- Finally, to prove the identity we're after, we just call the F* `sum_lemma` that has already been proven from within Pulse, and the proof is concluded.

The program is a bit artificial, but hopefully it illustrates how Pulse programs can be shown to first refine a pure F* function, and then to rely on mathematical reasoning on those pure functions to conclude properties about the Pulse program itself.

36.2 Recursion

Pulse also supports general recursion, i.e., recursive functions that may not terminate. Here is a simple example—we'll see more examples later.

Let's start with a standard F* (doubly) recursive definition that computes the nth Fibonacci number.

```
let rec fib (n:nat) : nat =
  if n <= 1 then 1
  else fib (n - 1) + fib (n - 2)
```

One can also implement it in Pulse, as `fib_rec` while using an out-parameter to hold that values of the last two Fibonacci numbers in the sequence.

```
fn rec fib_rec (n:pos) (out:ref (nat & nat))
requires
    pts_to out 'v
ensures
    exists* v.
        pts_to out v **
        pure (
          fst v == fib (n - 1) /\
          snd v == fib n
        )
{
  if ((n = 1))
  {
    //type inference in Pulse doesn't work well here:
    //it picks (1, 1) to have type (int & int)
    //so we have to annotate
    out := ((1 <: nat), (1 <: nat));
  }
  else
  {
    fib_rec (n - 1) out;
    let v = !out;
    out := (snd v, fst v + snd v);
  }
}
```

Some points to note here:

- Recursive definitions in Pulse are introduced with `fn rec`.
- So that we can easily memoize the last two values of `fib`, we expect the argument `n` to be a positive number, rather than also allowing `0`.
- A quirk shown in the comments: We need an additional type annotation to properly type `(1, 1)` as a pair of nats.

Of course, one can also define fibonacci iteratively, with a while loop, as shown below.

```
fn fib_loop (k:pos)
  requires emp
  returns r:nat
  ensures pure (r == fib k)
```

(continues on next page)

(continued from previous page)

```
{
  let mut i : nat = 1;
  let mut j : nat = 1;
  let mut ctr : nat = 1;
  while (
    let c = !ctr;
    (c < k)
  )
  invariant b .
    exists* vi vj vctr.
        pts_to i vi **
        pts_to j vj **
        pts_to ctr vctr **
        pure (
            1 <= vctr /\
            vctr <= k /\
            vi == fib (vctr - 1) /\
            vj == fib vctr /\
            b == (vctr < k)
        )
  {
      let vi = !i;
      let vj = !j;
      let c = !ctr;
      ctr := c + 1;
      i := vj;
      j := vi + vj;
  };
  !j
}
```

CHAPTER

THIRTYSEVEN

MUTABLE ARRAYS

In this chapter, we will learn about mutable arrays in Pulse. An array is a contiguous collection of values of the same type. Similar to ref, arrays in Pulse can be allocated in the stack frame of the current function or in the heap—while the stack allocated arrays are reclaimed automatically (e.g., when the function returns), heap allocated arrays are explicitly managed by the programmer.

Pulse provides two array types: Pulse.Lib.Array.array t as the basic array type and Pulse.Lib.Vec.vec t for heap allocated arrays. To provide code reuse, functions that may operate over both stack and heap allocated arrays can be written using Pulse.Lib.Array.array t—the Pulse.Lib.Vec library provides back-and-forth coercions between vec t and array t.

37.1 array t

We illustrate the basics of array t with the help of the following example that reads an array:

```
fn read_i
  (#[@@@ Rust_generics_bounds ["Copy"]] t:Type)
  (arr:array t)
  (#p:perm)
  (#s:erased (Seq.seq t))
  (i:SZ.t { SZ.v i < Seq.length s })
  requires pts_to arr #p s
  returns x:t
  ensures pts_to arr #p s ** pure (x == Seq.index s (SZ.v i))
{
  arr.(i)
}
```

The library provides a points-to predicate pts_to arr #p s with the interpretation that in the current memory, the contents of arr are same as the (functional) sequence s:FStar.Seq.seq t. Like the pts_to predicate on reference, it is also indexed by an implicit fractional permission p, which distinguished shared, read-only access from exclusive read/write access.

In the arguments of read_i, the argument `s is erased, since it is for specification only.

Arrays can be read and written-to using indexes of type FStar.SizeT.t, a model of C size_t[1] in F*, provided that the index is within the array bounds—the refinement SZ.v i < Seq.length s enforces that the index is in bounds, where module SZ = FStar.SizeT. The function returns the i-th element of the array, the asserted by the postcondition vprop pure (x == Seq.index s (SZ.v i)). The body of the function uses the array read operator arr.(i).

[1] size_t in C is an unsigned integer type that is at least 16 bits wide. The upper bound of size_t is platform dependent. FStar.SizeT.size_t models this type and is extracted to the primitive size_t type in C, similar to the other *bounded integer types* discussed previously.

As another example, let's write to the i-th element of an array:

```
fn write_i (#t:Type) (arr:array t) (#s:erased (Seq.seq t)) (x:t) (i:SZ.t { SZ.v i < Seq.
↪length s })
  requires pts_to arr s
  ensures pts_to arr (Seq.upd s (SZ.v i) x)
{
  arr.(i) <- x
}
```

The function uses the array write operator arr(i) <- x and the postcondition asserts that in the state when the function returns, the contents of the array are same as the sequence s updated at the index i.

While any permission suffices for reading, writing requires 1.0R. For example, implementing write_i without 1.0R is rejected, as shown below.

~~~
[@@ expect_failure]
```pulse
fn write_ip #t (arr:array t) (#p:perm) (#s:erased _) (x:t) (i:SZ.t { SZ.v i < Seq.length␣
↪s })
  requires pts_to arr #p s
  ensures pts_to arr #p (Seq.upd s (SZ.v i) x)
{
  arr.(i) <- x
}
```
~~~

The library contains share and gather functions, similar to those for references, to divide and combine permissions on arrays.

We now look at a couple of examples that use arrays with conditionals, loops, existentials, and invariants, using many of the Pulse constructs we have seen so far.

37.1.1 Compare

Let's implement a function that compares two arrays for equality:

```
fn compare
  (#[@@@ Rust_generics_bounds ["PartialEq"; "Copy"]] t:eqtype)
  #p1 #p2
  (a1 a2:A.array t)
  (l:SZ.t)
  requires (
    A.pts_to a1 #p1 's1 **
    A.pts_to a2 #p2 's2 **
    pure (Seq.length 's1 == Seq.length 's2 /\ Seq.length 's2 == SZ.v l)
  )
  returns res:bool
  ensures (
    A.pts_to a1 #p1 's1 **
    A.pts_to a2 #p2 's2 **
    pure (res <==> Seq.equal 's1 's2)
  )
```

The function takes two arrays `a1` and `a2` as input, and returns a boolean. The postcondition `pure (res <==> Seq.equal 's1 's2)` specifies that the boolean is true if and only if the sequence representations of the two arrays are equal. Since the function only reads the arrays, it is parametric in the permissions `p1` and `p2` on the two arrays. Note that the type parameter `t` has type *eqtype*, requiring that values of type `t` support decidable equality.

One way to implement `compare` is to use a `while` loop, reading the two arrays using a mutable counter and checking that the corresponding elements are equal.

```
{
  let mut i = 0sz;
  while (
    let vi = !i;
    if (vi <^ l) {
      let v1 = a1.(vi);
      let v2 = a2.(vi);
      (v1 = v2)
    } else {
      false
    }
  )
  invariant b.
  exists* vi. (
    R.pts_to i vi **
    A.pts_to a1 #p1 's1 **
    A.pts_to a2 #p2 's2 **
    pure (
      SZ.v vi <= SZ.v l /\
      (b == (SZ.v vi < SZ.v l && Seq.index 's1 (SZ.v vi) = Seq.index 's2 (SZ.v vi))) /\
      (forall (i:nat). i < SZ.v vi ==> Seq.index 's1 i == Seq.index 's2 i)
    )
  )
  {
    let vi = !i;
    i := vi +^ 1sz
  };
  let vi = !i;
  let res = vi = l;
  res
}
```

The loop invariant states that (a) the arrays are pointwise equal up to the current value of the counter, and (b) the boolean `b` is true if and only if the current value of the counter is less than the length of the arrays and the arrays are equal at that index. While (a) helps proving the final postcondition of `compare`, (b) is required to maintain the invariant after the counter is incremented in the loop body.

37.1.2 Copy

As our next example, let's implement a copy function that copies the contents of the array a2 to a1.

```
fn copy
  (#[@@@ Rust_generics_bounds ["Copy"]] t:Type0)
  (a1 a2:A.array t)
  (l:SZ.t)
  (#p2:perm)
  requires (
    A.pts_to a1 's1 **
    A.pts_to a2 #p2 's2 **
    pure (Seq.length 's1 == Seq.length 's2 /\ Seq.length 's2 == SZ.v l)
  )
  ensures (
    (exists* s1. A.pts_to a1 s1 ** pure (Seq.equal s1 's2)) **
    A.pts_to a2 #p2 's2
  )
{
  let mut i = 0sz;
  while (
    let vi = !i;
    (vi <^ l)
  )
  invariant b.
  exists* vi s1. (
    R.pts_to i vi **
    A.pts_to a1 s1 **
    A.pts_to a2 #p2 's2 **
    pure (
      Seq.length s1 == Seq.length 's2 /\
      SZ.v vi <= SZ.v l /\
      (b == (SZ.v vi < SZ.v l)) /\
      (forall (i:nat). i < SZ.v vi ==> Seq.index s1 i == Seq.index 's2 i)
    )
  )
  {
    let vi = !i;
    let v = a2.(vi);
    a1.(vi) <- v;
    i := vi +^ 1sz
  }
}
```

The loop invariant existentially abstracts over the contents of a1, and maintains that up to the current loop counter, the contents of the two arrays are equal. Rest of the code is straightforward, the loop conditional checks that the loop counter is less than the array lengths and the loop body copies one element at a time.

The reader will notice that the postcondition of copy is a little convoluted. A better signature would be the following, where we directly state that the contents of a1 are same as 's2:

```
fn copy2
  (#[@@@ Rust_generics_bounds ["Copy"]] t:Type0)
  (a1 a2:A.array t)
```

(continues on next page)

(continued from previous page)

```
  (l:SZ.t)
  (#p2:perm)
  requires (
    A.pts_to a1 's1 **
    A.pts_to a2 #p2 's2 **
    pure (Seq.length 's1 == Seq.length 's2 /\ Seq.length 's2 == SZ.v l)
  )
  ensures (
    A.pts_to a1 's2 **
    A.pts_to a2 #p2 's2
  )
```

We can implement this signature, but it requires one step of rewriting at the end after the `while` loop to get the postcondition in this exact shape:

```
  // after the loop
  with v1 s1. _; //bind existentially bound witnesses from the invariant
  Seq.lemma_eq_elim s1 's2; //call an F* lemma to prove that s1 == 's2
  ()
```

We could also rewrite the predicates explicitly, as we saw in a *previous chapter*.

37.2 Stack allocated arrays

Stack arrays can be allocated using the expression `[| v; n |]`. It allocates an array of size `n`, with all the array elements initialized to `v`. The size `n` must be compile-time constant. It provides the postcondition that the newly create array points to a length `n` sequence of `v`. The following example allocates two arrays on the stack and compares them using the `compare` function above.

```
fn compare_stack_arrays ()
  requires emp
  ensures emp
{
  // |- emp
  let mut a1 = [| 0; 2sz |];
  // a1:array int |- pts_to a1 (Seq.create 0 (SZ.v 2))
  let mut a2 = [| 0; 2sz |];
  // a1 a2:array int |- pts_to a1 (Seq.create 0 (SZ.v 2)) ** pts_to a2 (Seq.create 0 (SZ.
↪v 2))
  let b = compare a1 a2 2sz;
  assert (pure b)
}
```

As with the stack references, stack arrays don't need to be deallocated or dropped, they are reclaimed automatically when the function returns. As a result, returning them from the function is not allowed:

~~~
[@@ expect_failure]
```pulse
fn ret_stack_array ()
  requires emp
  returns a:array int
~~~

(continues on next page)

(continued from previous page)

~~~
  ensures pts_to a (Seq.create 2 0)
{
  let mut a1 = [| 0; 2sz |];
  a1  // cannot prove pts_to a (Seq.create 0 2) in the context emp
}
```
~~~

37.3 Heap allocated arrays

The library Pulse.Lib.Vec provides the type vec t, for heap-allocated arrays: vec is to array as box is to ref.

Similar to array, vec is accompanied with a pts_to assertion with support for fractional permissions, share and gather for dividing and combining permissions, and read and write functions. However, unlike array, the Vec library provides allocation and free functions.

~~~
module V = Pulse.Lib.Vec

```pulse
fn heap_arrays ()
  requires emp
  returns a:V.vec int
  ensures V.pts_to a (Seq.create 2 0)
{
  let a1 = V.alloc 0 2sz;
  let a2 = V.alloc 0 2sz;
  V.free a1;
  a2  // returning vec is ok
}
```
~~~

As with the heap references, heap allocated arrays can be coerced to array using the coercion vec_to_array. To use the coercion, it is often required to convert Vec.pts_to to Array.pts_to back-and-forth; the library provides to_array_pts_to and to_vec_pts_to lemmas for this purpose.

The following example illustrates the pattern. It copies the contents of a stack array into a heap array, using the copy2 function we wrote above.

```
fn copy_app ([@@@ Rust_mut_binder] v:V.vec int)
  requires exists* s. V.pts_to v s ** pure (Seq.length s == 2)
  ensures V.pts_to v (Seq.create 2 0)
{
  let mut a = [| 0; 2sz |];
  // v, s |- V.pts_to v s ** ...
  V.to_array_pts_to v;
  // v, s |- pts_to (vec_to_array v) s ** ...
  copy2 (V.vec_to_array v) a 2sz;
  // v, s |- pts_to (vec_to_array v) (Seq.create 2 0) ** ...
  V.to_vec_pts_to v
  // v, s |- V.pts_to v (Seq.create 2 0) ** ...
}
```

Note how the assertion for v transforms from V.pts_to to pts_to (the points-to assertion for arrays) and back. It means that array algorithms and routines can be implemented with the `array t` type, and then can be reused for both stack- and heap-allocated arrays.

Finally, though the name `vec a` evokes the Rust `std::Vec` library, we don't yet support automatic resizing.

CHAPTER

THIRTYEIGHT

GHOST COMPUTATIONS

Throughout the chapters on pure F*, we made routine use of the `Lemmas` and ghost functions to prove properties of our programs. Lemmas, you will recall, are pure, total functions that always return `unit`, i.e., they have no computational significance and are erased by the F* compiler. F* *ghost functions* are also pure, total functions, except that they are allowed to inspect erased values in a controlled way—they too are erased by the F* compiler.

As we've seen already, F* lemmas and ghost functions can be directly used in Pulse code. But, these are only useful for describing properties about the pure values in scope. Often, in Pulse, one needs to write lemmas that speak about the state, manipulate `vprops`, etc. For this purpose, Pulse provides its own notion of *ghost computations* (think of these as the analog of F* lemmas and ghost functions, except they are specified using `vprops`); and *ghost state* (think of these as the analog of F* erased types, except ghost state is mutable, though still computationally irrelevant). Ghost computations are used everywhere in Pulse—we've already seen a few example. Ghost state is especially useful in proofs of concurrent programs.

38.1 Ghost Functions

Here's a Pulse function that fails to check, with the error message below.

~~~
[@@expect_failure]
```pulse
fn incr_erased_non_ghost (x:erased int)
requires emp
returns y:int
ensures emp ** pure (y == x + 1)
{
  let x = reveal x;
  (x + 1)
}
```
~~~

```
Cannot bind ghost expression reveal x with ST computation
```

We should expect this to fail, since the program claims to be able to compute an integer `y` by incrementing an erased integer `x`—the `x:erased int` doesn't exist at runtime, so this program cannot be compiled.

But, if we tag the function with the `ghost` qualifier, then this works:

```
ghost
fn incr_erased (x:erased int)
requires emp
returns y:int
```

(continues on next page)

(continued from previous page)

```
ensures emp ** pure (y == x + 1)
{
  let x = reveal x;
  (x + 1)
}
```

The `ghost` qualifier indicates the the Pulse checker that the function is to be erased at runtime, so `ghost` functions are allowed to make use of F* functions with `GTot` effect, like `FStar.Ghost.reveal`.

However, for this to be sound, no compilable code is allowed to depend on the return value of a `ghost` function. So, the following code fails with the error below:

~~~
[@@expect_failure]
```pulse
fn use_incr_erased (x:erased int)
requires emp
returns y:int
ensures emp ** pure (y == x + 1)
{
  incr_erased x;
}
```
~~~

```
Expected a term with a non-informative (e.g., erased) type; got int
```

That is, when calling a `ghost` function from a non-ghost context, the return type of the ghost function must be non-informative, e.g, `erased`, or `unit` etc. The class of non-informative types and the rules for allowing F* *ghost computations to be used in total contexts is described here*, and the same rules apply in Pulse.

To use of `incr_erased` in non-ghost contexts, we have to erase its result. There are a few ways of doing this.

Here's a verbose but explicit way, where we define a nested ghost function to wrap the call to `incr_erased`.

```
fn use_incr_erased (x:erased int)
requires emp
returns y:erased int
ensures emp ** pure (y == x + 1)
{
  ghost
  fn wrap (x:erased int)
  requires emp
  returns y:erased int
  ensures emp ** pure (y == x + 1)
  {
    let y = incr_erased x;
    hide y
  };
  wrap x
}
```

The library also contains `Pulse.Lib.Pervasives.call_ghost` that is a higher-order combinator to erase the result of a ghost call.

```
fn use_incr_erased_alt (x:erased int)
requires emp
returns y:erased int
ensures emp ** pure (y == x + 1)
{
  call_ghost incr_erased x;
}
```

The `call_ghost` combinator can be used with ghost functions of different arities, though it requires the applications to be curried in the following way.

Suppose we have a binary ghost function, like `add_erased`:

```
ghost
fn add_erased (x y:erased int)
requires emp
returns z:int
ensures emp ** pure (z == x + y)
{
  let x = reveal x;
  let y = reveal y;
  (x + y)
}
```

To call it in a non-ghost context, one can do the following:

```
fn use_add_erased (x y:erased int)
requires emp
returns z:erased int
ensures emp ** pure (z == x + y)
{
  call_ghost (add_erased x) y
}
```

That said, since `ghost` functions must have non-informative return types to be usable in non-ghost contexts, it's usually best to define them that way to start with, rather than having to wrap them at each call site, as shown below:

```
ghost
fn add_erased_erased (x y:erased int)
requires emp
returns z:erased int
ensures emp ** pure (z == x + y)
{
  let x = reveal x;
  let y = reveal y;
  hide (x + y)
}
```

38.2 Some Primitive Ghost Functions

Pulse ghost functions with `emp` or `pure _` pre and postconditions are not that interesting—such functions can usually be written with regular F* ghost functions.

Ghost functions are often used as proof steps to prove equivalences among `vprops`. We saw a few *examples of ghost functions before*—they are ghost since their implementations are compositions of `ghost` functions from the Pulse library.

The `rewrite` primitive that we saw *previously* is in fact a defined function in the Pulse library. Its signature looks like this:

```
ghost
fn __rewrite (p q:vprop)
requires p ** pure (p == q)
ensures q
```

Many of the other primitives like `fold`, `unfold`, etc. are defined in terms of `rewrite` and are `ghost` computations.

Other primitives like `introduce exists*` are also implemented in terms of library `ghost` functions, with signatures like the one below:

```
ghost
fn intro_exists (#a:Type0) (p:a -> vprop) (x:erased a)
requires p x
ensures exists* x. p x
```

38.3 Recursive Predicates and Ghost Lemmas

We previously saw how to *define custom predicates*, e.g., for representation predicates on data structures. Since a `vprop` is just a regular type, one can also define `vprops` by recursion in F*. Working with these recursive predicates in Pulse usually involves writing recursive ghost functions as lemmas. We'll look at a simple example of this here and revisit in subsequent chapters as look at programming unbounded structures, like linked lists.

Say you have a list of references and want to describe that they all contain integers whose value is at most `n`. The recursive predicate `all_at_most l n` does just that:

```
let rec all_at_most (l:list (ref nat)) (n:nat)
: vprop
= match l with
  | [] -> emp
  | hd::tl -> exists* (i:nat). pts_to hd i ** pure (i <= n) ** all_at_most tl n
```

As we did when working with *nullable references*, it's useful to define a few helper ghost functions to introduce and eliminate this predicate, for each of its cases.

The elimination and introduction functions for the `[]` case are very similar to what had previously with nullable references:

```
ghost
fn intro_all_at_most_nil (l:list (ref nat)) (n:nat)
requires pure (l == [])
ensures all_at_most l n
{
```

(continues on next page)

(continued from previous page)

```
  fold (all_at_most [] n);
  rewrite each (Nil #(ref nat)) as l;
}
```

```
ghost
fn elim_all_at_most_nil (l:list (ref nat)) (n:nat)
requires all_at_most l n ** pure (l == [])
ensures emp
{
  rewrite (all_at_most l n) as (all_at_most [] n);
  unfold (all_at_most [] n);
}
```

For the `hd::tl` case, we have to be a bit more careful in how we do the rewriting: using a regular `rewrite` fails, since it involves proving an equality between two higher-order terms (the body of `exists*` is represented in F* as a lambda term). Pulse also provides a `rewrite_by` function in the library that allows rewriting terms using a user-provided tactic. In this case, the tactic `Pulse.Lib.Pervasives.vprop_equiv_norm` equates `vprops` by proving them definitionally equal by normalization.

```
ghost
fn intro_all_at_most_cons (l:list (ref nat)) (hd:ref nat) (tl:list (ref nat)) (n:nat)
requires pts_to hd 'i ** all_at_most tl n ** pure ('i <= n) ** pure (l == hd :: tl)
ensures all_at_most l n
{
  rewrite_by (exists* (i:nat). pts_to hd i ** pure (i <= n) ** all_at_most tl n)
             (all_at_most (hd::tl) n)
             vprop_equiv_norm ();
  rewrite each (hd::tl) as l;
}
```

```
ghost
fn elim_all_at_most_cons (l:list (ref nat)) (hd:ref nat) (tl:list (ref nat)) (n:nat)
requires all_at_most l n ** pure (l == hd :: tl)
ensures exists* (i:nat). pts_to hd i ** pure (i <= n) ** all_at_most tl n
{
  rewrite (all_at_most l n) as (all_at_most (hd :: tl) n);
  rewrite_by (all_at_most (hd::tl) n)
             (exists* (i:nat). pts_to hd i ** pure (i <= n) ** all_at_most tl n)
             vprop_equiv_norm ();
}
```

38.3.1 Recursive Ghost Lemmas

Pulse allows writing recursive ghost functions as lemmas for use in Pulse code. Like F* lemmas, recursive ghost functions must be proven to terminate on all inputs—otherwise, they would not be sound.

To see this in action, let's write a ghost function to prove that `all_at_most l n` can be weakened to `all_at_most l m` when `n <= m`.

```
ghost
fn rec weaken_at_most (l:list (ref nat)) (n:nat) (m:nat)
```

(continues on next page)

(continued from previous page)

```
requires all_at_most l n ** pure (n <= m)
ensures all_at_most l m
decreases l
{
  match l {
    Nil -> {
      elim_all_at_most_nil l n;
      intro_all_at_most_nil l m
    }
    Cons hd tl -> {
      elim_all_at_most_cons l hd tl n;
      weaken_at_most tl n m;
      intro_all_at_most_cons l hd tl m
    }
  }
}
```

A few points to note:

- Recursive functions in Pulse are defined using `fn rec`.
- Ghost recursive functions must also have a `decreases` annotation—unlike in F*, Pulse does not yet attempt to infer a default decreases annotation. In this case, we are recursing on the list `l`.
- List patterns in Pulse do not (yet) have the same syntactic sugar as in F*, i.e., you cannot write `[]` and `hd::tl` as patterns.
- The proof itself is fairly straightforward:
  - In the `Nil` case, we eliminate the `all_at_most` predicate at `n` and introduce it at `m`.
  - In the `Cons` case, we eliminate `all_at_most l n`,` use the induction hypothesis to weaken the `all_at_most` predicate on the `tl`; and then introduce it again, packaging it with assumption on `hd`.

38.4 Mutable Ghost References

The underlying logic that Pulse is based on actually supports a very general form of ghost state based on partial commutative monoids (PCMs). Users can define their own ghost state abstractions in F* using PCMs and use these in Pulse programs. The library `Pulse.Lib.GhostReference` provides the simplest and most common form of ghost state: references to erased values with a fractional-permission-based ownership discipline.

We'll use the module abbreviation `module GR = Pulse.Lib.GhostReference` in what follows. The library is very similar to `Pulse.Lib.Reference`, in that it provides:

- `GR.ref a`: The main type of ghost references. `GR.ref` is an erasable type and is hence considered non-informative.
- `GR.pts_to (#a:Type0) (r:GR.ref a) (#p:perm) (v:a) : vprop` is the main predicate provided by the library. Similar to the regular `pts_to`, the permission index defaults to `1.0R`.
- Unlike `ref a` (and more like `box a`), ghost references `GR.ref a` are not lexically scoped: they are allocated using `GR.alloc` and freed using `GR.free`. Of course, neither allocation nor free'ing has any runtime cost—these are just ghost operations.
- Reading a ghost reference using `!r` returns an `erased a`, when `r:GR.ref a`. Likewise, to update `r`, it is enough to provide a new value `v:erased a`.

- Operations to `share` and `gather` ghost references work just as with `ref`.

38.4.1 A somewhat contrived example

Most examples that require ghost state usually involve stating interesting invariants between multiple threads, or sometimes in a sequential setting to correlate knowledge among different components. We'll see examples of that in a later chapter. For now, here's a small example that gives a flavor of how ghost state can be used.

Suppose we want to give a function read/write access to a reference, but want to ensure that before it returns, it resets the value of the reference to its original value. The simplest way to do that would be to give the function the following signature:

```
fn uses_but_resets #a (x:ref a)
requires pts_to x 'v
ensures pts_to x 'v
```

Here's another way to do it, this time with ghost references.

First, we define a predicate `correlated` that holds full permission to a reference and half permission to a ghost reference, forcing them to hold the same value.

```
let correlated #a (x:ref a) (y:GR.ref a) (v:a)=
  pts_to x v ** GR.pts_to y #0.5R v
```

Now, here's the signature of a function `use_temp`: at first glance, from its signature alone, one might think that the witness `v0` bound in the precondition is unrelated to the `v1` bound in the postcondition.

```
fn use_temp (x:ref int) (y:GR.ref int)
requires exists* v0. correlated x y v0
ensures exists* v1. correlated x y v1
```

But, `use_temp` only has half-permission to the ghost reference and cannot mutate it. So, although it can mutate the reference itself, in order to return its postcondition, it must reset the reference to its initial value.

```
{
  unfold correlated;
  let v = !x;
  x := 17; //temporarily mutate x, give to to another function to use with full perm
  x := v; //but, we're forced to set it back to its original value
  fold correlated;
}
```

This property can be exploited by a caller to pass a reference to `use_temp` and be assured that the value is unchanged when it returns.

```
fn use_correlated ()
requires emp
ensures emp
{
  let mut x = 17;
  let g = GR.alloc 17;
  GR.share g;
  fold correlated;  // GR.pts_to g #0.5R 17 ** correlated x g 17
  use_temp x g;     // GR.pts_to g #0.5R 17 ** correlated x g ?v1
```

(continues on next page)

(continued from previous page)

```
  unfold correlated; // GR.pts_to g #0.5R 17 ** GR.pts_to g #0.5R ?v1 ** pts_to x ?v1
  GR.gather g;       //this is the crucial step
                     // GT.pts_to g 17 ** pure (?v1 == 17) ** pts_to x ?v1
  assert (pts_to x 17);
  GR.free g;
}
```

CHAPTER THIRTYNINE

HIGHER ORDER FUNCTIONS

Like F*, Pulse is higher order. That is, Pulse functions are first class and can be passed to other functions, returned as results, and some functions can even be stored in the heap.

39.1 Pulse Computation Types

Here's perhaps the simplest higher-order function: `apply` abstracts function application.

```
fn apply (#a:Type0)
         (#b:a -> Type0)
         (#pre:a -> vprop)
         (#post: (x:a -> b x -> vprop))
         (f: (x:a -> stt (b x) (pre x) (fun y -> post x y)))
         (x:a)
requires pre x
returns y:b x
ensures post x y
{
  f x
}
```

This function is polymorphic in the argument, result type, pre, and post-condition of a function `f`, which it applies to an argument `x`. This is the first time we have written the type of a Pulse function as an F* type. So far, we have been writing *signatures* of Pulse functions, using the `fn/requires/ensures` notation, but here we see that the type of Pulse function is of the form:

```
x:a -> stt b pre (fun y -> post)
```

where,

- like any F* function type, Pulse functions are dependent and the right hand side of the arrow can mention `x`
- immediately to the right of the arrow is a Pulse computation type tag, similar to F*'s `Tot`, or `GTot`, etc.
- The tag `stt` is the most permissive of Pulse computation type tags, allowing the function's body to read and write the state, run forever etc., but with pre-condition `pre`, return type `b`, and post-condition `fun y -> post`.

Pulse provides several other kinds of computation types. For now, the most important is the constructor for ghost computations. We show below `apply_ghost`, the analog of `apply` but for `ghost` functions.

```
ghost
fn apply_ghost
```

(continues on next page)

(continued from previous page)

```
          (#a:Type0)
          (#b:a -> Type0)
          (#pre:a -> vprop)
          (#post: (x:a -> b x -> vprop))
          (f: (x:a -> stt_ghost (b x) emp_inames (pre x) (fun y -> post x y)))
          (x:a)
requires pre x
returns y:b x
ensures post x y
{
  f x
}
```

The type of f is similar to what we had before, but this time we have:

- computation type tag stt_ghost, indication that this function reads or writes ghost state only, and always terminates.
- the return type is b x
- the next argument is emp_inames, describes the set of invariants that a computation may open, where emp_inames means that this computation opens no invariants. For now, let's ignore this.
- the precondition is pre x and the postcondition is fun y -> post x y.

39.1.1 Universes

For completeness, the signature of stt and stt_ghost are shown below:

```
val stt (a:Type u#a) (i:inames) (pre:vprop) (post: a -> vprop)
  : Type u#0

val stt_ghost (a:Type u#a) (i:inames) (pre:vprop) (post: a -> vprop)
  : Type u#4
```

A point to note is that stt computations live in universe u#0. This is because stt computations are allowed to infinitely loop, and are built upon *the effect of divergence*, or Div, which, as we learned earlier, lives in universe u#0. The universe of stt means that one can store an stt function in an reference, e.g., ref (unit -> stt unit p q) is a legal type in Pulse.

In contrast, stt_ghost functions are total and live in universe 4. You cannot store a stt_ghost function in the state, since that would allow writing non-terminating functions in stt_ghost.

39.2 Counters

For a slightly more interesting use of higher order programming, let's look at how to program a mutable counter. We'll start by defining the type ctr of a counter.

```
noeq
type ctr = {
    inv: int -> vprop;
    next: i:erased int -> stt int (inv i) (fun y -> inv (i + 1) ** pure (y == reveal i));
```

(continues on next page)

(continued from previous page)

```
    destroy: i:erased int -> stt unit (inv i) (fun _ -> emp)
}
```

A counter packages the following:

- A predicate inv on the state, where inv i states that the current value of the counter is i, without describing exactly how the counter's state is implemented.
- A stateful function next that expects the inv i, returns the current value i of the counter, and provides inv (i + 1).
- A stateful function destroy to deallocate the counter.

One way to implement a ctr is to represent the state with a heap-allocated reference. This is what new_counter does below:

```
fn new_counter ()
requires emp
returns c:ctr
ensures c.inv 0
{
    open Pulse.Lib.Box;
    let x = alloc 0;
    fn next (i:erased int)
    requires pts_to x i
    returns j:int
    ensures pts_to x (i + 1) ** pure (j == reveal i)
    {
        let j = !x;
        x := j + 1;
        j
    };
    fn destroy (i:erased int)
    requires pts_to x i
    ensures emp
    {
       free x
    };
    let c = { inv = pts_to x; next; destroy };
    rewrite (pts_to x 0) as (c.inv 0);
    c
}
```

Here's how it works.

First, we allocate a new heap reference x initialized to 0.

Pulse allows us to define functions within any scope. So, we define two local functions next and destroy, whose implementations and specifications are straightforward. The important bit is that they capture the reference x:box int in their closure.

Finally, we package next and destroy into a c:ctr, instantiating inv to Box.pts_to x, rewrite the context assertion to c.inv 0, and return c.

In a caller's context, such as test_counter below, the fact that the counter is implemented using a single mutable heap reference is completely hidden.

```
fn test_counter ()
requires emp
ensures emp
{
    let c = new_counter ();
    let next = c.next;
    let destroy = c.destroy;
    let x = next _; //FIXME: Should be able to write c.next
    assert pure (x == 0);
    let x = next _;
    assert pure (x == 1);
    destroy _;
}
```

CHAPTER

FORTY

IMPLICATION AND UNIVERSAL QUANTIFICATION

In this chapter, we'll learn about two more separation logic connectives, `@==>` and `forall*`. We show a few very simple examples using them, though these will be almost trivial. In the next chapter, on linked lists, we'll see more significant uses of these connectives.

40.1 Trades, or Separating Ghost Implication

The library `module I = Pulse.Lib.Stick.Util` defines the operator *trade* (`@==>`) and utilities for using it. In the literature, the operator `p --* q` is pronounced "p magic-wand q"; `p @==> q` is similar, though there are some important technical differences, as we'll see. We'll just pronounce it `p for q`, or `p trades for q`. Here's an informal description of what `p @==> q` means:

> `p @==> q` says that if you have `p` then you can *trade* it for `q`. In other words, from `p ** (p @==> q)`, you can derive `q`. This step of reasoning is performed using a ghost function `I.elim` with the signature below:

```
ghost
fn I.elim (p q:vprop)
requires p ** (p @==> q)
ensures q
```

Importantly, if you think of `p` as describing permission on a resource, the `I.elim` makes you *give up* the permission `p` and get `q` as a result. Note, during this step, you also lose permission on the implication, i.e., `p @==> q` lets you trade `p` for `q` just once.

But, how do you create a `p @==> q` in the first place? That's its introduction form, shown below:

```
ghost
fn I.intro (p q r:vprop)
           (elim: unit -> stt_ghost unit emp_inames (r ** p) (fun _ -> q))
requires r
ensures p @==> q
```

That is, to introduce `p @==> q`, one has to show hold permission `r`, such that a ghost function can transform `r ** p` into `q`.

40.1.1 Share and Gather

Here's a small example to see `p @==> q` at work.

```
let regain_half #a (x:GR.ref a) (v:a) =
  pts_to x #0.5R v @==> pts_to x v
```

The predicate `regain_half` says that you can trade a half-permission `pts_to x #one_half v` for a full permission `pts_to x v`. At first, this may seem counter-intuitive: how can you gain a full permission from half-permission. The thing to remember is that `p @==> q` itself holds permissions internally. In particular, `regain_half x v` holds `exists* u. pts_to x #one_half u` internally, such that if the context presents the other half, the eliminator can combine the two to return the full permission.

Let's look at how to introduce `regain_half`:

```
ghost
fn intro_regain_half (x:GR.ref int)
requires pts_to x 'v
ensures pts_to x #0.5R 'v ** regain_half x 'v
{
  ghost
  fn aux ()
  requires pts_to x #0.5R 'v ** pts_to x #0.5R 'v
  ensures pts_to x 'v
  {
    GR.gather x;
  };
  GR.share x;
  I.intro _ _ _ aux;
  fold regain_half;
}
```

The specification says that if we start out with `pts_to x 'v` then we can split it into `pts_to x #one_half v` and a `regain_half x 'v`. The normal way of splitting a permission a reference would split it into two halves—here, we just package the second half in a ghost function that allows us to gather the permission back when we need it.

In the implementation, we define an auxiliary ghost function that corresponds to the eliminator for `pts_to x #one_half 'v @==> pts_to x 'v`—it's just a `gather`. Then, we split `pts_to x 'v` into halves, call `I.intro` passing the eliminator, and the fold it into a `regain_half`. All `regain_half` has done is to package the ghost function `aux`, together the half permission on `x`, and put it into a `vprop`.

Later on, if want to use `regain_half`, we can call its eliminator—which, effectively, calls `aux` with the needed permission, as shown below.

```
ghost
fn use_regain_half (x:GR.ref int)
requires pts_to x #0.5R 'v ** regain_half x 'v
ensures pts_to x 'v
{
  unfold regain_half;
  I.elim _ _;
}
```

At this point, you may be wondering why we bother to use a `regain_half x 'v` in the first place, since one might as well have just used `pts_to x #one_half 'v` and `gather`, and you'd be right to wonder that! In this simple usage, the `(@==>)` hasn't bought us much.

40.2 Universal Quantification

Let's look at our `regain_half` predicate again:

```
let regain_half #a (x:GR.ref a) (v:a) =
  pts_to x #0.5R v @==> pts_to x v
```

This predicate is not as general as it could be: to eliminate it, it requires the caller to prove that they holds `pts_to x #one_half v`, for the same `v` as was used when the trade was introduced.

One could try to generalize `regain_half` a bit by changing it to:

```
let regain_half #a (x:GR.ref a) (v:a) =
  (exists* u. pts_to x #one_half u) @==> pts_to x v
```

This is an improvement, but it still is not general enough, since it does not relate `v` to the existentially bound `u`. What we really need is a universal quantifier.

Here's the right version of `regain_half`:

```
let regain_half_q #a (x:GR.ref a) =
  forall* u. pts_to x #0.5R u @==> pts_to x u
```

This says that no matter what `pts_to x #one_half u` the context has, they can recover full permission to it, *with the same witness* `u`.

The `forall*` quantifier and utilities to manipulate it are defined in `Pulse.Lib.Forall.Util`. The introduction and elimination forms have a similar shape to what we saw earlier for `@==>`:

```
ghost
fn FA.elim (#a:Type) (#p:a->vprop) (v:a)
requires (forall* x. p x)
ensures p v
```

The eliminator allows a *single* instantiation of the universally bound `x` to `v`.

```
ghost
fn FA.intro (#a:Type) (#p:a->vprop)
     (v:vprop)
     (f_elim : (x:a -> stt_ghost unit emp_inames v (fun _ -> p x)))
requires v
ensures (forall* x. p x)
```

The introduction form requires proving that one holds `v`, and that with `v` a ghost function can produce `p x`, for any `x`.

Note, it's very common to have universal quantifiers and trades together, so the library also provides the following combined forms:

```
ghost
fn elim_forall_imp (#a:Type0) (p q: a -> vprop) (x:a)
requires (forall* x. p x @==> q x) ** p x
ensures q x
```

and

```
ghost
fn intro_forall_imp (#a:Type0) (p q: a -> vprop) (r:vprop)
         (elim: (u:a -> stt_ghost unit emp_inames
                          (r ** p u)
                          (fun _ -> q u)))
requires r
ensures forall* x. p x @==> q x
```

40.2.1 Share and Gather, Again

Here's how one introduces regain_half_q:

```
ghost
fn intro_regain_half_q (x:GR.ref int)
requires pts_to x 'v
ensures pts_to x #0.5R 'v ** regain_half_q x
{
  ghost
  fn aux1 (u:int)
  requires pts_to x #0.5R 'v ** pts_to x #0.5R u
  ensures pts_to x u
  {
    gather x;
  };
  GR.share x;
  FA.intro_forall_imp _ _ _ aux1;
  fold regain_half_q;
}
```

Now, when we want to use it, we can trade in any half-permission on pts_to x #one_half u, for a full permission with the same u.

```
ghost
fn use_regain_half_q (x:GR.ref int)
requires pts_to x #0.5R 'u ** regain_half_q x
ensures pts_to x 'u
{
  unfold regain_half_q;
  FA.elim #_ #(fun u -> pts_to x #0.5R u @==> pts_to x u) 'u;
  I.elim _ _;
}
```

Note using the eliminator for FA.elim is quite verbose: we need to specify the quantifier term again. The way Pulse uses F*'s unifier currently does not allow it to properly find solutions to some higher-order unification problems. We expect to fix this soon.

40.3 Trades and Ghost Steps

As a final example in this section, we show that one can use package any ghost computation into a trade, including steps that may modify the ghost state. In full generality, this makes `@==>` behave more like a view shift (in Iris terminology) than a wand.

Here's a predicate `can_update` which says that one can trade a half permission to `pts_to x #one_half u` for a full permission to a *different value* `pts_to x v`.

```
let can_update (x:GR.ref int) =
  forall* u v. pts_to x #0.5R u @==>
               pts_to x v
```

In `make_can_update`, we package a ghost-state update function into a binary quantifier `forall* u v. ...`.

```
ghost
fn make_can_update (x:GR.ref int)
requires pts_to x 'w
ensures pts_to x #0.5R 'w ** can_update x
{
  ghost
  fn aux (u:int)
  requires pts_to x #0.5R 'w
  ensures forall* v. pts_to x #0.5R u @==> pts_to x v
  {
    ghost
    fn aux (v:int)
    requires pts_to x #0.5R 'w ** pts_to x #0.5R u
    ensures pts_to x v
    {
      gather x;
      x := v;
    };
    FA.intro_forall_imp _ _ _ aux;
  };
  share x;
  FA.intro _ aux;
  fold (can_update x);
}
```

And in `update`, below, we instantiate it to update the reference `x` from `'u` to `k`, and also return back a `can_update` predicate to the caller, for further use.

```
ghost
fn update (x:GR.ref int) (k:int)
requires pts_to x #0.5R 'u ** can_update x
ensures pts_to x #0.5R k ** can_update x
{
  unfold can_update;
  FA.elim #_ #(fun u -> forall* v. pts_to x #0.5R u @==> pts_to x v) 'u;
  FA.elim #_ #(fun v -> pts_to x #0.5R 'u @==> pts_to x v) k;
  I.elim _ _;
  make_can_update x;
}
```

CHAPTER

FORTYONE

LINKED LISTS

In this chapter, we develop a linked list library. Along the way, we'll see uses of recursive predicates, trades, and universal quantification.

41.1 Representing a Linked List

Let's start by defining the type of a singly linked list:

```
noeq
type node (t:Type0) = {
    head : t;
    tail : llist t;
}

and node_ptr (t:Type0) = ref (node t)

//A nullable pointer to a node
and llist (t:Type0) = option (node_ptr t)
```

A `node t` contains a `head:t` and a `tail:llist t`, a nullable reference pointing to the rest of the list. Nullable references are represented by an option, as *we saw before*.

Next, we need a predicate to relate a linked list to a logical representation of the list, for use in specifications.

```
let rec is_list #t (x:llist t) (l:list t)
: Tot vprop (decreases l)
= match l with
  | [] -> pure (x == None)
  | head::tl ->
    exists* (p:node_ptr t) (tail:llist t).
      pure (x == Some p) **
      pts_to p { head; tail } **
      is_list tail tl
```

The predicate `is_list x l` is a recursive predicate:

- When `l == []`, the reference `x` must be null.
- Otherwise, `l == head :: tl`, `x` must contains a valid reference `p`, where `p` points to `{ head; tail }` and, recursively , we have `is_list tail tl`.

41.2 Boilerplate: Introducing and Eliminating is_list

We've seen *recursive predicates in a previous chapter*, and just as we did there, we need some helper ghost functions to work with `is_list`. We expect the Pulse checker will automate these boilerplate ghost lemmas in the future, but, for now, we are forced to write them by hand.

~~~
```pulse
ghost
fn elim_is_list_nil (#t:Type0) (x:llist t)
requires is_list x 'l ** pure ('l == [])
ensures pure (x == None)
{
   rewrite each 'l as Nil #t;
   unfold (is_list x [])
}
```

```pulse
ghost
fn intro_is_list_nil (#t:Type0) (x:llist t)
requires pure (x == None)
ensures is_list x []
{
    fold (is_list x []);
}
```

```pulse
ghost
fn elim_is_list_cons (#t:Type0) (x:llist t) (l:list t) (head:t) (tl:list t)
requires is_list x l ** pure (l == head::tl)
ensures (
  exists* (p:node_ptr t) (tail:llist t).
    pure (x == Some p) **
    pts_to p { head; tail } **
    is_list tail tl
)
{
  rewrite each l as (head::tl);
  rewrite_by
    (is_list x (head::tl))
    (exists* (p:node_ptr t) (tail:llist t).
      pure (x == Some p) **
      pts_to p { head; tail } **
      is_list tail tl)
    vprop_equiv_norm
    ()
}
```



```pulse
ghost
~~~

(continues on next page)

(continued from previous page)

~~~
fn intro_is_list_cons (#t:Type0) (x:llist t) (v:node_ptr t) (#node:node t) (#tl:list t)
requires
  pts_to v node **
  is_list node.tail tl **
  pure (x == Some v)
ensures
  is_list x (node.head::tl)
{
    rewrite (pts_to v node) as (pts_to v { head=node.head; tail=node.tail });
    rewrite_by
        (exists* (v:node_ptr t) (tail:llist t).
          pure (x == Some v) **
          pts_to v { head=node.head; tail } **
          is_list tail tl)
        (is_list x (node.head::tl))
        vprop_equiv_norm
        ()
}
```
~~~

41.3 Case analyzing a nullable pointer

When working with a linked list, the first thing we'll do, typically, is to check whether a given `x:llist t` is null or not. However, the `is_list x l` predicate is defined by case analyzing `l:list t` rather than `x:llist t`, since that is makes it possible to write the predicate by recursing on the tail of `l`. So, below, we have a predicate `is_list_cases x l` that inverts `is_list x l` predicate based on whether or not `x` is null.

```
let is_list_cases #t (x:llist t) (l:list t)
: vprop
= match x with
  | None -> pure (l == [])
  | Some v ->
    exists* (n:node t) (tl:list t).
      pts_to v n **
      pure (l == n.head::tl) **
      is_list n.tail tl
```

Next, we define a ghost function to invert `is_list` into `is_list_cases`.

```
ghost
fn cases_of_is_list #t (x:llist t) (l:list t)
requires is_list x l
ensures is_list_cases x l
{
  match l {
    Nil -> {
      elim_is_list_nil x;
      fold (is_list_cases None l);
      rewrite each (None #(ref (node t))) as x;
    }
```

(continues on next page)

(continued from previous page)

```
    Cons head tl -> {
      elim_is_list_cons x l head tl;
      with w tail. _;
      let v = Some?.v x;
      rewrite each w as v;
      rewrite each tail as (({ head; tail }).tail) in (is_list tail tl);
      fold (is_list_cases (Some v) l);
      rewrite each (Some #(ref (node t)) v) as x;
    }
  }
}
```

We also define two more ghost functions that package up the call to `cases_of_is_list`.

```
ghost
fn is_list_case_none (#t:Type) (x:llist t) (#l:list t)
requires is_list x l ** pure (x == None)
ensures is_list x l ** pure (l == [])
{
  cases_of_is_list x l;
  rewrite each x as (None #(ref (node t)));
  unfold (is_list_cases None l);
  intro_is_list_nil x;
}
```

```
ghost
fn is_list_case_some (#t:Type) (x:llist t) (v:node_ptr t) (#l:list t)
requires is_list x l ** pure (x == Some v)
ensures
  exists* (node:node t) (tl:list t).
    pts_to v node **
    is_list node.tail tl **
    pure (l == node.head::tl)
{
  cases_of_is_list x l;
  rewrite each x as (Some v);
  unfold (is_list_cases (Some v) l);
}
```

41.4 Length, Recursively

With our helper functions in hand, let's get to writing some real code, starting with a function to compute the length of an `llist`.

```
fn rec length (#t:Type0) (x:llist t)
requires is_list x 'l
returns n:nat
ensures is_list x 'l ** pure (n == List.Tot.length 'l)
{
  match x {
```

(continues on next page)

(continued from previous page)

```
    None -> {
      is_list_case_none x;
      0
    }
    Some vl -> {
      is_list_case_some x vl;
      with _node _tl. _;
      let node = !vl;
      rewrite each _node as node;
      let n = length node.tail;
      intro_is_list_cons x vl;
      (1 + n)
    }
  }
}
```

The None case is simple.

Some points to note in the Some case:

- We use `with _node _tl. _` to "get our hands on" the existentially bound witnesses.
- After reading `let node = !vl`, we need `is_list node.tail _tl` to make the recursive call. But the context contains `is_list _node.tail _tl` and `node == _node`. So, we need a rewrite.
- We re-introduce the `is_list` predicate, and return `1 + n`. While the `intro_is_list_cons x vl` is a ghost step and will be erased before execution, the addition is not—so, this function is not tail recursive.

41.5 Exercise 1

Write a tail-recursive version of `length`.

41.6 Exercise 2

Index the `is_list` predicate with a fractional permission. Write ghost functions to share and gather fractional `is_list` predicates.

41.7 Length, Iteratively, with Trades

What if we wanted to implement `length` using a while loop, as is more idiomatic in a language like C. It will take us a few steps to get there, and we'll use the trade operator (`@==>`) to structure our proof.

41.7.1 Trade Tails

Our first step is to define `tail_for_cons`, a lemma stating that with permission on a node pointer (`pts_to v n`), we can build a trade transforming a permission on the tail into a permission for a cons cell starting at the given node.

```
ghost
fn tail_for_cons (#t:Type) (v:node_ptr t) (#n:node t) (tl:erased (list t))
requires
  pts_to v n
ensures
  (is_list n.tail tl @==> is_list (Some v) (n.head::tl))
{
  ghost
  fn aux ()
  requires
    pts_to v n ** is_list n.tail tl
  ensures
    is_list (Some v) (n.head::tl)
  {
    intro_is_list_cons (Some v) v
  };
  I.intro _ _ _ aux;
}
```

41.7.2 Tail of a list

Next, here's a basic operation on a linked list: given a pointer to a cons cell, return a pointer to its tail. Here's a small diagram:

```
x             tl
|             |
v             v
.---.---.     .---.---.
|   | --|---> |   | --|--> ...
.---.---.     .---.---.
```

We're given a pointer `x` to the cons cell at the head of a list, and we want to return `tl`, the pointer to the next cell (or `None`, of x this is the end of the list). But, if we want to return a pointer to `tl`, we a permission accounting problem:

- We cannot return permission to `x` to the caller, since then we would have two *aliases* pointing to the next cell in the list: the returned `tl` and `x -> next`.
- But, we cannot consume the permission to `x` either, since we would like to return permission to `x` once the return `tl` goes out of scope.

The solution here is to use a trade. The type of `tail` below says that if `x` is a non-null pointer satisfying `is_list x 'l`, then `tail` returns a pointer `y` such that `is_list y tl` (where `tl` is the tail of `'l`); and, one can trade `is_list y tl` to recover permission to `is_list x 'l`. The trade essentially says that you cannot have permission to `is_list x 'l` and `is_list y tl` at the same time, but once you give up permission on `y`, you can get back the original permission on `x`.

```
fn tail (#t:Type) (x:llist t)
requires is_list x 'l ** pure (Some? x)
returns y:llist t
```

(continues on next page)

(continued from previous page)

```
ensures exists* tl.
    is_list y tl **
    (is_list y tl @==> is_list x 'l) **
    pure (Cons? 'l /\ tl == Cons?.tl 'l)
{
    let np = Some?.v x;
    is_list_case_some x np;
    with node tl. _;
    let nd = !np;
    rewrite each node as nd;
    tail_for_cons np tl;
    nd.tail
}
```

41.7.3 length_iter

The code below shows our iterative implementation of `length`. The basic idea is simple, though the proof takes a bit of doing. We initialize a current pointer `cur` to the head of the list; and `ctr` to `0`. Then, while `cur` is not null, we move `cur` to the tail and increment `ctr`. Finally, we return the `!ctr`.

```
fn length_iter (#t:Type) (x: llist t)
requires is_list x 'l
returns n:nat
ensures is_list x 'l ** pure (n == List.Tot.length 'l)
{
  open I;
  let mut cur = x;
  let mut ctr = 0;
  I.refl (is_list x 'l); //initialize the trade for the invariant
  while (
    let v = !cur;
    Some? v
  )
  invariant b.
  exists* n ll suffix.
    pts_to ctr n **
    pts_to cur ll **
    is_list ll suffix **
    pure (n == List.Tot.length 'l - List.Tot.length suffix /\
          b == (Some? ll)) **
    (is_list ll suffix @==> is_list x 'l)
  {
    with _n _ll _suffix. _; //bind existential variables in the invariant
    let n = !ctr;
    let ll = !cur;
    rewrite each _ll as ll; //again, rewrite the context to use ll instead of _ll
    let next = tail ll;     //tail gives us back a trade
    with tl. _;
    I.trans (is_list next tl) _ _; //extend the trade, transitively
    cur := next;
    ctr := n + 1;
```

(continues on next page)

(continued from previous page)

```
  };
  with _n ll _sfx. _;
  is_list_case_none ll; //this tells us that suffix=[]; so n == List.Tot.length 'l
  I.elim _ _;           //regain ownership of x, giving up ll
  let n = !ctr;
  n
}
```

Now, for the proof. The main part is the loop invariant, which says:

- the current value of the counter is n;
- cur holds a list pointer, ll where ll contains the list represented by suffix;
- n is the the length of the prefix of the list traversed so far;
- the loop continues as long as b is true, i.e., the list pointer l is not None;
- and, the key bit: you can trade ownership on ll back for ownership on the original list x.

Some parts of this could be simplified, e.g., to avoid some of the rewrites.

One way to understand how trades have helped here is to compare length_iter to the recursive function length. In length, after each recursive call returns, we called a ghost function to repackage permission on the cons cell after taking out permission on the tail. The recursive function call stack kept track of all these pieces of ghost code that had to be executed. In the iterative version, we use the trade to package up all the ghost functions that need to be run, rather than using the call stack. When the loop terminates, we use I.elim to run all that ghost code at once.

Of course, the recursive length is much simpler in this case, but this pattern of using trades to package up ghost functions is quite broadly useful.

41.8 Append, Recursively

Here's another recursive function on linked lists: append concatenates y on to the end of x.

It's fairly straightforward: we recurse until we reach the last node of x (i.e., the tail field is None; and we set that field to point to y.

```
fn rec append (#t:Type0) (x y:llist t)
requires is_list x 'l1 ** is_list y 'l2 ** pure (Some? x)
ensures is_list x ('l1 @ 'l2)
{
  let np = Some?.v x;
  is_list_case_some x np;
  with _node _tl. _;
  let node = !np;
  rewrite each _node as node;
  match node.tail {
    None -> {
      is_list_case_none node.tail;
      elim_is_list_nil node.tail;
      np := { node with tail = y };
      rewrite each y as ({ node with tail = y }).tail in (is_list y 'l2);
      intro_is_list_cons x np;
    }
```

(continues on next page)

(continued from previous page)

```
    Some _ -> {
      append node.tail y;
      intro_is_list_cons x np;
    }
  }
}
```

The code is tail recursive in the Some _ case, but notice that we have a ghost function call *after* the recursive call. Like we did for length, can we implement an iterative version of append, factoring this ghost code on the stack into a trade?

41.9 Append, Iteratively

Let's start by defining a more general version of the tail function from before. In comparison, the postcondition of tail_alt uses a universal quantifier to say, roughly, that whatever list tl' the returns y points to, it can be traded for a pointer to x that cons's on to tl. Our previous function tail can be easily recovered by instantiating tl' to tl.

```
fn tail_alt (#t:Type) (x:llist t)
requires is_list x 'l ** pure (Some? x)
returns y:llist t
ensures exists* hd tl.
  is_list y tl **
  (forall* tl'. is_list y tl' @==> is_list x (hd::tl')) **
  pure ('l == hd::tl)
{
  let np = Some?.v x;
  is_list_case_some x np;
  with _node _tl. _;
  let node = !np;
  rewrite each _node as node;
  ghost
  fn aux (tl':list t)
    requires pts_to np node ** is_list node.tail tl'
    ensures is_list x (node.head::tl')
  {
    intro_is_list_cons x np;
  };
  FA.intro_forall_imp _ _ _ aux;
  node.tail
}
```

We'll use these quantified trades in our invariant of append_iter, shown below. The main idea of the implementation is to use a while loop to traverse to the last element of the first list x; and then to set y as the next pointer of this last element.

```
fn append_iter (#t:Type) (x y:llist t)
requires is_list x 'l1 ** is_list y 'l2 ** pure (Some? x)
ensures is_list x ('l1 @ 'l2)
{
  let mut cur = x;
  //the base case, set up the initial invariant
```

(continues on next page)

(continued from previous page)

```
  FA.intro emp (fun l -> I.refl (is_list x l));
  rewrite (forall* l. is_list x l @==> is_list x l)
      as  (forall* l. is_list x l @==> is_list x ([]@l));
  while (
    with _b ll pfx sfx. _;
    let l = !cur;
    rewrite each ll as l;
    let b = is_last_cell l; //check if we are at the last cell
    if b
    {
      false //yes, break out of the loop
    }
    else
    {
      let next = tail_alt l;
      with hd tl. _;
      (* this is the key induction step *)
      FA.trans_compose
          (is_list next) (is_list l) (is_list x)
          (fun tl -> hd :: tl)
          (fun tl -> pfx @ tl);
      //Use F* sugar for classical connectives to introduce a property
      //needed for the next rewrite
      (introduce forall tl. pfx @ (hd :: tl) == (pfx @ [hd]) @ tl
       with List.Tot.Properties.append_assoc pfx [hd] tl);
      rewrite (forall* tl. is_list next tl @==> is_list x (pfx@(hd::tl)))
           as (forall* tl. is_list next tl @==> is_list x ((pfx@[hd])@tl));
      cur := next;
      non_empty_list next; //need to prove that Some? next, for the invariant
      true
    }
  )
  invariant b.
  exists* ll pfx sfx.
    pts_to cur ll **   //cur holds the pointer to the current head of the traversal, ll
    is_list ll sfx **  //ll points to some suffix of the original list, since `pfx@sfx = 'l1` below
    //the main bit: whatever ll points to `sfx`, trade it for x pointing to the concatenation ``pfx @ sfx``
    (forall* sfx'. is_list ll sfx' @==> is_list x (pfx @ sfx')) **
    pure (
      (b==false ==> List.Tot.length sfx == 1) /\ //the loop ends when we reach the last cell
      pfx@sfx == 'l1 /\ //sfx is really the suffix
      Some? ll          //and the current list is always non-null
    )
  { () };
  with ll pfx sfx. _;
  let last = !cur;
  rewrite each ll as last;
  append_at_last_cell last y;
  FA.elim_forall_imp (is_list last) (fun sfx' -> is_list x (pfx @ sfx')) (sfx@'l2);
```

(continues on next page)

(continued from previous page)

```
  List.Tot.Properties.append_assoc pfx sfx 'l2;
  ()
}
```

There are few interesting points to note.

- The main part is the quantified trade in the invariant, which, as we traverse the list, encapsulates the ghost code that we need to run at the end to restore permission to the initial list pointer `x`.
- The library function, `FA.trans_compose` has the following signature:

  ```
  ghost
  fn trans_compose (#a #b #c:Type0)
                   (p: a -> vprop)
                   (q: b -> vprop)
                   (r: c -> vprop)
                   (f: a -> GTot b)
                   (g: b -> GTot c)
  requires
    (forall* x. p x @==> q (f x)) **
    (forall* x. q x @==> r (g x))
  ensures
    forall* x. p x @==> r (g (f x))
  ```

  We use it in the key induction step as we move one step down the list—similar to what we had in `length_iter`, but this time with a quantifier.
- Illustrating again that Pulse is a superset of pure F*, we make use of a *bit of F* sugar* in the `introduce forall` to prove a property needed for a Pulse rewrite.
- Finally, at the end of the loop, we use `FA.elim_forall_imp` to restore permission on `x`, now pointing to a concatenated list, effectively running all the ghost code we accumulated as we traversed the list.

Perhaps the lesson from all this is that recursive programs are much easier to write and prove correct that iterative ones? That's one takeaway. But, hopefully, you've seen how trades and quantifiers work and can be useful in some proofs—of course, we'll use them not just for rewriting recursive as iterative code.

41.9.1 Exercise 3

Write a function to insert an element in a list and a specific position.

41.9.2 Exercise 4

Write a function to reverse a list.

CHAPTER

FORTYTWO

ATOMIC OPERATIONS AND INVARIANTS

In this section, we finally come to some concurrency related constructs.

Concurrency in Pulse is built around two concepts:

- **Atomic operations**: operations that are guaranteed to be executed in a single-step of computation without interruption by other threads.
- **Invariants**: named predicates that are enforced to be true at all times. Atomic operations can make use of invariants, assuming they are true in the current state, and enforced to be true again once the atomic step concludes.

Based on this, and in conjunction with all the other separation logic constructs that we've learned about so far, notably the use of ghost state, Pulse enables proofs of concurrent programs.

42.1 Atomic Operations

We've learned so far about *two kinds of Pulse computations*:

- General purpose, partially correct computations, with the `stt` computation type
- Ghost computations, proven totally correct, and enforced to be computationally irrelevant with the `stt_ghost` computation type.

Pulse offers a third kind of computation, *atomic* computations, with the `stt_atomic` computation type. Here is the signature of `read_atomic` and `write_atomic` from `Pulse.Lib.Reference`:

```
atomic
fn read_atomic (r:ref U32.t) (#n:erased U32.t) (#p:perm)
requires pts_to r #p n
returns x:U32.t
ensures pts_to r #p n ** pure (reveal n == x)
```

```
atomic
fn write_atomic (r:ref U32.t) (x:U32.t) (#n:erased U32.t)
requires pts_to r n
ensures pts_to r x
```

The `atomic` annotation on these functions claims that reading and writing 32-bit integers can be done in a single atomic step of computation.

This is an assumption about the target architecture on which a Pulse program is executed. It may be that on some machines, 32-bit values cannot be read or written atomically. So, when using atomic operations, you should be careful to check that its safe to assume that these operations truly are atomic.

Pulse also provides a way for you to declare that other operations are atomic, e.g., maybe your machine supports 64-bit or 128-bit atomic operations—you can program the semantics of these operations in F* and add them to Pulse, marking them as atomic.

Sometimes, particularly at higher order, you will see atomic computations described by the computation type below:

```
val stt_atomic (t:Type) (i:inames) (pre:vprop) (post:t -> vprop)
  : Type u#4
```

Like `stt_ghost`, atomic computations are total and live in universe `u#4`. As such, you cannot store an atomic function in the state, i.e., `ref (unit -> stt_atomic t i p q)` is not a well-formed type.

Atomic computations and ghost computations are also indexed by `i:inames`, where `inames` is a set of invariant names. We'll learn about these next.

42.2 Invariants

In `Pulse.Lib.Core`, we have the following types:

```
[@@erasable]
val iref : Type0
val inv (i:iref) (p:vprop) : vprop
```

Think of `inv i p` as a predicate asserting that `p` is true in the current state and all future states of the program. Every invariant has a name, `i:iref`, though, the name is only relevant in specifications, i.e., it is erasable.

A closely related type is `iname`:

```
val iname : eqtype
let inames = erased (FStar.Set.set iname)
```

Every `iref` can be turned into an `iname`, with the function `iname_of (i:iref): GTot iname`.

Invariants are duplicable, i.e., from `inv i p` one can prove `inv i p ** inv i p`, as shown by type of `Pulse.Lib.Core.dup_inv` below:

```
val dup_inv (i:iref) (p:vprop)
  : stt_ghost unit emp_inames (inv i p) (fun _ -> inv i p ** inv i p)
```

42.2.1 Boxable predicates

Pulse's language of predicates, i.e., the type `vprop`, is stratified. The type `boxable` is a refinement of `vprop`, defined as shown below in `Pulse.Lib.Core`

```
let boxable = v:vprop { is_big v }
```

That is, certain `vprops`, i.e., those that satisfy `is_big`, are `boxable` predicates. All the predicates that we have encountered so far are boxable, except for the `inv i p` predicate. For example, `pts_to x v` is boxable; `exists* x. p x` is boxable if `p x` is boxable; etc. However `inv i p` is not boxable.

Why does this matter? It turns out that PulseCore, the logic on which Pulse is built, only allows turning boxable predicates into invariants. That is, while one can build an invariant such as `inv i (exists* v. pts_to x v)`, one **cannot** nest invariants, i.e., there is no meaningful way to construct an instance of `inv i (inv j p)`.

This restriction is a fundamental limitation of PulseCore: invariants cannot mention other invariants. In more technical terms, *invariants are predicative*. One might wonder whether this limitation is significant: after all, why might one want to construct an invariant that states that some p is already an invariant? It turns out that such predicates, although not very common, are useful and the inability to nest invariants in Pulse makes some styles of proofs awkward or perhaps even impossible. Nevertheless, forcing invariants to be predicative gives Pulse a simple foundational model in PulseCore in terms of a standard, predicative, dependently typed logic.

Let's look next at how to turn a boxable predicate into an invariant.

42.2.2 Creating an invariant and boxable predicates

Let's start by looking at how to create an invariant.

First, let's define a predicate owns x, to mean that we hold full-permission on x.

```
let owns (x:ref U32.t) : v:vprop { is_big v }= exists* v. pts_to x v
```

Notice the type annotation on owns claims that it is boxable, and indeed F*'s refinement type checker automatically proves that it is.

Now, if we can currently prove pts_to r x then we can turn it into an invariant inv i (owns r), as shown below.

```
ghost
fn create_invariant (r:ref U32.t) (v:erased U32.t)
requires pts_to r v
returns i:iref
ensures inv i (owns r)
{
    fold owns;
    new_invariant (owns r)
}
```

Importantly, when we turn pts_to r x into inv i (owns r), **we lose** ownership of pts_to r x. Remember, once we have inv i (owns r), Pulse's logic aims to prove that owns r remains true always. If we were allowed to retain pts_to r x, while also creating an inv i (owns r), we can clearly break the invariant, e.g., by freeing r.

Note: A tip: When using an inv i p, it's a good idea to make sure that p is a user-defined predicate. For example, one might think to just write inv i (exists* v. pts_to x v) instead of defining an auxiliary predicate for inv i (owns r). However, the some of the proof obligations produced by the Pulse checker are harder for the SMT solver to prove if you don't use the auxiliary predicate and you may start to see odd failures. This is something we're working to improve. In the meantime, use an auxiliary predicate.

Note, if one were to try to allocate an invariant for a non-boxable predicate, typechecking fails, as shown in the example below:

```
ghost
fn create_non_boxable_inv (p:vprop)
requires p
returns i:iref
ensures inv i p
{
  new_invariant p;
}
```

failing with an error pointing to the source location of the refinement precondition, `is_big`, at the call to `new_invariant`.

```
- Assertion failed
- The SMT solver could not prove the query. Use --query_stats for more details.
- See also ../../../lib/pulse/lib/Pulse.Lib.Core.fsti(536,29-536,37)
```

As you can see, although the language does not prevent you from writing `inv i p` for any predicate `p`, the only way to allocate an instance of `inv i p` is by provable that `p` is `boxable`. This design is convenient since the onus of proving that a predicate is boxable is only placed at the allocation site of the invariant—uses of invariants do not need to worry about the distinction between `boxable` and general `vprops`.

42.2.3 Opening an invariant

Now that we've allocated an `inv i (owns r)`, what can we do with it? As we said earlier, one can make use of the `owns r` in an atomic computation, so long as we restore it at the end of the atomic step.

The `with_invariants` construct gives us access to the invariant within the scope of at most one atomic step, preceded or succeeded by as many ghost or unobservable steps as needed.

The general form of `with_invariants` is as follows, to "open" invariants `i_1` to `i_k` in the scope of `e`.

```
with_invariants i_1 ... i_k
returns x:t
ensures post
{ e }
```

In many cases, the `returns` and `ensures` annotations are omitted, since it can be inferred.

This is syntactic sugar for the following nest:

```
with_invariants i_1 {
 ...
  with_invariants i_k
  returns x:t
  ensures post
  { e }
 ...
}
```

Here's the rule for opening a single invariant `inv i p` using `with_invariant i { e }` is as follows:

- `i` must have type `iref` and `inv i p` must be provable in the current context, for some `p:vprop`
- `e` must have the type `stt_atomic t j (p ** r) (fun x -> p ** s x)`.[1] That is, `e` requires and restores the invariant `p`, while also transforming `r` to `s x`, all in at most one atomic step. Further, the `name_of_inv i` must not be in the set `j`.
- `with_invariants i { e }` has type `stt_atomic t (add_inv i j) (inv i p ** r) (fun x -> inv i p ** s x)`. That is, `e` gets to use `p` for a step, and from the caller's perspective, the context was transformed from `r` to `s`, while the use of `p` is hidden.
- Pay attention to the `add_inv i j` index on `with_invariants`: `stt_atomic` (or `stt_ghost`) computation is indexed by the names of all the invariants that it may open.

[1] Alternatively `e` may have type `stt_ghost t j (p ** r) (fun x -> p ** s x)`, in which case the entire `with_invariants i { e }` block has type `stt_ghost t (add_inv i j) (inv i p ** r) (fun x -> inv i p ** s x)`, i.e., one can open an invariant and use it in either an atomic or ghost context.

Let's look at a few examples to see how with_invariants works.

Updating a reference

In the example below, given inv i (owns r), we can atomically update a reference with a pre- and postcondition of emp.

```
atomic
fn update_ref_atomic (r:ref U32.t) (i:iref) (v:U32.t)
requires inv i (owns r)
ensures inv i (owns r)
opens (singleton i)
{
  with_invariants i {    //owns r
     unfold owns;        //ghost step;  exists* u. pts_to r u
     write_atomic r v;   //atomic step; pts_to r v
     fold owns;          //ghost step;  owns r
  }
}
```

- At the start of the with_invariants scope, we have owns r in the context.
- The ghost step unfold owns unfolds it to its definition.
- Then, we do a single atomic action, write_atomic.
- And follow it up with a fold owns, another ghost step.
- The block within with_invariants i has type stt_atomic unit emp_inames (owns r ** emp) (fun _ -> owns r ** emp)
- Since we opened the invariant i, the type of update_ref_atomic records this in the opens (singleton i) annotation; equivalently, the type is stt_atomic unit (singleton i) (inv i (owns r)) (fun _ -> inv i (owns r)). When the opens annotation is omitted, it defaults to emp_inames, the empty set of invariant names.

Double opening is unsound

To see why we have to track the names of the opened invariants, consider the example below. If we opened the same invariant twice within the same scope, then it's easy to prove False:

~~~
[@@expect_failure]
```pulse
fn double_open_bad (r:ref U32.t) (i:inv (owns r))
requires emp
ensures pure False
{
    with_invariants i {
      with_invariants i {
        unfold owns;
        unfold owns;
        pts_to_dup_impossible r;
        fold owns;
        fold owns
~~~

(continues on next page)

(continued from previous page)

```
      }
    }
}
```
```

Here, we open the invariants `i` twice and get `owns r ** owns r`, or more than full permission to `r`—from this, it is easy to build a contradiction.

Subsuming atomic computations

Atomic computations can be silently converted to regular, `stt` computations, while forgetting which invariants they opened. For example, `update_ref` below is not marked atomic, so its type doesn't record which invariants were opened internally.

```
fn update_ref (r:ref U32.t) (i:iref) (v:U32.t)
requires inv i (owns r)
ensures inv i (owns r)
{
  with_invariants i {    //owns r
     unfold owns;        //ghost step;  exists* u. pts_to r u
     write_atomic r v;   //atomic step; pts_to r v
     fold owns;          //ghost step;  owns r
  }
}
```

This is okay, since a non-atomic computation can never appear within a `with_invariants` block—so, there's no fear of an `stt` computation causing an unsound double opening. Attempting to use a non-atomic computation in a `with_invariants` block produces an error, as shown below.

````
[@@expect_failure]
```pulse
fn update_ref_fail (r:ref U32.t) (i:iref) (v:U32.t)
requires inv i (owns r)
ensures inv i (owns r)
{
  with_invariants i {
    unfold owns;
    r := v; //not atomic
    fold owns;
  }
}
```
~~~

```
- This computation is not atomic nor ghost. `with_invariants`
  blocks can only contain atomic computations.
```
~~~

CHAPTER

# FORTYTHREE

# SPIN LOCKS

With atomic operations and invariants, we can build many useful abstractions for concurrency programming. In this chapter, we'll look at how to build a spin lock for mutual exclusion.

## 43.1 Representing a Lock

The main idea of the implementation is to represent a lock using a mutable machine word, where the value `0ul` signifies that the lock is currently released; and `1ul` signifies that the lock is currently acquired. To acquire a lock, we'll try to atomically compare-and-swap, repeating until we succeed in setting a `1ul` and acquiring the lock. Releasing the lock is simpler: we'll just set it to `0ul` (though we'll explore a subtlety on how to handle double releases).

From a specification perspective, a lock is lot like an invariant: the predicate type `lock_alive l p` states that the lock protects some property `p`. Acquiring the lock provides `p` to the caller; while releasing the lock requires the caller to give up ownership of `p`. The runtime mutual exclusion is enforced by the acquire spinning, or looping, until the lock is available.

We'll represent a lock as a pair of reference to a `U32.t` and an invariant:

```
let maybe (b:bool) (p:vprop) =
  if b then p else emp

let lock_inv (r:B.box U32.t) (p:vprop) : v:vprop { is_big p ==> is_big v } =
  exists* v. B.pts_to r v ** maybe (v = 0ul) p

noeq
type lock = {
  r:B.box U32.t;
  i:iref;
}

let lock_alive (l:lock) (p:vprop) =
  inv l.i (lock_inv l.r p)
```

The predicate `lock_inv r p` states:

- We hold full permission to the `r:box U32.t`; and
- If `r` contains `0ul`, then we also have `p`.

The lock itself pairs the concrete mutable state `box U32.t` with an invariant reference `i:iref`, where the `lock_alive l p` predicate states that `l.i` names an invariant for `lock_alive l.r p`.

## 43.2 Creating a lock

To create a lock, we implement `new_lock` below. It requires the caller to provide `p`, ceding ownership of `p` to the newly allocated `l:lock`

```
fn new_lock (p:boxable)
requires p
returns l:lock
ensures lock_alive l p
{
   let r = B.alloc 0ul;
   fold (maybe (0ul = 0ul) p);
   fold (lock_inv r p);
   let i = new_invariant (lock_inv r p);
   let l = {r;i};
   rewrite each i as l.i, r as l.r;
   fold lock_alive;
   l
}
```

Importantly, since allocating a lock involves allocating an invariant that protects the predicate `p`, we need `p` to be `boxable`.

Some notes on the implementation:

- We heap allocate a reference using `Box.alloc`, since clearly, the lock has to live beyond the scope of this function's activation.
- We use `new_invariant` to create an `inv i (lock_inv r p)`, and package it up with the newly allocated reference.

## 43.3 Duplicating permission to a lock

Locks are useful only if they can be shared between multiple threads. The `lock_alive l p` expresses ownership of a lock—but, since `lock_alive` is just an invariant, we can use `dup_inv` to duplicate `lock_alive`.

```
ghost
fn dup_lock_alive (l:lock) (p:vprop)
  requires lock_alive l p
  ensures lock_alive l p ** lock_alive l p
{
  unfold lock_alive;
  dup_inv l.i (lock_inv l.r p);
  fold lock_alive;
  fold lock_alive
}
```

## 43.4 Acquiring a lock

The signature of `acquire` is shown below: it says that with `lock_alive l p`, we can get back `p` without proving anything, i.e., the precondition is `emp`.

```
fn rec acquire (#p:vprop) (l:lock)
requires lock_alive l p
ensures lock_alive l p ** p
```

This may be seem surprising at first. But, recall that we've stashed `p` inside the invariant stored in the lock, and `acquire` is going to keep looping until such time as a CAS on the reference in the lock succeeds, allowing us to pull out `p` and return it to the caller.

The type of a compare-and-swap is shown below, from Pulse.Lib.Reference:

```
let cond b (p q:vprop) = if b then p else q
```

```
atomic
fn cas_box (r:Box.box U32.t) (u v:U32.t) (#i:erased U32.t)
requires Box.pts_to r i
returns b:bool
ensures cond b (Box.pts_to r v ** pure (reveal i == u))
               (Box.pts_to r i))
```

The specification of `cas_box r u v` says that we can try to atomically update `r` from `u` to `v`, and if the operation succeeds, we learn that the initial value (`i`) of `r` was equal to `u`.

Using `cas_box`, we can implement `acquire` using a tail-recursive function:

```
{
  unfold lock_alive;
  let b =
    with_invariants l.i
      returns b:bool
      ensures inv l.i (lock_inv l.r p) ** maybe b p
      opens (add_inv emp_inames l.i)
    {
      unfold lock_inv;
      let b = cas_box l.r 0ul 1ul;
      if b
      {
        elim_cond_true _ _ _;
        with _b. rewrite (maybe _b p) as p;
        fold (maybe false p);
        rewrite (maybe false p) as (maybe (1ul = 0ul) p);
        fold (lock_inv l.r p);
        fold (maybe true p);
        true
      }
      else
      {
        elim_cond_false _ _ _;
        fold (lock_inv l.r p);
        fold (maybe false p);
```

(continues on next page)

(continued from previous page)

```
        false
      }
    };
  fold lock_alive;
  if b { rewrite (maybe b p) as p; }
  else { rewrite (maybe b p) as emp; acquire l }
}
```

The main part of the implementation is the `with_invariants` block.

- Its return type `b:bool` and postcondition is `inv l.i (lock_inv l.r p) ** maybe b p`, signifying that after a single `cas`, we may have `p` if the `cas` succeeded, while maintaining the invariant.
- We open `l.i` to get `lock_inv`, and then try a `cas_box l.r 0ul 1ul`
- If the `cas_box` succeeds, we know that the lock was initially in the `0ul` state. So, from `lock_inv` we have `p`, and we can "take it out" of the lock and return it out of block as `maybe true p`. And, importantly, we can trivially restore the `lock_inv`, since we know its currently value is `1ul`, i.e., `maybe (1ul = 0ul) _ == emp`.
- If the `cas_box` fails, we just restore `lock_inv` and return false.

Outside the `with_invariants` block, if the CAS succeeded, then we're done: we have `p` to return to the caller. Otherwise, we recurse and try again.

## 43.5 Exercise

Rewrite the tail-recursive `acquire` using a while loop.

## 43.6 Releasing a lock

Releasing a lock is somewhat easier, at least for a simple version. The signature is the dual of `acquire`: the caller has to give up `p` to the lock.

```
fn release (#p:vprop) (l:lock)
requires lock_alive l p ** p
ensures lock_alive l p
{
  unfold lock_alive;
  with_invariants l.i
    returns _:unit
    ensures inv l.i (lock_inv l.r p)
    opens (add_inv emp_inames l.i)
  {
    unfold lock_inv;
    write_atomic_box l.r 0ul;
    drop_ (maybe _ _); //double release
    fold (maybe (0ul = 0ul) p);
    fold (lock_inv l.r p);
  };
  fold lock_alive
}
```

In this implementation, `release` unconditionally sets the reference to `0ul` and reproves the `lock_inv`, since we have `p` in context.

However, if the lock was already in the released state, it may already hold `p`—releasing an already released lock can allow the caller to leak resources.

## 43.7 Exercise

Rewrite `release` to spin until the lock is acquired, before releasing it. This is not a particularly realistic design for avoiding a double release, but it's a useful exercise.

## 43.8 Exercise

Redesign the lock API to prevent double releases. One way to do this is when acquiring to lock to give out a permission to release it, and for `release` to require and consume that permission.

## 43.9 Exercise

Add a liveness predicate, with fractional permissions, to allow a lock to be allocated, then shared among several threads, then gathered, and eventually free'd.

CHAPTER

# FORTYFOUR

# PARALLEL INCREMENT

In this chapter, we look at an example first studied by Susan Owicki and David Gries, in a classic paper titled Verifying properties of parallel programs: an axiomatic approach. The problem involves proving that a program that atomically increments an integer reference `r` twice in parallel correctly adds 2 to `r`. There are many ways to do this—Owicki & Gries' approach, adapted to a modern separation logic, involves the use of additional ghost state and offers a modular way to structure the proof.

While this is a very simple program, it captures the essence of some of the reasoning challenges posed by concurrency: two threads interact with a shared resource, contributing to it in an undetermined order, and one aims to reason about the overall behavior, ideally without resorting to directly analyzing each of the possible interleavings.

## 44.1 Parallel Blocks

Pulse provides a few primitives for creating new threads. The most basic one is parallel composition, as shown below:

```
parallel
requires p1 and p2
ensures q1 and q2
{ e1 }
{ e2 }
```

The typing rule for this construct requires:

```
val e1 : stt a p1 q1
val e2 : stt b p2 q2
```

and the `parallel` block then has the type:

```
stt (a & b) (p1 ** p2) (fun (x, y) -> q1 x ** q2 y)
```

In other words, if the current context can be split into separate parts `p1` and `p2` satisfying the preconditions of `e1` and `e2`, then the parallel block executes `e1` and `e2` in parallel, waits for both of them to finish, and if they both do, returns their results as a pair, with their postconditions on each component.

Using `parallel`, one can easily program the `par` combinator below:

```
fn par (#pf #pg #qf #qg:_)
       (f: unit -> stt unit pf (fun _ -> qf))
       (g: unit -> stt unit pg (fun _ -> qg))
requires pf ** pg
ensures qf ** qg
```

(continues on next page)

(continued from previous page)

```
{
  parallel
  requires pf and pg
  ensures qf and qg
  { f () }
  { g () };
  ()
}
```

As we saw in the *introduction to Pulse*, it's easy to increment two separate references in parallel:

```
fn par_incr (x y:ref int)
requires pts_to x 'i ** pts_to y 'j
ensures pts_to x ('i + 1) ** pts_to y ('j + 1)
{
   par (fun _ -> incr x)
       (fun _ -> incr y)
}
```

But, what if we wanted to increment the same reference in two separate threads? That is, we wanted to program something like this:

```
fn add2 (x:ref int)
requires pts_to x 'i
ensures pts_to x ('i + 2)
{
   par (fun _ -> incr x)
       (fun _ -> incr x)
}
```

But, this program doesn't check. The problem is that we have only a single `pts_to x 'i`, and we can't split it to share among the threads, since both threads require full permission to `x` to update it.

Further, for the program to correctly add `2` to `x`, each increment operations must take place atomically, e.g., if the two fragments below were executed in parallel, then they may both read the initial value of `x` first, bind it to `v, and then both update it to `v + 1`.

```
let v = !x;      ||    let v = !x;
x := v + 1;      ||    x := v + 1;
```

Worse, without any synchronization, on modern processors with weak memory models, this program could exhibit a variety of other behaviors.

To enforce synchronization, we could use a lock, e.g., shown below:

```
fn attempt (x:ref int)
requires pts_to x 'i
ensures exists* v. pts_to x v
{
  let l = L.new_lock (exists* v. pts_to x v);
  fn incr ()
  requires L.lock_alive l #0.5R (exists* v. pts_to x v)
  ensures L.lock_alive l #0.5R (exists* v. pts_to x v)
  {
```

(continues on next page)

(continued from previous page)

```
    L.acquire l;
    let v = !x;
    x := v + 1;
    L.release l
  };
  L.share2 l;
  par incr incr;
  L.gather2 l;
  L.acquire l;
  L.free l
}
```

This program is type correct and free from data races. But, since the lock holds the entire permission on `x`, there's no way to give this function a precise postcondition.

**Note:** In this section, we use an implementation of spin locks from the Pulse library, Pulse.Lib.SpinLock. Unlike the version we developed in the previous chapter, these locks use a fraction-indexed permission, `lock_alive l #f p`. The also provide a predicate, `lock_acquired l`, that indicates when the lock has been taken. With full-permission to the lock, and `lock_acquired l`, the lock can be freed—reclaiming the underlying memory. Additionally, the `lock_acquired` predicate ensures that locks cannot be double freed. As such, `Pulse.Lib.SpinLock` fixes the problems with the spin locks we introduced in the previous chapter and also provides a solution to the exercises given there.

## 44.2 A First Take, with Locks

Owicki and Gries' idea was to augment the program with auxiliary variables, or ghost state, that are purely for specification purposes. Each thread gets its own piece of ghost state, and accounts for how much that thread has contributed to the current value of shared variable. Let's see how this works in Pulse.

The main idea is captured by `lock_inv`, the type of the predicate protected by the lock:

```
let contributions (left right: GR.ref int) (i v:int) : v:vprop { is_big v }=
  exists* (vl vr:int).
    GR.pts_to left #0.5R vl **
    GR.pts_to right #0.5R vr **
    pure (v == i + vl + vr)

let lock_inv (x:ref int) (init:int) (left right:GR.ref int) : v:vprop { is_big v } =
  exists* v.
    pts_to x v **
    contributions left right init v
```

Our strawman `lock` in the `attempt` shown before had type `lock (exists* v. pts_to x v)`. This time, we add a conjunct that refines the value `v`, i.e., the predicate `contributions l r init v` says that the current value of `x` protected by the lock (i.e., `v`) is equal to `init + vl + vr`, where `init` is the initial value of `x`; `vl` is the value of the ghost state owned by the "left" thread; and `vr` is the value of the ghost state owned by the "right" thread. In other words, the predicate `contributions l r init v` shows that `v` always reflects the values of the contributions made by each thread.

Note, however, the `contributions` predicate only holds half-permission on the left and right ghost variables. The other half permission is held outside the lock and allows us to keep track of each threads contribution in our specifications.

Here's the code for the left thread, `incr_left`:

```
fn incr_left (x:ref int)
             (#p:perm)
             (#left:GR.ref int)
             (#right:GR.ref int)
             (#i:erased int)
             (lock:L.lock )
requires L.lock_alive lock #p (lock_inv x i left right) ** GR.pts_to left #0.5R 'vl
ensures L.lock_alive lock #p (lock_inv x i left right) ** GR.pts_to left #0.5R ('vl + 1)
{
  L.acquire lock;
  unfold lock_inv;
  unfold contributions;
  let v = !x;
  x := v + 1;
  GR.gather left;
  GR.write left ('vl + 1);
  GR.share left;
  fold (contributions left right i (v + 1));
  fold lock_inv;
  L.release lock
}
```

- Its arguments include `x` and the `lock`, but also both pieces of ghost state, `left` and `right`, and an erased value `i` for the initial value of `x`.
- Its precondition holds half permission on the ghost reference `left`
- Its postcondition returns half-permission to `left`, but proves that it was incremented, i.e., the contribution of the left thread to the value of `x` increased by `1`.

Notice that even though we only had half permission to `left`, the specifications says we have updated `left`—that's because we can get the other half permission we need by acquiring the lock.

- We acquire the lock and update increment the value stored in `x`.
- And then we follow the increment with several ghost steps:
  - Gain full permission on `left` by combining the half permission from the precondition with the half permission gained from the lock.
  - Increment `left`.
  - Share it again, returning half permission to the lock when we release it.
- Finally, we `GR.pts_to left #one_half (`vl + 1)` left over to return to the caller in the postcondition.

The code of the right thread is symmetrical, but in this, our first take, we have to essentially repeat the code—we'll see how to remedy this shortly.

```
fn incr_right (x:ref int)
              (#p:perm)
              (#left:GR.ref int)
              (#right:GR.ref int)
              (#i:erased int)
```

(continues on next page)

(continued from previous page)

```
              (lock:L.lock)
requires L.lock_alive lock #p (lock_inv x i left right) ** GR.pts_to right #0.5R 'vl
ensures L.lock_alive lock #p (lock_inv x i left right) ** GR.pts_to right #0.5R ('vl + 1)
{
  L.acquire lock;
  unfold lock_inv;
  unfold contributions;
  let v = !x;
  x := v + 1;
  GR.gather right;
  GR.write right ('vl + 1);
  GR.share right;
  fold (contributions left right i (v + 1));
  fold (lock_inv x i left right);
  L.release lock
}
```

Finally, we can implement `add2` with the specification we want:

```
fn add2 (x:ref int)
requires pts_to x 'i
ensures  pts_to x ('i + 2)
{
  let left = GR.alloc 0;
  let right = GR.alloc 0;
  GR.share left;
  GR.share right;
  fold (contributions left right 'i 'i);
  fold (lock_inv x 'i left right);
  let lock = L.new_lock (lock_inv x 'i left right);
  L.share2 lock;
  par (fun _ -> incr_left x lock)
      (fun _ -> incr_right x lock);
  L.gather2 lock;
  L.acquire lock;
  L.free lock;
  unfold lock_inv;
  unfold contributions;
  GR.gather left;
  GR.gather right;
  GR.free left;
  GR.free right;
}
```

- We allocate `left` and `right` ghost references, initializing them to `0`.
- Then we split them, putting half permission to both in the lock, retaining the other half.
- Then spawn two threads for `incr_left` and `incr_right`, and get as a postcondition that contributions of both threads and increased by one each.
- Finally, we acquire the lock, get `pts_to x v`, for some `v`, and `contributions left right i v`. Once we gather up the permission on `left` and `right`, and now the `contributions left right i v` tells us that `v == i + 1 + 1`, which is what we need to conclude.

## 44.3 Modularity with higher-order ghost code

Our next attempt aims to write a single function incr, rather than incr_left and incr_right, and to give incr a more abstract, modular specification. The style we use here is based on an idea proposed by Bart Jacobs and Frank Piessens in a paper titled Expressive modular fine-grained concurrency specification.

The main idea is to observe that incr_left and incr_right only differ by the ghost code that they execute. But, Pulse is higher order: so, why not parameterize a single function by incr and let the caller instantiate incr twice, with different bits of ghost code. Also, while we're at it, why not also generalize the specification of incr so that it works with any user-chosen abstract predicate, rather than contributions and left/right ghost state. Here's how:

```
fn incr (x: ref int)
        (#p:perm)
        (#refine #aspec: int -> vprop)
        (l:L.lock)
        (ghost_steps:
          (v:int -> vq:int -> stt_ghost unit
               emp_inames
               (refine v ** aspec vq ** pts_to x (v + 1))
               (fun _ -> refine (v + 1) ** aspec (vq + 1) ** pts_to x (v + 1))))
requires L.lock_alive l #p (exists* v. pts_to x v ** refine v) ** aspec 'i
ensures L.lock_alive l #p (exists* v. pts_to x v ** refine v) ** aspec ('i + 1)
 {
    L.acquire l;
    with _v. _;
    let vx = !x;
    rewrite each _v as vx;
    x := vx + 1;
    ghost_steps vx 'i;
    L.release l;
}
```

As before, incr requires x and the lock, but, this time, it is parameterized by:

- A predicate refine, which generalizes the contributions predicate from before, and refines the value that x points to.
- A predicate aspec, an abstract specification chosen by the caller, and serves as the main specification for incr, which transitions from aspec 'i to aspec ('i + 1).
- And, finally, the ghost function itself, ghost_steps, now specified abstractly in terms of the relationship between refine, aspec and pts_to x—it says, effectively, that once x has been updated, the abstract predicates refine and aspec can be updated too.

Having generalized incr, we've now shifted the work to the caller. But, incr, now verified once and for all, can be used with many different callers just by instantiating it differently. For example, if we wanted to do a three-way parallel increment, we could reuse our incr as is. Whereas, our first take would have to be completely revised, since incr_left and incr_right assume that there are only two ghost references, not three.

Here's one way to instantiate incr, proving the same specification as add2.

```
fn add2_v2 (x: ref int)
requires pts_to x 'i
ensures pts_to x ('i + 2)
{
    let left = GR.alloc 0;
```

(continues on next page)

(continued from previous page)

```
  let right = GR.alloc 0;
  GR.share left;
  GR.share right;
  fold (contributions left right 'i 'i);
  let lock = L.new_lock (
    exists* (v:int).
      pts_to x v ** contributions left right 'i v
  );
  ghost
  fn step
      (lr:GR.ref int)
      (b:bool { if b then lr == left else lr == right })
      (v vq:int)
    requires
      contributions left right 'i v **
      GR.pts_to lr #0.5R vq **
      pts_to x (v + 1)
    ensures
      contributions left right 'i (v + 1) **
      GR.pts_to lr #0.5R (vq + 1) **
      pts_to x (v + 1)
  {
    unfold contributions;
    if b
    {
      with _p _v. rewrite (GR.pts_to lr #_p _v) as (GR.pts_to left #_p _v);
      GR.gather left;
      GR.write left (vq + 1);
      GR.share left;
      with _p _v. rewrite (GR.pts_to left #_p _v) as (GR.pts_to lr #_p _v);
      fold (contributions left right 'i (v + 1));
    }
    else
    {
      with _p _v. rewrite (GR.pts_to lr #_p _v) as (GR.pts_to right #_p _v);
      GR.gather right;
      GR.write right (vq + 1);
      GR.share right;
      with _p _v. rewrite (GR.pts_to right #_p _v) as (GR.pts_to lr #_p _v);
      fold (contributions left right 'i (v + 1));
    }
  };
  L.share2 lock;
  par (fun _ -> incr x lock (step left true))
      (fun _ -> incr x lock (step right false));
  L.gather2 lock;
  L.acquire lock;
  L.free lock;
  unfold (contributions left right 'i);
  GR.gather left;
  GR.gather right;
  GR.free left;
```

(continues on next page)

(continued from previous page)

```
    GR.free right;
}
```

The code is just a rearrangement of what we had before, factoring the ghost code in `incr_left` and `incr_right` into a ghost function `step`. When we spawn our threads, we pass in the ghost code to either update the left or the right contribution.

This code still has two issues:

- The ghost `step` function is a bloated: we have essentially the same code and proof twice, once in each branch of the conditional. We can improve this by defining a custom bit of ghost state using Pulse's support for partial commutative monoids—but that's for another chapter.
- We allocate and free memory for a lock, which is inefficient—could we instead do things with atomic operations? We'll remedy that next.

### 44.3.1 Exercise

Instead of explicitly passing a ghost function, use a quantified trade.

## 44.4 A version with invariants

As a final example, in this section, we'll see how to program `add2` using invariants and atomic operations, rather than locks.

Doing this properly will require working with bounded, machine integers, e.g., `U32.t`, since these are the only types that support atomic operations. However, to illustrate the main ideas, we'll assume two atomic operations on unbounded integers—this will allow us to not worry about possible integer overflow. We leave as an exercise the problem of adapting this to `U32.t`.

```
assume
val atomic_read (r:ref int) (#p:_) (#i:erased int)
  : stt_atomic int emp_inames
    (pts_to r #p i)
    (fun v -> pts_to r #p i ** pure (reveal i == v))

assume
val cas (r:ref int) (u v:int) (#i:erased int)
  : stt_atomic bool emp_inames
    (pts_to r i)
    (fun b ->
      cond b (pts_to r v ** pure (reveal i == u))
             (pts_to r i))
```

### 44.4.1 Cancellable Invariants

The main idea of doing the add2 proof is to use an invariant instead of a lock. Just as in our previous code, add2 starts with allocating an invariant, putting `exists* v. pts_to x v ** contribution left right i v` in the invariant. Then call incr twice in different threads. However, finally, to recover `pts_to x (v + 2)`, where previously we would acquire the lock, with a regular invariant, we're stuck, since the `pts_to x v` permission is inside the invariant and we can't take it out to return to the caller.

An invariant `inv i p` guarantees that the property `p` is true and remains true for the rest of a program's execution. But, what if we wanted to only enforce `p` as an invariant for some finite duration, and then to cancel it? This is what the library `Pulse.Lib.CancellableInvariant` provides. Here's the relevant part of the API:

```
[@@ erasable]
val cinv : Type0
val iref_of (c:cinv) : GTot iref
```

The main type it offers is `cinv`, the name of a cancellable invariant.

```
ghost
fn new_cancellable_invariant (v:boxable)
requires v
returns c:cinv
ensures inv (iref_of c) (cinv_vp c v) ** active c 1.0R
```

Allocating a cancellable invariant is similar to allocating a regular invariant, except one gets an invariant for an abstract predicate `cinv_cp c v`, and a fraction-indexed predicate `active c 1.0R` which allows the cancellable invariant to be shared and gathered between threads.

The `cinv_cp c v` predicate can be used in conjunction with `active` to recover the underlying predicate `v`—but only when the invariant has not been cancelled yet—this is what `unpack_cinv_vp`, and its inverse, `pack_cinv_vp`, allow one to do.

```
ghost
fn unpack_cinv_vp (#p:perm) (#v:vprop) (c:cinv)
requires cinv_vp c v ** active c p
ensures v ** unpacked c ** active c p

ghost
fn pack_cinv_vp (#v:vprop) (c:cinv)
requires v ** unpacked c
ensures cinv_vp c v
```

Finally, if one has full permission to the invariant (`active c 1.0R`) it can be cancelled and the underlying predicate `v` can be obtained as postcondition.

```
ghost
fn cancel (#v:vprop) (c:cinv)
requires inv (iref_of c) (cinv_vp c v) ** active c 1.0R
ensures v
opens add_inv emp_inames (iref_of c)
```

### 44.4.2 An increment operation

Our first step is to build an increment operation from an `atomic_read` and a `cas`. Here is its specification:

```
fn incr_atomic
        (x: ref int)
        (#p:perm)
        (#refine #aspec: int -> vprop)
        (c:C.cinv)
        (f: (v:int -> vq:int -> stt_ghost unit
                  emp_inames
                  (refine v ** aspec vq ** pts_to x (v + 1))
                  (fun _ -> refine (v + 1) ** aspec (vq + 1) ** pts_to x (v + 1))))
requires inv (C.iref_of c) (C.cinv_vp c (exists* v. pts_to x v ** refine v)) ** aspec 'i␣
↪** C.active c p
ensures inv (C.iref_of c) (C.cinv_vp c (exists* v. pts_to x v ** refine v)) ** aspec ('i␣
↪+ 1) ** C.active c p
```

The style of specification is similar to the generic style we used with `incr`, except now we use cancellable invariant instead of a lock.

For its implementation, the main idea is to repeatedly read the current value of `x`, say `v`; and then to `cas` in `v+1` if the current value is still `v`.

The `read` function is relatively easy:

```
  atomic
  fn read ()
  requires inv (C.iref_of c) (C.cinv_vp c (exists* v. pts_to x v ** refine v)) ** C.
↪active c p
  returns v:int
  ensures inv (C.iref_of c) (C.cinv_vp c (exists* v. pts_to x v ** refine v)) ** C.
↪active c p
  opens (add_inv emp_inames (C.iref_of c))
  {
    with_invariants (C.iref_of c)
    {
        C.unpack_cinv_vp c;
        let v = atomic_read x;
        C.pack_cinv_vp #(exists* v. pts_to x v ** refine v) c;
        v
    }
  };
```

- We open the invariant `l`; then, knowing that the invariant is still active, we can unpack` it; then read the value `v`; pack it back; and return `v`.

The main loop of `incr_atomic` is next, shown below:

```
  let mut continue = true;
  fold (cond true (aspec 'i) (aspec ('i + 1)));
  while (
    with _b. _;
    let b = !continue;
    rewrite each _b as b;
```

(continues on next page)

(continued from previous page)

```
    b
  )
  invariant b.
    inv (C.iref_of c) (C.cinv_vp c (exists* v. pts_to x v ** refine v)) **
    pts_to continue b **
    C.active c p **
    cond b (aspec 'i) (aspec ('i + 1))
  {
    let v = read ();
    let next =
      with_invariants (C.iref_of c)
      returns b1:bool
      ensures inv (C.iref_of c) (C.cinv_vp c (exists* v. pts_to x v ** refine v))
          ** cond b1 (aspec 'i) (aspec ('i + 1))
          ** pts_to continue true
          ** C.active c p
      {
        C.unpack_cinv_vp c;
        let b = cas x v (v + 1);
        if b
        {
          elim_cond_true b _ _;
          elim_cond_true true _ _;
          f _ _;
          intro_cond_false (aspec 'i) (aspec ('i + 1));
          C.pack_cinv_vp #(exists* v. pts_to x v ** refine v) c;
          false
        }
        else
        {
          with p q. rewrite (cond b p q) as q;
          C.pack_cinv_vp #(exists* v. pts_to x v ** refine v) c;
          true
        }
      };
    continue := next
  };
```

The loop invariant says:

- the invariant remains active
- the local variable `continue` determines if the loop iteration continues
- and, so long as the loop continues, we still have `aspec 'i`, but when the loop ends we have `aspec ('i + 1)`

The body of the loop is also interesting and consists of two atomic operations. We first `read` the value of `x` into `v`. Then we open the invariant again try to `cas` in `v+1`. If it succeeds, we return `false` from the `with_invariants` block; otherwise `true`. And, finally, outside the `with_invariants` block, we set the `continue` variable accordingly. Recall that `with_invariants` allows at most a single atomic operation, so we having done a `cas`, we are not allowed to also set `continue` inside the `with_invariants` block.

### 44.4.3 add2_v3

Finally, we implement our parallel increment again, add2_v3, this time using invariants, though it has the same specification as before.

```
fn add2_v3 (x: ref int)
requires pts_to x 'i
ensures pts_to x ('i + 2)
{
    let left = GR.alloc 0;
    let right = GR.alloc 0;
    GR.share left;
    GR.share right;
    fold (contributions left right 'i 'i);
    let c = C.new_cancellable_invariant (
      exists* (v:int).
          pts_to x v **
          contributions left right 'i v
    );
    ghost
    fn step
        (lr:GR.ref int)
        (b:bool { if b then lr == left else lr == right })
        (v vq:int)
      requires
        contributions left right 'i v **
        GR.pts_to lr #0.5R vq **
        pts_to x (v + 1)
      ensures
        contributions left right 'i (v + 1) **
        GR.pts_to lr #0.5R (vq + 1) **
        pts_to x (v + 1)
    {
      unfold contributions;
      if b
      {
        with _p _v. rewrite (GR.pts_to lr #_p _v) as (GR.pts_to left #_p _v);
        GR.gather left;
        GR.write left (vq + 1);
        GR.share left;
        with _p _v. rewrite (GR.pts_to left #_p _v) as (GR.pts_to lr #_p _v);
        fold (contributions left right 'i (v + 1));
      }
      else
      {
        with _p _v. rewrite (GR.pts_to lr #_p _v) as (GR.pts_to right #_p _v);
        GR.gather right;
        GR.write right (vq + 1);
        GR.share right;
        with _p _v. rewrite (GR.pts_to right #_p _v) as (GR.pts_to lr #_p _v);
        fold (contributions left right 'i (v + 1));
      }
    };
```

(continues on next page)

(continued from previous page)

```
    C.share2 c;
    with pred. assert (inv (C.iref_of c) (C.cinv_vp c (exists* v. pts_to x v ** pred v)));
    dup_inv (C.iref_of c) (C.cinv_vp c (exists* v. pts_to x v ** pred v));
    par (fun _ -> incr_atomic x c (step left true))
        (fun _ -> incr_atomic x c (step right false));

    C.gather2 c;
    C.cancel c;
    unfold contributions;
    GR.gather left;
    GR.gather right;
    GR.free left;
    GR.free right;
    drop_ (inv _ _)
}
```

The code too is very similar to add2_v2, except instead of allocating a lock, we allocate a cancellable invariant. And, at the end, instead of acquiring, and leaking, the lock, we simply cancel the invariant and we're done.

## 44.5 Exercise

Implement add2 on a ref U32.t. You'll need a precondition that 'i + 2 < pow2 32 and also to strengthen the invariant to prove that each increment doesn't overflow.

CHAPTER

# FORTYFIVE

# EXTRACTION

Pulse programs can be extracted to OCaml, C, and Rust. We illustrate the extraction capabilities with the help of the Boyer-Moore majority vote algorithm implemented in Pulse.

## 45.1 Boyer-Moore majority vote algorithm

The algorithm finds majority vote in an array of votes in linear time (2n comparisons, where n is the length of the array) and constant extra memory.

We implement the algorithm in Pulse with the following specification:

~~~
let count (#a:eqtype) (x:a) (s:Seq.seq a) : GTot nat = Seq.count x s

noextract
let has_majority_in (#a:eqtype) (x:a) (s:Seq.seq a) = Seq.length s < 2 * count x s

noextract
let no_majority (#a:eqtype) (s:Seq.seq a) = forall (x:a). ~(x `has_majority_in` s)

```pulse
fn majority
  (#[@@@ Rust_generics_bounds ["Copy"; "PartialEq"]] a:eqtype)
  #p (#s:G.erased _) (votes:array a) (len:SZ.t { SZ.v len == Seq.length s })
  requires pts_to votes #p s ** pure (0 < SZ.v len /\ SZ.fits (2 * SZ.v len))
  returns x:option a
  ensures pts_to votes #p s **
          pure ((x == None ==> no_majority s) /\ (Some? x ==> (Some?.v x) `has_majority_
↪in` s))
~~~

The precondition `SZ.fits (2 * SZ.v len)` ensures safe arithmetic when counting for majority.

The algorithm consists of two phases. The first phase, called the pairing phase, pairs off disagreeing votes (cancels them) until the remaining votes are all same. The main idea of the algorithm is to do this pairing with n comparisons. After the pairing phase, the remaining vote must be the majority, *if the majority exists*. The second phase, called the counting phase, checks if the remaining vote is indeed in majority with n more comparisons.

For the first phase, the algorithm maintains three auxiliary variables, `i` for the loop counter, `cand` the current majority candidate, and a count `k`. It visits the votes in a loop, where for the `i-th` element of the array, if `k = 0`, the algorithm assigns the `i-th` vote as the new majority candidate and assigns `k = 1`. Otherwise, if the `i-th` vote is same as `cand`, it increments `k` by one, otherwise it decrements `k` by one.

The second phase then is another while loop that counts the number of votes for the majority candidate from the first phase.

```
{
  let mut i = 1sz;
  let mut k = 1sz;
  let votes_0 = votes.(0sz);
  let mut cand = votes_0;
  assert (pure (count_until votes_0 s 1 == 1));
  // while loop for phase 1
  while (
    let vi = !i;
    (vi <^ len)
  )
  invariant b.
    pts_to votes #p s **
    (exists* vi vk vcand.
       R.pts_to i vi       **
       R.pts_to k vk       **
       R.pts_to cand vcand **
       pure (
         v vi <= Seq.length s /\
         // v is a spec function that returns nat representation of an FStar.SizeT
         0 <= v vk /\ v vk <= count_until vcand s (v vi) /\
         // constraint for the current candidate,
         2 * (count_until vcand s (v vi) - v vk) <= v vi - v vk /\
         // constraint for the rest of the candidates
         (forall (vcand':a). vcand' =!= vcand ==> 2 * count_until vcand' s (v vi) <= v␣
↪vi - v vk) /\
         b == (v vi < v len)))
  {
    let vi = !i;
    let vk = !k;
    let vcand = !cand;
    let votes_i = votes.(vi);
    // count_until_next is a lemma that captures the behavior of
    // count when the sequence index is incremented
    count_until_next vcand s (SZ.v vi);
    if (vk = 0sz) {
      cand := votes_i;
      k := 1sz;
      i := vi +^ 1sz
    } else if (votes_i = vcand) {
      k := vk +^ 1sz;
      i := vi +^ 1sz
    } else {
      k := vk -^ 1sz;
      i := vi +^ 1sz
    }
  };
  let vk = !k;
  let vcand = !cand;
  // a couple of optimizations
  if (vk = 0sz) {
    None #a
  } else if (len <^ 2sz *^ vk) {
```

(continues on next page)

(continued from previous page)

```
    Some vcand
  } else {
    i := 0sz;
    k := 0sz;
    // while loop for phase 2
    while (
      let vi = !i;
      (vi <^ len)
    )
    invariant b.
      pts_to votes #p s **
      (exists* vi vk.
         R.pts_to i vi **
         R.pts_to k vk **
         pure (SZ.v vi <= Seq.length s /\
               SZ.v vk == count_until vcand s (SZ.v vi) /\
               b == (SZ.v vi < SZ.v len)))
    {
      let vi = !i;
      let vk = !k;
      let votes_i = votes.(vi);
      count_until_next vcand s (SZ.v vi);
      if (votes_i = vcand) {
        k := vk +^ 1sz;
        i := vi +^ 1sz
      } else {
        i := vi +^ 1sz
      }
    };

    let vk = !k;
    if (len <^ 2sz *^ vk) {
      Some vcand
    } else {
      None #a
    }
  }
}
```

The loop invariant for the first phase specifies majority constraints *within* the prefix of the array that the loop has visited so far. The second phase loop invariant is a simple counting invariant.

Pulse automatically proves the program, with an hint for the behavior of the count function as we increment the loop counter, the following count_until_next lemma captures the behavior, and we invoke the lemma in both the while loops:

```
let count_until_next (#a:eqtype) (x:a) (s:Seq.seq a) (j:nat { j < Seq.length s })
  : Lemma
      (ensures count_until (Seq.index s j) s (j + 1) == count_until (Seq.index s j) s j + 1 /\
               (forall (y:a). y =!= Seq.index s j ==> count_until y s (j + 1) == count_until y s j))
```

## 45.2 Rust extraction

Pulse toolchain is accompanied with a tool to extract Pulse programs to Rust. The extraction pipeline maps the Pulse syntactic constructs such as `let mut`, `while`, `if-then-else`, etc. to corresponding Rust constructs. Further, Pulse libraries are mapped to their Rust counterparts, e.g. `Pulse.Lib.Vec` to `std::vec`, `Pulse.Lib.Array` to Rust slices etc.

To extract a Pulse file to Rust, we first invoke the F* extraction pipeline with the command line option `--codegen Extension`. This emits a `.ast` file containing an internal AST representation of the file. We then invoke the Rust extraction tool that takes as input the `.ast` files and outputs the extracted Rust code (by-default the output is written to `stdout`, if an `-o <file>` option is provided to the tool, the output is written to `file`). For example, the first command produces the `.ast` file from `PulseTutorial.Algorithms.fst` (which contains the Boyer-Moore algorithm implementation), and then the second command extracts the Rust code to `voting.rs`. (These commands are run in the `pulse` root directory, change the location of main.exe according to your setup.)

```
$ fstar.exe --include lib/pulse/ --include lib/pulse/lib
  --include share/pulse/examples/by-example/ --include share/pulse/examples/_output/
↪cache/
  --load_cmxs pulse  --odir . PulseTutorial.Algorithms.fst
  --extract 'PulseTutorial.Algorithms' --codegen Extension

$ ./pulse2rust/main.exe PulseTutorial_Algorithms.ast -o voting.rs
```

The output Rust code is as shown below:

```
pub fn majority<A: Clone + Copy + PartialEq>(
    p: (),
    s: (),
    votes: &mut [A],
    len: usize,
) -> std::option::Option<A> {
    let mut i = 1;
    let mut k = 1;
    let votes_0 = votes[0];
    let mut cand = votes_0;
    while {
        let vi = i;
        vi < len
    } {
        let vi = i;
        let vk = k;
        let vcand = cand;
        let votes_i = votes[vi];
        if vk == 0 {
            cand = votes_i;
            k = 1;
            i = vi + 1
        } else if votes_i == vcand {
            k = vk + 1;
            i = vi + 1
        } else {
            k = vk - 1;
            i = vi + 1
```

(continues on next page)

(continued from previous page)

```
        };
    }
    let vk = k;
    let vcand = cand;
    let _bind_c = if vk == 0 {
        None
    } else if len < 2 * vk {
        Some(vcand)
    } else {
        i = 0;
        k = 0;
        while {
            let vi = i;
            vi < len
        } {
            let vi = i;
            let vk1 = k;
            let votes_i = votes[vi];
            if votes_i == vcand {
                k = vk1 + 1;
                i = vi + 1
            } else {
                i = vi + 1
            };
        }
        let vk1 = k;
        if len < 2 * vk1 {
            Some(vcand)
        } else {
            None
        }
    };
    let cand1 = _bind_c;
    let k1 = cand1;
    let i1 = k1;
    i1
}
```

We can test it by adding the following in `voting.rs` and running the tests (using `cargo test`, it requires a `Cargo.toml` file, we provide an example file in the repo that can be used):

```
#[derive(Copy, Clone, PartialEq, Debug)]
enum Candidate {
    A,
    B,
    C,
}

#[test]
fn test() {
    let mut votes = [0, 1, 0, 0, 0, 1];
    let winner = majority((), (), &mut votes, 6);
```

(continues on next page)

(continued from previous page)

```
    assert_eq!(winner, Some(0));

    let mut str_votes = ["a", "b", "a", "a", "c"];
    let str_winner = majority((), (), &mut str_votes, 5);
    assert_eq!(str_winner, Some("a"));

    let mut cand_votes = [Candidate::A, Candidate::B, Candidate::C, Candidate::B];
    let cand_winner = majority((), (), &mut cand_votes, 4);
    assert_eq!(cand_winner, None);
}
```

A few notes about the extracted Rust code:

- The Pulse function and the Rust function are generic in the type of the votes. In Rust, the extracted code required the type argument to implement the `Clone`, `Copy`, and `PartialEq` traits. Currently we hardcode these traits. We plan to specify these traits in Pulse through attribute mechanism
- The ghost arguments `p` and `s` appear in the Rust code as `unit` arguments, we plan to make it so that these arguments are completely erased.
- Whereas `majority` needs only read permission for the `votes` array in the Pulse signature, the extracted Rust code specifies the argument as `&mut`. The Rust extraction pipeline currently passes all the references as `mut`, we plan to make it more precise by taking into account the permissions from the Pulse signature.

## 45.3 C extraction

Pulse programs can also be extracted to C. The extraction pipeline is based on the Karamel tool. The process to extract Pulse programs to C is similar to that of extracting Low* to C, described in this tutorial. In summary, we first generate `.krml` files from using the F* extraction command line option `--codegen krml`, and then run the Karamel tool on those files.

One catch with extracting our Boyer-Moore implementation to C is that due to the lack of support of polymorphism in C, Karamel monomorphizes polymorphic functions based on their uses. So, we write a monomorphic version of the `majority` function for `u32`, that internally calls the polymorphic `majority` function:

```
fn majority_mono #p (#s:G.erased _) (votes:array u32_t) (len:SZ.t { SZ.v len == Seq.
↪length s })
  requires pts_to votes #p s ** pure (0 < SZ.v len /\ SZ.fits (2 * SZ.v len))
  returns x:option u32_t
  ensures pts_to votes #p s **
          pure ((x == None ==> no_majority s) /\ (Some? x ==> (Some?.v x) `has_majority_
↪in` s))
{
  majority #u32_t #p #s votes len
}
```

Then we extract it to C as follows (the commands are run in the `pulse` root directory as before):

```
$ fstar.exe --include lib/pulse/ --include lib/pulse/lib
  --include share/pulse/examples/by-example/ --include share/pulse/examples/_output/
↪cache/
  --load_cmxs pulse  --odir . PulseTutorial.Algorithms.fst
  --extract 'FStar.Pervasives.Native PulseTutorial.Algorithms' --codegen krml
```

(continues on next page)

(continued from previous page)

```
$ ../karamel/krml -skip-compilation out.krml
```

This produces `PulseTutorial_Algorithms.h` and `PulseTutorial_Algorithms.c` files, with the following implementation of `majority`:

```
FStar_Pervasives_Native_option__uint32_t
PulseTutorial_Algorithms_majority__uint32_t(uint32_t *votes, size_t len)
{
  size_t i = (size_t)1U;
  size_t k = (size_t)1U;
  uint32_t votes_0 = votes[0U];
  uint32_t cand = votes_0;
  size_t vi0 = i;
  bool cond = vi0 < len;
  while (cond)
  {
    size_t vi = i;
    size_t vk = k;
    uint32_t vcand = cand;
    uint32_t votes_i = votes[vi];
    if (vk == (size_t)0U)
    {
      cand = votes_i;
      k = (size_t)1U;
      i = vi + (size_t)1U;
    }
    else if (votes_i == vcand)
    {
      k = vk + (size_t)1U;
      i = vi + (size_t)1U;
    }
    else
    {
      k = vk - (size_t)1U;
      i = vi + (size_t)1U;
    }
    size_t vi0 = i;
    cond = vi0 < len;
  }
  size_t vk = k;
  uint32_t vcand = cand;
  if (vk == (size_t)0U)
    return ((FStar_Pervasives_Native_option__uint32_t){ .tag = FStar_Pervasives_Native_
↪None });
  else if (len < (size_t)2U * vk)
    return
      (
        (FStar_Pervasives_Native_option__uint32_t){
          .tag = FStar_Pervasives_Native_Some,
          .v = vcand
        }
```

(continues on next page)

(continued from previous page)

```
   );
 else
 {
  i = (size_t)0U;
  k = (size_t)0U;
  size_t vi0 = i;
  bool cond = vi0 < len;
  while (cond)
  {
   size_t vi = i;
   size_t vk1 = k;
   uint32_t votes_i = votes[vi];
   if (votes_i == vcand)
   {
    k = vk1 + (size_t)1U;
    i = vi + (size_t)1U;
   }
   else
    i = vi + (size_t)1U;
   size_t vi0 = i;
   cond = vi0 < len;
  }
  size_t vk1 = k;
  if (len < (size_t)2U * vk1)
   return
    (
     (FStar_Pervasives_Native_option__uint32_t){
      .tag = FStar_Pervasives_Native_Some,
      .v = vcand
     }
    );
  else
   return ((FStar_Pervasives_Native_option__uint32_t){ .tag = FStar_Pervasives_Native_
↪None });
 }
}
```

We can now test it with a client like:

```
#include "PulseTutorial_Algorithms.h"

int main(int argc, char **argv) {
 uint32_t votes[4] = {1, 1, 0, 1};
 FStar_Pervasives_Native_option__uint32_t result = PulseTutorial_Algorithms_majority__
↪uint32_t(votes, 4);
 if (result.tag == FStar_Pervasives_Native_None) {
  printf("No majority\n");
 } else {
  printf("Majority: %d\n", result.v);
 }
 return 0;
}
```

```
$ gcc PulseTutorial_Algorithms.c PulseTutorial_Algorithms_Client.c -I ../karamel/include/
  -I ../karamel/krmllib/c -I ../karamel/krmllib/dist/minimal/

$ ./a.out
Majority: 1

$
```

## 45.4 OCaml extraction

As with all F* programs, Pulse programs can be extracted to OCaml. One caveat with using the OCaml backend for Pulse programs is that the explicit memory management from Pulse programs does not carry over to OCaml. For example, the extracted OCaml programs rely on the OCaml garbage collector for reclaiming unused heap memory, `let mut` variables are allocated on the heap, etc.

For the Boyer-Moore example, we can extract the program to OCaml as follows:

```
$ fstar.exe --include lib/pulse/ --include lib/pulse/lib
  --include share/pulse/examples/by-example/ --include share/pulse/examples/_output/
↪cache/
  --load_cmxs pulse  --odir . PulseTutorial.Algorithms.fst
  --extract 'PulseTutorial.Algorithms' --codegen OCaml
```

and the extracted `majority` function looks like:

```
let majority p s votes len =
  let i = Pulse_Lib_Reference.alloc (FStar_SizeT.uint_to_t Prims.int_one) in
  let k = Pulse_Lib_Reference.alloc (FStar_SizeT.uint_to_t Prims.int_one) in
  let votes_0 =
    Pulse_Lib_Array_Core.op_Array_Access votes Stdint.Uint64.zero () () in
  let cand = Pulse_Lib_Reference.alloc votes_0 in
  let uu___ =
    Pulse_Lib_Core.while_
      (fun _ ->
         let vi = Pulse_Lib_Reference.op_Bang i () () in
         FStar_SizeT.lt vi len)
      (fun _ ->
         let vi = Pulse_Lib_Reference.op_Bang i () () in
         let vk = Pulse_Lib_Reference.op_Bang k () () in
         let vcand = Pulse_Lib_Reference.op_Bang cand () () in
         let votes_i = Pulse_Lib_Array_Core.op_Array_Access votes vi () () in
         let uu___ = () in
         let _bind_c =
           if vk = Stdint.Uint64.zero
           then
             let uu___1 = Pulse_Lib_Reference.op_Colon_Equals cand votes_i () in
             let uu___2 =
               Pulse_Lib_Reference.op_Colon_Equals k Stdint.Uint64.one () in
             Pulse_Lib_Reference.op_Colon_Equals i
               (FStar_SizeT.add vi Stdint.Uint64.one) ()
           else
```

(continues on next page)

(continued from previous page)

```
           if votes_i = vcand
           then
             (let uu___1 =
                Pulse_Lib_Reference.op_Colon_Equals k
                  (FStar_SizeT.add vk Stdint.Uint64.one) () in
              Pulse_Lib_Reference.op_Colon_Equals i
                (FStar_SizeT.add vi Stdint.Uint64.one) ())
           else
             (let uu___1 =
                Pulse_Lib_Reference.op_Colon_Equals k
                  (FStar_SizeT.sub vk Stdint.Uint64.one) () in
              Pulse_Lib_Reference.op_Colon_Equals i
                (FStar_SizeT.add vi Stdint.Uint64.one) ()) in
       let _while_b = () in ()) in
let vk = Pulse_Lib_Reference.op_Bang k () () in
let vcand = Pulse_Lib_Reference.op_Bang cand () () in
let _bind_c =
  if vk = Stdint.Uint64.zero
  then FStar_Pervasives_Native.None
  else
    if FStar_SizeT.lt len (FStar_SizeT.mul (Stdint.Uint64.of_int (2)) vk)
    then FStar_Pervasives_Native.Some vcand
    else
      (let uu___1 =
         Pulse_Lib_Reference.op_Colon_Equals i Stdint.Uint64.zero () in
       let uu___2 =
         Pulse_Lib_Reference.op_Colon_Equals k Stdint.Uint64.zero () in
       let uu___3 =
         Pulse_Lib_Core.while_
           (fun _ ->
              let vi = Pulse_Lib_Reference.op_Bang i () () in
              FStar_SizeT.lt vi len)
           (fun _ ->
              let vi = Pulse_Lib_Reference.op_Bang i () () in
              let vk1 = Pulse_Lib_Reference.op_Bang k () () in
              let votes_i =
                Pulse_Lib_Array_Core.op_Array_Access votes vi () () in
              let uu___3 = () in
              let _bind_c =
                if votes_i = vcand
                then
                  let uu___4 =
                    Pulse_Lib_Reference.op_Colon_Equals k
                      (FStar_SizeT.add vk1 Stdint.Uint64.one) () in
                  Pulse_Lib_Reference.op_Colon_Equals i
                    (FStar_SizeT.add vi Stdint.Uint64.one) ()
                else
                  Pulse_Lib_Reference.op_Colon_Equals i
                    (FStar_SizeT.add vi Stdint.Uint64.one) () in
              let _while_b = () in ()) in
       let vk1 = Pulse_Lib_Reference.op_Bang k () () in
       if
```

(continues on next page)

(continued from previous page)

```
         FStar_SizeT.lt len
           (FStar_SizeT.mul (Stdint.Uint64.of_int (2)) vk1)
       then FStar_Pervasives_Native.Some vcand
       else FStar_Pervasives_Native.None) in
let cand1 = _bind_c in let k1 = cand1 in let i1 = k1 in i1
```

# Part VIII

# Under the hood

In this part of the book, we'll look at some of the inner workings of F*, things that you will eventually need to know to become an expert user of the system. We'll cover F*'s SMT encoding, its two normalization engines, its plugin framework, and other topics.

CHAPTER

# FORTYSIX

# UNDERSTANDING HOW F* USES Z3

As we have seen throughout, F* relies heavily on the Z3 SMT (Satifiability Modulo Theories) solver for proof automation. Often, on standalone examples at the scale covered in earlier chapters, the automation just works out of the box, but as one builds larger developments, proof automation can start becoming slower or unpredictable—at that stage, it becomes important to understand how F*'s encoding to SMT works to better control proofs.

At the most abstract level, one should realize that finding proofs in the SMT logic F* uses (first-order logic with uninterpreted functions and arithmetic) is an undecidable problem. As such, F* and the SMT solver relies on various heuristics and partial decision procedures, and a solver like Z3 does a remarkable job of being able to effectively solve the very large problems that F* presents to it, despite the theoretical undecidability. That said, the proof search that Z3 uses is computationally expensive and can be quite sensitive to the choice of heuristics and syntactic details of problem instances. As such, if one doesn't chose the heuristics well, a small change in a query presented to Z3 can cause it to take a different search path, perhaps causing a proof to not be found at all, or to be found after consuming a very different amount of resources.

Some background and resources:

- F*'s SMT encoding uses the SMT-LIB v2 language. We refer to the "SMT-LIB v2" language as SMT2.
- Alejandro Aguirre wrote a tech report describing work in progress towards formalizing F*'s SMT encoding.
- Michal Moskal's Programming with Triggers describes how to pick triggers for quantifier instantiation and how to debug and profile the SMT solver, in the context of Vcc and the relation Hypervisor Verification project.
- Leonardo de Moura and Nikolaj Bjorner describe how E-matching is implemented in Z3 (at least circa 2007).

## 46.1 A Primer on SMT2

SMT2 is a standardized input language supported by many SMT solvers. Its syntax is based on S-expressions, inspired by languages in the LISP family. We review some basic elements of its syntax here, particularly the parts that are used by F*'s SMT encoding.

- Multi-sorted logic

  The logic provided by the SMT solver is multi-sorted: the sorts provide a simple type system for the logic, ensuring, e.g., that terms from two different sorts can never be equal. A user can define a new sort T, as shown below:

  ```
  (declare-sort T)
  ```

  Every sort comes with a built-in notion of equality. Given two terms p and q of the same sort T, (= p q) is a term of sort Bool expressing their equality.

- Declaring uninterpreted functions

A new function symbol `F`, with arguments in sorts `sort_1 .. sort_n` and returning a result in `sort` is declared as shown below,

```
(declare-fun F (sort_1 ... sort_n) sort)
```

The function symbol `F` is *uninterpreted*, meaning that the only information the solver has about `F` is that it is a function, i.e., when applied to equal arguments `F` produces equal results.

- Theory symbols

  Z3 provides support for several *theories*, notably integer and real arithmetic. For example, on terms `i` and `j` of `Int` sort, the sort of unbounded integers, the following terms define the expected arithmetic functions:

  ```
  (+ i j)        ; addition
  (- i j)        ; subtraction
  (* i j)        ; multiplication
  (div i j)      ; Euclidean division
  (mod i j)      ; Euclidean modulus
  ```

- Logical connectives

  SMT2 provides basic logical connectives as shown below, where `p` and `q` are terms of sort `Bool`

  ```
  (and p q)                 ; conjunction
  (or p q)                  ; disjunction
  (not p)                   ; negation
  (implies p q)             ; implication
  (iff p q)                 ; bi-implication
  ```

  SMT2 also provides support for quantifiers, where the terms below represent a term `p` with the variables `x1 ... xn` universally and existentially quantified, respectively.

  ```
  (forall ((x1 sort_1) ... (xn sort_n)) p)
  (exists ((x1 sort_1) ... (xn sort_n)) p)
  ```

- Attribute annotations

  A term `p` can be decorated with attributes names `a_1 .. a_n` with values `v_1 .. v_n` using the following syntax—the `!` is NOT to be confused with logical negation.

  ```
  (! p
     :a_1 v_1
     ...
     :a_n v_n)
  ```

  A common usage is with quantifiers, as we'll see below, e.g.,

  ```
  (forall ((x Int))
          (! (implies (>= x 0) (f x))
             :qid some_identifier))
  ```

- An SMT2 theory and check-sat

  An SMT2 theory is a collection of sort and function symbol declarations, and assertions of facts about them. For example, here's a simple theory declaring a function symbol `f` and an assumption that `f x y` is equivalent to `(>= x y)`—note, unlike in F*, the `assert` keyword in SMT2 assumes that a fact is true, rather than checking that it is valid, i.e., `assert` in SMT2 is like `assume` in F*.

```
(declare-fun f (Int Int) Bool)

(assert (forall ((x Int) (y Int))
                (iff (>= y x) (f x y))))
```

In the context of this theory, one can ask whether some facts about `f` are valid. For example, to check if `f` is a transitive function, one asserts the *negation* of the transitivity property for `f` and then asks Z3 to check (using the `(check-sat)` directive) if the resulting theory is satisfiable.

```
(assert (not (forall ((x Int) (y Int) (z Int))
                     (implies (and (f x y) (f y z))
                              (f x z)))))
(check-sat)
```

In this case, Z3 very quickly responds with `unsat`, meaning that there are no models for the theory that contain an interpretation of `f` compatible with both assertions, or, equivalently, the transitivity of `f` is true in all models. That is, we expect successful queries to return `unsat`.

## 46.2 A Brief Tour of F*'s SMT Encoding

Consider the following simple F* code:

```
let id x = x
let f (x:int) =
   if x < 0
   then assert (- (id x) >= 0)
   else assert (id x >= 0)
```

To encode the proof obligation of this program to SMT, F* generates an SMT2 file with the following rough shape.

```
;; Some basic scaffoling

(declare-sort Term)
...

;; Encoding of some basic modules

(declare-fun Prims.bool () Term)
...

;; Encoding of background facts about the current module

(declare-fun id (Term) Term)
(assert (forall ((x Term)) (= (id x) x)))

;; Encoding the query, i.e., negated proof obligation

(assert (not (forall ((x Term))
                     (and (implies (lt x 0) (geq (minus (M.id x)) 0))
                          (implies (not (lt x 0)) (geq (M.id x) 0))))))

(check-sat)
```

(continues on next page)

(continued from previous page)

```
;; Followed by some instrumentation for error reporting
;; in case the check-sat call fails (i.e., does not return unsat)
```

That was just just to give you a rough idea—the details of F*'s actual SMT encoding are a bit different, as we'll see below.

To inspect F*'s SMT encoding, we'll work through several small examples and get F* to log the SMT2 theories that it generates. For this, we'll use the file shown below as a skeleton, starting with the `#push-options "--log_queries"` directive, which instructs F* to print out its encoding to `.smt2` file. The `force_a_query` definition at the end ensures that F* actually produces at least one query—without it, F* sends nothing to the Z3 and so prints no output in the .smt2 file. If you run `fstar.exe SMTEncoding.fst` on the command line, you will find a file `queries-SMTEncoding.smt2` in the current directory.

```
module SMTEncoding
open FStar.Mul
#push-options "--log_queries"
let false_boolean = false
let true_boolean = true

let rec factorial (n:nat) : nat =
  if n = 0 then 1
  else n * factorial (n - 1)

let id (x:Type0) = x
let force_a_query = assert (id True)
```

Even for a tiny module like this, you'll see that the .smt2 file is very large. That's because, by default, F* always includes the modules `prims.fst`, `FStar.Pervasives.Native.fst`, and `FStar.Pervasives.fsti` as dependences of other modules. Encoding these modules consumes about 150,000 lines of SMT2 definitions and comments.

The encoding of each module is delimited in the .smt2 file by comments of the following kind:

```
;;; Start module Prims
...
;;; End module Prims (1334 decls; total size 431263)

;;; Start module FStar.Pervasives.Native
...
;;; End module FStar.Pervasives.Native (2643 decls; total size 2546449)

;;; Start interface FStar.Pervasives
...
;;; End interface FStar.Pervasives (2421 decls; total size 1123058)
```

where each *End* line also describes the number of declarations in the module and its length in characters.

### 46.2.1 Term sort

Despite SMT2 being a multi-sorted logic, aside from the pervasive use the SMT sort Bool, F*'s encoding to SMT (mostly) uses just a single sort: every pure (or ghost) F* term is encoded to the SMT solver as an instance of an uninterpreted SMT sort called Term. This allows the encoding to be very general, handling F*'s much richer type system, rather than trying to map F*'s complex type system into the much simpler type system of SMT sorts.

### 46.2.2 Booleans

One of the most primitive sorts in the SMT solver is Bool, the sort of Booleans. All the logical connectives in SMT are operations on the Bool sort. To encode values of the F* type bool to SMT, we use the Bool sort, but since all F* terms are encoded to the Term sort, we "box" the Bool sort to promote it to Term, using the SMT2 definitions below.

```
(declare-fun BoxBool (Bool) Term)
(declare-fun BoxBool_proj_0 (Term) Bool)
(assert (! (forall ((@u0 Bool))
                   (! (= (BoxBool_proj_0 (BoxBool @u0))
                          @u0)
                       :pattern ((BoxBool @u0))
                       :qid projection_inverse_BoxBool_proj_0))
            :named projection_inverse_BoxBool_proj_0))
```

This declares two uninterpreted functions BoxBool and BoxBool_proj_0 that go back and forth between the sorts Bool and Term.

The axiom named projection_inverse_BoxBool_proj_0 states that BoxBool_proj_0 is the inverse of BoxBool, or, equivalently, that BoxBool is an injective function from Bool to Term.

The qid is the quantifier identifier, usually equal to or derived from the name of the assumption that includes it—qids come up when we look at *profiling quantifier instantiation*.

### 46.2.3 Patterns for quantifier instantiation

The projection_inverse_BoxBool_proj_0 axiom on booleans shows our first use of a quantified formula with a pattern, i.e., the part that says :pattern ((BoxBool @u0)). These patterns are the main heuristic used to control the SMT solver's proof search and will feature repeatedly in the remainder of this chapter.

When exploring a theory, the SMT solver has a current partial model which contains an assignment for some of the variables in a theory to ground terms. All the terms that appear in this partial model are called *active* terms and these active terms play a role in quantifier instantiation.

Each universally quantified formula in scope is a term of the form below:

```
(forall ((x1 s1) ... (xn sn))
        (! (  body  )
         :pattern ((p1) ... (pm))))
```

This quantified formula is inert and only plays a role in the solver's search once the bound variables x1 ... xn are instantiated. The terms p1 ... pm are called patterns, and collectively, p1 ... pm must mention *all* the bound variables. To instantiate the quantifier, the solver aims to find active terms v1...vm that match the patterns p1..pm, where a match involves finding a substitution S for the bound variables x1...xm, such that the substituted patterns S(p1...pm) are equal to the active terms v1..vm. Given such a substitution, the substituted term S(body) becomes active and may refine the partial model further.

Existentially quantified formulas are dual to universally quantified formulas. Whereas a universal formula in the *context* (i.e., in negative position, or as a hypothesis) is inert until its pattern is instantiated, an existential *goal* (or, in positive position) is inert until its pattern is instantiated. Existential quantifiers can be decorated with patterns that trigger instantiation when matched with active terms, just like universal quantifiers

Returning to `projection_inverse_BoxBool_proj_0`, what this means is that once the solver has an active term `BoxBool b`, it can instantiate the quantified formula to obtain `(= (BoxBool_proj_0 (BoxBool b)) b)`.

### 46.2.4 Integers

The encoding of the F* type `int` is similar to that of `bool`—the primitive SMT sort `Int` (of unbounded mathematical integers) are coerced to `Term` using the injective function `BoxInt`.

```
(declare-fun BoxInt (Int) Term)
(declare-fun BoxInt_proj_0 (Term) Int)
(assert (! (forall ((@u0 Int))
                   (! (= (BoxInt_proj_0 (BoxInt @u0))
                         @u0)
                      :pattern ((BoxInt @u0))
                      :qid projection_inverse_BoxInt_proj_0))
           :named projection_inverse_BoxInt_proj_0))
```

The primitive operations on integers are encoded by unboxing the arguments and boxing the result. For example, here's the encoding of `Prims.(+)`, the addition operator on integers.

```
(declare-fun Prims.op_Addition (Term Term) Term)
(assert (! (forall ((@x0 Term) (@x1 Term))
                   (! (= (Prims.op_Addition @x0
                                            @x1)
                         (BoxInt (+ (BoxInt_proj_0 @x0)
                                    (BoxInt_proj_0 @x1))))
                      :pattern ((Prims.op_Addition @x0
                                                   @x1))
                      :qid primitive_Prims.op_Addition))
         :named primitive_Prims.op_Addition))
```

This declares an uninterpreted function `Prims.op_Addition`, a binary function on `Term`, and an assumption relating it to the SMT primitive operator from the integer arithmetic theory `(+)`. The pattern allows the SMT solver to instantiate this quantifier for every active application of the `Prims.op_Addition`.

The additional boxing introduces some overhead, e.g., proving `x + y == y + x` in F* amounts to proving `Prims.op_Addition x y == Prims.op_Addition y x` in SMT2. This in turn involves instantiation quantifiers, then reasoning in the theory of linear arithmetic, and finally using the injectivity of the `BoxInt` function to conclude. However, this overhead is usually not perceptible, and the uniformity of encoding everything to a single `Term` sort simplifies many other things. Nevertheless, F* provides a few options to control the way integers and boxed and unboxed, described *ahead*.

### 46.2.5 Functions

Consider the F* function below:

```
let add3 (x y z:int) : int = x + y + z
```

Its encoding to SMT has several elements.

First, we have have a declaration of an uninterpreted ternary function on Term.

```
(declare-fun SMTEncoding.add3 (Term Term Term) Term)
```

The semantics of add3 is given using the assumption below, which because of the pattern on the quantifier, can be interpreted as a rewrite rule from left to right: every time the solver has SMTEncoding.add3 x y z as an active term, it can expand it to its definition.

```
(assert (! (forall ((@x0 Term) (@x1 Term) (@x2 Term))
                   (! (= (SMTEncoding.add3 @x0
                                           @x1
                                           @x2)
                         (Prims.op_Addition (Prims.op_Addition @x0
                                                               @x1)
                                            @x2))
                    :pattern ((SMTEncoding.add3 @x0
                                                @x1
                                                @x2))
                    :qid equation_SMTEncoding.add3))

         :named equation_SMTEncoding.add3))
```

In addition to its definition, F* encodes *the type of* add3 to the solver too, as seen by the assumption below. One of the key predicates of F*'s SMT encoding is HasType, which relates a term to its type. The assumption typing_SMTEncoding.add3 encodes the typing of the application based on the typing hypotheses on the arguments.

```
(assert (! (forall ((@x0 Term) (@x1 Term) (@x2 Term))
                   (! (implies (and (HasType @x0
                                             Prims.int)
                                    (HasType @x1
                                             Prims.int)
                                    (HasType @x2
                                             Prims.int))
                               (HasType (SMTEncoding.add3 @x0 @x1 @x2)
                                        Prims.int))
                      :pattern ((SMTEncoding.add3 @x0 @x1 @x2))
                 :qid typing_SMTEncoding.add3))
      :named typing_SMTEncoding.add3))
```

This is all we'd need to encode add3 if it was never used at higher order. However, F* treats functions values just like any other value and allows them to be passed as arguments to, or returned as results from, other functions. The SMT logic is, however, a first-order logic and functions like add3 are not first-class values. So, F* introduces another layer in the encoding to model higher-order functions, but we don't cover this here.

### 46.2.6 Recursive functions and fuel

Non-recursive functions are similar to macro definitions—F* just encodes encodes their semantics to the SMT solver as a rewrite rule. However, recursive functions, since they could be unfolded indefinitely, are not so simple. Let's look at the encoding of the `factorial` function shown below.

```
open FStar.Mul
let rec factorial (n:nat) : nat =
  if n = 0 then 1
  else n * factorial (n - 1)
```

First, we have, as before, an uninterpreted function symbol on `Term` and an assumption about its typing.

```
(declare-fun SMTEncoding.factorial (Term) Term)

(assert (! (forall ((@x0 Term))
                   (! (implies (HasType @x0 Prims.nat)
                               (HasType (SMTEncoding.factorial @x0) Prims.nat))
                    :pattern ((SMTEncoding.factorial @x0))
                    :qid typing_SMTEncoding.factorial))
         :named typing_SMTEncoding.factorial))
```

However, to define the semantics of `factorial` we introduce a second "fuel-instrumented" function symbol with an additional parameter of `Fuel` sort.

```
(declare-fun SMTEncoding.factorial.fuel_instrumented (Fuel Term) Term)
```

The `Fuel` sort is declared at the very beginning of F*'s SMT encoding and is a representation of unary integers, with two constructors `ZFuel` (for zero) and `SFuel f` (for successor).

The main idea is to encode the definition of `factorial` guarded by patterns that only allow unfolding the definition if the fuel argument of `factorial.fuel_instrumented` is not zero, as shown below. Further, the assumption defining the semantics of `factorial.fuel_instrumented` is guarded by a typing hypothesis on the argument `(HasType @x1 Prims.nat)`, since the recursive function in F* is only well-founded on `nat`, not on all terms. The `:weight` annotation is an SMT2 detail: setting it to zero ensures that the SMT solver can instantiate this quantifier as often as needed, so long as the the fuel instrumentation argument is non-zero. Notice that the equation peels off one application of `SFuel`, so that the quantifier cannot be repeatedly instantiated infinitely.

```
(assert (! (forall ((@u0 Fuel) (@x1 Term))
                   (! (implies (HasType @x1 Prims.nat)
                               (= (SMTEncoding.factorial.fuel_instrumented (SFuel @u0)␣
↪@x1)
                                  (let ((@lb2 (Prims.op_Equality Prims.int @x1 (BoxInt␣
↪0))))
                                       (ite (= @lb2 (BoxBool true))
                                            (BoxInt 1)
                                            (Prims.op_Multiply
                                                   @x1
                                                   (SMTEncoding.factorial.fuel_
↪instrumented
                                                        @u0
                                                        (Prims.op_Subtraction @x1␣
↪(BoxInt 1))))))))
                     :weight 0
```

(continues on next page)

(continued from previous page)

```
                     :pattern ((SMTEncoding.factorial.fuel_instrumented (SFuel @u0) @x1))
                     :qid equation_with_fuel_SMTEncoding.factorial.fuel_instrumented))
         :named equation_with_fuel_SMTEncoding.factorial.fuel_instrumented))
```

We also need an assumption that tells the SMT solver that the fuel argument, aside from controlling the number of unfoldings, is semantically irrelevant.

```
(assert (! (forall ((@u0 Fuel) (@x1 Term))
                   (! (= (SMTEncoding.factorial.fuel_instrumented (SFuel @u0) @x1)
                         (SMTEncoding.factorial.fuel_instrumented ZFuel @x1))
                    :pattern ((SMTEncoding.factorial.fuel_instrumented (SFuel @u0) @x1))
                    :qid @fuel_irrelevance_SMTEncoding.factorial.fuel_instrumented))
         :named @fuel_irrelevance_SMTEncoding.factorial.fuel_instrumented))
```

And, finally, we relate the original function to its fuel-instrumented counterpart.

```
(assert (! (forall ((@x0 Term))
                   (! (= (SMTEncoding.factorial @x0)
                         (SMTEncoding.factorial.fuel_instrumented MaxFuel @x0))
                    :pattern ((SMTEncoding.factorial @x0))
                    :qid @fuel_correspondence_SMTEncoding.factorial.fuel_instrumented))
         :named @fuel_correspondence_SMTEncoding.factorial.fuel_instrumented))
```

This definition uses the constant `MaxFuel`. The value of this constant is determined by the F* options `--initial_fuel n` and `--max_fuel m`. When F* issues a query to Z3, it tries the query repeatedly with different values of `MaxFuel` ranging between `n` and `m`. Additionally, the option `--fuel n` sets both the initial fuel and max fuel to `n`.

This single value of `MaxFuel` controls the number of unfoldings of *all* recursive functions in scope. Of course, the patterns are arranged so that if you have a query involving, say, `List.map`, quantified assumptions about an unrelated recursive function like `factorial` should never trigger. Neverthless, large values of `MaxFuel` greatly increase the search space for the SMT solver. If your proof requires a setting greater than `--fuel 2`, and if it takes the SMT solver a long time to find the proof, then you should think about whether things could be done differently.

However, with a low value of `fuel`, the SMT solver cannot reason about recursive functions beyond that bound. For instance, the following fails, since the solver can unroll the definition only once to conclude that `factorial 1 == 1 * factorial 0`, but being unable to unfold `factorial 0` further, the proof fails.

```
#push-options "--fuel 1"
let _ = assert (factorial 1 == 1) (* fails *)
```

As with regular functions, the rest of the encoding of recursive functions has to do with handling higher-order uses.

### 46.2.7 Inductive datatypes and ifuel

Inductive datatypes in F* allow defining unbounded structures and, just like with recursive functions, F* encodes them to SMT by instrumenting them with fuel, to prevent infinite unfoldings. Let's look at a very simple example, an F* type of unary natural numbers.

```
type unat =
  | Z : unat
  | S : (prec:unat) -> unat
```

Although Z3 offers support for a built-in theory of datatypes, F* does not use it (aside for Fuel), since F* datatypes are more complex. Instead, F* rolls its own datatype encoding using uninterpreted functions and the encoding of unat begins by declaring these functions.

```
(declare-fun SMTEncoding.unat () Term)
(declare-fun SMTEncoding.Z () Term)
(declare-fun SMTEncoding.S (Term) Term)
(declare-fun SMTEncoding.S_prec (Term) Term)
```

We have one function for the type unat; one for each constructor (Z and S); and one "projector" for each argument of each constructor (here, only S_prec, corresponding to the F* projector S?.prec).

The type unat has its typing assumption, where Tm_type is the SMT encoding of the F* type Type—note F* does not encode the universe levels to SMT.

```
(assert (! (HasType SMTEncoding.unat Tm_type)
         :named kinding_SMTEncoding.unat@tok))
```

The constructor S_prec is assumed to be an inverse of S. If there were more than one argument to the constructor, each projector would project out only the corresponding argument, encoding that the constructor is injective on each of its arguments.

```
(assert (! (forall ((@x0 Term))
                   (! (= (SMTEncoding.S_prec (SMTEncoding.S @x0)) @x0)
                    :pattern ((SMTEncoding.S @x0))
                    :qid projection_inverse_SMTEncoding.S_prec))
         :named projection_inverse_SMTEncoding.S_prec))
```

The encoding defines two macros is-SMTEncoding.Z and is-SMTEncoding.S that define when the head-constructor of a term is Z and S respectively. These two macros are used in the definition of the inversion assumption of datatypes, namely that given a term of type unat, one can conclude that its head constructor must be either Z or S. However, since the type unat is unbounded, we want to avoid applying this inversion indefinitely, so it uses a quantifier with a pattern that requires non-zero fuel to be triggered.

```
(assert (! (forall ((@u0 Fuel) (@x1 Term))
              (! (implies (HasTypeFuel (SFuel @u0) @x1 SMTEncoding.unat)
                          (or (is-SMTEncoding.Z @x1)
                              (is-SMTEncoding.S @x1)))
                 :pattern ((HasTypeFuel (SFuel @u0) @x1 SMTEncoding.unat))
                 :qid fuel_guarded_inversion_SMTEncoding.unat))
      :named fuel_guarded_inversion_SMTEncoding.unat))
```

Here, we see a use of HasTypeFuel, a fuel-instrumented version of the HasType we've seen earlier. In fact, (HasType x t) is just a macro for (HasTypeFuel MaxIFuel x t), where much like for recursive functions and fuel, the constant MaxIFuel is defined by the current value of the F* options --initial_ifuel, --max_ifuel, and --ifuel (where ifuel stands for "inversion fuel").

The key bit in ensuring that the inversion assumption above is not indefinitely applied is in the structure of the typing assumptions for the data constructors. These typing assumptions come in two forms, introduction and elimination.

The introduction form for the S constructor is shown below. This allows deriving that S x has type unat from the fact that x itself has type unat. The pattern on the quantifier makes this goal-directed: if (HasTypeFuel @u0 (SMTEncoding.S @x1) SMTEncoding.unat) is already an active term, then the quantifier fires to make (HasTypeFuel @u0 @x1 SMTEncoding.unat) an active term, peeling off one application of the S constructor. If we were to use (HasTypeFuel @u0 @x1 SMTEncoding.unat) as the pattern, this would lead to an infinite quantifier instantiation loop, since every each instantiation would lead a new, larger active term that could instantiate the quantifier again. Note, using the introduction form does not vary the fuel parameter, since the the number of applications of the constructor S decreases at each instantiation anyway.

```
(assert (! (forall ((@u0 Fuel) (@x1 Term))
              (! (implies (HasTypeFuel @u0 @x1 SMTEncoding.unat)
                          (HasTypeFuel @u0 (SMTEncoding.S @x1) SMTEncoding.unat))
                 :pattern ((HasTypeFuel @u0 (SMTEncoding.S @x1) SMTEncoding.unat))
                 :qid data_typing_intro_SMTEncoding.S@tok))
         :named data_typing_intro_SMTEncoding.S@tok))
```

The elimination form allows concluding that the sub-terms of a well-typed application of a constructor are well-typed too. This time note that the conclusion of the rule decreases the fuel parameter by one. If that were not the case, then we would get a quantifier matching loop between `data_elim_SMTEncoding.S` and `fuel_guarded_inversion_SMTEncoding.unat`, since each application of the latter would contribute an active term of the form `(HasTypeFuel (SFuel _) (S (S_prec x)) unat)`, allowing the former to be triggered again.

```
(assert (! (forall ((@u0 Fuel) (@x1 Term))
                   (! (implies (HasTypeFuel (SFuel @u0) (SMTEncoding.S @x1) SMTEncoding.
↪unat)
                               (HasTypeFuel @u0 @x1 SMTEncoding.unat))
                    :pattern ((HasTypeFuel (SFuel @u0) (SMTEncoding.S @x1) SMTEncoding.
↪unat))
                    :qid data_elim_SMTEncoding.S))
         :named data_elim_SMTEncoding.S))
```

A final important element in the encoding of datatypes has to do with the well-founded ordering used in termination proofs. The following states that if `S x1` is well-typed (with non-zero fuel) then `x1` precedes `S x1` in F*'s built-in sub-term ordering.

```
(assert (! (forall ((@u0 Fuel) (@x1 Term))
                   (! (implies (HasTypeFuel (SFuel @u0)
                                            (SMTEncoding.S @x1)
                                            SMTEncoding.unat)
                               (Valid (Prims.precedes Prims.lex_t Prims.lex_t
                                                      @x1 (SMTEncoding.S @x1))))
                    :pattern ((HasTypeFuel (SFuel @u0) (SMTEncoding.S @x1) SMTEncoding.
↪unat))
                    :qid subterm_ordering_SMTEncoding.S))
      :named subterm_ordering_SMTEncoding.S))
```

Once again, a lot of the rest of the datatype encoding has to do with handling higher order uses of the constructors.

As with recursive functions, the single value of `MaxIFuel` controls the number of inversions of all datatypes in scope. It's a good idea to try to use an `ifuel` setting that is as low as possible for your proofs, e.g., a value less than `2`, or even `0`, if possible. However, as with fuel, a value of ifuel that is too low will cause the solver to be unable to prove some facts. For example, without any `ifuel`, the solver cannot use the inversion assumption to prove that the head of `x` must be either S or Z, and F* reports the error "Patterns are incomplete".

```
#push-options "--ifuel 0"
let rec as_nat (x:unat) : nat =
   match x with (* fails exhaustiveness check *)
   | S x -> 1 + as_nat x (* fails termination check *)
   | Z -> 0
```

Sometimes it is useful to let the solver arbitrarily invert an inductive type. The `FStar.Pervasives.`

`allow_inversion` is a library function that enables this, as shown below. Within that scope, the ifuel guards on the `unat` type are no longer imposed and SMT can invert `unat` freely—F* accepts the code below.

```
#push-options "--ifuel 0"
let rec as_nat (x:unat) : nat =
   allow_inversion unat;
   match x with
   | S x -> 1 + as_nat x
   | Z -> 0
```

This can be useful sometimes, e.g., one could set the ifuel to 0 and allow inversion within a scope for only a few selected types, e.g., `option`. However, it is rarely a good idea to use `allow_inversion` on an unbounded type (e.g., `list` or even `unat`).

### 46.2.8 Logical Connectives

The *logical connectives* that F* offers are all derived forms. Given the encodings of datatypes and functions (and arrow types, which we haven't shown), the encodings of all these connectives just fall out naturally. However, all these connectives also have built-in support in the SMT solver as part of its propositional core and support for E-matching-based quantifier instantiation. So, rather than leave them as derived forms, a vital optimization in F*'s SMT encoding is to recognize these connectives and to encode them directly to the corresponding forms in SMT.

The term `p /\ q` in F* is encoded to `(and [[p]] [[q]])` where `[[p]]` and `[[q]]` are the *logical* encodings of `p` and `q` respectively. However, the SMT connective `and` is a binary function on the SMT sort `Bool`, whereas all we have been describing so far is that every F* term `p` is encoded to the SMT sort `Term`. To bridge the gap, the logical encoding of a term `p` interprets the `Term` sort into `Bool` by using a function `Valid p`, which deems a `p : Term` to be valid if it is inhabited, as per the definitions below.

```
(declare-fun Valid (Term) Bool)
(assert (forall ((e Term) (t Term))
             (! (implies (HasType e t) (Valid t))
                :pattern ((HasType e t) (Valid t))
                :qid __prelude_valid_intro)))
```

The connectives `p \/ q`, `p ==> q`, `p <==> q`, and `~p` are similar.

The quantified forms `forall` and `exists` are mapped to the corresponding quantifiers in SMT. For example,

```
let fact_positive = forall (x:nat). factorial x >= 1
```

is encoded to:

```
(forall ((@x1 Term))
        (implies (HasType @x1 Prims.nat)
                 (>= (BoxInt_proj_0 (SMTEncoding.factorial @x1))
                     (BoxInt_proj_0 (BoxInt 1)))))
```

Note, this quantifier does not have any explicitly annotated patterns. In this case, Z3's syntactic trigger selection heuristics pick a pattern: it is usually the smallest collection of sub-terms of the body of the quantifier that collectively mention all the bound variables. In this case, the choices for the pattern are `(SMTEncoding.factorial @x1)` and `(HasType @x1 Prims.nat)`: Z3 picks both of these as patterns, allowing the quantifier to be triggered if an active term matches either one of them.

For small developments, leaving the choice of pattern to Z3 is often fine, but as your project scales up, you probably want to be more careful about your choice of patterns. F* lets you write the pattern explicitly on a quantifier and translates it down to SMT, as shown below.

```
let fact_positive_2 = forall (x:nat).{:pattern (factorial x)} factorial x >= 1
```

This produces:

```
(forall ((@x1 Term))
        (! (implies (HasType @x1 Prims.nat)
                    (>= (BoxInt_proj_0 (SMTEncoding.factorial @x1))
                        (BoxInt_proj_0 (BoxInt 1))))
         :pattern ((SMTEncoding.factorial.fuel_instrumented ZFuel @x1))))
```

Note, since `factorial` is fuel instrumented, the pattern is translated to an application that requires no fuel, so that the property also applies to any partial unrolling of factorial also.

Existential formulas are similar. For example, one can write:

```
let fact_unbounded = forall (n:nat). exists (x:nat). factorial x >= n
```

And it gets translated to:

```
(forall ((@x1 Term))
        (implies (HasType @x1 Prims.nat)
                 (exists ((@x2 Term))
                         (and (HasType @x2 Prims.nat)
                              (>= (BoxInt_proj_0 (SMTEncoding.factorial @x2))
                                  (BoxInt_proj_0 @x1))))))
```

### 46.2.9 Options for Z3 and the SMT Encoding

F* provides two ways of passing options to Z3.

The option `--z3cliopt <string>` causes F* to pass the string as a command-line option when starting the Z3 process. A typical usage might be `--z3cliopt 'smt.random_seed=17'`.

In contrast, `--z3smtopt <string>` causes F* to send the string to Z3 as part of its SMT2 output and this option is also reflected in the .smt2 file that F* emits with `--log_queries`. As such, it can be more convenient to use this option if you want to debug or profile a run of Z3 on an .smt2 file generated by F*. A typical usage would be `--z3smtopt '(set-option :smt.random_seed 17)'`. Note, it is possible to abuse this option, e.g., one could use `--z3smtopt '(assert false)'` and all SMT queries would trivially pass. So, use it with care.

F*'s SMT encoding also offers a few options.

- `--smtencoding.l_arith_repr native`

This option requests F* to inline the definitions of the linear arithmetic operators (`+` and `-`). For example, with this option enabled, F* encodes the term `x + 1 + 2` as the SMT2 term below.

```
(BoxInt (+ (BoxInt_proj_0 (BoxInt (+ (BoxInt_proj_0 @x0)
                                     (BoxInt_proj_0 (BoxInt 1)))))
           (BoxInt_proj_0 (BoxInt 2))))
```

- `--smtencoding.elim_box true`

This option is often useful in combination with `smtencoding.l_arith_repr native`, enables an optimization to remove redundant adjacent box/unbox pairs. So, adding this option to the example above, the encoding of `x + 1 + 2` becomes:

```
(BoxInt (+ (+ (BoxInt_proj_0 @x0) 1) 2))
```

- `--smtencoding.nl_arith_repr [native|wrapped|boxwrap]`

This option controls the representation of non-linear arithmetic functions (`*, /, mod`) in the SMT encoding. The default is `boxwrap` which uses the encoding of `Prims.op_Multiply, Prims.op_Division, Prims.op_Modulus` analogous to `Prims.op_Addition`.

The `native` setting is similar to the `smtencoding.l_arith_repr native`. When used in conjuction with `smtencoding.elim_box true`, the F* term `x * 1 * 2` is encoded to:

```
(BoxInt (* (* (BoxInt_proj_0 @x0) 1) 2))
```

However, a third setting `wrapped` is also available with provides an intermediate level of wrapping. With this setting enabled, the encoding of `x * 1 * 2` becomes

```
(BoxInt (_mul (_mul (BoxInt_proj_0 @x0) 1) 2))
```

where `_mul` is declared as shown below:

```
(declare-fun _mul (Int Int) Int)
(assert (forall ((x Int) (y Int)) (! (= (_mul x y) (* x y)) :pattern ((_mul x y)))))
```

Now, you may wonder why all these settings are useful. Surely, one would think, `--smtencoding.l_arith_repr native --smtencoding.nl_arith_repr native --smtencoding.elim_box true` is the best setting. However, it turns out that the additional layers of wrapping and boxing actually enable some proofs to go through, and, empirically, no setting strictly dominates all the others.

However, the following is a good rule of thumb if you are starting a new project:

1. Consider using `--z3smtopt '(set-option :smt.arith.nl false)'`. This entirely disables support for non-linear arithmetic theory reasoning in the SMT solver, since this can be very expensive and unpredictable. Instead, if you need to reason about non-linear arithmetic, consider using the lemmas from `FStar.Math.Lemmas` to do the non-linear steps in your proof manually. This will be more painstaking, but will lead to more stable proofs.
2. For linear arithmetic, the setting `--smtencoding.l_arith_repr native --smtencoding.elim_box true` is a good one to consider, and may yield some proof performance boosts over the default setting.

## 46.3 Designing a Library with SMT Patterns

In this section, we look at the design of `FStar.Set`, a module in the standard library, examining, in particular, its use of SMT patterns on lemmas for proof automation. The style used here is representative of the style used in many proof-oriented libraries—the interface of the module offers an abstract type, with some constructors and some destructors, and lemmas that relate their behavior.

To start with, for our interface, we set the fuel and ifuel both to zero—we will not need to reason about recursive functions or invert inductive types here.

```
module SimplifiedFStarSet
(** Computational sets (on eqtypes): membership is a boolean function *)
#set-options "--fuel 0 --ifuel 0"
```

Next, we introduce the signature of the main abstract type of this module, `set`:

```
val set (a:eqtype)
  : Type0
```

Sets offer just a single operation called mem that allows testing whether or not a given element is in the set.

```
val mem (#a:eqtype) (x:a) (s:set a)
  : bool
```

However, there are several ways to construct sets:

```
val empty (#a:eqtype)
  : set a

val singleton (#a:eqtype) (x:a)
  : set a

val union (#a:eqtype) (s0 s1: set a)
  : set a

val intersect (#a:eqtype) (s0 s1: set a)
  : set a

val complement (#a:eqtype) (s0:set a)
  : set a
```

Finally, sets are equipped with a custom equivalence relation:

```
val equal (#a:eqtype) (s0 s1:set a)
  : prop
```

The rest of our module offers lemmas that describe the behavior of mem when applied to each of the constructors.

```
val mem_empty (#a:eqtype) (x:a)
  : Lemma
    (ensures (not (mem x empty)))
    [SMTPat (mem x empty)]

val mem_singleton (#a:eqtype) (x y:a)
  : Lemma
    (ensures (mem y (singleton x) == (x=y)))
    [SMTPat (mem y (singleton x))]

val mem_union (#a:eqtype) (x:a) (s1 s2:set a)
  : Lemma
    (ensures (mem x (union s1 s2) == (mem x s1 || mem x s2)))
    [SMTPat (mem x (union s1 s2))]

val mem_intersect (#a:eqtype) (x:a) (s1:set a) (s2:set a)
  : Lemma
    (ensures (mem x (intersect s1 s2) == (mem x s1 && mem x s2)))
    [SMTPat (mem x (intersect s1 s2))]

val mem_complement (#a:eqtype) (x:a) (s:set a)
```

(continues on next page)

(continued from previous page)

```
  : Lemma
    (ensures (mem x (complement s) == not (mem x s)))
    [SMTPat (mem x (complement s))]
```

Each of these lemmas should be intuitive and familiar. The extra bit to pay attention to is the SMTPat annotations on each of the lemmas. These annotations instruct F*'s SMT encoding to treat the lemma like a universal quantifier guarded by the user-provided pattern. For instance, the lemma mem_empty is encoded to the SMT solver as shown below.

```
(assert (! (forall ((@x0 Term) (@x1 Term))
                   (! (implies (and (HasType @x0 Prims.eqtype)
                                    (HasType @x1 @x0))
                               (not (BoxBool_proj_0
                                       (SimplifiedFStarSet.mem @x0
                                                               @x1
                                                              (SimplifiedFStarSet.empty␣
↪@x0)))))
                    :pattern ((SimplifiedFStarSet.mem @x0
                                                       @x1
                                                       (SimplifiedFStarSet.empty @x0)))
                    :qid lemma_SimplifiedFStarSet.mem_empty))
        :named lemma_SimplifiedFStarSet.mem_empty))
```

That is, from the perspective of the SMT encoding, the statement of the lemma mem_empty is analogous to the following assumption:

```
forall (a:eqtype) (x:a). {:pattern (mem x empty)} not (mem x empty)
```

As such, lemmas decorated with SMT patterns allow the user to inject new, quantified hypotheses into the solver's context, where each of those hypotheses is justified by a proof in F* of the corresponding lemma. This allows users of the FStar.Set library to treat set almost like a new built-in type, with proof automation to reason about its operations. However, making this work well requires some careful design of the patterns.

Consider mem_union: the pattern chosen above allows the solver to decompose an active term mem x (union s1 s2) into mem x s1 and mem x s2, where both terms are smaller than the term we started with. Suppose instead that we had written:

```
val mem_union (#a:eqtype) (x:a) (s1 s2:set a)
  : Lemma
    (ensures (mem x (union s1 s2) == (mem x s1 || mem x s2)))
    [SMTPat (mem x s1); SMTPat (mem x s2)]
```

This translates to an SMT quantifier whose patterns are the pair of terms mem x s1 and mem x s2. This choice of pattern would allow the solver to instantiate the quantifier with all pairs of active terms of the form mem x s, creating more active terms that are themselves matching candidates. To be explicit, with a single active term mem x s, the solver would derive mem x (union s s), mem x (union s (union s s)), and so on. This is called a matching loop and can be disastrous for solver performance. So, carefully chosing the patterns on quantifiers and lemmas with SMTPat annotations is important.

Finally, to complete our interface, we provide two lemmas to characterize equal, the equivalence relation on sets. The first says that sets that agree on the mem function are equal, and the second says that equal sets are provably equal (==), and the patterns allow the solver to convert reasoning about equality into membership and provable equality.

```
val equal_intro (#a:eqtype) (s1 s2: set a)
  : Lemma
    (requires  (forall x. mem x s1 = mem x s2))
    (ensures (equal s1 s2))
    [SMTPat (equal s1 s2)]

val equal_elim (#a:eqtype) (s1 s2:set a)
  : Lemma
    (requires (equal s1 s2))
    (ensures  (s1 == s2))
    [SMTPat (equal s1 s2)]
```

Of course, all these lemmas can be easily proven by F* under a suitable representation of the abstract type set, as shown in the module implementation below.

```
module SimplifiedFStarSet
(** Computational sets (on eqtypes): membership is a boolean function *)
#set-options "--fuel 0 --ifuel 0"
open FStar.FunctionalExtensionality
module F = FStar.FunctionalExtensionality

let set (a:eqtype) = F.restricted_t a (fun _ -> bool)

(* destructors *)

let mem #a x s = s x

(* constructors *)
let empty #a = F.on_dom a (fun x -> false)
let singleton #a x = F.on_dom a (fun y -> y = x)
let union #a s1 s2 = F.on_dom a (fun x -> s1 x || s2 x)
let intersect #a s1 s2 = F.on_dom a (fun x -> s1 x && s2 x)
let complement #a s = F.on_dom a (fun x -> not (s x))

(* equivalence relation *)
let equal (#a:eqtype) (s1:set a) (s2:set a) = F.feq s1 s2

(* Properties *)
let mem_empty      #a x       = ()
let mem_singleton  #a x y     = ()
let mem_union      #a x s1 s2 = ()
let mem_intersect  #a x s1 s2 = ()
let mem_complement #a x s     = ()

(* extensionality *)
let equal_intro #a s1 s2 = ()
let equal_elim  #a s1 s2 = ()
```

### 46.3.1 Exercise

Extend the set library with another constructor with the signature shown below:

```
val from_fun (#a:eqtype) (f: a -> bool) : Tot (set a)
```

and prove a lemma that shows that a an element `x` is in `from_fun f` if and only if `f x = true`, decorating the lemma with the appropriate SMT pattern.

This interface file and its implementation provides the definitions you need.

**Answer**

Look at FStar.Set.intension if you get stuck

---

## 46.4 Profiling Z3 and Solving Proof Performance Issues

At some point, you will write F* programs where proofs start to take much longer than you'd like: simple proofs fail to go through, or proofs that were once working start to fail as you make small changes to your program. Hopefully, you notice this early in your project and can try to figure out how to make it better before slogging through slow and unpredictable proofs. Contrary to the wisdom one often receives in software engineering where early optimization is discouraged, when developing proof-oriented libraries, it's wise to pay attention to proof performance issues as soon as they come up, otherwise you'll find that as you scale up further, proofs become so slow or brittle that your productivity decreases rapidly.

### 46.4.1 Query Statistics

Your first tool to start diagnosing solver performance is F*'s `--query_stats` option. We'll start with some very simple artificial examples.

With the options below, F* outputs the following statistics:

```
#push-options "--initial_fuel 0 --max_fuel 4 --ifuel 0 --query_stats"
let _ = assert (factorial 3 == 6)
```

```
(<input>(20,0-20,49))        Query-stats (SMTEncoding._test_query_stats, 1)  failed
   {reason-unknown=unknown because (incomplete quantifiers)} in 31 milliseconds
   with fuel 0 and ifuel 0 and rlimit 2723280
   statistics={mk-bool-var=7065 del-clause=242 num-checks=3 conflicts=5
               binary-propagations=42 arith-fixed-eqs=4 arith-pseudo-nonlinear=1
               propagations=10287 arith-assert-upper=21 arith-assert-lower=18
               decisions=11 datatype-occurs-check=2 rlimit-count=2084689
               arith-offset-eqs=2 quant-instantiations=208 mk-clause=3786
               minimized-lits=3 memory=21.41 arith-pivots=6 max-generation=5
               arith-conflicts=3 time=0.03 num-allocs=132027456 datatype-accessor-ax=3
               max-memory=21.68 final-checks=2 arith-eq-adapter=15 added-eqs=711}

(<input>(20,0-20,49))        Query-stats (SMTEncoding._test_query_stats, 1)  failed
   {reason-unknown=unknown because (incomplete quantifiers)} in 47 milliseconds
   with fuel 2 and ifuel 0 and rlimit 2723280
   statistics={mk-bool-var=7354 del-clause=350 arith-max-min=10 interface-eqs=3
```

(continues on next page)

(continued from previous page)

```
               num-checks=4 conflicts=8 binary-propagations=56 arith-fixed-eqs=17
               arith-pseudo-nonlinear=3 arith-bound-prop=2 propagations=13767
               arith-assert-upper=46 arith-assert-lower=40 decisions=25
               datatype-occurs-check=5 rlimit-count=2107946 arith-offset-eqs=6
               quant-instantiations=326 mk-clause=4005 minimized-lits=4
               memory=21.51 arith-pivots=20 max-generation=5 arith-add-rows=34
               arith-conflicts=4 time=0.05 num-allocs=143036410 datatype-accessor-ax=5
               max-memory=21.78 final-checks=6 arith-eq-adapter=31 added-eqs=1053}

(<input>(20,0-20,49))        Query-stats (SMTEncoding._test_query_stats, 1)  succeeded
    in 48 milliseconds with fuel 4 and ifuel 0 and rlimit 2723280
    statistics={arith-max-min=26 num-checks=5 binary-propagations=70 arith-fixed-eqs=47
                arith-assert-upper=78 arith-assert-lower=71 decisions=40
                rlimit-count=2130332 max-generation=5 arith-nonlinear-bounds=2
                time=0.05 max-memory=21.78 arith-eq-adapter=53 added-eqs=1517
                mk-bool-var=7805 del-clause=805 interface-eqs=3 conflicts=16
                arith-pseudo-nonlinear=6 arith-bound-prop=4 propagations=17271
                datatype-occurs-check=5 arith-offset-eqs=20 quant-instantiations=481
                mk-clause=4286 minimized-lits=38 memory=21.23 arith-pivots=65
                arith-add-rows=114 arith-conflicts=5 num-allocs=149004462
                datatype-accessor-ax=9 final-checks=7}
```

There's a lot of information here:

- We see three lines of output, each tagged with a source location and an internal query identifer (`(SMTEncoding._test_query_stats, 1)`, the first query for verifying `_test_query_stats`).
- The first two attempts at the query failed, with Z3 reporting the reason for failure as `unknown because (incomplete quantifiers)`. This is a common response from Z3 when it fails to prove a query—since first-order logic is undecidable, when Z3 fails to find a proof, it reports "unknown" rather than claiming that the theory is satisfiable. The third attempt succeeded.
- The attempts used `0`, `2`, and `4` units of fuel. Notice that our query was `factorial 3 == 6` and this clearly requires at least 4 units of fuel to succeed. In this case it didn't matter much, since the two failed attempts took only `47` and `48` milliseconds. But, you may sometimes find that there are many attempts of a proof with low fuel settings and finally success with a higher fuel number. In such cases, you may try to find ways to rewrite your proof so that you are not relying on so many unrollings (if possible), or if you decide that you really need that much fuel, then setting the `--fuel` option to that value can help avoid several slow failures and retries.
- The rest of the statistics report internal Z3 statistics.
  - The `rlimit` value is a logical resource limit that F* sets when calling Z3. Sometimes, as we will see shortly, a proof can be "cancelled" in case Z3 runs past this resource limit. You can increase the `rlimit` in this case, as we'll see below.
  - Of the remaning statistics, perhaps the main one of interest is `quant_instantiations`. This records a cumulative total of quantifiers instantiated by Z3 so far in the current session—here, each attempt seems to instantiate around 100–150 quantifiers. This is a very low number, since the query is so simple. You may be wondering why it is even as many as that, since 4 unfolding of factorial suffice, but remember that there are many other quantifiers involved in the encoding, e.g., those that prove that `BoxBool` is injective etc. A more typical query will see quantifier instantiations in the few thousands.

**Note:** Note, since the `quant-instantiations` metric is cumulative, it is often useful to precede a query with something like the following:

```
#push-options "--initial_fuel 0 --max_fuel 4 --ifuel 0 --query_stats"
#restart-solver
let _dummy = assert (factorial 0 == 1)

let _test_query_stats = assert (factorial 3 == 6)
```

The `#restart-solver` creates a fresh Z3 process and the `dummy` query "warms up" the process by feeding it a trivial query, which will run somewhat slow because of various initialization costs in the solver. Then, the query stats reported for the real test subject starts in this fresh session.

### 46.4.2 Working though a slow proof

Even a single poorly chosen quantified assumption in the prover's context can make an otherwise simple proof take very long. To illustrate, consider the following variation on our example above:

```
assume Factorial_unbounded: forall (x:nat). exists (y:nat). factorial y > x

#push-options "--fuel 4 --ifuel 0 --query_stats"
#restart-solver
let _test_query_stats = assert (factorial 3 == 6)
```

We've now introduced the assumption `Factorial_unbounded` into our context. Recall from the SMT encoding of quantified formulas, from the SMT solver's perspective, this looks like the following:

```
(assert (! (forall ((@x0 Term))
                   (! (implies (HasType @x0 Prims.nat)
                               (exists ((@x1 Term))
                                       (! (and (HasType @x1 Prims.nat)
                                          (> (BoxInt_proj_0 (SMTEncoding.factorial @x1))
                                             (BoxInt_proj_0 @x0)))
                                        :qid assumption_SMTEncoding.Factorial_unbounded.
↪1)))
                :qid assumption_SMTEncoding.Factorial_unbounded))
         :named assumption_SMTEncoding.Factorial_unbounded))
```

This quantifier has no explicit patterns, but Z3 picks the term `(HasType @x0 Prims.nat)` as the pattern for the `forall` quantifier. This means that it can instantiate the quantifier for active terms of type `nat`. But, a single instantiation of the quantifier, yields the existentially quantified formula. Existentials are immediately skolemized by Z3, i.e., the existentially bound variable is replaced by a fresh function symbol that depends on all the variables in scope. So, a fresh term `a @x0` corresponding `@x1` is introduced, and immediately, the conjunct `HasType (a @x0) Prims.nat` becomes an active term and can be used to instantiate the outer universal quantifier again. This "matching loop" sends the solver into a long, fruitless search and the simple proof about `factorial 3 == 6` which previously succeeded in a few milliseconds, now fails. Here's are the query stats:

```
(<input>(18,0-18,49))        Query-stats (SMTEncoding._test_query_stats, 1)  failed
  {reason-unknown=unknown because canceled} in 5647 milliseconds
  with fuel 4 and ifuel 0 and rlimit 2723280
  statistics={ ... quant-instantiations=57046 ... }
```

A few things to notice:

- The failure reason is "unknown because canceled". That means the solver reached its resource limit and halted the proof search. Usually, when you see "canceled" as the reason, you could try raising the rlimit, as we'll see shortly.

- The failure took 5.6 seconds.
- There were 57k quantifier instantiations, as compared to just the 100 or so we had earlier. We’ll soon seen how to pinpoint which quantifiers were instantiated too much.

#### Increasing the rlimit

We can first retry the proof by giving Z3 more resources—the directive below doubles the resource limit given to Z3.

```
#push-options "--z3rlimit_factor 2"
```

This time it took 14 seconds and failed. But if you try the same proof a second time, it succeeds. That’s not very satisfying.

#### Repeating Proofs with Quake

Although this is an artificial example, unstable proofs that work and then suddenly fail do happen. Z3 does guarantee that it is deterministic in a very strict sense, but even the smallest change to the input, e.g., a change in variable names, or even asking the same query twice in a succession in the same Z3 session, can result in different answers.

There is often a deeper root cause (in our case, it’s the `Factorial_unbounded` assumption, of course), but a first attempt at determining whether or not a proof is “flaky” is to use the F* option `--quake`.

```
#push-options "--quake 5/k"
let _test_query_stats = assert (factorial 3 == 6)
```

This tries the query 5 times and reports the number of successes and failures.

In this case, F* reports the following:

```
Quake: query (SMTEncoding._test_query_stats, 1) succeeded 4/5 times (best fuel=4, best␣
↪ifuel=0)
```

If you’re working to stabilize a proof, a good criterion is to see if you can get the proof to go through with the `--quake` option.

You can also try the proof by varying the Z3’s random seed and checking that it works with several choices of the seed.

```
#push-options "--z3smtopt '(set-option :smt.random_seed 1)'"
```

### 46.4.3 Profiling Quantifier Instantiation

We have a query that’s taking much longer than we’d like and from the query-stats we see that there are a lot of quantifier instances. Now, let’s see how to pin down which quantifier is to blame.

1. Get F* to log an .smt2 file, by adding the `--log_queries` option. It’s important to also add a `#restart-solver` before just before the definition that you’re interested in profiling.

   ```
   #push-options "--fuel 4 --ifuel 0 --query_stats --log_queries --z3rlimit_factor 2"
   #restart-solver
   let _test_query_stats = assert (factorial 3 == 6)
   ```

   F* reports the name of the file that it wrote as part of the query-stats. For example:

```
(<input>(18,0-18,49)@queries-SMTEncoding-7.smt2)      Query-stats ...
```

2. Now, from a terminal, you run Z3 on this generated .smt2 file, while passing it the following option and save the output in a file.

   ```
   z3 queries-SMTEncoding-7.smt2 smt.qi.profile=true > sample_qiprofile
   ```

3. The output contains several lines that begin with `[quantifier_instances]`, which is what we're interested in.

   ```
   grep quantifier_instances sample_qiprofile | sort -k 4 -n
   ```

   The last few lines of output look like this:

   ```
   [quantifier_instances] bool_inversion :    352 :  10 : 11
   [quantifier_instances] bool_typing :    720 :  10 : 11
   [quantifier_instances] constructor_distinct_BoxBool :    720 :  10 : 11
   [quantifier_instances] projection_inverse_BoxBool_proj_0 :   1772 :  10 : 11
   [quantifier_instances] primitive_Prims.op_Equality :   2873 :  10 : 11
   [quantifier_instances] int_typing :   3168 :  10 : 11
   [quantifier_instances] constructor_distinct_BoxInt :   3812 :  10 : 11
   [quantifier_instances] typing_SMTEncoding.factorial :   5490 :  10 : 11
   [quantifier_instances] int_inversion :   5506 :  11 : 12
   [quantifier_instances] @fuel_correspondence_SMTEncoding.factorial.fuel_instrumented␣
   ↪:   5746 :  10 : 11
   [quantifier_instances] Prims_pretyping_ae567c2fb75be05905677af440075565 :   5835 : ␣
   ↪11 : 12
   [quantifier_instances] projection_inverse_BoxInt_proj_0 :   6337 :  10 : 11
   [quantifier_instances] primitive_Prims.op_Multiply :   6394 :  10 : 11
   [quantifier_instances] primitive_Prims.op_Subtraction :   6394 :  10 : 11
   [quantifier_instances] token_correspondence_SMTEncoding.factorial.fuel_instrumented␣
   ↪:   7629 :  10 : 11
   [quantifier_instances] @fuel_irrelevance_SMTEncoding.factorial.fuel_instrumented : ␣
   ↪ 9249 :  10 : 11
   [quantifier_instances] equation_with_fuel_SMTEncoding.factorial.fuel_instrumented :␣
   ↪ 13185 :  10 : 10
   [quantifier_instances] refinement_interpretation_Tm_refine_
   ↪542f9d4f129664613f2483a6c88bc7c2 :  15346 :  10 : 11
   [quantifier_instances] assumption_SMTEncoding.Factorial_unbounded :  15890 :  10 :␣
   ↪11
   ```

   Each line mentions is of the form:

   ```
   qid : number of instances : max generation :  max cost
   ```

   where,

   - qid is the identifer of quantifier in the .smt2 file
   - the number of times it was instantiated, which is the number we're most interested in
   - the generation and cost are other internal measures, which Nikolaj Bjorner explains here

4. Interpreting the results

   Clearly, as expected, `assumption_SMTEncoding.Factorial_unbounded` is instantiated the most.

Next, if you search in the .smt2 file for ":qid refinement_interpretation_Tm_refine_542f9d4f129664613f2483a6c88bc7c2", you'll find the assumption that gives an interpretation to the `HasType x Prims.nat` predicate, where each instantiation of `Factorial_unbounded` yields another instance of this fact.

Notice that `equation_with_fuel_SMTEncoding.factorial.fuel_instrumented` is also instantiated a lot. This is because aside from the matching loop due to `HasType x Prims.nat`, each instantiation of `Factorial_unbounded` also yields an occurrence of `factorial` as a new active term, which the solver then unrolls up to four times.

We also see instantiations of quantifiers in `Prims` and other basic facts like `int_inversion`, `bool_typing` etc. Sometimes, you may even find that these quantifiers fire the most. However, these quantifiers are inherent to F*'s SMT encoding: there's not much you can do about it as a user. They are usually also not to blame for a slow proof—they fire a lot when other terms are instantiated too much. You should try to identify other quantifiers in your code or libraries that fire a lot and try to understand the root cause of that.

### Z3 Axiom Profiler

The Z3 Axiom Profiler can also be used to find more detailed information about quantifier instantiation, including which terms we used for instantiation, dependence among the quantifiers in the form of instantiation chains, etc.

However, there seem to be some issues with using it at the moment with Z3 logs generated from F*.

## 46.4.4 Splitting Queries

In the next two sections, we look at a small example that Alex Rozanov reported, shown below. It exhibits similar proof problems to our artificial example with factorial. Instead of just identifying the problematic quantifier, we look at how to remedy the performance problem by revising the proof to be less reliant on Z3 quantifier instantiation.

```
module Alex

let unbounded (f: nat -> int) = forall (m: nat). exists (n:nat). abs (f n) > m

assume
val f : (f:(nat -> int){unbounded f})

let g : (nat -> int) = fun x -> f (x+1)

#push-options "--query_stats --fuel 0 --ifuel 0 --split_queries always"
let find_above_for_g (m:nat) : Lemma(exists (i:nat). abs(g i) > m) =
  assert (unbounded f); // apply forall to m
  eliminate exists (n:nat). abs(f n) > m
  returns exists (i:nat). abs(g i) > m with _. begin
    let m1 = abs(f n) in
    assert (m1 > m); //prover hint
    if n>=1 then assert (abs(g (n-1)) > m)
    else begin
      assert (n<=0); //arithmetics hint
      eliminate exists (n1:nat). abs (f n1) > m1
      returns exists (i:nat). abs(g i) > m with _.
      begin
        assert (n1>0);
        assert (abs (g (n1-1)) > m);
        ()
```

(continues on next page)

(continued from previous page)

```
      end
    end
  end
```

The hypothesis is that `unbounded f` has exactly the same problem as the our unbounded hypothesis on factorial—the `forall/exists` quantifier contains a matching loop.

This proof of `find_above_for_g` succeeds, but it takes a while and F* reports:

```
(Warning 349) The verification condition succeeded after splitting
it to localize potential errors, although the original non-split
verification condition failed. If you want to rely on splitting
queries for verifying your program please use the '--split_queries
always' option rather than relying on it implicitly.
```

By default, F* collects all the proof obligations in a top-level F* definition and presents it to Z3 in a single query with several conjuncts. Usually, this allows Z3 to efficiently solve all the conjuncts together, e.g., the proof search for one conjunct may yield clauses useful to complete the search for other clauses. However, sometimes, the converse can be true: the proof search for separate conjuncts can interfere with each other negatively, leading to the entire proof to fail even when every conjunct may be provable if tried separately. Additionally, when F* calls Z3, it applies the current rlimit setting for every query. If a query contains N conjuncts, splitting the conjuncts into N separate conjuncts is effectively a rlimit multiplier, since each query can separately consume resources as much as the current rlimit.

If the single query with several conjunct fails without Z3 reporting any further information that F* can reconstruct into a localized error message, F* splits the query into its conjuncts and tries each of them in isolation, so as to isolate the failing conjunct it any. However, sometimes, when tried in this mode, the proof of all conjuncts can succeed.

One way to respond to Warning 349 is to follow what it says and enable `--split_queries always` explicitly, at least for the program fragment in question. This can sometimes stabilize a previously unstable proof. However, it may also end up deferring an underlying proof-performance problem. Besides, even putting stability aside, splitting queries into their conjuncts results in somewhat slower proofs.

### 46.4.5 Taking Control of Quantifier Instantiations with Opaque Definitions

Here is a revision of Alex's program that addresses the quantifier instantiation problem. There are a few elements to the solution.

```
[@@"opaque_to_smt"]
let unbounded (f: nat → int) = forall (m: nat). exists (n:nat). abs (f n) > m

let instantiate_unbounded (f:nat → int { unbounded f }) (m:nat)
  : Lemma (exists (n:nat). abs (f n) > m)
  = reveal_opaque (`%unbounded) (unbounded f)

assume
val f : (z:(nat → int){unbounded z})

let g : (nat -> int) = fun x -> f (x+1)

#push-options "--query_stats --fuel 0 --ifuel 0"
let find_above_for_g (m:nat) : Lemma(exists (i:nat). abs(g i) > m) =
  instantiate_unbounded f m;
  eliminate exists (n:nat). abs(f n) > m
```

(continues on next page)

(continued from previous page)

```
  returns exists (i:nat). abs(g i) > m with _. begin
    let m1 = abs(f n) in
    if n>=1 then assert (abs(g (n-1)) > m)
    else begin
      instantiate_unbounded f m1;
      eliminate exists (n1:nat). abs (f n1) > m1
      returns exists (i:nat). abs(g i) > m with _.
      begin
        assert (abs (g (n1-1)) > m)
      end
    end
  end
```

1. Marking definitions as opaque

   The attribute `[@@"opaque_to_smt"]` on the definition of `unbounded` instructs F* to not encode that definition to the SMT solver. So, the problematic alternating quantifier is no longer in the global scope.

2. Selectively revealing the definition within a scope

   Of course, we still want to reason about the unbounded predicate. So, we provide a lemma, `instantiate_unbounded`, that allows the caller to explicity instantiate the assumption that `f` is unbounded on some lower bound `m`.

   To prove the lemma, we use `FStar.Pervasives.reveal_opaque`: its first argument is the name of a symbol that should be revealed; its second argument is a term in which that definition should be revealed. It this case, it proves that `unbounded f` is equal to `forall m. exists n. abs (f n) > m`.

   With this fact available in the local scope, Z3 can prove the lemma. You want to use `reveal_opaque` carefully, since with having revealed it, Z3 has the problematic alternating quantifier in scope and could go into a matching loop. But, here, since the conclusion of the lemma is exactly the body of the quantifier, Z3 quickly completes the proof. If even this proves to be problematic, then you may have to resort to tactics.

3. Explicitly instantiate where needed

   Now, with our instantiation lemma in hand, we can precisly instantiate the unboundedness hypothesis on `f` as needed.

   In the proof, there are two instantiations, at `m` and `m1`.

   Note, we are still relying on some non-trivial quantifier instantiation by Z3. Notably, the two assertions are important to instantiate the existential quantifier in the `returns` clause. We'll look at that in more detail shortly.

   But, by making the problematic definition opaque and instantiating it explicitly, our performance problem is gone—here's what query-stats shows now.

   ```
   (<input>(18,2-31,5))    Query-stats (AlexOpaque.find_above_for_g, 1)
                succeeded in 46 milliseconds
   ```

This wiki page provides more information on selectively revealing opaque definitions.

### 46.4.6 Other Ways to Explicitly Trigger Quantifiers

For completeness, we look at some other ways in which quantifier instantiation works.

#### An Artificial Trigger

Instead of making the definition of unbounded opaque, we could protect the universal quantifier with a pattern using some symbol reserved for this purpose, as shown below.

```
let trigger (x:int) = True

let unbounded_alt (f: nat → int) = forall (m: nat). {:pattern (trigger m)} (exists (n:nat). abs (f n) > m)

assume
val ff : (z:(nat → int){unbounded_alt z})

let gg : (nat -> int) = fun x -> ff (x+1)

#push-options "--query_stats --fuel 0 --ifuel 0"
let find_above_for_gg (m:nat) : Lemma(exists (i:nat). abs(gg i) > m) =
  assert (unbounded_alt ff);
  assert (trigger m);
  eliminate exists (n:nat). abs(ff n) > m
  returns exists (i:nat). abs(gg i) > m with _. begin
    let m1 = abs(ff n) in
    if n>=1 then assert (abs(gg (n-1)) > m)
    else begin
      assert (trigger m1);
      eliminate exists (n1:nat). abs (ff n1) > m1
      returns exists (i:nat). abs(gg i) > m with _.
      begin
        assert (abs (gg (n1-1)) > m)
      end
    end
  end
```

1. We define a new function trigger x that is trivially true.
2. In unbounded_alt we decorate the universal quantifier with an explicit pattern, {:pattern (trigger x)}. The pattern is not semantically relevant—it's only there to control how the quantifier is instantiated
3. In find_above_for_gg, whenever we want to instantiate the quantifier with a particular lower bound k, we assert trigger k. That gives Z3 an active term that mentions trigger which it then uses to instantiate the quantifier with our choice of k.

This style is not particularly pleasant, because it involves polluting our definitions with semantically irrelevant triggers. The selectively revealing opaque definitions style is much preferred. However, artificial triggers can sometimes be useful.

### Existential quantifiers

We have an existential formula in the goal `exists (i:nat). abs(g i) > m` and Z3 will try to solve this by finding an active term to instantiate `i`. In this case, the patterns Z3 picks is `(g i)` as well the predicate `(HasType i Prims.nat)`, which the SMT encoding introduces. Note, F* does not currently allow the existential quantifier in a `returns` annoation to be decorated with a pattern—that will likely change in the future.

Since `g i` is one of the patterns, by asserting `abs (g (n - 1)) > m` in one branch, and `abs (g (n1 - 1)) > m` in the other, Z3 has the terms it needs to instantiate the quantifier with `n - 1` in one case, and `n1 - 1` in the other case.

In fact, any assertion that mentions the `g (n - 1)` and `g (n1 - 1)` will do, even trivial ones, as the example below shows.

```
let find_above_for_g1 (m:nat) : Lemma(exists (i:nat). abs(g i) > m) =
  instantiate_unbounded f m;
  eliminate exists (n:nat). abs(f n) > m
  returns exists (i:nat). abs(g i) > m with _. begin
    let m1 = abs(f n) in
    if n>=1 then assert (trigger (g (n-1)))
    else begin
      instantiate_unbounded f m1;
      eliminate exists (n1:nat). abs (f n1) > m1
      returns exists (i:nat). abs(g i) > m with _.
      begin
        assert (trigger (g (n1-1)))
      end
    end
  end
```

We assert `trigger (g (n - 1)))` and `trigger (g (n1 - 1))`, this gives Z3 active terms for `g (n - 1))` and `g (n1 - 1)`, which suffices for the instantiation. Note, asserting `trigger (n - 1)` is not enough, since that doesn't mention `g`.

However, recall that there's a second pattern that's also applicable `(HasType i Prims.nat)`–we can get Z3 to instantiate the quantifier if we can inject the predicate `(HasType (n - 1) nat)` into Z3's context. By using `trigger_nat`, as shown below, does the trick, since F* inserts a proof obligation to show that the argument `x` in `trigger_nat x` validates `(HasType x Prims.nat)`.

```
let trigger_nat (x:nat) = True
let find_above_for_g2 (m:nat) : Lemma(exists (i:nat). abs(g i) > m) =
  instantiate_unbounded f m;
  eliminate exists (n:nat). abs(f n) > m
  returns exists (i:nat). abs(g i) > m with _. begin
    let m1 = abs(f n) in
    if n>=1 then assert (trigger_nat (n-1))
    else begin
      instantiate_unbounded f m1;
      eliminate exists (n1:nat). abs (f n1) > m1
      returns exists (i:nat). abs(g i) > m with _.
      begin
        assert (trigger_nat (n1-1))
      end
    end
  end
```

Of course, rather than relying on implicitly chosen triggers for the existentials, one can be explicit about it and provide the instance directly, as shown below, where the `introduce exists ...` in each branch directly provides the witness rather than relying on Z3 to find it. This style is much preferred, if possible, than relying implicit via various implicitly chosen patterns and artificial triggers.

```
let find_above_for_g' (m:nat) : Lemma(exists (i:nat). abs(g i) > m) =
  instantiate_unbounded f m;
  eliminate exists (n:nat). abs(f n) > m
  returns _ // exists (i:nat). abs(g i) > m
  with _. begin
    let m1 = abs(f n) in
    if n>=1 then (
      introduce exists (i:nat). abs(g i) > m
      with (n - 1)
      and ()
    )
    else begin
      instantiate_unbounded f m1;
      eliminate exists (n1:nat). abs (f n1) > m1
      returns _ //exists (i:nat). _ abs(g i) > m
      with _.
        begin
          introduce exists (i:nat). abs (g i) > m
          with (n1 - 1)
          and ()
        end
    end
  end
```

Here is a link to the the full file with all the variations we have explored.

### 46.4.7 Overhead due to a Large Context

Consider the following program:

```
module T = FStar.Tactics
module B = LowStar.Buffer
module SA = Steel.Array
open FStar.Seq

#push-options "--query_stats"
let warmup1 (x:bool { x == true }) = assert x

let test1 (a:Type) (s0 s1 s2: seq a)
  : Lemma (Seq.append (Seq.append s0 s1) s2 `Seq.equal`
           Seq.append s0 (Seq.append s1 s2))
  = ()
```

The lemma `test1` is a simple property about `FStar.Seq`, but the lemma occurs in a module that also depends on a large number of other modules—in this case, about 177 modules from the F* standard library. All those modules are encoded to the SMT solver producing about 11MB of SMT2 definitions with nearly 20,000 assertions for the solver to process. This makes for a large search space for the solver to explore to find a proof, however, most of those assertions are quantified formulas guarded by patterns and they remain inert unless some active term triggers them. Nevertheless, all these definitions impose a noticeable overhead to the solver. If you turn `--query_stats` on (after a single warm-up query), it takes Z3 about 300 milliseconds (and about 3000 quantifier instantiations) to find a proof for `test1`.

You probably won't really notice the overhead of a proof that takes 300 milliseconds—the F* standard library doesn't have many quantifiers in scope with things like bad quantifier alternation that lead to matching loops. However, as your development starts to depend on an ever larger stack of modules, there's the danger that at some point, your proofs are impacted by some bad choice of quantifiers in some module that you have forgotten about. In that case, you may find that seemingly simple proofs take many seconds to go through. In this section, we'll look at a few things you can do to diagnose such problems.

### Filtering the context

The first thing we'll look at is an F* option to remove facts from the context.

```
#push-options "--using_facts_from 'Prims FStar.Seq'"
let warmup2 (x:bool { x == true }) = assert x

let test2 (a:Type) (s0 s1 s2: seq a)
  : Lemma (Seq.append (Seq.append s0 s1) s2 `Seq.equal`
           Seq.append s0 (Seq.append s1 s2))
  = ()
```

The `--using_facts_from` option retains only facts from modules that match the namespace-selector string provided. In this case, the selector shrinks the context from 11MB and 20,000 assertions to around 1MB and 2,000 assertions and the query stats reports that the proof now goes through in just 15 milliseconds—a sizeable speedup even though the absolute numbers are still small.

Of course, deciding which facts to filter from your context is not easy. For example, if you had only retained `FStar.Seq` and forgot to include `Prims`, the proof would have failed. So, the `--using_facts_from` option isn't often very useful.

### Unsat Core and Hints

When Z3 finds a proof, it can report which facts from the context were relevant to the proof. This collection of facts is called the unsat core, because Z3 has proven that the facts from the context and the negated goal are unsatisfiable. F* has an option to record and replay the unsat core for each query and F* refers to the recorded unsat cores as "hints".

Here's how to use hints:

1. Record hints

   ```
   fstar.exe --record_hints ContextPollution.fst
   ```

   This produces a file called `ContextPollution.fst.hints`

   The format of a hints file is internal and subject to change, but it is a textual format and you can roughly see what it contains. Here's a fragment from it:

   ```
   [
      "ContextPollution.test1",
      1,
      2,
      1,
      [
   ```

(continues on next page)

(continued from previous page)

```
    "@MaxIFuel_assumption", "@query", "equation_Prims.nat",
    "int_inversion", "int_typing", "lemma_FStar.Seq.Base.lemma_eq_intro",
    "lemma_FStar.Seq.Base.lemma_index_app1",
    "lemma_FStar.Seq.Base.lemma_index_app2",
    "lemma_FStar.Seq.Base.lemma_len_append",
    "primitive_Prims.op_Addition", "primitive_Prims.op_Subtraction",
    "projection_inverse_BoxInt_proj_0",
    "refinement_interpretation_Tm_refine_542f9d4f129664613f2483a6c88bc7c2",
    "refinement_interpretation_Tm_refine_ac201cf927190d39c033967b63cb957b",
    "refinement_interpretation_Tm_refine_d83f8da8ef6c1cb9f71d1465c1bb1c55",
    "typing_FStar.Seq.Base.append", "typing_FStar.Seq.Base.length"
  ],
  0,
  "3f144f59e410fbaa970cffb0e20df75d"
]
```

This is the hint entry for the query with whose id is `(ContextPollution.test1, 1)`

The next two fields are the fuel and ifuel used for the query, `2` and `1` in this case.

Then, we have the names of all the facts in the unsat core for this query: you can see that it was only about 20 facts that were needed, out of the 20,000 that were originally present.

The second to last field is not used—it is always 0.

And the last field is a hash of the query that was issued.

2. Replaying hints

   The following command requests F* to search for `ContextPollution.fst.hints` in the include path and when attempting to prove a query with a given id, it looks for a hint for that query in the hints file, uses the fuel and ifuel settings present in the hints, and prunes the context to include only the facts present in the unsat core.

   ```
   fstar.exe --use_hints ContextPollution.fst
   ```

   Using the hints usually improves verification times substantially, but in this case, we see that the our proof now goes through in about 130 milliseconds, not nearly as fast as the 15 milliseconds we saw earlier. That's because when using a hint, each query to Z3 spawns a new Z3 process initialized with just the facts in the unsat core, and that incurs some basic start-up time costs.

Many F* projects use hints as part of their build, including F*'s standard library. The .hints files are checked in to the repository and are periodically refreshed as proofs evolve. This helps improve the stability of proofs: it may take a while for a proof to go through, but once it does, you can record and replay the unsat core and subsequent attempts of the same proof (or even small variations of it) can go through quickly.

Other projects do not use hints: some people (perhaps rightfully) see hints as a way of masking underlying proof performance problems and prefer to make proofs work quickly and robustly without hints. If you can get your project to this state, without relying on hints, then so much the better for you!

### Differential Profiling with qprofdiff

If you have a proof that takes very long without hints but goes through quickly with hints, then the hints might help you diagnose why the original proof was taking so long. This wiki page describes how to compare two Z3 quantifier instantiation profiles with a tool that comes with Z3 called qprofdiff.

### Hints that fail to replay

Sometimes, Z3 will report an unsat core, but when F* uses it to try to replay a proof, Z3 will be unable to find a proof of unsat, and F* will fall back to trying the proof again in its original context. The failure to find a proof of unsat from a previously reported unsat core is not a Z3 unsoundness or bug—it's because although the report core is really logically unsat, finding a proof of unsat may have relied on quantifier instantiation hints from facts that are not otherwise semantically relevant. The following example illustrates.

```
module HintReplay

assume
val p (x:int) : prop

assume
val q (x:int) : prop

assume
val r (x:int) : prop

assume P_Q : forall (x:int). {:pattern q x} q x ==> p x
assume Q_R : forall (x:int). {:pattern p x} q x ==> r x

let test (x:int { q x }) = assert (r x)
```

Say you run the following:

```
fstar --record_hints HintReplay.fst
fstar --query_stats --use_hints HintReplay.fst
```

You will see the following output from the second run:

```
(HintReplay.fst(15,27-15,39))   Query-stats (HintReplay.test, 1)        failed
  {reason-unknown=unknown because (incomplete quantifiers)} (with hint)
  in 42 milliseconds ..

(HintReplay.fst(15,27-15,39))   Query-stats (HintReplay.test, 1)        succeeded
  in 740 milliseconds ...
```

The first attempt at the query failed when using the hint, and the second attempt at the query (without the hint) succeeded.

To see why, notice that to prove the assertion `r x` from the hypothesis `q x`, logically, the assumption `Q_R` suffices. Indeed, if you look in the hints file, you will see that it only mentions `HintReplay.Q_R` as part of the logical core. However, `Q_R` is guarded by a pattern `p x` and in the absence of the assumption `P_Q`, there is no way for the solver to derive an active term `p x` to instantiate `Q_R`—so, with just the unsat core, it fails to complete the proof.

Failures for hint replay usually point to some unusual quantifier triggering pattern in your proof. For instance, here we used `p x` as a pattern, even though `p x` doesn't appear anywhere in `Q_R`—that's not usually a good choice, though sometimes, e.g., when using *artificial triggers* it can come up.

This wiki page on hints provides more information about diagnosing hint-replay failures, particularly in the context of the Low* libraries.