Dijkstra Monads for Free

Danel Ahman[1,2] Cătălin Hriţcu[1,3] Kenji Maillard[1,3,4] Guido Martínez[3,5]
Gordon Plotkin[1,2] Jonathan Protzenko[1] Aseem Rastogi[6] Nikhil Swamy[1]

[1]Microsoft Research, USA [2]University of Edinburgh, UK [3]Inria Paris, France
[4]ENS Paris, France [5]UNR, Argentina [6]Microsoft Research, India

Abstract

Dijkstra monads enable a dependent type theory to be enhanced with support for specifying and verifying effectful code via weakest preconditions. Together with their closely related counterparts, *Hoare monads*, they provide the basis on which verification tools like F⋆, Hoare Type Theory (HTT), and Ynot are built.

We show that Dijkstra monads can be derived "for free" by applying a continuation-passing style (CPS) translation to the standard monadic definitions of the underlying computational effects. Automatically deriving Dijkstra monads in this way provides a correct-by-construction and efficient way of reasoning about user-defined effects in dependent type theories.

We demonstrate these ideas in EMF⋆, a new dependently typed calculus, validating it via both formal proof and a prototype implementation within F⋆. Besides equipping F⋆ with a more uniform and extensible effect system, EMF⋆ enables a novel mixture of intrinsic and extrinsic proofs within F⋆.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

Keywords verification; proof assistants; effectful programming; dependent types

1. Introduction

In Dijkstra's (1975) weakest precondition semantics, stateful computations transform postconditions, relating results and final states to preconditions on input states. One can express such semantics via a monad of predicate transformers, a so-called "Dijkstra monad" (Jacobs 2015; Swamy et al. 2013). For instance, in the case of state, the following monad arises:

```
WP_ST a = post a → pre      where post a = (a ∗ state) → Type
                                  pre = state → Type

return_WP_ST x post s0 = post (x, s0)
bind_WP_ST f g post s0 = f (λ (x, s1) → g x post s1) s0
```

The *weakest precondition (WP)* of a pure term `e` is computed to be `return_WP_ST e`, and the WP of the sequential composition `let x = e1 in e2` is computed to be `bind_WP_ST wp1 (λx. wp2)`, where `wp1` and `wp2` are the WPs of `e1` and `e2` respectively.

Building on previous work by Nanevski et al. (2008), Swamy et al. (2013, 2016) designed and implemented F⋆, a dependently typed programming language whose type system can express WPs for higher-order, effectful programs via Dijkstra monads.

While this technique of specifying and verifying programs has been relatively successful, there is still room for improvement. Notably, in the version of F⋆ described by Swamy et al. (2016) specifying Dijkstra monads in F⋆ is a tedious, manual process, requiring delicate meta-theoretic arguments to establish the soundness of a user-provided predicate-transformer semantics with respect to the semantics of effectful programs. These typically require proofs of various correctness and admissibility conditions, including the correspondence to the operational semantics and the monad laws. Furthermore, only a handful of primitively supported effects are provided by the previous version of F⋆, and extending it with user-defined effects is not possible.

Rather than being given manually, we show that these predicate transformers can be automatically derived by CPS'ing purely functional definitions of monadic effects (with answer type `Type`). For instance, rather than defining `WP_ST`, one can simply compute it by CPS'ing the familiar ST monad (i.e., `state → a ∗ state`), deriving

```
WP_ST a = ((a ∗ state) → Type) → state → Type        (unfolded)
```

We apply this technique of deriving Dijkstra monads to F⋆. Our goal is to make F⋆'s effect system easier to configure and extensible beyond its primitive effects. To do so, we proceed as follows:

A monadic metalanguage We introduce DM, a simply typed, pure, monadic metalanguage in which one can, in the spirit of Wadler (1992), define a variety of monadic effects, ranging from state and exceptions, to continuations.

A core dependent type theory with monadic reflection To formally study our improvements to F⋆, we define a new dependently typed core calculus, EMF⋆ (for Explicitly Monadic F⋆) that features an extensible effect system. EMF⋆ is loosely based on the Calculus of Constructions (Coquand and Huet 1988) with (among other features): (1) a predicative hierarchy of non-cumulative universes; (2) a weakest-precondition calculus for pure programs; (3) refinement types; and (4) a facility for representing user-defined effects using the monadic reflection and reification of Filinski (1994), adapted to the dependently typed setting. New effects can be introduced into the language by defining them in terms of the built-in pure constructs, related to each other via monad morphisms; each such effect obtains a suitable weakest precondition calculus derived from the underlying pure WPs. We prove the calculus strongly normalizing and the WP calculus sound for total correctness verification, for both pure and effectful programs.

A CPS translation We give a type-directed CPS translation from DM to EMF⋆. This can be used to extend EMF⋆ with a new effect. One starts by defining a monadic effect (say ST) in DM. Next, via the translation, one obtains the Dijkstra variant of that effect (WP_ST) as a monotone, conjunctive predicate EMF⋆ transformer monad. Finally, a second translation from DM produces expression-level terms representing monadic computations in EMF⋆. A logical relations proof shows that monadic computations are correctly specified by their predicate transformers. We give examples of these translations for monadic effects, such as state, exceptions, information-flow control, continuations, and some combinations thereof.

Intrinsic and extrinsic proofs in EMF* Effectful programs in EMF* can be proven correct using one or both of two different reasoning styles. First, using the WP calculus, programs can be proven intrinsically, by decorating their definitions with specifications that must be proven to be at least as strong as their WPs. We refer to this as the *intrinsic* style, already familiar to users of F*, and other tools like HTT (Nanevski et al. 2008), Dafny (Leino 2010), and Why3 (Filliâtre and Paskevich 2013).

Second, through monadic reification, EMF* allows terminating effectful programs to be revealed as their underlying pure implementations. Once reified, one can reason about them via the computational behavior of their definitions. As such, one may define effectful programs with relatively uninformative types, and prove properties about them as needed, via reification. This *extrinsic* style of proving is familiar to users of systems like Coq or Isabelle, where it is routinely employed to reason about pure functions; using monads this style extends smoothly to terminating effectful programs. As in Coq an Isabelle, this extrinsic style only works for terminating code; in this paper we do not consider divergent computations and only discuss divergence as future work (§7)

Primitive effects in a call-by-value semantics We see EMF* as a meta-language in which to analyze and describe the semantics of terms in an object language, EMF$^\star_{\mathsf{ST}}$, a call-by-value programming language with primitive state. In the spirit of Moggi (1989), we show that EMF* programs that treat their ST effect abstractly soundly model EMF$^\star_{\mathsf{ST}}$ reductions—technically, we prove a simulation between EMF$^\star_{\mathsf{ST}}$ and EMF*. As such, our work is a strict improvement on the prior support for primitive effects in F*: despite programming and proving programs in a pure setting, stateful programs can still be compiled to run efficiently in the primitively effectful EMF$^\star_{\mathsf{ST}}$, while programs with other user-defined effects (e.g., information-flow control) can, unlike before, be executed via their pure encodings.

A prototype implementation for F* We have adapted F* to benefit from the theory developed in this paper, using a subset of F* itself as an implementation of DM, and viewing EMF$^\star_{\mathsf{ST}}$ as a model of its existing extraction mechanism to OCaml. Programmers can now configure F*'s effect system using simple monadic definitions, use F* to prove these definitions correct, and then use our CPS transformation to derive the Dijkstra monads required to configure F*'s existing type-checker. To benefit from the new extrinsic proving capabilities, we also extended F* with two new typing rules, and changed its normalizer, to handle monadic reflection and reification.

Several examples show how our work allows F* to be easily extended beyond the primitive effects already supported, without compromising its efficient primitive effect compilation strategy; and how the new extrinsic proof style places reasoning about terminating effectful programs in F* on an equal footing with its support for reasoning about pure programs.

1.1 Summary of Contributions

The central contribution of our work is designing three closely related lambda calculi, studying their metatheory and the connections between them, and applying them to provide a formal and practical foundation for a user-extensible effect system for F*. Specifically,

(1) EMF*: A new dependent type theory with user-extensible, monadic effects; monadic reflection and reification; WPs; and refinement types. We prove that EMF* is strongly normalizing and that its WPs are sound for total correctness (§3).

(2) DM: A simply typed language to define the expression-level monads that we use to extend EMF* with effects. We define a CPS transformation of DM terms to derive Dijkstra monads from expression-level monads, as well as an elaboration of DM terms to EMF*. Moreover, elaborated terms are proven to be in relation with their WPs (§4). This is the first formal characterization of the relation between WPs and CPS at arbitrary order.

(3) EMF$^\star_{\mathsf{ST}}$: A call-by-value language with primitive state, whose reductions are simulated by well-typed EMF* terms (§5).

(4) An implementation of these ideas within F* (§3.5, §4.6) and several examples of free Dijkstra monads for user-defined effects (§2). We highlight, in particular, the new ability to reason extrinsically about effectful terms.

The auxiliary materials (`https://www.fstar-lang.org/papers/dm4free`) contain appendices with complete definitions and proofs for the formal results in §3, §4 (Appendix A below), and §5. The F* source code (`https://github.com/FStarLang/FStar`) now includes the extensions from §3.5 and §4.6 and the examples from §2.

2. Illustrative Examples

We illustrate our main ideas using several examples from F*, contrasting with the state of affairs in F* prior to our work. We start by presenting the core WP calculus for pure programs (§2.1), then show how state and exceptions can be added to it (§2.2, §2.3, §2.4 and §2.5). Thereafter, we present several additional examples, including modeling dynamically allocated references (§2.6), reasoning about primitive state (§2.7), information-flow control (§2.8), and continuations (§2.9)—sections §3 and §4 may be read mostly independently of these additional examples.

Notation: The syntax $\lambda(\mathsf{b}_1)\,...\,(\mathsf{b}_n) \rightarrow \mathsf{t}$ introduces a lambda abstraction, where b_i ranges over binding occurrences $\mathsf{x{:}t}$ declaring a variable x at type t. The type $\mathsf{b}_1 \rightarrow ... \rightarrow \mathsf{b}_n \rightarrow \mathsf{c}$ is the type of a curried function, where c is a computation type—we emphasize the lack of enclosing parentheses on the b_i. We write just the type in b when the name is irrelevant, and $\mathsf{t} \rightarrow \mathsf{t}'$ for $\mathsf{t} \rightarrow \mathsf{Tot}\ \mathsf{t}'$.

2.1 WPs for Pure Programs

Reasoning about purely functional programs is a relatively well-understood activity: the type theories underlying systems like Coq, Agda, and F* are already well-suited to the task. Consider proving that pure term $\mathsf{sqr} = \lambda(\mathsf{x{:}int}) \rightarrow \mathsf{x} * \mathsf{x}$ always returns a non-negative integer. A natural strategy is an *extrinsic* proof, which involves giving sqr a simple type such as $\mathsf{int} \rightarrow \mathsf{Tot\ int}$, the type of total functions on integers, and then proving a lemma $\forall \mathsf{x}.\ \mathsf{sqr\ x} \geq 0$. In the case of F*, the proof of the lemma involves, first, a little computation to turn the goal into $\forall \mathsf{x}.\ \mathsf{x}{*}\mathsf{x} \geq 0$, and then reasoning in the theory of integer arithmetic of the Z3 SMT solver (de Moura and Bjørner 2008) to discharge the proof.

An alternative *intrinsic* proof style in F* involves giving sqr type $\mathsf{x{:}int} \rightarrow \mathsf{Pure\ int}\ (\lambda \mathsf{post} \rightarrow \forall \mathsf{y}.\ \mathsf{y} \geq 0 \Longrightarrow \mathsf{post\ y})$, a dependent function type of the form $\mathsf{x{:}t} \rightarrow c$, where the formal parameter $\mathsf{x{:}t}$ is in scope in the *computation type* c to the right of the arrow. Computation types c are either $\mathsf{Tot\ t}$ (for some type t) or of the form $\mathsf{M\ t\ wp}$, where M is an effect label, t is the result type of the computation, and wp is a predicate transformer specifying the semantics of the computation. The computation type we give to sqr is of the form $\mathsf{Pure\ t\ wp}$, the type of t-returning pure computations described by the predicate transformer wp: $(\mathsf{t} \rightarrow \mathsf{Type}) \rightarrow \mathsf{Type}$, a function taking postconditions on the result (predicates of type $\mathsf{t} \rightarrow \mathsf{Type}$), to preconditions. These predicate transformers form a Dijkstra monad. In this case, the wp states that to prove any property post of $\mathsf{sqr\ x}$, it suffices to prove $\mathsf{post\ y}$, for all non-negative y—as such, it states our goal that $\mathsf{sqr\ x}$ is non-negative. To prove sqr can be given this type, F* infers a weakest precondition for sqr x, namely λpost $\rightarrow$ post (x $*$ x) and aims to prove that the predicate transformer we specified is at least as strong as the weakest one it inferred: $\forall$post. ($\forall$y. y$\geq 0 \Longrightarrow$ post y) $\Longrightarrow$ post (x$*$x), which is discharged automatically by Z3. For pure programs, this intrinsic proof style may seem like overkill and, indeed, it often is. But, as we will see, this mechanism for reasoning about pure terms via WPs is a basic capability that we can leverage for reasoning about terms with more complex, effectful semantics.

2.2 Adding WPs for State

Consider proving that incr _ = let x = get() in put (x + 1) produces an output state greater than its input state. Since this program has the state effect, a proof by extrinsic reasoning is not completely straightforward, because reducing an effectful computation within a logic may not be meaningful. Instead, tools like Ynot (Chlipala et al. 2009), HTT (Nanevski et al. 2008), and F* only support the intrinsic proof style. In the case of F*, this involves the use of a computation type ST$'$ t wp, where wp: WP_ST t and for our simple example we take WP_ST t = ((t $*$ int) $\rightarrow$ Type) $\rightarrow$ int $\rightarrow$ Type, i.e., the Dijkstra state monad from §1 with state=int.

Using the ST$'$ computation type in F*, one can specify for incr the type unit $\rightarrow$ST$'$ unit (λpost s0 $\rightarrow\forall$s1. s1 $>$ s0 $\Longrightarrow$ post ((), s1)). That is, to prove any postcondition post of incr, it suffices to prove post ((), s1) for every s1 greater than s0, the initial state—this is the statement of our goal. The proof in F* currently involves:

(1) As discussed already in §1, one must define WP_ST t, its return and bind combinators, proving that these specifications are sound with respect to the operational semantics of state.

(2) The primitive effectful actions, get and put are assumed to have the types below—again, these types must be proven sound with respect to the operational semantics of F*.

    get : unit $\rightarrow$ST$'$ int (λ post s0 $\rightarrow$ post (s0, s0))
    put : x:int $\rightarrow$ST$'$ unit (λ post _ $\rightarrow$ post ((), x))

(3) Following the rule for sequential composition sketched in §1, F* uses the specifications of get and put to compute bind_ST_WP wp_get (λx $\rightarrow$ wp_put (x + 1)) as the WP of incr, which reduces to λpost s0 $\rightarrow$ post ((), s0 + 1).

(4) The final step requires proving that the computed WP is at least as weak as the specified goal, which boils down to showing that s0 + 1 $>$ s0, which F* and Z3 handle automatically.

The first two steps above correspond to adding a new effect to F*. The cost of this is amortized by the much more frequent and relatively automatic steps 3 and 4. However, adding a new effect to F* is currently an expert activity, carried out mainly by the language designers themselves. This is in large part because the first two steps above are both tedious and highly technical: a dangerous mixture that can go wrong very easily.

Our primary goal is to simplify those first two steps, allowing effects to be added to F* more easily and with fewer meta-level arguments to trust. Besides, although F* supports customization of its effect system, it only allows programmers to specify refinements of a fixed set of existing effects inherited from ML, namely, state, exceptions, and divergence. For example, an F* programmer can refine the state effect into three sub-effects for reading, writing, and allocation; but, she cannot add a new effect like alternative combinations of state and exceptions, non-determinism, continuations, etc. We aim for a more flexible, trustworthy mechanism for extending F* beyond the primitive effects it currently supports. Furthermore, we wish to place reasoning about terminating effectful programs on an equal footing with pure ones, supporting mixtures of intrinsic and extrinsic proofs for both.

2.3 CPS'ing Monads to Dijkstra Monads

Instead of manually specifying WP_ST, we program a traditional ST monad and derive WP_ST using a CPS transform. In §4.1 we formally present DM, a simply typed language in which to define monadic effects. DM itself contains a single primitive identity monad τ, which (as will be explained shortly) is used to control the CPS transform. We have implemented DM as a subset of F*, and for the informal presentation here we use the concrete syntax of our implementation. What follows is an unsurprising definition of a state monad st a, the type of total functions from s to identity computations returning a pair (a $*$ s).

```
let st a = s → τ (a ∗ s)
let return (x:a) : st a = λs0 → x, s0
let bind (f:st a) (g:a → st b) : st b = λs0 → let x,s1 = f s0 in g x s1
let get () : st a = λs0 → s0, s0
let put (x:s) : st unit = λ_ → (), x
```

This being a subset of F*, we can use it to prove that this definition is indeed a monad: proofs of the three monad laws for st are discharged automatically by F* below (feq is extensional equality on functions, and assert p requests F* to prove p statically). Other identities relating combinations of get and put can be proven similarly.

```
let right_unit_st (f:st α) = assert (feq (bind f return) f)
let left_unit_st (x:α) (f:(α → st β)) = assert (feq (bind (return x) f) (f x))
let assoc_st (f:st α) (g:(α → st β)) (h:(β → st γ))
   = assert (feq (bind f (λ x → bind (g x) h)) (bind (bind f g) h))
```

We then follow a two-step recipe to add an effect like st to F*:

Step 1 To derive the Dijkstra monad variant of st, we apply a selective CPS transformation called the $\star$-translation (§4.2); first, on type st a; then, on the various monadic operations. CPS'ing only those arrows that have τ-computation co-domains, we obtain:

$$
\begin{array}{lcl}
(\mathsf{st\ a})^\star & = & \mathsf{s} \rightarrow ((\mathsf{a} * \mathsf{s}) \rightarrow \mathsf{Type}) \rightarrow \mathsf{Type} \\
\mathsf{return}^\star & = & \lambda \mathsf{x\ s0\ post} \rightarrow \mathsf{post\ (x, s0)} \\
\mathsf{bind}^\star & = & \lambda \mathsf{f\ g\ s0\ post} \rightarrow \mathsf{f\ s0}\ (\lambda (\mathsf{x,s1}) \rightarrow \mathsf{g\ x\ s1\ post}) \\
\mathsf{get}^\star & = & \lambda ()\ \mathsf{s0\ post} \rightarrow \mathsf{post\ (s0, s0)} \\
\mathsf{put}^\star & = & \lambda \mathsf{x}\ \_\ \mathsf{post} \rightarrow \mathsf{post\ ((), x)}
\end{array}
$$

Except for a reordering of arguments, the terms above are identical to the analogous definitions for WP_ST. We prove that the $\star$-translation preserves equality: so, having shown the monad laws for st a, we automatically obtain the monad laws for (st a)*. We also prove that every predicate transformer produced by the $\star$-translation is monotone (it maps weaker postconditions to weaker preconditions) and conjunctive (it distributes over conjunctions and universals, i.e., infinite conjunctions, on the postcondition).

Step 2 The $\star$-translation yields a predicate transformer semantics for a new monadic effect, however, we still need a way to extend F* with the computational behavior of the new effect. For this, we define a second translation, which elaborates the definitions of the new monad and its associated actions to Pure computations in F*. A first rough approximation of what we prove is that for a well-typed DM computation e : τ t, its elaboration $\underline{\mathsf{e}}$ has type Pure $\underline{\mathsf{t}}$ e* in EMF*.

The first-order cases are particularly simple: for example, $\underline{\mathsf{return}}$ = return has type x:a $\rightarrow$Pure a (return* x) in EMF*; and $\underline{\mathsf{get}}$ = get has type u:unit $\rightarrow$Pure s (get* u) in EMF*. For a higher-order example, we sketch the elaboration of bind below, writing $\underline{\mathsf{st}}$ t wp for s0:s $\rightarrow$Pure t (wp s0):

$$
\begin{array}{l}
\underline{\mathsf{bind}}\ :\ \mathsf{wpf{:}(st\ a)^\star} \rightarrow \mathsf{f{:}\underline{st}\ a\ wpf} \\
\quad \rightarrow \mathsf{wpg{:}(a} \rightarrow \mathsf{(st\ b)^\star)} \rightarrow \mathsf{g{:}(x{:}a} \rightarrow \mathsf{\underline{st}\ b\ wpg\,x)} \\
\quad \rightarrow \underline{\mathsf{st}}\ \mathsf{b}\ (\mathsf{bind}^\star\ \mathsf{wpf\ wpg}) \\
\quad = \lambda \mathsf{wpf\ f\ wpg\ g\ s0} \rightarrow \mathsf{let\ x,\ s1 = f\ s0\ in\ g\ x\ s1}
\end{array}
$$

Intuitively, a function in DM (like bind) that abstracts over computations (f and g) is elaborated to a function ($\underline{\mathsf{bind}}$) in EMF*

that abstracts both over those computations (f and g again, but at their elaborated types) as well as the WP specifications of those computations (wpf and wpg). The result type of bind shows that it returns a computation whose specification matches bind*, i.e., the result of the CPS'ing $\star$-translation.

In other words, the WPs computed by F* for monads implemented as Pure programs correspond exactly to what one gets by CPS'ing the monads. At first, this struck us as just a happy coincidence, although, of course, we now know that it must be so. We see our proof of this fact as providing a precise characterization of the close connection between and WPs and CPS transformations.

2.4 Reify and Reflect, for Abstraction and Proving

Unlike prior F* formalizations which included divergence, primitive exception and state effects, the only primitive monad in EMF* is for Pure computations. Except for divergence, we can encode other effects using their pure representations; we leave divergence for future work. Although the translations from DM yield pure definitions of monads in F*, programming directly against those pure implementations is undesirable, since this may break abstractions. For instance, consider an integer-state monad whose state is expected to monotonically increase: revealing its representation as a pure term makes it hard to enforce this invariant. We rely on Filinski's (1994) monadic reflection for controlling abstraction.

Continuing our example, introducing the state effect in F* produces a new computation type ST (a:Type) (wp: (st a)*) and two coercions:

```
reify : ST a wp → s0:s → Pure (a ∗ s) (wp s0)
reflect : (s0:s → Pure (a ∗ s) (wp s0)) → ST a wp
```

The reify coercion reveals the representation of an ST computation as a Pure function, while reflect encapsulates a Pure function as a stateful computation. As we will see in subsequent sections (§2.5 and §2.6), in some cases to preserve abstractions, one or both of these coercions will need to be removed, or restricted in various ways (§2.7).

To introduce the actions from DM as effectful actions in F*, we reflect the pure terms produced by the elaboration from DM to EMF*, obtaining actions for the newly introduced computation type. For example, after reflection the actions get and put appear within F* at the types below:

```
get : unit → ST s (get⋆ ())
put : s1:s → ST unit (put⋆ s1)
```

As in §2.2, we can still program stateful functions and prove them intrinsically, by providing detailed specifications to augment their definitions—of course, the first two steps of the process there are now automatic. However, we now have a means of doing extrinsic proofs by reifying stateful programs, as shown below (taking s=int).

```
let StNull a = ST a (λ s0 post → ∀x. post x)
let incr _ : StNull unit = let n = get() in put (n + 1)
let incr_increases (s0:s) = assert (snd (reify (incr()) s0) = s0 + 1)
```

The StNull unit annotation on the second line above gives a weak specification for incr. However, later, when a particular property of incr is required, we can recover it by reasoning extrinsically about the reification of incr() as a pure term.

2.5 Combining Monads: State and Exceptions, in Two Ways

To add more effects to F*, one can simply repeat the methodology outlined above. For instance, one can use DM to define exn a = unit → τ(option a) in the obvious way (the unit is necessary, cf. §4.1), our automated two-step recipe extends F* with an effect for terminating programs that may raise exceptions. Of course, we would like to combine the effects to equip stateful programs with exceptions and, here, we come to a familiar fork in the road.

State and exceptions can be combined in two mutually incompatible ways. In DM, we can define both stexn a = s → τ((option a) ∗ s) and exnst a = s → τ(option (a ∗ s)). The former is more familiar to most programmers: raising an exception preserves the state; the latter discards the state when an exception is raised, which though less common, is also useful. We focus first on exnst and then discuss a variant of stexn.

***Relating* st *and* exnst** Translating st (as before) and exnst to F* gives us two unrelated effects ST and ExnST. To promote ST computations to ExnST, we define a lift relating st to exnst, their pure representations in DM, and prove that it is a monad morphism.

```
let lift (f:st a) : exnst a = λs0 → Some (f s0)
let lift_is_an_st_exnst_morphism =
  assert (∀ x. feq (lift (ST.return x)) (ExnST.return x));
  assert (∀ f g. feq (lift (ST.bind f g)) (ExtST.bind (lift f) (λ x → lift (g x))))
```

Applying our two-step translation to lift, we obtain in F* a computation-type coercion from ST a wp to ExnST a (lift* wp). Through this coercion, and through F*'s existing inference algorithm (Swamy et al. 2011, 2016), ST computations are implicitly promoted to ExnST computations whenever needed. In particular, the ST actions, get and put, are implicitly available with ExnST. All that remains is to define an additional action, raise = λ() s0 → None, which gets elaborated and reflected to F* at the type unit → ExnST a (λ _ p → p None).

ExnST programs in F* can be verified intrinsically and extrinsically. For an intrinsic proof, we show div_intrinsic below, which raises an exception on a divide-by-zero. To prove it, we make use of an abbreviation ExnSt a pre post, which lets us write specifications using pre- and postconditions instead of predicate transformers.

```
let ExnSt a pre post =
  ExnST a (λ s0 p → pre s0 ∧ ∀x. post s0 x =⇒ p x)
let div_intrinsic i j : ExnSt int
  (requires (λ _ → True))
  (ensures (λ s0 x → match x with
                     | None → j=0
                     | Some (z, s1) → s0 = s1 ∧ j <> 0 ∧ z = i / j))
  = if j=0 then raise () else i / j
```

Alternatively, for an extrinsic proof, we give a weak specification for div_extrinsic and verify it by reasoning about its reified definition separately. This time, we add a call to incr in the ST effect in case of a division-by-zero. F*'s type inference lifts incr to ExnST as required by the context. However, as the proof shows, the incr has no effect, since the raise that follows it discards the state.

```
let ExnStNull a = ExnST a (λ s0 post → ∀x. post x)
let div_extrinsic i j : ExnStNull int = if j=0 then (incr(); raise ()) else i / j
let lemma_div_extrinsic i j =
  assert (match reify (div_extrinsic i j) 0 with
          | None → j = 0
          | Some (z, 0) → j <> 0 ∧ z = i / j)
```

Using reify and reflect we can also build exception handlers, following ideas of Filinski (Filinski 1999). For example, in try_div below, we use a handler and (under-)specify that it never raises an exception.

```
let try_div i j : ExnSt int
    (requires (λ _ → True))
    (ensures (λ _ x → Option.isSome x))
  = reflect (λ s0 → match reify (div_intrinsic i j) s0 with
                    | None → Some (0, s0)
                    | x → x)
```

More systematically, we can first program a Benton and Kennedy (2001) exception handler in DM, namely, as a term of type

```
exnst a → (unit → exnst b) → (a → exnst b) → exnst b
```

and then translate it to F⋆, thereby obtaining a weakest precondition rule for it for free. More generally, adapting the algebraic effect handlers of Plotkin and Pretnar (2009) to user-defined monads m, handlers can be programmed in DM as terms of type

$$\mathsf{m\ a \to (m\ b \to b) \to (a \to b) \to b}$$

and then imported to F⋆. We leave a more thorough investigation of such effect handlers for Dijkstra monads to the future.

***An exception-counting state monad:* stexnC** For another combination of state and exceptions, we define stexnC, which in addition to combining state and exceptions (in the familiar way), also introduces an additional integer output that counts the number of exceptions that are raised. In DM, we write:

```
let stexnC a = s → τ(option a ∗ (s ∗ int))
let return (x:a) = λs → Some x, (s, 0)
let bind (m:stexnC a) (f:a → stexnC b) = λs0 → let r0 = m s0 in
    match r0 with
    | None, (s1, c1) → None, (s1, c1)
    | Some r, (s1, c1) → let res, (s, c2) = f r s1
                         in res (s, c1 + c2)
let raise () : stexnC a = λs → None, (s, 1)
let lift (f:st a) : stexnC a = λs → let x, s1 = f s in Some x, (s1, 0)
```

Notice that raise returns an exception count of 1. This count is added up in bind, à la writer monad. Adding stexnC to F⋆ proceeds as before. But, we need to be a bit careful with how we use reflection. In particular, an implicit invariant of stexnC is that the exception count field in the result is non-negative and actually counts the number of raised exceptions. If a programmer is allowed to reflect any s → Pure (option a ∗ (s ∗ int)) wp into an stexnC computation, then this invariant can be broken. Programmers can rely on F⋆'s module system to simply forbid the use of stexnC.reflect in client modules. Depending on the situation, the module providing the effect may still reveal a restricted version of the reflect operator to a client, e.g., we may only provide reflect_nonneg to clients, which only supports reflecting computations whose exception count is not negative. Of course, this only guarantees that the counter over-approximates the number of exceptions raised, which may or may not be acceptable.

```
let reflect_nonneg (f: s → Pure (option a ∗ (s ∗ int)) wp)
    : stexnC a (λ s0 post →
      wp s0 (λ (r, (s1, n)) → post (r, (s1, n)) ∧ n ≥ 0))
    = reflect f
```

The standard combination of state and exceptions (i.e., stexn) was already provided primitively in F⋆. The other two combinations shown here were not previously supported, since F⋆ only allowed OCaml effects. In the following more advanced subsections, we present a heap model featuring dynamic allocation (§2.6), the reconciliation of primitive state and extrinsic reasoning via reify and reflect (§2.7), and encodings of two other user-defined effects (a dynamic information-flow control monitor in §2.8 and continuations in §2.9).

2.6 State with References and Dynamic Allocation

The state monads that we have seen so far provide global state, with get and put as the only actions. Using just these actions, we can encode references and dynamic allocation by choosing a suitable representation for the global state. There are many choices for this representation with various tradeoffs, but a (simplified) model of memory that we use in F⋆ is the type heap shown below:[1]

```
type pre_heap = {
  next_addr: nat;
  mem : nat → Tot (option (a:Type & a))
}
type heap = h:pre_heap{∀ (n:nat). n ≥ h.next_addr
                                  ⟹ h.mem n==None}
```

A pre_heap is a pair of next_addr, the next free memory location and a memory mem mapping locations to possibly allocated values. ("a:Type & a" is a dependent pair type of some type a:Type and a value at that type). A heap is a pre_heap with an invariant (stated as a refinement type) that nothing is allocated beyond next_addr.

By taking s=heap in the ST monad of the previous section, we can program derived actions for allocation, reading, writing and deallocating references—we show just alloc below; deallocation is similar, while reading and writing require their references to be allocated in the current state. First, however, we define an abbreviation St a pre post, which lets us write specifications using pre- and postconditions instead of predicate transformers, which can be more convenient—the F⋆ keywords, requires and ensures are only there for readability and have no semantic content.

```
let St a pre post = ST a (λ h0 p →
    pre h0 ∧ (∗ pre: a predicate on the input state ∗)
    ∀x h1. post h0 x h1 (∗ post relates result, initial and final states ∗)
      ⟹ p (x, h1))

abstract let ref (a:Type) = nat (∗ other modules cannot treat ref as nat ∗)

let alloc (a:Type) (init:a) : St (ref a)
 (requires (λ h → True)) (∗ can allocate, assuming infinite mem. ∗)
 (ensures (λ h0 r h1 →
   h0.mem r == None ∧ (∗ the ref r is fresh ∗)
   h1.mem r == Some (| a, init |) ∧ (∗ initialized to init ∗)
   (∀ s. r≠s ⟹ h0.mem s == h1.mem s))) (∗ other refs not modified ∗)
   = let h0 = get () in (∗ get the current heap ∗)
     let r = h0.next_addr in (∗ allocate at next_addr ∗)
     let h1 = {
       next_addr=h0.next_addr + 1; (∗ bump and update mem ∗)
       mem = (λ r′ → if r = r′ then Some (| a , x |) else h0.mem r′)
     } in
     put h1; r (∗ put the new state and return the ref ∗)
```

Forbidding recursion through the store The reader may wonder if adding mutable references would allow stateful programs to diverge by recursing through the memory. This is forbidden due to universe constraints. The type Type in F⋆ includes an implicit level drawn from a predicative, countable hierarchy of universes. Written explicitly, the type heap lives in universe Type_{i+1} since it contains a map whose co-domain is in Type_i, for some universe level i. As such, while one can allocate references like ref nat or ref (nat → Tot nat), importantly, ref (a → ST b wp) is forbidden, since the universe of a → ST b wp is the universe of its representation a → h:heap → Pure (b ∗ heap) (wp h), which is Type_{i+1}. Thus, our heap model forbids storing stateful functions altogether. More fine-grained encodings are possible too, e.g., stratifying the heap into fragments and storing stateful functions that can only read from lower strata.

2.7 Relating heap to a Primitive Heap

While one can execute programs using the ST monad instantiated with heap as its state, in practice, for efficiency, the F⋆ compiler provides primitive support for state via its extraction facility to OCaml. In such a setting, one needs a leap of faith to believe that our model of the heap is faithful to the concrete implementation of the OCaml heap, e.g., the abstraction of ref a is important to ensure that F⋆ programs are parametric in the representation of OCaml references.

More germane to this paper, compiling ST programs primitively requires that they do not rely concretely on the representation of

[1] Although expressible in F⋆, types like heap are not expressible in the EMF⋆ calculus of §3 since it lacks support for features like inductive types and universe polymorphism.

ST a wp as h:heap → Pure (a ∗ heap) (wp h), since the OCaml heap cannot be reified to a value. In §5, we show that source programs that are free of reflect and reify can indeed be safely compiled using primitive state (Theorem 10). Without reflect and reify, one may rely on intrinsic proof to show generally useful properties of programs. For example, one may use intrinsic proofs to show that ST computations never use references after they are deallocated, since reading and writing require their references to be allocated in the current state. Note, we write r ∈ h for h.mem r == Some _, indicating that r is allocated in h.

```
let incr (r:ref int) : St unit (requires (λ h → r ∈ h))
                                (ensures (λ h0 s h1 → r ∈ h1))
    = r := !r + 1
```

Of course, one would still like to show that incr increments its reference. In the rest of this section, we show how we can safely restore reflect and reify in the presence of primitive state.

Restoring reify and reflect for extrinsic proofs F⋆ programs using primitive state are forbidden from using reify and reflect only in the *executable* part of a program—fragments of a program that are computationally irrelevant (aka "ghost" code) are erased by the F⋆ compiler and are free to use these operators. As such, within specifications and proofs, ST programs can be reasoned about extrinsically via reification and reflection (say, for functional correctness), while making use of intrinsically proven properties like memory safety.

To restrict their use, as described in §2.5, we rely on F⋆'s module system to hide both the reify and reflect operators from clients of a module FStar.State defining the ST effect. Instead, we expose to clients only ghost_reify, a function equivalent to reify, but at the signature shown below. Notice that the function's co-domain is marked with the Ghost effect, meaning that it can only be used within specifications (e.g., WPs and assertions)—any other use will be flagged as a typing error by F⋆.

```
ghost_reify: (x:a → ST b (wp x))
           → Ghost (x:a → s0:s → Pure (b ∗ s) (wp x s0))
```

The FStar.State module also provides refine_St, a total function that allows a client to strengthen the postcondition of an effectful function f to additionally record that the returned value of f on any argument and input state h0 corresponds to the computational behavior of the (ghostly) reification of f. This allows a client to relate f to its reification while remaining in a computationally relevant context.

```
let refine_St (f :(x:a → St b (pre x) (post x)))
  : Tot (x:a → St b (pre x) (λ h0 z h1 → post x h0 z h1 ∧
                                ghost_reify f x h0 == z, h1)
  = λx → STATE.reflect (reify (f x))
```

Reasoning using ghost_reify instead of reify, clients can still prove incr_increases as in §2.4, making use of incr's intrinsic specification to show that if the reference r is allocated before calling incr it will still be allocated afterwards.

```
let incr_increases (r:ref int) (h0:heap{r ∈ h0}) : Ghost unit =
    let Some x0 = h0.mem r in
    let _, h1 = ghost_reify incr r h0 in
    let Some x1 = h1.mem r in
    assert (x1 = x0 + 1)
```

Further, in computationally relevant (non-ghost) code, refine_St allows us to reason using the concrete definition of incr:

```
let r = ST.alloc 42 in
let n0 = !r in
refine_St incr r;
let n1 = !r in
assert(n1 == n0 + 1)
```

The intrinsic specification of incr does not constrain the final value of r, so calling incr directly here would not be enough for proving the final assertion. By tagging the call-site with refine_St, we strengthen the specification of incr extrinsically, allowing the proof to complete as in incr_increases.

2.8 Information Flow Control

Information-flow control (Sabelfeld and Myers 2006) is a paradigm in which a program is deemed secure when one can prove that its behavior observable to an adversary is independent of the secrets the program may manipulate, i.e., it is *non-interferent*. Monadic reification allows us to prove non-interference properties directly, by relating multiple runs of an effectful program (Benton 2004). For example, take the simple stateful program below:

```
let ifc h = if h then (incr(); let y = get() in decr(); y) else get() + 1
```

It is easy to prove this program non-interferent via the extrinsic, relational proof below, which states that regardless of its secret input (h0, h1), ifc when run in the same public initial state (s0) produces identical public outputs. This generic extrinsic proof style is in contrast to Barthe et al. (2014), whose rF⋆ is a custom extension to F⋆ supporting only intrinsic relational proofs.

```
let ni_ifc = assert (∀ h0 h1 s0. reify (ifc h0) s0 = reify (ifc h1) s0)
```

Aside from such relational proofs, with user-defined effects, it is also possible to define monadic, dynamic information-flow control monitors in DM, deferring non-interference checks to runtime, and to reason about monitored programs in F⋆. Here's a simplified example, inspired by the floating label approach of LIO (Stefan et al. 2011). For simplicity, we take the underlying monad to be exnst, where the state is a security label from a two-point lattice that represents the secrecy of data that a computation may have observed so far.

```
type label = Low | High
let difc a = label → τ(option (a ∗ label))
```

Once added to F⋆, we can provide two primitive actions to interface with the outside world, where DIFC is the effect corresponding to difc. Importantly, writing to a public channel using write Low when the current label is High causes a dynamic failure signaling a potential Leak of secret information.

```
let join l1 l2 = match l1, l2 with | _, High | High, _ → High | _ → Low
val read : l:label → DIFC bool (λ l0 p → ∀b. p (Some (b, join l0 l)))
let flows l1 l2 = match l1, l2 with | High, Low → false | _ → true
val write : l:label → bool → DIFC unit (λ l0 p →
  if flows l0 l then p (Some ((), l0)) else p None)
```

As before, it is important to not allow untrusted client code to reflect on DIFC, since that may allow it to declassify arbitrary secrets. Arguing that DIFC soundly enforces a form of termination-insensitive non-interference requires a meta-level argument, much like that of Stefan et al. (2011).

We can now write programs like the one below, and rely on the dynamic checks to ensure they are secure.

```
let b1, b2 = read Low, read Low in write Low (b1 && b2)
let b3 = read High in write High (b1 || b3); write Low (xor b3 b3)
```

In this case, we can also prove that the program fails with a None at the last write Low. In contrast to the relational proof sketched earlier, dynamic information-flow control is conservative: even though the last write reveals no information on the low channel, the monitor still raises an error.

2.9 CPS'ing the Continuation Monad

As a final example before our formal presentation, we ask the irresistible question of whether we can get a Dijkstra monad for free for the continuation monad itself—indeed, we can.

We start by defining the standard continuation monad, cont, in DM. Being a subset of F*, we can prove that it is indeed a monad, automatically.

```
let cont a = (a → τ ans) → τ ans
let return x = λk → k x
let bind f g k = f (λ x → g x k)
(∗ cont is a monad ∗)
let right_unit_cont (f:cont α) = assert (bind f return == f);
let left_unit_cont (x:α) (f:(α → cont β)) = assert (bind (return x) f == f x)
let assoc_cont (f:cont α) (g:(α → cont β)) (h:(β → cont γ)) =
    assert (bind f (λ x → bind (g x) h) == bind (bind f g) h)
```

Following our two-step recipe, we derive the Dijkstra variant of cont, but first we define some abbreviations to keep the notation manageable. The type kwp a is the type of a predicate transformer specifying a continuation a $\to \tau$ ans; and kans is the type of a predicate transformer of the computation that yields the final answer.

$$\begin{array}{lclcl} \mathsf{kwp\ a} & = & \mathsf{a} \to \mathsf{kans} & = & (\mathsf{a} \to \tau\,\mathsf{ans})^\star \\ \mathsf{kans} & = & (\mathsf{ans} \to \mathsf{Type}) \to \mathsf{Type} & = & (\tau\,\mathsf{ans})^\star \end{array}$$

Using these abbreviations, we show the $\star$-translation of cont, return and bind. Instead of being just a predicate transformer, $(\mathsf{cont\ a})^\star$ is a predicate-transformer transformer.

$$\begin{array}{lcl} (\mathsf{cont\ a})^\star & = & \mathsf{kwp\ a} \to \mathsf{kans} \\ \mathsf{return}^\star & = & \lambda(\mathsf{x{:}a})\ (\mathsf{wp\_k{:}kwp\ a}) \to \mathsf{wp\_k\ x} \\ \mathsf{bind}^\star & = & \lambda \mathsf{f\ g}\ (\mathsf{wp\_k{:}kwp\ b}) \to \mathsf{f}\ (\lambda(\mathsf{x{:}a}) \to \mathsf{g\ x\ wp\_k}) \end{array}$$

For step 2, we show the elaboration of return and bind to F*, using the abbreviation kt a wp for the type of the elaborated term $\underline{k}$, where the DM term k is a continuation of type a $\to \tau$ ans and wp=$k^\star$. As illustrated in §2.3, elaborating higher-order functions from DM to F* introduces additional arguments corresponding to the predicate transformers of abstracted computations.

$$\begin{array}{lcl}
\mathsf{kt\ a\ wp} & = & \mathsf{x{:}a} \to \mathsf{Pure\ ans\ (wp\ x)} \\
\underline{\mathsf{return}} & : & \mathsf{x{:}a} \to \mathsf{wpk{:}kwp\ a} \to \mathsf{k{:}kt\ a\ wpk} \to \mathsf{Pure\ ans}\ (\mathsf{return}^\star\ \mathsf{x\ wpk}) \\
 & = & \lambda \mathsf{x\ wpk\ k} \to \mathsf{k\ x} \\
\underline{\mathsf{bind}} & : & \mathsf{wpf{:}(cont\ a)}^\star \\
 & \to & \mathsf{f{:}(wpk{:}kwp\ a} \to \mathsf{k{:}kt\ a\ wpk} \to \mathsf{Pure\ ans\ (wpf\ wpk))} \\
 & \to & \mathsf{wpg{:}(a} \to \mathsf{(cont\ b)}^\star) \\
 & \to & \mathsf{g{:}(x{:}a} \to \mathsf{wpk{:}kwp\ b} \to \mathsf{k{:}kt\ b\ wpk} \to \mathsf{Pure\ ans\ (wpg\ x\ wpk))} \\
 & \to & \mathsf{wpk{:}kwp\ b} \\
 & \to & \mathsf{k{:}kt\ b\ wpk} \\
 & \to & \mathsf{Pure\ ans}\ (\mathsf{bind}^\star\ \mathsf{wpf\ wpg\ wpk}) \\
 & = & \lambda \mathsf{wpf\ f\ wpg\ g\ wpk\ k} \to \mathsf{f}\ (\lambda \mathsf{x} \to \mathsf{wpg\ x\ wpk})\ (\lambda \mathsf{x} \to \mathsf{g\ x\ wpk\ k})
\end{array}$$

In the case of $\underline{\mathsf{return}}$, we have one additional argument for the predicate transformer of the continuation k—the type of the result shows how $\underline{\mathsf{return}}$ relates to return*. The elaboration $\underline{\mathsf{bind}}$ involves many such additional parameters, but the main point to take away is that its specification is given in terms of bind*, which is applied to the predicate transformers wpf, wpg, wpk, while $\underline{\mathsf{bind}}$ was applied to the computations f, g, k. In both cases, the definitions of $\underline{\mathsf{return}}$ and $\underline{\mathsf{bind}}$ match their pre-images in DM aside from abstracting over and passing around the additional WP arguments.

To better see the monadic structure in the types of $\underline{\mathsf{return}}$ and $\underline{\mathsf{bind}}$ we repeat these types, but this time writing $\underline{\mathsf{cont}}$ a wp for the type wpk:kwp a $\to$ k:kt a wpk $\to$ Pure ans (wp wpk):

$$\begin{array}{lcl}
\underline{\mathsf{return}} & : & \mathsf{x{:}a} \to \underline{\mathsf{cont}}\ \mathsf{a}\ (\mathsf{return}^\star\ \mathsf{x}) \\
\underline{\mathsf{bind}} & : & \mathsf{wpf{:}(cont\ a)}^\star \to \mathsf{f{:}}\underline{\mathsf{cont}}\ \mathsf{a\ wpf} \\
 & \to & \mathsf{wpg{:}(a} \to \mathsf{(cont\ b)}^\star) \to \mathsf{g{:}(x{:}a} \to \underline{\mathsf{cont}}\ \mathsf{b\ (wpg\ x))} \\
 & \to & \underline{\mathsf{cont}}\ \mathsf{b}\ (\mathsf{bind}^\star\ \mathsf{wpf\ wpg})
\end{array}$$

3. Explicitly Monadic F*

We begin our formal development by presenting EMF*, an explicitly typed, monadic core calculus intended to serve as a model of F*. As seen above, the F* implementation includes an inference algorithm

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textit{Terms}} \\
e, t, wp, \phi & ::= & x \mid T \mid x{:}t\{\phi\} \mid \lambda x{:}t.e \mid x{:}t \to c \mid e_1\ e_2 \\
 & \mid & \mathsf{case}_t\,(e\ \mathsf{as}\ y)\ x.e_1\ x.e_2 \mid \mathsf{run}\ e \mid \mathsf{reify}\ e \\
 & \mid & \mathsf{reflect}\ e \mid M.\mathsf{lift}_{M'}\ t\ wp\ e \mid F.\mathsf{act}\ \bar{e} \\
 & \mid & M.\mathsf{return}\ t\ e \mid M.\mathsf{bind}\ t_1\ t_2\ wp_1\ e_1\ wp_2\ x.e_2 \\
\multicolumn{3}{l}{\textit{Computation types}} \\
c & ::= & \mathsf{Tot}\ t \mid M\ t\ wp \ \ \text{where } M \in \{\mathsf{Pure},\ F\}
\end{array}$$

Signatures of monadic effects and lifts

$$\begin{array}{lcl}
S & ::= & D \mid S, D \mid S, L \\
D & ::= & F \left\{ \begin{array}{lclclcl} repr & = & t & ; & wp_type & = & t \\ \underline{return} & = & e & ; & return^\star & = & wp \\ \underline{bind} & = & e & ; & bind^\star & = & wp \\ \underline{act}_j & = & e & ; & act^\star_j & = & \overline{x_j{:}t_j} \to c_j \end{array} \right\} \\
L & ::= & \{\ \underline{M.lift}_{M'} = e;\ M.lift^\star_{M'} = wp\ \}
\end{array}$$

Figure 1. Syntax of EMF*

(Swamy et al. 2016) so that source programs may omit all explicit uses of the monadic return, bind and lift operators. We do not revisit that inference algorithm here. Furthermore, EMF* lacks F*'s support for divergent and ghost computations, fixed points and their termination check, inductive types, and universe polymorphism. We leave extending EMF* to accommodate all these features as future work, together with a formal proof that after inference, F* terms can be elaborated into EMF* (along the lines of the elaboration of Swamy et al. (2011)).

3.1 Syntax

Figure 1 shows the EMF* syntax. We highlight several key features.

Expressions, types, WPs, and formulae are all represented uniformly as terms; however, to evoke their different uses, we often write e for expressions, t for types, wp for WPs, and ϕ for logical formulae. Terms include variables (x, y, a, b, w etc.); refinement types $x{:}t\{\phi\}$; λ abstractions; dependent products with computation-type co-domains, $x{:}t \to c$ (with the sugar described in §2); and applications. Constants T include Type$_i$, the ith level from a countable hierarchy of predicative universes.[2] We also include constants for non-dependent pairs and disjoint unions; the former are eliminated using fst and snd (also constants), while the latter are eliminated using $\mathsf{case}_t\,(e\ \mathsf{as}\ y)\ x.e_1\ x.e_2$, which is standard dependent pattern matching with an explicit return type t and a name for the scrutinee y, provided only when the dependency is necessary.

Computation types (c) include Tot t, the type of total t-returning terms, and $M\ t\ wp$, the type of a computation with effect M, return type t, and behavior specified by the predicate transformer wp. Let M range over the Pure effect as well as user-defined effects F.

Explicit monadic returns, binds, actions, lifts, reify, and reflect. M.return and M.bind are the monad operations for the effect M, with explicit arguments for the types and predicate transformers. $M.\mathsf{lift}_{M'}\ t\ wp\ e$ lifts the $e : M\ t\ wp$ to M'. A fully applied F action is written $F.\mathsf{act}\ \bar{e}$. The reify and reflect operators are for monadic reflection, and run coerces a Pure computation to Tot.

Signatures for user-defined effects EMF* is parameterized by a signature S. A user-defined effect $F\ t\ wp$ is specified using D, the result of translating a DM monad. A definition D is a record containing several fields: *repr* is the type of an F computation reified

[2] We have yet to model F*'s universe polymorphism, making the universes in EMF* less useful than the ones in F*. Lacking universe polymorphism, we restrict computation to have results in Type$_0$. A simple remediation would be replicate the monad definitions across the universe levels.

as a pure term, *wp_type* is the type of the *wp* argument to F; $\underline{return}$, $\underline{bind}$, and $\underline{act}_j$ are EMF* expressions, and $return^\star$, $bind^\star$, and $act^\star_j$ are EMF* WPs (act_j is the j^{th} action of F). We use $S.F.\underline{return}$ to denote the lookup of the $\underline{return}$ field from F's definition in the signature S, and similar notation for the other fields.

For example, for the ST monad from §2.3, we have[3]:

$$
\begin{array}{llcl}
ST\{ & wp\_type & = & \lambda a.s \to (a * s \to \mathsf{Type}_0) \to \mathsf{Type}_0 \\
& repr & = & \lambda a\ w.s_0{:}s \to \mathsf{Pure}\ (a*s)\ (w\ s_0) \\
& \underline{return} & = & \lambda a.\underline{\mathsf{return}} \\
& return^\star & = & \lambda a.\mathsf{return}^\star \\
& \underline{bind} & = & \lambda a\ b.\underline{\mathsf{bind}} \\
& bind^\star & = & \lambda a\ b.\mathsf{bind}^\star \\
& \underline{get} & = & \underline{\mathsf{get}} \\
& get^\star & = & \mathsf{get}^\star \\
& \underline{put} & = & \underline{\mathsf{put}} \\
& put^\star & = & \mathsf{put}^\star \qquad \}
\end{array}
$$

where (as described in §2.3) $\underline{\mathsf{return}} : a \to x{:}a \to repr\ a\ (return^\star\ a\ x)$; and similarly for $\underline{\mathsf{bind}}$, $\underline{\mathsf{get}}$, and $\underline{\mathsf{put}}$.

In addition to the monad definitions D, the signature S contains the definitions of lifts that contain an EMF* expression and an EMF* WP. We use notations $S.\underline{M.lift}_{M'}$ and $S.M.lift^\star_{M'}$ to look these up in S. Finally, the signature always includes a fixed partial definition for the Pure monad, only containing the following definitions:

$$
\begin{array}{lll}
Pure\,\{\, wp\_type & = & \lambda a{:}\mathsf{Type}_0.\ (a \to \mathsf{Type}_0) \to \mathsf{Type}_0 \\
\quad return^\star & = & \lambda a{:}\mathsf{Type}_0.\ \lambda x{:}a.\ \lambda p{:}(a \to \mathsf{Type}_0).\ p\ x \\
\quad bind^\star & = & \lambda a.\ \lambda b.\ \lambda w_1.\ \lambda w_2.\ \lambda p.\ w_1\ (\lambda x.\ (w_2\ x)\ p) \quad \}
\end{array}
$$

The other fields are not defined, since Pure is handled primitively in the EMF* dynamic semantics (§3.3).

The well-formedness conditions on the signature S (shown in the auxiliary material) check that the fields in definitions D and the lifts in L are well-typed as per their corresponding WPs. In addition, each effect definition can make use of the previously defined effects, enabling a form of layering. However, in this paper, we mainly focus on combining effects using the lift operations.

3.2 Static Semantics

The expression typing judgment in EMF* has the form $S;\Gamma \vdash e : c$, where Γ is the list of bindings $x : t$ as usual. Selected rules for the judgment are shown in Figure 2. In the rules, we sometimes write $S;\Gamma \vdash e : t$ as an abbreviation for $S;\Gamma \vdash e : \mathsf{Tot}\ t$.

Monadic returns, binds, lifts, and actions. Rules T-RETURN, T-BIND, and T-LIFT simply use the corresponding *wp* specification from the signature for M to compute the final *wp*. For example, in the case of the ST monad from §2.3, $S.ST.return^\star\ t = \lambda x{:}t.\lambda s_0{:}s.\lambda post.post\ (x, s_0)$. Rule T-ACT is similar; it looks up the type of the action from the signature, and then behaves like the standard function application rule.

Monadic reflection and reification. Rules T-REIFY and T-REFLECT are dual, coercing between a computation type and its underlying pure representation. Rule T-RUN coerces e from type $\mathsf{Pure}\ t\ wp$ to $\mathsf{Tot}\ t$. However, since the Tot type is unconditionally total, the second premise of the rule checks that the *wp* is satisfiable.

Refinements, computations types, and proof irrelevance. EMF*'s refinement and computation types include a form of proof irrelevance. In T-REFINE, the universe of $x{:}t\{\phi\}$ is determined by the universe of t alone, since a witness for the proposition ϕ is never materialized. Refinement formulas ϕ and *wp*s are manipulated using an entailment relation, $S;\Gamma \models \phi$, for a proof-irrelevant, classical logic where all the connectives are "squashed" (Nogin 2002), e.g., $\mathsf{p} \wedge \mathsf{q}$ and $\mathsf{p} \Longrightarrow \mathsf{q}$ from §2, are encoded as $\mathsf{x{:}unit\{p * q\}}$ and $\mathsf{x{:}unit\{p \to q\}}$, and reside in Type_0. Similar to T-REFINE, in C-PURE, the universe of a computation type is determined only by the result type. Since the *wp* is proof irrelevant, the use of Type_0 in the type of *wp* is quite natural, because its proof content is always squashed. For user-defined monads F, the rule C-F delegates to their underlying representation $S.F.repr$.

$$\textsc{T-Return}\ \frac{S;\Gamma \vdash e : \mathsf{Tot}\ t}{S;\Gamma \vdash M.\mathsf{return}\ t\ e : M\ t\ (S.M.return^\star\ t\ e)} \qquad \textsc{T-Refine}\ \frac{S;\Gamma \vdash t : Type_i \quad S;\Gamma, x{:}t \vdash \phi : Type_j}{S;\Gamma \vdash x{:}t\{\phi\} : Type_i}$$

$$\textsc{T-Bind}\ \frac{S;\Gamma \vdash t_2 : \mathsf{Type}_0 \quad S;\Gamma \vdash wp_2 : x{:}t_1 \to S.M.wp\_type\ t_2 \quad S;\Gamma \vdash e_1 : M\ t_1\ wp_1 \quad S;\Gamma, x{:}t_1 \vdash e_2 : M\ t_2\ (wp_2\ x)}{S;\Gamma \vdash M.\mathsf{bind}\ t_1\ t_2\ wp_1\ e_1\ wp_2\ x.e_2 : M\ t_2\ (S.M.bind^\star\ t_1\ t_2\ wp_1\ wp_2)}$$

$$\textsc{T-Lift}\ \frac{S;\Gamma \vdash e : M\ t\ wp}{S;\Gamma \vdash M.\mathsf{lift}_{M'}\ t\ wp\ e : M'\ t\ (S.M.lift^\star_{M'}\ wp)} \qquad \textsc{T-Act}\ \frac{S.F.act^\star = \overline{x{:}t} \to c \quad \forall i.\ S;\Gamma \vdash e_i : t_i}{S;\Gamma \vdash F.\mathsf{act}\ \bar{e} : c[\bar{e}/\bar{x}]}$$

$$\textsc{T-Reify}\ \frac{S;\Gamma \vdash e : F\ t\ wp}{S;\Gamma \vdash \mathsf{reify}\ e : \mathsf{Tot}\ (S.F.repr\ t\ wp)} \qquad \textsc{T-Reflect}\ \frac{S;\Gamma \vdash e : \mathsf{Tot}\ (S.F.repr\ t\ wp)}{S;\Gamma \vdash \mathsf{reflect}\ e : F\ t\ wp}$$

$$\textsc{T-Run}\ \frac{S;\Gamma \vdash e : \mathsf{Pure}\ t\ wp \quad S;\Gamma \models \exists p.wp\ p}{S;\Gamma \vdash \mathsf{run}\ e : \mathsf{Tot}\ t} \qquad \textsc{T-Sub}\ \frac{S;\Gamma \vdash e : c' \quad S;\Gamma \vdash c' <: c}{S;\Gamma \vdash e : c}$$

$$\textsc{C-Pure}\ \frac{S;\Gamma \vdash t : \mathsf{Type}_0 \quad S;\Gamma \vdash wp : (t \to \mathsf{Type}_0) \to \mathsf{Type}_0}{S;\Gamma \vdash \mathsf{Pure}\ t\ wp : \mathsf{Type}_0} \qquad \textsc{C-F}\ \frac{S;\Gamma \vdash S.F.repr\ t\ wp : \mathsf{Type}_0}{S;\Gamma \vdash F\ t\ wp : \mathsf{Type}_0}$$

Figure 2. Selected typing rules for EMF*

$$\textsc{S-Tot}\ \frac{S;\Gamma \vdash t' <: t}{S;\Gamma \vdash \mathsf{Tot}\ t' <: \mathsf{Tot}\ t} \qquad \textsc{S-Pure}\ \frac{S;\Gamma \vdash t' <: t \quad S;\Gamma \models \forall p.wp\ p \Rightarrow wp'\ p}{S;\Gamma \vdash \mathsf{Pure}\ t'\ wp' <: \mathsf{Pure}\ t\ wp}$$

$$\textsc{S-F}\ \frac{S;\Gamma \vdash S.F.repr\ t'\ wp' <: S.F.repr\ t\ wp}{S;\Gamma \vdash F\ t'\ wp' <: F\ t\ wp} \qquad \textsc{S-Prod}\ \frac{S;\Gamma \vdash t <: t' \quad S;\Gamma, x : t \vdash c' <: c}{S;\Gamma \vdash x{:}t' \to c' <: x{:}t \to c}$$

$$\textsc{S-RefineL}\ \frac{}{S;\Gamma \vdash x{:}t\{\phi\} <: t} \qquad \textsc{S-RefineR}\ \frac{S;\Gamma, x : t \models \phi}{S;\Gamma \vdash t <: x{:}t\{\phi\}} \qquad \textsc{S-Conv}\ \frac{S \vdash t' \longrightarrow^* t \ \vee\ S \vdash t \longrightarrow^* t'}{S;\Gamma \vdash t' <: t}$$

Figure 3. Selected subtyping rules for EMF*

Subsumption and subtyping judgment. T-SUB is a subsumption rule for computations, which makes use of the two judgments $S;\Gamma \vdash c <: c'$ and $S;\Gamma \vdash t <: t'$, shown (selectively) in Figure 3. Rule S-PURE checks that $t' <: t$, and makes use of the $S;\Gamma \models \phi$ relation to check that *wp* is stronger than wp', i.e. for all postconditions, the precondition computed by *wp* implies the precondition computed by wp'.

Similar to C-F, the rule S-F delegates the check to the underlying representation of F. Rule S-PROD is the standard dependent function subtyping. Rule S-REFINEL permits dropping the refinement from the subtype, and rule S-REFINER allows subtyping to a refinement

[3] We use $\mathsf{sans\ serif}$ font for the actual field values.

$$\frac{}{S \vdash (\lambda x{:}t.e)\ e' \longrightarrow e[e'/x]}\ \text{R-APP} \qquad \frac{}{S \vdash \mathsf{run}\ (\mathsf{Pure.return}\ t\ e) \longrightarrow e}\ \text{R-RUN}$$

$$\frac{}{S \vdash \mathsf{Pure.bind}\ t_1\ t_2\ wp_1\ (\mathsf{Pure.return}\ t\ e_1)\ wp_2\ x.e_2 \longrightarrow e_2[e_1/x]}\ \text{R-PUREBIND}$$

$$\frac{}{S \vdash \mathsf{reify}\ (F.\mathsf{return}\ t\ e) \longrightarrow S.F.\underline{return}\ t\ e}\ \text{R-REIFYRET} \qquad \frac{}{S \vdash \mathsf{reify}\ (\mathsf{reflect}\ e) \longrightarrow e}\ \text{R-REIFYREFLECT}$$

$$\frac{e' = S.F.\underline{bind}\ t_1\ t_2\ wp_1\ (\mathsf{reify}\ e_1)\ wp_2\ x.(\mathsf{reify}\ e_2)}{S \vdash \mathsf{reify}\ (F.\mathsf{bind}\ t_1\ t_2\ wp_1\ e_1\ wp_2\ x.e_2) \longrightarrow e'}\ \text{R-REIFYBIND}$$

$$\frac{}{S \vdash \mathsf{reify}(F.\mathsf{act}\ \bar{e}) \longrightarrow S.F.\underline{act}\ \bar{e}}\ \text{R-REIFYACT}$$

$$\frac{}{S \vdash \mathsf{reify}(M.\mathsf{lift}_{M'}\ t\ wp\ e) \longrightarrow S.\underline{M.lift}_{M'}\ t\ wp\ (\mathsf{reify}\ e)}\ \text{R-REIFYLIFT}$$

Figure 4. Dynamic semantics of EMF* (selected reduction rules)

type, if we can prove the formula ϕ for an arbitrary x. Finally, rule S-CONV states that the beta-convertible types are subtypes of each other ($S \vdash t \longrightarrow t'$ is the small-step evaluation judgment, introduced in the next section).

3.3 EMF* Dynamic Semantics

We now turn to the dynamic semantics of EMF*, which is formalized as a strong small-step reduction relation. Evaluation context are defined as follows:

$$\begin{array}{rcl} E & ::= & \bullet \mid \lambda x{:}t.E \mid E\ e \mid e\ E \mid \mathsf{run}\ E \mid \mathsf{reify}\ E \mid \mathsf{reflect}\ E \\ & \mid & M.\mathsf{bind}\ t_1\ t_2\ wp_1\ E\ wp_2\ x.e_2 \mid M.\mathsf{return}\ t\ E \\ & \mid & M.\mathsf{lift}_{M'}\ t\ wp\ E \mid F.\mathsf{act}\ \bar{e}\ E\ \bar{e}' \mid \mathsf{case}_t(E\ \mathsf{as}\ \_)\ x.e_1\ x.e_2 \\ & \mid & \mathsf{case}_t(e\ \mathsf{as}\ \_)\ x.E_1\ x.e_2 \mid \mathsf{case}_t(e\ \mathsf{as}\ \_)\ x.e_1\ x.E_2 \end{array}$$

The judgment has the form $S \vdash e \longrightarrow e'$. We show some selected rules in Figure 4. The main ideas of the judgment are: (a) the Tot terms reduce primitively in using a strong reduction semantics, (b) Pure.bind is also given a primitive semantics, however (c) to β-reduce other monadic operations (binds, returns, actions, and lifts), they need to be reified first, which then makes progress using their underlying implementation in the signature.

Order of evaluation. Since the effectful terms reduce via reification, the semantics does not impose any evaluation order on the effects—reification yields Tot terms (T-REIFY), that reduce using the strong reduction semantics. However, the more familiar sequencing semantics of effects can be recovered by controlled uses of reify that do not break the abstraction of effects arbitrarily. Indeed, we formalize this notion in Section 5, and prove that by sequencing the effects as usual using bind, and then reifying and reducing the entire effectful term, one gets the expected strict evaluation semantics (Theorem 10).

Semantics for* Pure *terms. Rule R-PUREBIND reduces similarly to the usual β-reduction. For run e, the semantics first evaluates e to Pure.return $t\ e'$, and then run removes the Pure.return and steps to the underlying total computation e' via R-RUN.

Semantics for monadic returns and binds. Rule R-REIFYBIND looks up the underlying implementation $S.F.\underline{bind}$ in the signature, and applies it to e_1 and e_2 but after reifying them so that their effects are handled properly. In a similar manner, rule R-REIFYRET looks up the underlying implementation $S.F.\underline{return}$ and applies it to e. Note that in this case, we don't need to reify e (as we did in bind), because e is already a Tot term.

Semantics for monadic lifts and actions. Rules R-REIFYACT and R-REIFYLIFT also lookup the underlying implementations of the lifts and actions in the signature and use them. Rule R-REIFYLIFT in addition reifies the computation e. For lifts, the arguments $\bar{e}$ are already Tot.

3.4 EMF* Metatheory

We prove several metatheoretical results for EMF*. First, we prove strong normalization for EMF* via a translation to the calculus of inductive constructions (CiC) (Paulin-Mohring 2015).

Theorem 1 (Strong normalization)**.** *If $S;\Gamma \vdash e : c$ and CiC is strongly normalizing, then e is strongly normalizing.*

Proof. (sketch) The proof proceeds by defining a translation from EMF* to CiC, erasing refinements and WPs, inlining the pure implementations of each monad, and removing the reify and reflect operators. We show that this translation is a type-preserving, forward simulation. If CiC is strongly normalizing, then EMF* must also be, since otherwise an infinite reduction sequence in EMF* could not be matched by CiC, contradicting the forward simulation. □

Theorem 2 (Subject Reduction)**.** *If $S;\Gamma \vdash e : c$ and $S \vdash e \longrightarrow e'$, then $S;\Gamma \vdash e' : c$.*

This allows us to derive a total correctness property for the Pure monad saying that run-ing a Pure computation produces a value which satisfies all the postconditions that are consistent with the *wp* of the Pure computation.

Corollary 3 (Total Correctness of Pure)**.** *If $S;\cdot \vdash e : \mathsf{Pure}\ t\ wp$, then $\forall p.\ S;\cdot \vdash p : t \to \mathsf{Type}_0$ and $S;\cdot \models wp\ p$, we have $S \vdash \mathsf{run}\ e \longrightarrow^* v$ such that $S;\cdot \models p\ v$.*

For the user-defined monads F, we can derive their total correctness property by appealing to the total correctness of the Pure monad. For instance, for the ST monad from §2.3, we can derive the following corollary simply by using the typing of reify and Corollary 3.

Corollary 4 (Total Correctness of *ST*)**.** *If $S;\cdot \vdash e : ST\ t\ wp$, then $\forall p, s_0.\ S;\cdot \vdash s_0 : s,\ S;\cdot \vdash p : t \times s \to \mathsf{Type}_0$ and $S;\cdot \models wp\ s_0\ p$, then $S \vdash \mathsf{run}\ ((\mathsf{reify}\ e)\ s_0) \longrightarrow^* v$ such that $S;\cdot \models p\ v$.*

3.5 Implementation in F*

The implementation of F* was relatively easy to adapt to EMF*. In fact, EMF* and DM and the translation between them were designed to match F*'s existing type system, as much as possible. We describe the main changes that were made.

User-defined non-primitive effects are, of course, the main new feature. Effect configurations closely match the D form from Figure 1, the main delta being that non-primitive effects include pure implementations or $M.\underline{bind}$, $M.\underline{return}$, $M.\underline{lift}_{M'}$ etc.

***Handling* reify *and* reflect** in the type-checker involved implementing the two relatively simple rules for them in Figure 2. A more significant change was made to F*'s normalization machinery, extending it to support rules that trigger evaluation for reified, effectful programs. In contrast, before our changes, F* would never reduce effectful terms. The change to the normalizer is exploited by F*'s encoding of proof obligations to an SMT solver—it now encodes the semantics of effectful terms to the solver, after using the normalizer to partially evaluate a reified effectful term to its pure form.

4. Dijkstra Monads for Free

This section formally presents DM, a language for defining effects by giving monads with their actions and lifts between them. Via a pair of translations, we export such definitions to EMF* as effect configurations. The first translation of a term e, a CPS, written $e^\star$ produces a predicate-transformer from DM term; the second one is an *elaboration*, $\underline{e}$, which produces an EMF* implementation of a DM term. The main result shows that for any DM term the result of the $\star$-translation is in a suitable logical relation to the elaboration of the term, and thus a valid specification for this elaboration. We also show that the $\star$-translation always produces monotonic and conjunctive predicates, properties that should always hold for WPs. Finally, we show that the $\star$-translation preserves all equalities in DM, and thus translates DM monads into EMF* Dijkstra monads.

4.1 Source: DM Effect Definition Language

The source language DM is a simply-typed lambda calculus augmented with an abstract monad τ, as in §2.3. The language is essentially that of Filinski (1994) with certain restrictions on allowed types to ensure the correctness of elaboration.

There are two effect symbols: n (non-effectful) and τ. The typing judgment is split accordingly, and ε ranges over both of them. Every monadic term needs to be bound via $\mathbf{bind}_\tau$ to be used.[4] Functions can only take non-effectful terms as arguments, but may return a monadic result.

The set of DM types is divided into *A types*, *H types*, and *C types*, ranged over by A, C, and H, respectively. They are given by the grammar:

$$\begin{array}{lcl} A & ::= & X \mid b \mid A \xrightarrow{n} A \mid A + A \mid A \times A \\ H & ::= & A \mid C \\ C & ::= & H \xrightarrow{\tau} A \mid H \xrightarrow{n} C \mid C \times C \end{array}$$

Here X ranges over type variables (needed to define monads) and b are base types. The τ-arrows represent functions with a monadic result, and our translations will provide WPs for these arrows. A types are referred to as "τ-free", since they contain no monadic operations. C types are inherently computational in the sense that they cannot be eliminated into an A type: every possible elimination will lead to a monadic term. They are referred to as "computational types". H types are the union of both, and are called "hypothesis" types, as they represent the types of possible functional arguments. As an example, the state monad is represented as the type $S \xrightarrow{\tau} (X \times S)$, where X is a type variable and S is some type representing the state. We will exemplify our main results for terms of this type, thus covering every stateful computation definable in DM.

DM types do not include "mixed" $A \times C$ pairs, computational sums $C + H$, functions of type $C \xrightarrow{n} A$, or types with right-nested τ-arrows. We do allow nesting τ-arrows to the left, providing the generality needed for the continuation monad, and others. These restrictions are crafted to carefully match EMF*. Without them, our translations, would generate ill-typed or logically unrelated EMF* terms, and these restrictions do not appear to be severe in practice, as evidenced by the examples in §2.

[4] In this formalization, **bind** and **return** appear explicitly in source programs. When using our implementation, however, the user need not call **bind** and **return**; rather, they write programs in a direct style, and **let**-bindings are turned into **bind**s as needed. §4.6 provides some details on the interpretation and elaboration of concrete F* terms as DM terms.

The syntax for terms is (κ standing for constants):

$$\begin{array}{lcl} e & ::= & x \mid e\ e \mid \lambda x{:}H.\ e \mid \kappa(e,\ldots,e) \\ & \mid & (e,e) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \\ & \mid & \mathbf{inl}(e) \mid \mathbf{inr}(e) \mid \mathbf{case}\ e\ \mathbf{inl}\ x{:}A.\ e;\ \mathbf{inr}\ y{:}A.\ e \\ & \mid & \mathbf{return}_\tau\ e \mid \mathbf{bind}_\tau\ e\ \mathbf{to}\ x\ \mathbf{in}\ e \end{array}$$

Typing judgments have the forms $\Delta \mid \Gamma \vdash e : H \,!\, n$ and $\Delta \mid \Gamma \vdash e : A \,!\, \tau$, where Δ is a finite sequence of type variables and Γ is a normal typing context, whose types only use type variables from Δ. Here are some example rules:

$$\frac{\Delta \mid \Gamma, x{:}H \vdash e : H'!\varepsilon}{\Delta \mid \Gamma \vdash \lambda x{:}H.\ e : H \xrightarrow{\varepsilon} H'!n} \qquad \frac{\Delta \mid \Gamma \vdash f : H \xrightarrow{\varepsilon} H'!n \quad \Delta \mid \Gamma \vdash e : H!n}{\Delta \mid \Gamma \vdash f e : H'!\varepsilon}$$

$$\frac{\Delta \mid \Gamma \vdash e : A!n}{\Delta \mid \Gamma \vdash \mathbf{return}_\tau\ e : A!\tau} \qquad \frac{\Delta \mid \Gamma \vdash e_1 : A!\tau \quad \Delta \mid \Gamma, x : A \vdash e_2 : A'!\tau}{\Delta \mid \Gamma \vdash \mathbf{bind}_\tau\ e_1\ \mathbf{to}\ x\ \mathbf{in}\ e_2 : A'!\tau}$$

In these rules we implicitly assume that all appearing types are well-formed with respect to the grammar, e.g., one cannot form a function of type $C \xrightarrow{n} A$ by the abstraction rule.

As an example, $\text{return}_{\text{ST}} = \lambda x{:}X.\ \lambda s{:}S.\ \mathbf{return}_\tau\ (x,s)$ has type $X \xrightarrow{n} S \xrightarrow{\tau} (X \times S)$, using these rules.

When defining effects and actions, one deals (at a top level) with non-effectful C types ($C \,!\, n$).

4.2 The $\star$-translation

The essence of the $\star$-translation is to translate $\mathbf{return}_\tau\ e$ and $\mathbf{bind}_\tau\ e_1\ \mathbf{to}\ x\ \mathbf{in}\ e_2$ to the returns and binds of the continuation monad. We begin by defining a translation $H^\star$, that translates any H type to the type of its predicates by CPS'ing the τ-arrows. First, for any τ-free type A, $A^\star$ is essentially the identity, except we replace every arrow $\xrightarrow{n}$ by a $\rightarrow$. Then, for computation types, we define:

$$\begin{array}{lcl} (H \xrightarrow{n} C)^\star & = & H^\star \rightarrow C^\star \\ (C \times C')^\star & = & C^\star \times C'^\star \\ (H \xrightarrow{\tau} A)^\star & = & H^\star \rightarrow (A^\star \rightarrow \mathsf{Type}_0) \rightarrow \mathsf{Type}_0 \end{array}$$

Note that all arrows on the right hand side have a Tot codomain, as per our notational convention.

In essence, the codomains of τ-arrows are CPS'd into a WP, which takes as argument a predicate on the result and produces a predicate representing the "precondition". All other constructs are just translated recursively: the real work is for the τ-arrows.

For example, for the state monad $S \xrightarrow{\tau} (X \times S)$, the $\star$-translation produces the EMF* type $S \rightarrow (X \times S \rightarrow \mathsf{Type}_0) \rightarrow \mathsf{Type}_0$. It is the type of predicates that map an initial state and a postcondition (on both result and state) into a proposition. Modulo isomorphism (of the order of the arguments and currying)[5] this is exactly the type of WPs in current F*'s state monad (cf. §1, §2.3).

The two main cases for the $\star$-translation for well-typed DM terms are shown below; every other case is simply a homomorphic application of $\star$ on the sub-terms.

$$\begin{array}{ll} (\mathbf{return}_\tau\ e)^\star = \lambda p{:}(A^\star \rightarrow \mathsf{Type}_0).\ p\ e^\star & {\scriptstyle\text{when}}\ \Delta \mid \Gamma \vdash e : A!n \\ (\mathbf{bind}_\tau\ e_1\ \mathbf{to}\ x\ \mathbf{in}\ e_2)^\star = \lambda p{:}(A'^\star \rightarrow \mathsf{Type}_0).\ e_1^\star\ (\lambda x{:}A.\ e_2^\star\ p) & \\ & {\scriptstyle\text{when}}\ \Delta \mid \Gamma, x : A \vdash e_2 : A'!\tau \end{array}$$

Formally, the $\star$-translation and elaboration are defined over a typing derivation, as one needs more information than what is present in the term. The $\star$-translations of terms and types are related in the following sense, where we define the environments $\underline{\Delta}$ as $X_1 : \mathsf{Type}_0, \ldots, X_n : \mathsf{Type}_0$ when $\Delta = X_1, \ldots, X_n$; and $\Gamma^\star$ as $x_1 : t_1^\star, \ldots x_n : t_n^\star$ when $\Gamma = x_1 : t_1, \ldots x_n : t_n$ (we assume that variables and type variables are also EMF* variables).

[5] One can tweak our translation to generate WPs that have the usual postcondition to precondition shape. However we found the current shape to be generally easier to work with.

Theorem 5 (well-typing of $\star$-translation)**.**
$\Delta \mid \Gamma \vdash e : C\,!\,n$ *implies* $\underline{\Delta}, \Gamma^\star \vdash e^\star : C^\star$.

After translating a closed term e, one can abstract over the variables in $\underline{\Delta}$ to introduce the needed polymorphism in EMF*. This will also be the case for elaboration.

As an example, for the previous definition of $\mathsf{return}_{\mathrm{ST}}$ we get the translation $\lambda x{:}X.\ \lambda s{:}S.\ \lambda p{:}(X \times S \to \mathsf{Type}_0).\ p(x,s)$, which has the required transformer type: $X \to S \to (X \times S \to \mathsf{Type}_0) \to \mathsf{Type}_0$ (both with X as a free type variable). It is what one would expect: to prove a postcondition p about the result of running $\mathsf{return}_{\mathrm{ST}}\ x$, one needs to prove $p(x,s)$ where s is the initial state.

4.3 Elaboration

Elaboration is merely a massaging of the source term to make it properly typed in EMF*. During elaboration, monadic operations are translated to those of the identity monad in EMF*, namely Pure.

Elaboration of types We define two elaboration translations for DM types, which produce the EMF* types of the elaborated expression-level terms. The first translation $\underline{A}$ maps an A type to a simple EMF* type, while the second one $\mathrm{F}_C\ wp$ maps a C type and a *specification wp* of type $C^\star$ into an EMF* computation type containing Tot and Pure arrows. The $\underline{A}$ translation is the same as the CPS one, i.e., $\underline{A} = A^\star$.

The $\mathrm{F}_C\ wp$ (where $wp : C^\star$) translation is defined by:

$$
\begin{array}{lll}
(1) & \mathrm{F}_{C\times C'}\ wp =_{\mathbf{def}} & \mathrm{F}_C\ (\mathsf{fst}\ wp) \times \mathrm{F}_{C'}\ (\mathsf{snd}\ wp)\\
(2) & \mathrm{F}_{C \xrightarrow{\varepsilon} H}\ wp =_{\mathbf{def}} & w'{:}C^\star \to \mathrm{F}_C\ w' \to \mathrm{G}^{\varepsilon}_{H}(wp\ w')\\
(3) & \mathrm{F}_{A \xrightarrow{\varepsilon} H}\ wp =_{\mathbf{def}} & x{:}\underline{A} \to \mathrm{G}^{\varepsilon}_{H}(wp\ x)
\end{array}
$$

Here we define $\mathrm{G}^n_C(wp) = \mathrm{F}_C\ wp$ and $\mathrm{G}^\tau_A(wp) = \mathsf{Pure}\ \underline{A}\ wp$.

The main idea is that if an EMF* term e has type $\mathrm{F}_C\ wp$, then wp is a proper specification of the final result. Putting pairs aside for a moment, this means that if one applies enough arguments e_i to e in order to eliminate it into a Pure computation, then $e\ \bar{e}_i : \mathsf{Pure}\ A\ (wp\ \bar{s}_i)$, where each s_i is the specification for each e_i. This naturally extends to pairs, for which the specification is a pair of proper specifications, as shown by case (1) above.

In case (2), the $w' : C^\star$ arguments introduced by F are relevant for the higher-order cases, and serve the following purpose, as illustrated in §2.3 (for the translation of bind for the ST monad) and §2.9 (for the continuation monad): when taking computations as arguments, we first require their specification in order to be able to reason about them at the type level. Taking these specification arguments is also the only way for being WP-polymorphic in EMF*. Note that, according to the dependencies, only the $C^\star$ argument is used in the specifications, while we shall see in the elaboration of terms that only the $\mathrm{F}_C\ wp$ argument is used in terms. When elaborating terms, we pass this specification as an extra argument where needed.

In case (3), when elaborating functions taking an argument of A type there is no need to take a specification, since the argument is completely non-effectful and can be used at both the expression and the type levels. Informally, a non-effectful term is its own specification.

Returning to our state monad example, the result of $\mathrm{F}_{S \xrightarrow{\tau} (X\times S)}\ wp$ is $s{:}S \to \mathsf{Pure}\ (X \times S)\ (wp\ s)$, i.e., the type of a function f such that for any postcondition p and states s for which one can prove the precondition $wp\ s\ p$, we have that $f\ s$ satisfies p.

Elaboration of terms is defined in Figure 5 and is, as expected, mostly determined by the translation of types. The translation is formally defined over typing derivations, however, for brevity, we present each translation rule simply on the terms, with the important side-conditions we rely on from the derivation shown in parenthesis. We describe only the most interesting cases.

Computational abstractions and applications (cases 4 and 10) Case (4) translates a function with a computational argument $x{:}C$ to a function that expects two arguments, a specification $x^w{:}C^\star$ and x itself, related to x^w at a suitably translated type. We track the association between x and x^w using a substitution s_Γ, which maps every computational hypothesis $x : C$ in Γ to x^w (of type $C^\star$) in $\underline{\Gamma}$, In case (10), when passing a computation argument e_2, we need to eliminate the double abstraction introduced in case (4), passing both $e_2^\star\ s_\Gamma$, *i.e.* the specification of e_2 where we substitute the free computation variables, and $\underline{e_2}$ itself.

Return and bind (cases 14 and 15) The last two rules show the translation of return and bind for τ to return and bind for Pure in EMF*. This is one of the key points: in the elaboration, we interpret the τ as the identity monad in EMF*, whereas in the $\star$-translation, we interpret τ as the continuation monad. Theorem 6, our main theorem, shows that EMF*'s WP computation in the Pure monad for $\underline{e}$ produces a WP that is logically related to the $\star$-translation of e, i.e., WPs and the CPS coincide formally, at arbitrary order.

Theorem 6 (Logical relations lemma)**.**

1. $\Delta \mid \Gamma \vdash e : C\,!\,n \implies \underline{\Delta}, \underline{\Gamma} \vdash \underline{e} : \mathrm{F}_C\ (e^\star \mathrm{s}_\Gamma)$
2. $\Delta \mid \Gamma \vdash e : A\,!\,\tau \implies \underline{\Delta}, \underline{\Gamma} \vdash \underline{e} : \mathsf{Pure}\ \underline{A}\ (e^\star \mathrm{s}_\Gamma)$

Where $\underline{\Gamma}$ is defined by mapping any "$x : A$" binding in Γ to "$x : \underline{A}$" and any "$y : C$" binding to "$y^w : C^\star, y : \mathrm{F}_C\ y^w$". Instantiating (1) for an empty Γ, we get as corollary that $\underline{\Delta} \vdash \underline{e} : \mathrm{F}_C\ e^\star$, representing the fact that $e^\star$ is a proper specification for $\underline{e}$. Following the ST monad example, this implies that for any source term e such that $X \mid \cdot \vdash e : S \xrightarrow{\tau} (X \times S)$ holds, then $X : \mathsf{Type}_0 \vdash \underline{e} : s_0{:}S \to \mathsf{Pure}\ (X \times S)\ (e^\star\ s_0)$, will hold in EMF*, as intuitively expected.

4.4 Monotonicity and Conjunctivity

A key property of WPs is monotonicity: weaker postconditions should map to weaker preconditions. This is also an important F* invariant that allows for logical optimizations of WPs. Similarly, WPs are conjunctive: they distribute over conjunction and universal quantification in the postcondition. We show that any EMF* term obtained from the $\star$-translation is monotonic and conjunctive, for higher-order generalizations of the usual definitions of these properties (Dijkstra 1997).

We first introduce a hereditarily-defined relation between EMF* terms $t_1 \lesssim_t t_2$, read "t_1 stronger than t_2 at type t" and producing an EMF* formula in Type_0, by recursion on the structure of t:

$$
\begin{array}{ll}
x \lesssim_{\mathsf{Type}_0} y & =_{\mathbf{def}} x \Rightarrow y\\
x \lesssim_b y & =_{\mathbf{def}} x == y\\
x \lesssim_X y & =_{\mathbf{def}} x == y\\
f \lesssim_{t_1 \to t_2} g & =_{\mathbf{def}} \forall x, y : t_1 .\, x \lesssim_{t_1} x \wedge x \lesssim_{t_1} y \wedge y \lesssim_{t_1} y \Rightarrow f\ x \lesssim_{t_2} g\ y\\
x \lesssim_{t_1 \times t_2} y & =_{\mathbf{def}} \mathsf{fst}\ x \lesssim_{t_1} \mathsf{fst}\ y \wedge \mathsf{snd}\ x \lesssim_{t_2} \mathsf{snd}\ y\\
x \lesssim_{t_1 + t_2} y & =_{\mathbf{def}} (\exists v_1, v_2 : t_1, x == \mathsf{inl}\ v_1 \wedge y == \mathsf{inl}\ v_2 \wedge v_1 \lesssim_{t_1} v_2) \vee\\
 & \qquad (\exists v_1, v_2 : t_2, x == \mathsf{inr}\ v_1 \wedge y == \mathsf{inr}\ v_2 \wedge v_1 \lesssim_{t_2} v_2)
\end{array}
$$

where b represents any EMF* base type (i.e., a type constant in Type_0) and X any type variable[6]. The symbol $==$ represents EMF*'s squashed propositional equality. The $\lesssim$ relation is only defined for the subset of EMF* types that are all-Tot and non-dependent. All types resulting from the $\star$-translation are in this subset, so this not a limitation for our purposes. A type t in this subset is called *predicate-free* when it does not mention Type_0. For any predicate-free type t the relation $\lesssim_t$ reduces to extensional equality.

The $\lesssim$ relation is not reflexive. We say that an EMF* term e of type t is *monotonic* when $e \lesssim_t e$. Note that monotonicity is preserved by application. For first-order WPs this coincides with the standard

[6] We can get a stronger result if we don't restrict the relation on type variables to equality and treat it abstractly instead. For our purposes this is not needed as we plan to instantiate type variables with predicate-free types.

$$
\begin{array}{llcllllcl}
(1) & \underline{x} & = & x & & (5) & \underline{\mathbf{fst}(e)} & = & \mathsf{fst}\ \underline{e} \\
(2) & \underline{\kappa(e_1,\ldots,e_n)} & = & \kappa\ \underline{e_1}\ \ldots\ \underline{e_n} & & (6) & \underline{\mathbf{snd}(e)} & = & \mathsf{snd}\ \underline{e} \\
(3) & \underline{\lambda x{:}A.\,e} & = & \lambda x{:}\underline{A}.\,\underline{e} & & (7) & \underline{\mathbf{inl}(e)} & = & \mathsf{inl}\ \underline{e} \\
(4) & \underline{\lambda x{:}C.\,e} & = & \lambda x^w{:}C^\star.\,\lambda x{:}\mathrm{F}_C\ x^w.\,\underline{e} & & (8) & \underline{\mathbf{inr}(e)} & = & \mathsf{inr}\ \underline{e} \\
(9) & \underline{e_1 e_2} & = & \underline{e_1}\ \underline{e_2} & & & & & (\Delta \mid \Gamma \vdash e_2 : A\,!\,n) \\
(10) & \underline{e_1 e_2} & = & \underline{e_1}\ (e_2^\star\ \mathrm{s}_\Gamma)\ \underline{e_2} & & & & & (\Delta \mid \Gamma \vdash e_2 : C\,!\,n) \\
(11) & \underline{(e_1,e_2)} & = & (\underline{e_1},\underline{e_2}) & & & & & \\
(12) & \underline{\mathbf{case}\ e\ \mathbf{inl}\ x{:}A_1.\,e_1;\ \mathbf{inr}\ y{:}A_2.\,e_2} & = & \mathsf{case}(\underline{e})\ x.\underline{e_1}\ y.\underline{e_2} & & & & & (\Delta \mid \Gamma, x{:}A_1 \vdash e_1 : A\,!\,\varepsilon) \\
(13) & \underline{\mathbf{case}\ e\ \mathbf{inl}\ x{:}A_1.\,e_1;\ \mathbf{inr}\ y{:}A_2.\,e_2} & = & \mathsf{case}_{\mathrm{F}_C\ \mathsf{case}(z)\ x.(e_1^\star\ \mathrm{s}_\Gamma)\ y.(e_2^\star\ \mathrm{s}_\Gamma)}(\underline{e}\ \mathsf{as}\ z)\ x.\underline{e_1}\ y.\underline{e_2} & & & & & (\Delta \mid \Gamma, x{:}A_1 \vdash e_1 : C\,!\,n) \\
(14) & \underline{\mathbf{return}_\tau\ e} & = & \mathsf{Pure.return}\ \underline{A}\ \underline{e} & & & & & (\Delta \mid \Gamma \vdash e : A\,!\,\tau) \\
(15) & \underline{\mathbf{bind}_\tau\ e_1\ \mathbf{to}\ x{:}A\ \mathbf{in}\ e_2} & = & \mathsf{Pure.bind}\ \underline{A}\ \underline{A'}\ (e_1^\star\ \mathrm{s}_\Gamma)\ \underline{e_1}\ (\lambda x{:}A^\star.\,e_2^\star\ \mathrm{s}_\Gamma)\ x.\underline{e_2} & & & & & (\Delta \mid \Gamma, x : A \vdash e_2 : A'\,!\,\tau)
\end{array}
$$

Figure 5. The elaboration of DM terms to EMF*

definitions, and for higher-order predicates it gives a reasonable extension. Since the relation reduces to equality on predicate-free types, every term of such a type is trivially monotonic. The reader can also check that every term of a type $t = d_1 \to \cdots \to d_n \to \mathsf{Type}_0$ (where each d_i is predicate-free) is monotonic; it is only at higher-order that monotonicity becomes interesting.

For a first-order example, let's take the type of WPs for programs in the ST monad: $S \to (X \times S \to \mathsf{Type}_0) \to \mathsf{Type}_0$, making use of the previous simplification:

$$
\begin{array}{rl}
& f \lesssim_{S \to (X \times S \to \mathsf{Type}_0) \to \mathsf{Type}_0} f \\
\equiv & \forall s_1, s_2.\, s_1 = s_1 \wedge s_1 = s_2 \wedge s_2 = s_2 \Rightarrow f\ s_1 \lesssim f\ s_2 \\
\iff & \forall s.\, f\ s \lesssim_{(X \times S \to \mathsf{Type}_0) \to \mathsf{Type}_0} f\ s \\
\equiv & \forall s, p_1, p_2.\, p_1 \lesssim p_2 \Rightarrow f\ s\ p_1 \lesssim_{\mathsf{Type}_0} f\ s\ p_2 \\
\iff & \forall s, p_1, p_2.\, (\forall x, s'.\, p_1\ (x,s') \Rightarrow p_2\ (x,s')) \Rightarrow (f\ s\ p_1 \Rightarrow f\ s\ p_2)
\end{array}
$$

This is exactly the usual notion of monotonicity for imperative programs (Dijkstra 1997): "if p_2 is weaker than p_1, then $f\ s\ p_2$ is weaker than $f\ s\ p_1$ for any s".

Now, for a higher-order example, consider the continuation monad in DM: $\mathsf{Cont}\ X = (X \xrightarrow{\tau} R) \xrightarrow{\tau} R$, where X is the type variable and R some other variable representing the end result of the computation. The type of WPs for this type is

$$\mathsf{Cont}_{wp}\ X = (X \to (R \to \mathsf{Type}_0) \to \mathsf{Type}_0) \to (R \to \mathsf{Type}_0) \to \mathsf{Type}_0$$

Modulo argument swapping, this maps a postcondition on R to a precondition on the specification of the continuation function. The condition $wp \lesssim_{\mathsf{Cont}_{wp}\ X} wp$ reduces and simplifies to:

$$
\begin{array}{c}
kw_1 \lesssim kw_1\ \wedge\ kw_1 \lesssim kw_2\ \wedge\ kw_2 \lesssim kw_2\ \wedge\ p_1 \lesssim p_2 \\
\Longrightarrow\ wp\ kw_1\ p_1\ \Longrightarrow\ wp\ kw_2\ p_2
\end{array}
$$

for any kw_1, kw_2, p_1, p_2 of appropriate types. Intuitively, this means that wp behaves monotonically on both arguments, but requiring that the first one is monotonic. In particular, this implies that for any monotonic kw, $wp\ kw$ is monotonic at type $(R \to \mathsf{Type}_0) \to \mathsf{Type}_0$.

We proved that the $\star$-translation of any well-typed source term $e : C\,!\,n$ gives a monotonic $e^\star$ at the type $C^\star$. This result is more general than it appears at a first glance: not only does it mean that the WPs of *any* defined return and bind are monotonic, but also those of any action or function are. Also, lifts between monads and other higher-level computations will preserve this monotonicity. Furthermore, the relation $\vDash$ in the conclusion of the theorem below is EMF*'s validity judgment, i.e., we show that these properties are actually provable within F* without relying on meta-level reasoning.

Theorem 7 (Monotonicity of $\star$-translation)**.**
For any e and C, $\Delta \mid \cdot \vdash e : C\,!\,n$ implies $\underline{\Delta} \vDash e^\star \leq_{C^\star} e^\star$.

We give a similar higher-order definition of conjunctivity, and prove similar results ensuring the $\star$-translation produces conjunctive WPs. The definition for conjunctivity is given below, where a describes the predicate-free types (including variables).

$$
\begin{array}{ll}
\mathbb{C}_{(a \to \mathsf{Type}_0) \to \mathsf{Type}_0}(w) & =_{\mathbf{def}} \forall p_1, p_2.\, w\ p_1 \wedge w\ p_2 = w\ (\lambda x.\, p_1\ x \wedge p_2\ x) \\
\mathbb{C}_a(x) & =_{\mathbf{def}} \mathsf{true} \\
\mathbb{C}_{t_1 \to t_2}(f) & =_{\mathbf{def}} \forall x : t_1.\, \mathbb{C}_{t_1}(x) \Rightarrow \mathbb{C}_{t_2}(f x) \\
\mathbb{C}_{t_1 \times t_2}(p) & =_{\mathbf{def}} \mathbb{C}_{t_1}(\mathsf{fst}\ p) \wedge \mathbb{C}_{t_2}(\mathsf{snd}\ p)
\end{array}
$$

Again, the relation is not defined on all types, but it does include the image of the type-level $\star$-translation, so it is enough for our purposes. This relation is trivially preserved by application, which allows us to prove the following theorem:

Theorem 8 (Conjunctivity of $\star$-translation)**.**
For any e and C, $\Delta \mid \cdot \vdash e : C\,!\,n$ implies $\underline{\Delta} \vDash \mathbb{C}_{C^\star}(e^\star)$

For the ST monad, this implies that for any e such that $e : S \xrightarrow{\tau} X \times S$ we know, again within EMF*, that $e^\star\ s\ p_1 \wedge e^\star\ s\ p_2 = e^\star\ s\ (\lambda x.\, p_1\ x \wedge p_2\ x)$ for any s, p_1, p_2. This is the usual notion of conjunctivity for WPs of this type.

4.5 The $\star$-translation Preserves Equality and Monad Laws

We define an equality judgment on DM terms that is basically $\beta\eta$-equivalence, augmented with the monad laws for the abstract τ monad. We show that the $\star$-translation preserves this equality.

Theorem 9 (Preservation of equality by CPS)**.**
If $\Delta \mid \cdot \vdash e_1 = e_2 : H\,!\,\varepsilon$ then $\underline{\Delta} \vDash e_1^\star == e_2^\star$.

Since the monad laws are equalities themselves, any source monad will be translated to a specification-level monad of WPs. This also applies to lifts: source monad morphisms are mapped to monad morphisms between Dijkstra monads.

4.6 Implementing the Translations in F*

We devised a prototype implementation of the two translations in F*. Users define their monadic effects as F* terms in direct style, as done in §2, and these definitions get automatically rewritten into DM. As explained in §2, instead of τ-arrows ($H \xrightarrow{\tau} A$), we use a distinguished F* effect τ to indicate where the CPS should occur. The effect τ is defined to be an alias for F*'s Tot effect, which allows the programmer to reason extrinsically about the definitions and prove that they satisfy various properties within F*, e.g., the monad laws. Once the definitions have been type-checked in F*, another minimalist type-checker kicks in, which has a twofold role. First, it ensures that the definitions indeed belong to DM, e.g., distinguishing A types from C types. Second, it performs bidirectional inference to distinguish monadic computations from pure computations, starting from top-level annotations, and uses this type information to automatically introduce $\mathbf{return}_\tau$ and $\mathbf{bind}_\tau$ as needed. For instance, in the st example from §2.3, the type-checker rewrites `x, s0` into $\mathbf{return}_\tau\ (x, s_0)$; and `let x, s1 = f s0 in ...` into $\mathbf{bind}_\tau\ f\ s_0\ \mathbf{to}\ x, s_1\ \mathbf{in}\ \ldots$; and `g x s1` into $\mathbf{return}_\tau\ (g\ x\ s_1)$. The elaboration maps let-bindings in DM to let-bindings in F*; the general inference mechanism in F* takes care of synthesizing the WPs, meaning that the elaboration, really, is only concerned about extra arguments for abstractions and applications.

Once the effect definition is rewritten to DM, our tool uses the $\star$-translation and elaboration to generate the WP transformers for the Dijkstra monad, which previously would be written by hand. Moreover, several other WP combinators are derived from the WP type and used internally by the F* type-checker; again previously these had to be written by hand.

5. EMF* with Primitive State

As we have seen in §3, EMF* encodes all its effects using pure functions. However, one would like to be able to run F* programs efficiently using primitively implemented effects. In this section, we show how EMF*'s pure monads apply to F*'s existing compilation strategy, which provides primitive support for state via compilation to OCaml, which, of course, has state natively.[7] The main theorem of §5.2 states that well-typed EMF* programs using the state monad abstractly (i.e., not breaking the abstraction of the state monad with arbitrary uses of reify and reflect) are related by a simulation to EMF$^\star_{\mathsf{ST}}$ programs that execute with a primitive notion of state. This result exposes a basic tension: although very useful for proofs, reify and reflect can break the abstractions needed for efficient compilation. However, as noted in §2.6, this restriction on the use of reify and reflect only applies to the *executable* part of a program—fragments of a program that are computationally irrelevant are erased by the F* compiler and are free to use these operators.

5.1 EMF$^\star_{\mathsf{ST}}$: A Sub-Language of EMF* with Primitive State

The syntax of EMF$^\star_{\mathsf{ST}}$ corresponds to EMF*, except, we configure it to just use the ST monad. Other effects that may be added to EMF* can be expanded into their encodings in its primitive Pure monad—as such, we think of EMF$^\star_{\mathsf{ST}}$ as modeling a compiler target for EMF* programs, with ST implemented primitively, and other arbitrary effects implemented purely. We thus exclude reify and reflect from EMF$^\star_{\mathsf{ST}}$, also dropping type and WP arguments of return, bind and lift operators, since these are no longer relevant here.

The operational semantics of EMF$^\star_{\mathsf{ST}}$ is a small-step, call-by-value reduction relation between pairs (s,e) of a state s and a term e. The relation includes the pure reduction steps of EMF* that simply carry the state along (we only show ST-beta), and three primitive reduction rules for ST, shown below. The only irreducible ST computation is ST.return v. Since the state is primitive in EMF$^\star_{\mathsf{ST}}$, the term $\mathsf{ST.bind}\ e\ x.e'$ reduces without needing an enclosing reify.

$$
\begin{array}{ll}
(s,(\lambda x{:}t.e)v) \rightsquigarrow (s,e[v/x]) & \text{ST-beta}\\
(s,\mathsf{ST.bind}\ (\mathsf{ST.return}\ v)\ x.e) \rightsquigarrow (s,e[v/x]) & \text{ST-bind}\\
(s,\mathsf{ST.get}\ ()) \rightsquigarrow (s,\mathsf{ST.return}\ s) & \text{ST-get}\\
(s,\mathsf{ST.put}\ s') \rightsquigarrow (s',\mathsf{ST.return}\ ()) & \text{ST-put}
\end{array}
$$

5.2 Relating EMF* to EMF$^\star_{\mathsf{ST}}$

We relate EMF* to EMF$^\star_{\mathsf{ST}}$ by defining a (partial) translation from the former to the latter, and show that one or more steps of reduction in EMF$^\star_{\mathsf{ST}}$ are matched by one or more steps in EMF*. This result guarantees that it is sound to verify a program in EMF* and execute it in EMF$^\star_{\mathsf{ST}}$: the verification holds for all EMF* reduction sequences, and EMF$^\star_{\mathsf{ST}}$ evaluation corresponds to one such reduction.

The main intuition behind our proof is that the reduction of reflect-free EMF* programs maintains terms in a very specific structure—a stateful redex (an ST computation wrapped in reify) reduces in a context structured like a telescope of binds, with the state threaded sequentially as the telescope evolves. We describe this invariant structure as an EMF* context, K, parameterized by a state s. In the definition, $\hat{E}$ is a single-hole, reify-and-reflect-free EMF* context, a refinement of the evaluation contexts of §3, to be filled by a reify-and-reflect free EMF* term, f. Additionally, we separate the $\hat{E}$ contexts by their effect into several sorts: $\hat{E}$: Tot and $\hat{E}$: Pure are contexts which when filled by a suitably typed term produce in EMF* a Tot or Pure term, respectively; the case $\hat{E}$: Inert is for an un-reified stateful EMF* term. The last two cases are the most interesting: they represent the base and inductive case of the telescope of a stateful term "caught in the act" of reducing—we refer to them as the Active contexts. We omit the sort of a context when it is irrelevant.

$$
\begin{array}{rcl}
K\ s & ::= & \hat{E} : \mathsf{Tot} \mid \hat{E} : \mathsf{Pure} \mid \hat{E} : \mathsf{Inert} \mid \mathsf{reify}\ \hat{E}\ s : \mathsf{Active}\\
& \mid & \mathsf{Pure.bind}\ (K\ s)\ p.((\lambda x.\mathsf{reify}\ f)\ (\mathsf{fst}\ p)\ (\mathsf{snd}\ p)) : \mathsf{Active}\\
& & \hfill (\text{if } K\ s : \mathsf{Active})
\end{array}
$$

Next, we define a simple translation $\{\![\cdot]\!\}$ from contexts $K\ s$ to EMF$^\star_{\mathsf{ST}}$.

$$
\begin{array}{l}
\{\![\hat{E}]\!\} = \hat{E}\\
\{\![\mathsf{reify}\ \hat{E}\ s]\!\} = \hat{E}\\
\{\![\mathsf{Pure.bind}\ (K\ s)\ p.((\lambda x.\mathsf{reify}\ f)\ (\mathsf{fst}\ p)\ (\mathsf{snd}\ p))]\!\}\\
\hfill = \mathsf{ST.bind}\ \{\![K\ s]\!\}\ x.f
\end{array}
$$

The definition of $\{\![\cdot]\!\}$ further illustrates why we need to structure the Active contexts as a telescope—because not every stateful computation that can reduce in EMF* is of the form reify e. For example, the reduction rule R-REIFYBIND pushes reify inside the arguments of bind. As a result, one needs to perform several "administrative" steps of reduction to get the resulting term back to being of the form reify e. However, in order to show that EMF$^\star_{\mathsf{ST}}$ can indeed be used as a compiler target for EMF*, we crucially need to relate all such intermediate redexes to ST computations in EMF$^\star_{\mathsf{ST}}$—thus the telescope-like definition of the Active contexts.

Finally, we prove the simulation theorem for EMF* and EMF$^\star_{\mathsf{ST}}$, which shows that one or more steps of reduction in EMF$^\star_{\mathsf{ST}}$ are matched by one or more steps in EMF*, in a compatible way.

Theorem 10 (Simulation)**.** *For all well-typed, closed, filled contexts $K\ s\ f$, either $K\ s$ is Inert, or one of the following is true:*

(1) $\exists K' s' f'.\ (s, \{\![K\ s]\!\}\ f) \rightsquigarrow^+ (s', \{\![K'\ s']\!\}\ f')$ *and* $K\ s\ f \longrightarrow^+ K'\ s'\ f'$ *and* $sort\,(K\ s) = sort\,(K'\ s')$ *and if* $K'\ s'$ *is not Active then* $s = s'$.
(2) $K\ s$ *is Active and* $\exists v\ s'.\ (s, \{\![K\ s]\!\}\ f) \rightsquigarrow^* (s', \mathsf{ST.return}\ v)$ *and* $K\ s\ f \longrightarrow^+ \mathsf{Pure.return}\ (v,s')$.
(3) $K\ s$ *is Pure and* $\exists v.\ \{\![K\ s]\!\}\ f = K\ s\ f = \mathsf{Pure.return}\ v$.
(4) $K\ s$ *is Tot and* $\exists v.\ \{\![K\ s]\!\}\ f = K\ s\ f = v$.

6. Related Work

We have already discussed many elements of related work throughout the paper. Here we focus on a few themes not covered fully elsewhere.

Our work builds on the many uses of monads for programming language semantics found in the literature. Moggi (1989) was the first to use monads to give semantics to call-by-value reduction—our Theorem 10 makes use of the monadic structure of EMF* to show that it can safely be executed in a strict semantics with primitive state. Moggi (1989), Wadler (1990, 1992), Filinski (1994, 1999, 2010), Benton et al. (2000) and others, use monads to introduce effects into a functional language—our approach of adding user-defined effects to the pure EMF* calculus follows this well-trodden path. Moggi (1989), Flanagan et al. (1993), Wadler (1994) and others, have used monads to provide a foundation on which to understand program transformations, notably CPS—we show that weakest precondition semantics can be formally related to CPS via our main logical relation theorem (Theorem 6).

[7] F* also compiles exceptions natively to OCaml, however we focus only on state here, leaving a formalization of primitive exceptions to the future—we expect it to be similar to the development here.

Representing monads Our work also draws a lot from Filinski's (1994) monadic reflection methodology, for representing and controlling the abstraction of monads. In particular, our DM monad definition language is essentially the language of (Filinski 1994) with some restrictions on the allowed types. Beyond controlling abstraction, Filinski shows how monadic reflection enables a universal implementation of monads using composable continuations and a single mutable cell. We do not (yet) make use of that aspect of his work, partly because deploying this technique in practice is challenging, since it requires compiling programs to a runtime system that provides composable continuations. Filinski's (1999) work on representing layered monads generalizes his technique to the setting of multiple monads. We also support multiple monads, but instead of layering monads, we define each monad purely, and relate them via morphisms. This style is better suited to our purpose, since one of our primary uses of reification is purification, i.e., revealing the pure representation of an effectful term for reasoning purposes. With layering, multiple steps of reification may be necessary, which may be inconvenient for purification. Finally, Filinski (2010) gives an operational semantics that is extensible with monadic actions, taking the view of effects as being primitive, rather than encoded purely. We take a related, but slightly different view: although effects are encoded purely in EMF*, we see it as language in which to analyze and describe the semantics of a primitively effectful object language, EMF$^\star_{\mathrm{ST}}$, relating the two via a simulation.

Dependent types and effects Nanevski et al. developed Hoare type theory (HTT) (Nanevski et al. 2008) and Ynot (Chlipala et al. 2009) as a way of extending Coq with effects. The strategy there is to provide an axiomatic extension of Coq with a single catch-all monad in which to encapsulate imperative code. Being axiomatic, their approach lacks the ability to reason extrinsically about effectful terms by computation. However, their approach accommodates effects like non-termination, which EMF* currently lacks. Interestingly, the internal semantics of HTT is given using predicate transformers, similar in spirit to EMF*'s WP semantics. It would be interesting to explore whether or not our free proofs of monotonicity and conjunctivity simplify the proof burden on HTT's semantics.

Zombie (Casinghino et al. 2014) is a dependently typed language with general recursion, which supports reasoning extrinsically about potentially divergent code—this approach may be fruitful to apply to EMF* to extend its extrinsic reasoning to divergent code.

Another point in the spectrum between extrinsic and intrinsic reasoning is Charguéraud's (2011) characteristic formulae, which provide a precise formula in higher-order logic capturing the semantics of a term, similar in spirit to our WPs. However, as opposed to WPs, characteristic formulae are used interactively to prove program properties after definition, although not via computation, but via logical reasoning. Interestingly enough, characteristic formulae are structured in a way that almost gives the illusion that they are the terms themselves. CFML is tool in Coq based on these ideas, providing special tactics to manipulate formulas structured this way.

Brady (2013, 2014) encodes algebraic effects with pre- and postconditions in Idris in the style of Atkey's (2009) parameterized monads. Rather than speaking about the computations themselves, the pre- and postconditions refer to some implicit state of the world, e.g., whether or not a file is closed. In contrast, F*'s WPs give a full logical characterization of a computation. Additionally, the WP style is better suited to computing verification conditions, instead of explicitly chaining indices in the parameterized monad.

It would be interesting, and possibly clarifying, to link up with recent work on the denotational semantics of effectful languages with dependent types (Ahman et al. 2016); in our case one would investigate the semantics of EMF* and EMF$^\star_{\mathrm{ST}}$, which has state, but extended with recursion (and so with nontermination).

Continuations and predicate transformers We are not the first to study the connection between continuations and predicate transformers. For example, Jensen (1978) and Audebaud and Zucca (1999) both derive WPs from a continuation semantics of first-order imperative programs. While they only consider several primitive effects, we allow arbitrary monadic definitions of effects. Also while their work is limited to the first-order case, we formalize the connection between WPs and CPS also for higher-order. The connection between WPs and the continuation monad also appears in Keimel (2015); Keimel and Plotkin (2016).

7. Looking Back, Looking Ahead

While our work has yielded the pleasant combination of both a significant simplification and boost in expressiveness for F*, we believe it can also provide a useful foundation on which to add user-defined effects to other dependently typed languages. All that is required is the Pure monad upon which everything else can be built, mostly for free.

On the practical side, going forward, we hope to make use of the new extrinsic proving capabilities in F* to simplify specifications and proofs in several ongoing program verification efforts that use F*. We are particularly interested in furthering the relational verification style, sketched in §2.8. We also hope to scale EMF* to be a definitive semantics of all of F*—the main missing ingredients are recursion and its semantic termination check, inductive types, universe polymorphism, and the extensional treatment of equality. Beyond the features currently supported by F*, we would like to investigate adding indexed effects and effect polymorphism.

We would also like to generalize the current work to divergent computations. For this we do not plan any changes to DM. However, we plan to extend EMF* with general recursion and a primitive Div effect (for divergence), following the current F* implementation (Swamy et al. 2016). Each monad in DM will be elaborated in two ways: first, to Pure for total correctness, as in the current paper; and second, to Div, for partial correctness. The reify operator for a partial correctness effect will produce a Div computation, not a Pure one. With the addition of Div, the dynamic semantics of EMF* will force a strict evaluation order for Div computations, rather than the non-deterministic strong reduction that we allow for Pure computations.

Along another axis, we have already mentioned our plans to investigate translations of effect handlers (§2.5). We also hope to enhance DM in other ways, e.g., relaxing the stratification of types and adding inductive types. The latter would allow us to define monads for some forms of nondeterminism and probabilities, as well as many forms of I/O, provided we can overcome the known difficulties with CPS'ing inductive types (Barthe and Uustalu 2002). Enriching DM further, one could also add dependent types, reducing the gap between it and F*, and bringing within reach examples like Ahman and Uustalu's (2013) dependently typed update monads.

Acknowledgments

We are grateful to Clément Pit-Claudel for all his help with the F* interactive mode; to Pierre-Evariste Dagand and Michael Hicks for interesting discussions; and to the anonymous reviewers for their helpful feedback. This work was, in part, supported by the European Research Council under ERC Starting Grant SECOMP (715753).

A. Appendix

In this appendix we provide proofs and auxiliary results for the theorems that appear in the body of the paper. We also show the full type system for the source language.

A.1 The Definitional Language DM

In the typing judgment, the metavariable Δ represents a set of type variables that remains fixed throughout typing. It is used to introduce top-level let-polymorphism on all CPS'd/elaborated terms. A type is well-formed in the context Δ if all of its variables are in Δ. In rigor, all judgments from here onwards are subject to that constraint, which we do not write down. A context Γ is well-formed if both (1) all of its types are well-formed according to Δ (2) no variable names are repeated. This last condition simplifies reasoning about substitution and does not limit the language in any way.

We assume that every base type in DM is also a base type in EMF* (or that there exists a mapping from them, formally), and that source constants are also present and with the same type (formally, also a mapping for constants that respects the previous one).

The typing judgment for DM is given in Figure 6. We assume that the types appearing in the rules are well-formed. For example, in the (ST-PAIR) rule, either both H and H' are in A or both are in C etc.

A.2 CPS Translation (WP Generation)

The full $\star$-translation for DM expressions is given in Figure 7. The one for types was previously defined. We define translation on environments in the following way:

$$\frac{\Delta = X_1,\ldots,X_m}{\Delta^\star = X_1 : \mathsf{Type}_0,\ldots,X_m : \mathsf{Type}_0} \qquad \frac{\Gamma = x_1 : H_1,\ldots,x_n : H_1}{\Gamma^\star = x_1 : H_1^\star,\ldots,x_n : H_1^\star}$$

One can then prove the following:

Lemma 11 (Well-typing of $\star$-translation)**.** *For any Γ, e, A and H:*

$$\begin{array}{ll} \Delta \mid \Gamma \vdash e : H\,!\,n & \Longrightarrow \Delta^\star,\Gamma^\star \vdash e^\star : H^\star \\ \Delta \mid \Gamma \vdash e : A\,!\,\tau & \Longrightarrow \Delta^\star,\Gamma^\star \vdash e^\star : (A^\star \to \mathsf{Type}_0) \to \mathsf{Type}_0 \end{array}$$

Proof. By induction on the typing derivation. □

In this lemma statement, and in those that follow, when writing $e^\star$ we refer to the translation of e using the typing derivation from the premise.

A.3 Elaboration

The definitions of $\underline{A}$ and the F relation were previously given. For elaboration, we also translate environments, in the following manner:

$$\frac{\Delta = X_1,\ldots,X_m}{\underline{\Delta} = X_1 : \mathsf{Type}_0,\ldots,X_m : \mathsf{Type}_0}$$

$$\frac{}{\underline{x : A} = x : \underline{A}} \qquad \frac{}{\underline{x : C} = x^w : C^\star, x : \mathrm{F}_C\; x^w} \qquad \frac{\Gamma = x_1 : H_1,\ldots,x_n : H_n}{\underline{\Gamma} = \underline{x_1 : H_1},\ldots,\underline{x_n : H_n}}$$

Note that for any computational variable in the context, we introduce two variables: one for its WP and one for its actual expression. The x^w variable, which is assumed to be fresh, is used only at the WP level. Also note that $\underline{\Delta} = \Delta^\star$.

For any Γ, we define the substitution s_Γ as $[x_{i_1}^w/x_{i_1},\ldots,x_{i_k}^w/x_{i_k}]$, for the computational variables $x_{i_1},\ldots,x_{i_k} \in \Gamma$.

Similarly to Lemma 11 we show that:

Lemma 12 (Well-typing of $\star$-translation — elaboration contexts)**.** *For any Γ, e, A and C we have:*

$$\begin{array}{lll} \Delta \mid \Gamma \vdash e : H\,!\,n & \Longrightarrow & \underline{\Delta},\underline{\Gamma} \vdash e^\star \mathrm{s}_\Gamma : H^\star \\ \Delta \mid \Gamma \vdash e : A\,!\,\tau & \Longrightarrow & \underline{\Delta},\underline{\Gamma} \vdash e^\star \mathrm{s}_\Gamma : (A^\star \to \mathsf{Type}_0) \to \mathsf{Type}_0 \end{array}$$

For expression elaboration we aim to show that:

$$\frac{\Delta \mid \Gamma \vdash e : A\,!\,n}{\underline{\Delta},\underline{\Gamma} \vdash \underline{e} : \underline{A}} \qquad \frac{\Delta \mid \Gamma \vdash e : C\,!\,n}{\underline{\Delta},\underline{\Gamma} \vdash \underline{e} : \mathrm{F}_C\;(e^\star \mathrm{s}_\Gamma)} \qquad \frac{\Delta \mid \Gamma \vdash e : A\,!\,\tau}{\underline{\Delta},\underline{\Gamma} \vdash \underline{e} : \mathsf{Pure}\;\underline{A}\;(e^\star \mathrm{s}_\Gamma)}$$

A.4 Proof of the Logical Relation Lemma

We shall prove some intermediate lemmas before.

Theorem 13. *For any A, $\Delta \mid \Gamma \vdash e : A\,!\,n \implies \underline{e} = e^\star\ \mathrm{s}_\Gamma$. (That is, syntactic equality).*

Proof. By induction on the typing derivation. The cases for (ST-RET) and (ST-BIND) do not apply.

(1) (ST-VAR)
Our goal is to show $x = x\ \mathrm{s}_\Gamma$. Since the type of x is A the substitution does not affect x, thus they're trivially both x.

(2) (ST-CONST)
Say $\Delta \mid \Gamma \vdash \kappa(b_1,\ldots,b_n) : b\,!\,n$. By the induction hypothesis we know that $\underline{b_i} = b_i^\star \mathrm{s}_\Gamma$ for each i. We thus trivially get our goal by substitution of the arguments.

(3) (ST-ABS)
Say we concluded $\Delta \mid \Gamma \vdash \lambda x : A.\, e : A \xrightarrow{n} A'\,!\,n$. Our premise is (note the substitution from the IH does not affect x, as it has an A-type) the fact that $\underline{e} = e^\star\ \mathrm{s}_\Gamma$. We need to show that:

$$\underline{\lambda x : A.\ e} = (\lambda x : A.\ e)^\star$$

which is just

$$\lambda x : \underline{A}.\ \underline{e} = \lambda x : A^\star.\ e^\star$$

which is trivial from our hypothesis and since $\underline{A} =_{\mathbf{def}} A^\star$.

(4) (ST-APP)
Say we concluded $\Delta \mid \Gamma \vdash f\ e : A'\,!\,n$ by the premises

$$\Delta \mid \Gamma \vdash f : A \xrightarrow{n} A'\,!\,n \qquad \Delta \mid \Gamma \vdash e : A\,!\,n$$

(it cannot be the case that e has some C type, because of the type restrictions). Using the inductive hypotheses we have:

$$\underline{f\ e} = \underline{f}\ \underline{e} = (f^\star\ \mathrm{s}_\Gamma)\ (e^\star\ \mathrm{s}_\Gamma) = (f\ e)^\star\ \mathrm{s}_\Gamma$$

As required.

(5) (ST-FST), (ST-SND), (ST-PAIR), (ST-INL), (ST-INR)
All of these are trivial by applying the IH. For (ST-PAIR) one needs to note that the restrictions will ensure that the type of the pair will be an A-type.

(6) (ST-CASE)
Say we concluded $\Delta \mid \Gamma \vdash \mathbf{case}\ e\ \mathbf{inl}\ x : A_0.\,e_1;\ \mathbf{inr}\ y : A_1.\,e_2 : A_2\,!\,n$. As inductive hypothesis we have:

$$\underline{e} = e^\star\ \mathrm{s}_\Gamma \qquad \underline{e_1} = e_1^\star\ \mathrm{s}_\Gamma \qquad \underline{e_2} = e_2^\star\ \mathrm{s}_\Gamma$$

(e_1 and e_2 are typed in Γ extended with x and y respectively, however since they are A-typed the s_Γ substitution is the same)
The goal is:

$$\begin{array}{ll} & (\mathsf{case}(\underline{e})\ x.\underline{e_1}\ y.\underline{e_2}) \\ = & (\mathsf{case}(e^\star\ \mathrm{s}_\Gamma)\ x.e_1^\star\ \mathrm{s}_\Gamma\ y.e_2^\star\ \mathrm{s}_\Gamma) \end{array}$$

We trivially get our goal from the IHs.

□

Theorem 14. *If $\Delta \mid \Gamma \vdash e : A\,!\,n$, then $\underline{\Delta},\underline{\Gamma} \vdash \underline{e} : \underline{A}$.*

Proof. By induction on the typing derivation.

(1) (ST-VAR)
We have $\Delta \mid \Gamma \vdash x : A\,!\,n$, with $x \in \Gamma$. By the translation for environments, we have $x : \underline{A}$ in Γ, so this is trivial.

$$\frac{x : H \in \Gamma}{\Delta \mid \Gamma \vdash x : H \,!\, n}\ \text{ST-VAR} \qquad \frac{\Delta \mid \Gamma \vdash e_i : b_i \,!\, n \qquad \kappa : b_1, \ldots, b_n \to b}{\Delta \mid \Gamma \vdash \kappa(e_1, \ldots, e_n) : b \,!\, n}\ \text{ST-CONST} \qquad \frac{\Delta \mid \Gamma, x : H \vdash e : H' \,!\, \varepsilon}{\Delta \mid \Gamma \vdash \lambda x : H . e : H \xrightarrow{\varepsilon} H' \,!\, n}\ \text{ST-ABS}$$

$$\frac{\Delta \mid \Gamma \vdash e : H \xrightarrow{\varepsilon} H' \,!\, n \qquad \Delta \mid \Gamma \vdash e' : H \,!\, n}{\Delta \mid \Gamma \vdash e e' : H' \,!\, \varepsilon}\ \text{ST-APP} \qquad \frac{\Delta \mid \Gamma \vdash e : H \,!\, n \qquad \Delta \mid \Gamma \vdash e' : H' \,!\, n}{\Delta \mid \Gamma \vdash (e, e') : H \times H' \,!\, n}\ \text{ST-PAIR} \qquad \frac{\Delta \mid \Gamma \vdash e : H \times H' \,!\, n}{\Delta \mid \Gamma \vdash \mathbf{fst}(e) : H \,!\, n}\ \text{ST-FST}$$

$$\frac{\Delta \mid \Gamma \vdash e : A \,!\, n}{\Delta \mid \Gamma \vdash \mathbf{inl}(e) : A + A' \,!\, n}\ \text{ST-INL} \qquad \frac{\Delta \mid \Gamma \vdash e : A + A' \,!\, n \qquad \Delta \mid \Gamma, x : A \vdash e_1 : H \,!\, \varepsilon \qquad \Delta \mid \Gamma, x : A' \vdash e_2 : H \,!\, \varepsilon}{\Delta \mid \Gamma \vdash \mathbf{case}\ e\ \mathbf{inl}\ x : A . e_1;\ \mathbf{inr}\ y : A' . e_2 : H \,!\, \varepsilon}\ \text{ST-CASE}$$

$$\frac{\Delta \mid \Gamma \vdash e : A \,!\, n}{\Delta \mid \Gamma \vdash \mathbf{return}_\tau\ e : A \,!\, \tau}\ \text{ST-RET} \qquad \frac{\Delta \mid \Gamma \vdash e : A \,!\, \tau \qquad \Delta \mid \Gamma, x : A \vdash e : A' \,!\, \tau}{\Delta \mid \Gamma \vdash \mathbf{bind}_\tau\ e\ \mathbf{to}\ x : A\ \mathbf{in}\ e' : A' \,!\, \tau}\ \text{ST-BIND}$$

Figure 6. Typing rules of DM

$$\begin{array}{lcllcl}
x^\star & = & x & K(e_1, \ldots, e_n)^\star & = & K\ e_1^\star\ \ldots\ e_n^\star \\
(f\ e)^\star & = & f^\star\ e^\star & (\lambda x : H .\ e)^\star & = & \lambda x : H^\star .\ e^\star \\
\mathbf{fst}(e)^\star & = & \mathsf{fst}\ e^\star & \mathbf{snd}(e)^\star & = & \mathsf{snd}\ e^\star \\
\mathbf{inl}(e)^\star & = & \mathsf{inl}\ e^\star & \mathbf{inr}(e)^\star & = & \mathsf{inr}\ e^\star \\
(e_1, e_2)^\star & = & (e_1^\star, e_2^\star) & (\mathbf{case}\ e_0\ \mathbf{inl}\ x : A .\ e_1;\ \mathbf{inr}\ y : A' .\ e_2)^\star & = & \mathsf{case}(e_0^\star)\ x.e_1^\star\ y.e_2^\star
\end{array}$$

$$\begin{array}{lcll}
(\mathbf{return}_\tau\ e)^\star & = & \lambda p : A^\star \to \mathsf{Type}_0 .\ p\ e^\star & (\text{when } \Delta \mid \Gamma \vdash e : A ! n) \\
(\mathbf{bind}_\tau\ e_1\ \mathbf{to}\ x\ \mathbf{in}\ e_2)^\star & = & \lambda p : A'^\star \to \mathsf{Type}_0 .\ e_1^\star\ (\lambda x : A .\ e_2^\star\ p) & (\text{when } \Delta \mid \Gamma, x : A \vdash e_2 : A' ! \tau)
\end{array}$$

Figure 7. Definition of the $\star$-translation for DM terms

(2) (ST-CONST)
For any constant $\kappa : (b_1, \ldots, b_n) \to b$ say we have $\Delta \mid \Gamma \vdash \kappa(e_1, \ldots, e_n) : b \,!\, n$ by (ST-CONST) (note that b and all the b_i are in A). This means that for every i we have as inductive hypothesis:

$$\underline{\Delta}, \underline{\Gamma} \vdash \underline{e_i} : b_i$$

Since κ is also a target constant of the same type, we thus have:

$$\underline{\Delta}, \underline{\Gamma} \vdash \kappa\ \underline{e_1}\ \ldots\ \underline{e_n} : b$$

Which is exactly our goal as $\underline{b} = b$.

(3) (ST-FST), (ST-SND), (ST-PAIR), (ST-INL), (ST-INR)
Trivial by using IH.

(4) (ST-CASE)
Say we concluded $\Delta \mid \Gamma \vdash \mathbf{case}\ e\ \mathbf{inl}\ x : A_0 . e_1;\ \mathbf{inr}\ y : A_1 . e_2 : A_2 \,!\, n$ by (ST-CASE). Our IHs give us

$$\begin{array}{l}
\underline{\Delta}, \underline{\Gamma} \qquad\quad \vdash \underline{e}\ \ : \underline{A_0} + \underline{A_1} \\
\underline{\Delta}, \underline{\Gamma}, x : \underline{A_0} \vdash \underline{e_1} : \underline{A_2} \\
\underline{\Delta}, \underline{\Gamma}, y : \underline{A_1} \vdash \underline{e_2} : \underline{A_2}
\end{array}$$

By a non-dependent application of T-CaseTot we get

$$\underline{\Delta}, \underline{\Gamma} \vdash \mathsf{case}(\underline{e})\ x.\underline{e_1}\ y.\underline{e_2} : \underline{A_2}$$

Which is our goal.

(5) (ST-ABS), (ST-APP)
Both trivial from IHs.

□

Before jumping into the logical relation lemma, we will require the following auxiliary lemma, of which we make heavy use.

Lemma 15 (Invariancy of $\mathrm{F}_C\ w$)**.** *If* $\Gamma \vDash w_1 = w_2$, *then* $\Gamma \vdash \mathrm{F}_C\ w_1 <: \mathrm{F}_C\ w_2$.

Proof. By induction on C.

(1) $C \xrightarrow{\tau} A$
We need to show that

$$\Gamma \vdash \mathrm{F}_{C \xrightarrow{\tau} A}\ w_1 <: \mathrm{F}_{C \xrightarrow{\tau} A}\ w_2$$

Which is

$$\begin{array}{l}
\Gamma \vdash x^w : C^\star \to \mathrm{F}_C\ x^w \to \mathsf{Pure}\ \underline{A}\ (w_1\ x^w) \\
\quad <: x^w : C^\star \to \mathrm{F}_C\ x^w \to \mathsf{Pure}\ \underline{A}\ (w_2\ x^w)
\end{array}$$

After two applications of (ST-PROD) (and some trivial reflexivity discharges), the required premise to show is:

$$\Gamma, x^w : C^\star, \_ : \mathrm{F}_C\ x^w \vdash \mathsf{Pure}\ \underline{A}\ (w_1\ x^w) <: \mathsf{Pure}\ \underline{A}\ (w_2\ x^w)$$

By (S-PURE) we're required to show that $\underline{A}$ is a subtype of itself (which is trivial by reflexivity of subtyping (S-CONV)) and that w_2 is stronger than w_1, which can be easily proven as they are equal.

(2) $A \xrightarrow{\tau} A$
Very similar to the previous case, but simpler.

(3) $C \xrightarrow{n} C'$
We need to show that

$$\Gamma \vdash \mathrm{F}_{C \xrightarrow{n} C'}\ w_1 <: \mathrm{F}_{C \xrightarrow{n} C'}\ w_2$$

Which is

$$\begin{array}{l}
\Gamma \vdash x^w : C^\star \to \mathrm{F}_C\ x^w \to \mathrm{F}_{C'}\ (w_1\ x^w) \\
\quad <: x^w : C^\star \to \mathrm{F}_C\ x^w \to \mathrm{F}_{C'}\ (w_2\ x^w)
\end{array}$$

After two applications of (ST-PROD) (and some trivial reflexivity discharges), the required premise to show is:

$$\Gamma, x^w : C^\star, \_ : \mathrm{F}_C\ x^w \vdash \mathrm{F}_{C'}\ (w_1\ x^w) <: \mathrm{F}_{C'}\ (w_2\ x^w)$$

As in this context we can show $w_1\ x^w = w_2\ x^w$ we apply our IH to the type C' and are done.

(4) $A \xrightarrow{n} C'$
Also very similar to the previous case, but simpler.

(5) $C \times C'$
Trivial by IHs and concluding that $\mathsf{fst}\ w_1 = \mathsf{fst}\ w_2$, and similarly for snd.

□

Proof of Theorem 6 (Logical relations lemma)

Proof. The two parts are proved by a joint structural induction.

(1) (ST-VAR)
We have $\Delta \mid \Gamma \vdash x : C\,!\,n$, with $x \in \Gamma$. By the translation for environments, we have $x^w : C^\star$ and $x : \mathrm{F}_C\ x^w$ in $\underline{\Gamma}$. Since x is covered by the substitution s_Γ, what we need to prove is $\underline{\Delta}, \underline{\Gamma} \vdash x : \mathrm{F}_C\ x^w$, which is exactly what we have in the environment.

(2) (ST-PAIR)
Suppose we proved $(e_1, e_2) : C_1 \times C_2\,!\,n$ by (ST-PAIR). We want to show: $\underline{\Delta}, \underline{\Gamma} \vdash (\underline{e_1}, \underline{e_2}) : \mathrm{F}_{C_1 \times C_2}\ ((e_1^\star, e_2^\star)\ \mathrm{s}_\Gamma)$, i.e., that (after reduction inside F):

$$\underline{\Delta}, \underline{\Gamma} \vdash (\underline{e_1}, \underline{e_2}) : \mathrm{F}_{C_1}\ (e_1^\star \mathrm{s}_\Gamma) \times \mathrm{F}_{C_2}\ (e_2^\star \mathrm{s}_\Gamma)$$

This is trivial by applying both IHs.

(3) (ST-FST), (ST-SND)
Suppose we proved $\Delta \mid \Gamma \vdash \mathbf{fst}(e) : C_1\,!\,n$ by (ST-FST). We need to then show

$$\underline{\Delta}, \underline{\Gamma} \vdash \mathsf{fst}\ \underline{e} : \mathrm{F}_{C_1}\ (\mathsf{fst}\ e^\star \mathrm{s}_\Gamma)$$

By our induction hypothesis we have $\underline{\Delta}, \underline{\Gamma} \vdash \underline{e} : \mathrm{F}_{C_1 \times C_2}\ (e^\star \mathrm{s}_\Gamma)$, which is

$$\underline{\Delta}, \underline{\Gamma} \vdash \underline{e} : \mathrm{F}_{C_1}\ (\mathsf{fst}\ e^\star \mathrm{s}_\Gamma) \times \mathrm{F}_{C_2}\ (\mathsf{snd}\ e^\star \mathrm{s}_\Gamma)$$

It is therefore easy to see that we have our goal.

(4) (ST-ABS)
There are two cases:

- $A \xrightarrow{\varepsilon} H$
  Suppose we concluded $\Delta \mid \Gamma \vdash \lambda x : A.\, e : A \xrightarrow{\varepsilon} H\,!\,n$. Then we have $\Delta \mid \Gamma, x : A \vdash e : H\,!\,\varepsilon$ and so, by the induction hypothesis, in both cases for ε we have

  $$\underline{\Delta}, \underline{\Gamma}, x : \underline{A} \vdash \underline{e} : \mathrm{G}_H^\varepsilon(e^\star\ \mathrm{s}_\Gamma)$$

  And we have to show:

  $$\underline{\Delta}, \underline{\Gamma} \vdash \lambda x : \underline{A}.\, \underline{e} : \mathrm{F}_{A \xrightarrow{n} H}\ ((\lambda x : A^\star . e^\star)\ \mathrm{s}_\Gamma)$$

  Which is

  $$\underline{\Delta}, \underline{\Gamma} \vdash \lambda x : \underline{A}.\, \underline{e} : x : \underline{A} \to \mathrm{G}_H^\varepsilon((\lambda x : A^\star . e^\star)\ \mathrm{s}_\Gamma\ x)$$

  Since the substitution does not cover x, the argument to G is just $e^\star\ \mathrm{s}_\Gamma$, thus we use our IH to conclude this easily.
- $C \xrightarrow{\varepsilon} H$
  Suppose we concluded $\Delta \mid \Gamma \vdash \lambda x : C.\, e : C \xrightarrow{\varepsilon} H\,!\,n$. Then we have $\Delta \mid \Gamma, x : C \vdash e : H\,!\,\varepsilon$ and so, by the induction hypothesis we have, in either case for ε:

  $$\underline{\Delta}, \underline{\Gamma}, x^w : C^\star, x : \mathrm{F}_C\ x^w \vdash \underline{e} : \mathrm{G}_H^\varepsilon(e^\star\ \mathrm{s}_\Gamma\ [x^w/x])$$

  And we have to show:

  $$\underline{\Delta}, \underline{\Gamma} \vdash \lambda x^w : C^\star.\, \lambda x : \mathrm{F}_C\ x^w.\, \underline{e} : \mathrm{F}_{C \xrightarrow{\varepsilon} H}\ ((\lambda x : C^\star . e^\star)\ \mathrm{s}_\Gamma)$$

  Which is

  $$\begin{array}{l} \underline{\Delta}, \underline{\Gamma} \vdash \lambda x^w : C^\star.\, \lambda x : \mathrm{F}_C\ x^w.\, \underline{e} \\ \quad : x^w : C^\star \to \mathrm{F}_C\ x^w \to \mathrm{G}_H^\varepsilon((\lambda x : C^\star . e^\star)\ \mathrm{s}_\Gamma\ x^w) \end{array}$$

  Using T-Abs twice we can conclude this via

  $$\underline{\Delta}, \underline{\Gamma}, x^w : C^\star, x : \mathrm{F}_C\ x^w \vdash \underline{e} : \mathrm{G}_H^\varepsilon((\lambda x : C^\star . e^\star)\ \mathrm{s}_\Gamma\ x^w)$$

  Since the substitution does not cover x, the argument to F reduces to $e^\star\ \mathrm{s}_\Gamma\ [x^w/x]$, thus we use our IH to conclude this easily.

(5) (ST-APP)
Again, There are two possible cases:

- $A \xrightarrow{\varepsilon} H$
  We concluded $\Delta \mid \Gamma \vdash f\,e : G\,!\,\varepsilon$. Our premises are $\Delta \mid \Gamma \vdash f : A \xrightarrow{\varepsilon} C\,!\,n$ and $\Delta \mid \Gamma \vdash e : A\,!\,n$. The IH for f is, expanding F:

  $$\underline{\Delta}, \underline{\Gamma} \vdash \underline{f} : x : \underline{A} \to \mathrm{G}_H^\varepsilon((f^\star\ \mathrm{s}_\Gamma)\ x)$$

  By T-App, and since $\underline{e} : \underline{A}$, this is just:

  $$\underline{\Delta}, \underline{\Gamma} \vdash \underline{f}\ \underline{e} : \mathrm{G}_H^\varepsilon((f^\star\ \mathrm{s}_\Gamma)\ \underline{e})$$

  Since from a previous theorem we know we have $\underline{e} = e^\star\ \mathrm{s}_\Gamma$ (syntactically), we can conclude:

  $$\underline{\Delta}, \underline{\Gamma} \vdash \underline{f}\ \underline{e} : \mathrm{G}_H^\varepsilon((f^\star\ \mathrm{s}_\Gamma)\ (e^\star\ \mathrm{s}_\Gamma))$$

  This is exactly:

  $$\underline{\Delta}, \underline{\Gamma} \vdash \underline{f\ e} : \mathrm{G}_H^\varepsilon((f\ e)^\star\ \mathrm{s}_\Gamma)$$

  which is our goal, in either the $C\,!\,n$ or the $A\,!\,\tau$ case.
- $C \xrightarrow{\varepsilon} H$

  We concluded $\Delta \mid \Gamma \vdash f\,e : H\,!\,\varepsilon$. Our premises are $\Delta \mid \Gamma \vdash f : C \xrightarrow{\varepsilon} H\,!\,n$ and $\Delta \mid \Gamma \vdash e : C\,!\,n$ The IHs are, expanding F:

  $$\begin{array}{lcl} \underline{\Delta}, \underline{\Gamma} & \vdash & \underline{f} : x^w : C^\star \to \mathrm{F}_C\ x^w \to \mathrm{G}_H^\varepsilon((f^\star\ \mathrm{s}_\Gamma)\ x^w) \\ \underline{\Delta}, \underline{\Gamma} & \vdash & \underline{e} : \mathrm{F}_C\ (e^\star\ \mathrm{s}_\Gamma) \end{array}$$

  Thus by two uses of T-App (noting that it's well typed by our IH for e), we can conclude:

  $$\underline{\Delta}, \underline{\Gamma} \vdash \underline{f}\ (e^\star\ \mathrm{s}_\Gamma)\ \underline{e} : \mathrm{G}_{C'}^\varepsilon((f^\star\ \mathrm{s}_\Gamma)\ (e^\star\ \mathrm{s}_\Gamma))$$

  This is just, syntactically:

  $$\underline{\Delta}, \underline{\Gamma} \vdash \underline{f\ e} : \mathrm{G}_{C'}^\varepsilon((f^\star\ \mathrm{s}_\Gamma)\ (e^\star\ \mathrm{s}_\Gamma))$$

  which is our goal, in either the $C\,!\,n$ or the $A\,!\,\tau$ case.

(6) (ST-CASE)
There are two cases depending on wether we eliminate into $C\,!\,n$ or $A\,!\,\tau$. Both of these cases are quite dull, and deal mostly with the typing judgment on the target. This may be skipped without hindering any of the main ideas.

- $C\,!\,n$
  Suppose that $\Delta \mid \Gamma \vdash e : A + A'\,!\,n$, $\Delta \mid \Gamma, x : A \vdash e_1 : C\,!\,n$, and $\Delta \mid \Gamma, y : A' \vdash e_2 : C\,!\,n$, so that $\Delta \mid \Gamma \vdash \textbf{case}\ e\ \textbf{inl}\ x : A.\, e_1;\ \textbf{inr}\ y : A'.\, e_2 : C\,!\,n$. We will go into detail only for e_1 as the typing and reasoning for e_2 is exactly analogous. As inductive hypothesis for e_1 we have

  $$\underline{\Delta}, \underline{\Gamma}, x : \underline{A} \vdash \underline{e_1} : \mathrm{F}_C\ (e_1^\star\ \mathrm{s}_\Gamma)$$

  We wish to show:

  $$\begin{array}{l} \underline{\Delta}, \underline{\Gamma} \vdash \mathsf{case}_{\mathrm{F}_C\ (\mathsf{case}(z)\ e_1^\star\ \mathrm{s}_\Gamma\ e_2^\star\ \mathrm{s}_\Gamma)}(\underline{e}\ \mathsf{as}\ z)\ x.\underline{e_1}\ y.\underline{e_2} \\ \qquad : \ \mathrm{F}_C\ ((\mathsf{case}(e^\star)\ x.e_1^\star\ y.e_2^\star)\ \mathrm{s}_\Gamma) \end{array}$$

  Which is

  $$\begin{array}{l} \underline{\Delta}, \underline{\Gamma} \vdash \mathsf{case}_{\mathrm{F}_C\ (\mathsf{case}(z)\ x.e_1^\star\ \mathrm{s}_\Gamma\ y.e_2^\star\ \mathrm{s}_\Gamma)}(\underline{e}\ \mathsf{as}\ z)\ x : \underline{A}.\, \underline{e_1}\ y : \underline{A'}.\, \underline{e_2} \\ \qquad : \ \mathrm{F}_C\ (\mathsf{case}(e^\star\ \mathrm{s}_\Gamma)\ x.e_1^\star\ \mathrm{s}_\Gamma\ y.e_2^\star\ \mathrm{s}_\Gamma) \end{array}$$

  Since $\underline{e} = e^\star\ \mathrm{s}_\Gamma$ we will prove this has type

  $$\mathrm{F}_C\ (\mathsf{case}(\underline{e})\ x.e_1^\star\ \mathrm{s}_\Gamma\ y.e_2^\star\ \mathrm{s}_\Gamma)$$

  By T-CaseTot, we should show:

  $$\begin{array}{ll} \underline{\Delta}, \underline{\Gamma} & \vdash \underline{e} : \underline{A} + \underline{A'} \\ \underline{\Delta}, \underline{\Gamma}, x : \underline{A} & \vdash \underline{e_1} : \mathrm{F}_C\ (\mathsf{case}(\mathsf{inl}\ x)\ x.e_1^\star\ \mathrm{s}_\Gamma\ y.e_2^\star\ \mathrm{s}_\Gamma) \end{array}$$

  (And the one for e_2). We get the first one trivially by theorem 14. The second is by reduction equivalent to:

  $$\underline{\Delta}, \underline{\Gamma}, x : \underline{A} \vdash \underline{e_1} : \mathrm{F}_C\ (e_1^\star\ \mathrm{s}_\Gamma)$$

Which is exactly our IH for e_1, so we're done.

- $A \,!\, \tau$

  Our hypotheses are:

$$\begin{array}{ll} \underline{\Delta},\underline{\Gamma} & \vdash \underline{e} : \underline{A_0} + \underline{A_1} \\ \underline{\Delta},\underline{\Gamma}, x : \underline{A_0} & \vdash \underline{e_1} : \mathsf{Pure}\ \underline{A_2}\ (e_1^\star\ \mathrm{s}_\Gamma) \\ \underline{\Delta},\underline{\Gamma}, y : \underline{A_1} & \vdash \underline{e_2} : \mathsf{Pure}\ \underline{A_2}\ (e_2^\star\ \mathrm{s}_\Gamma) \end{array}$$

  Applying T-Case (non-dependently) we get.

$$\begin{array}{l} \underline{\Delta},\underline{\Gamma} \vdash \mathsf{case}(\underline{e})\ x.\underline{e_1}\ y.\underline{e_2} : \\ \qquad \mathsf{Pure}\ \underline{A_2}\ (\overline{\mathsf{case}(\underline{e})\ x.e_1^\star\ \mathrm{s}_\Gamma\ y.e_2^\star\ \mathrm{s}_\Gamma}) \end{array}$$

  Since we know $\underline{e} = e^\star\ \mathrm{s}_\Gamma$ and since $\underline{A} =_{\mathbf{def}} A^\star$ this is exactly:

$$\begin{array}{l} \underline{\Delta},\underline{\Gamma} \vdash \mathsf{case}(\underline{e})\ x.\underline{e_1}\ y.\underline{e_2} : \\ \qquad \mathsf{Pure}\ \underline{A_2}\ ((\overline{\mathsf{case}(e^\star)\ x.e_1^\star\ y.e_2^\star})\ \mathrm{s}_\Gamma) \end{array}$$

  Which is our goal.

(7) (ST-RET)
We have $\Delta \mid \Gamma \vdash e : A \,!\, n$. We need to show:

$$\underline{\Delta},\underline{\Gamma} \vdash \underline{\mathbf{return}_\tau\ e} : \mathsf{Pure}\ \underline{A}\ (\mathbf{return}_\tau\ e^\star\ \mathrm{s}_\Gamma)$$

i.e.

$$\underline{\Delta},\underline{\Gamma} \vdash \mathsf{Pure.return}\ \underline{A}\ \underline{e} : \mathsf{Pure}\ \underline{A}\ (\lambda p : A^\star \to \mathsf{Type}_0.\ (e^\star\ \mathrm{s}_\Gamma))$$

This is a trivial consequence of Theorem 13 by using the T-Ret rule of EMF*, and the fact that $\underline{A} =_{\mathbf{def}} A^\star$.

(8) (ST-BIND)
Suppose we have $\Delta \mid \Gamma \vdash e_1 : A \,!\, \tau$, and $\Delta \mid \Gamma, x : A \vdash e_2 : A' \,!\, \tau$, and so $\Delta \mid \Gamma \vdash \mathbf{bind}_\tau\ e_1\ \mathbf{to}\ x : A\ \mathbf{in}\ e_2 : A' \,!\, \tau$. We have to show:

$$\underline{\Delta},\underline{\Gamma} \vdash \underline{\mathbf{bind}_\tau\ e_1\ \mathbf{to}\ x : A\ \mathbf{in}\ e_2} : \mathsf{Pure}\ \underline{A'}\ ((\mathbf{bind}_\tau\ e_1\ \mathbf{to}\ x : A\ \mathbf{in}\ e_2)^\star\ \mathrm{s}_\Gamma)$$

that is:

$$\begin{array}{l} \underline{\Delta},\underline{\Gamma} \vdash \mathsf{Pure.bind}\ \underline{A}\ \underline{A'}\ (e_1^\star\ \mathrm{s}_\Gamma)\ \underline{e_1}\ (\lambda x : A^\star.\, e_2^\star\ \mathrm{s}_\Gamma)\ (\lambda x : \underline{A}.\, \underline{e_2}) : \\ \qquad \mathsf{Pure}\ \underline{A'}\ (\lambda p : A'^\star \to Ty.\ e_1^\star(\lambda x : A^\star.\, e_2^\star\ p)) \end{array}$$

By our IHs we have:

$$\begin{array}{ll} \underline{\Delta},\underline{\Gamma} & \vdash \underline{e_1} : \mathsf{Pure}\ \underline{A}\ (e_1^\star\ \mathrm{s}_\Gamma) \\ \underline{\Delta},\underline{\Gamma}, x : \underline{A} & \vdash \underline{e_2} : \mathsf{Pure}\ \underline{A'}\ (e_2^\star\ \mathrm{s}_\Gamma) \end{array}$$

So we get:

$$\underline{\Delta},\underline{\Gamma} \vdash \lambda x : \underline{A}.\, \underline{e_2} : x : \underline{A} \to \mathsf{Pure}\ \underline{A'}\ (e_2^\star\ \mathrm{s}_\Gamma)$$

By the T-Bind rule we can conclude:

$$\begin{array}{l} \underline{\Delta},\underline{\Gamma} \vdash \mathsf{Pure.bind}\ \underline{A}\ \underline{A'}\ (e_1^\star\ \mathrm{s}_\Gamma)\ \underline{e_1}\ (\lambda x : A^\star.\, e_2^\star\ \mathrm{s}_\Gamma)\ (\lambda x : \underline{A}.\, \underline{e_2}) : \\ \qquad \mathsf{Pure}\ \underline{A'}\ (\lambda p : \underline{A'} \to Ty.\ e_1^\star(\lambda x : \underline{A}.\, e_2^\star\ p)) \end{array}$$

Since $\underline{A} = A^\star$ and $\underline{A'} = A'^\star$ this is exactly our goal.

□

Note: in this proof, we didn't use any specific fact about Pure, except the relation between monad operations and their WPs, so this is all generalizable to another target monad that's already defined and satisfies the base conditions for return and bind.

A.5 Equality Preservation

We want to show that any source monad will give rise to specification-level monads in the target. This will be a consequence on the fact that equality is preserved by the $\star$-translation. From equality preservation we also get the property that lifts are monad morphisms without further effort.

First we define equality for the source language. It is basically standard $\beta\eta$-equivalence adding the monad laws for the base monad T. We keep the type and effect of each equality. It is an invariant that if we can derive an equality, both sides are well-typed at the specified type and effect.

The definition of the equality judgment is in Figure 8. Besides those rules, there is a congruence rule for every source construct and computation rules for pairs and sums, as expected.

We then prove that:

$$\frac{\Delta \mid \Gamma \vdash e_1 = e_2 : H \,!\, \varepsilon}{\underline{\Delta},\underline{\Gamma} \vDash e_1^\star\ \mathrm{s}_\Gamma == e_2^\star\ \mathrm{s}_\Gamma}$$

Where by $\vDash$ it is meant the validity judgment of EMF*.

Theorem 16 (Preservation of equality by CPS). *If $\Delta \mid \Gamma \vdash e_1 = e_2 : H \,!\, \varepsilon$ for any $\Delta, \Gamma, e_1, e_2, H, \varepsilon$, then one has $\underline{\Delta},\underline{\Gamma} \vDash e_1^\star\ \mathrm{s}_\Gamma == e_2^\star\ \mathrm{s}_\Gamma$.*

Proof. By induction on the equality derivation. Most of the cases are trivial, since EMF* has very similar rules for equality. The interesting cases are the monadic equalities, which we show here:

(1) (EQ-M1)
We concluded

$$\Delta \mid \Gamma \vdash \mathbf{bind}_\tau\ m\ \mathbf{to}\ x\ \mathbf{in}\ (\mathbf{return}_\tau\ x) = m : A \,!\, \tau$$

Thus we need to show that

$$\underline{\Delta},\underline{\Gamma} \vDash (\mathbf{bind}_\tau\ m\ \mathbf{to}\ x\ \mathbf{in}\ (\mathbf{return}_\tau\ x))^\star\ \mathrm{s}_\Gamma == m^\star\ \mathrm{s}_\Gamma$$

That is:

$$\underline{\Delta},\underline{\Gamma} \vDash (\lambda p.\ (m^\star\ \mathrm{s}_\Gamma)(\lambda x.\ (\lambda p'.\ p'\ x)\ p)) == m^\star\ \mathrm{s}_\Gamma$$

This is trivially provable by $\beta\eta$-reduction.

(2) (EQ-M2)
We concluded

$$\Delta \mid \Gamma \vdash \mathbf{bind}_\tau\ (\mathbf{return}_\tau\ e)\ \mathbf{to}\ x\ \mathbf{in}\ f\ x = f\ e : A' \,!\, \tau$$

thus we need to show that

$$\underline{\Delta},\underline{\Gamma} \vDash (\mathbf{bind}_\tau\ (\mathbf{return}_\tau\ e)\ \mathbf{to}\ x\ \mathbf{in}\ f\ x)^\star\ \mathrm{s}_\Gamma == (f\ e)^\star\ \mathrm{s}_\Gamma$$

That is:

$$\underline{\Delta},\underline{\Gamma} \vDash (\lambda p.\ (\lambda p'.\ p'\ e^\star)\ (\lambda x.\ f^\star\ x\ p))\ \mathrm{s}_\Gamma = (f^\star\ \mathrm{s}_\Gamma)(e^\star\ \mathrm{s}_\Gamma)$$

Note that since $x \notin FV(f) \implies x \notin FV(f^\star\ \mathrm{s}_\Gamma)$, this is easily shown by $\beta\eta$-reduction as well.

(3) (EQ-M3)
We concluded

$$\begin{array}{l} \Delta \mid \Gamma \vdash \mathbf{bind}_\tau\ (\mathbf{bind}_\tau\ m\ \mathbf{to}\ x\ \mathbf{in}\ e_1)\ \mathbf{to}\ y\ \mathbf{in}\ e_2 \\ \qquad = \mathbf{bind}_\tau\ m\ \mathbf{to}\ x\ \mathbf{in}\ (\mathbf{bind}_\tau\ e_1\ \mathbf{to}\ y\ \mathbf{in}\ e_2) : A'' \,!\, \tau \end{array}$$

thus we need to show that

$$\begin{array}{l} \underline{\Delta},\underline{\Gamma} \vDash (\mathbf{bind}_\tau\ (\mathbf{bind}_\tau\ m\ \mathbf{to}\ x\ \mathbf{in}\ e_1)\ \mathbf{to}\ y\ \mathbf{in}\ e_2)^\star\ \mathrm{s}_\Gamma \\ \qquad == (\mathbf{bind}_\tau\ m\ \mathbf{to}\ x\ \mathbf{in}\ (\mathbf{bind}_\tau\ e_1\ \mathbf{to}\ y\ \mathbf{in}\ e_2))^\star\ \mathrm{s}_\Gamma \end{array}$$

That is:

$$\begin{array}{l} \underline{\Delta},\underline{\Gamma} \vDash (\lambda p.\ (\lambda p'.\ (m^\star\ \mathrm{s}_\Gamma)\ (\lambda x.\ (e_1^\star\ \mathrm{s}_\Gamma) p'))\ (\lambda y.\ (e_2^\star\ \mathrm{s}_\Gamma)\ p)) \\ \qquad == (\lambda p.\ (m^\star\ \mathrm{s}_\Gamma)\ (\lambda x.\ (\lambda p'.\ (e_1^\star\ \mathrm{s}_\Gamma)\ (\lambda y.\ (e_2^\star\ \mathrm{s}_\Gamma)\ p'))\ p)) \end{array}$$

Note that since $x \notin FV(e_2) \implies x \notin FV(e_2^\star)$, this is also easily shown by $\beta\eta$-reduction.

The proof above is easy, and that should not be surprising, as we are translating our abstract monadic operations into a concrete monad (continuations), thus our source equalities should be trivially satisfied after translation.

□

A.6 Monotonicity

We're interested in the monotonicity of WPs. Firstly, we need a higher-order definition for this property. Throughout this section we mostly ignore DM's typing restrictions and work with a larger source language. This gives us a stronger result than strictly necessary.

The types where the translation is defined are those non-dependent and monad-free (meaning every arrow in them is a Tot-arrow). No ocurrence of Pure is allowed. The types of specifications are always of this shape, so this is not a limitation.

For non-empty environments the theorem states:

$$\textsc{Eq-Beta}\quad \frac{\Delta \mid \Gamma, x : H \vdash e_1 : H' \,!\, \varepsilon \qquad \Delta \mid \Gamma \vdash e_2 : H \,!\, n}{\Delta \mid \Gamma \vdash (\lambda x : H.\, e_1) e_2 = e_1[e_2/x] : H' \,!\, \varepsilon}$$

$$\textsc{Eq-Eta}\quad \frac{\Delta \mid \Gamma \vdash e : H \xrightarrow{\varepsilon} H' \,!\, n \qquad x \notin FV(e)}{\Delta \mid \Gamma \vdash (\lambda x : H.\, e\, x) = e : H \xrightarrow{\varepsilon} H' \,!\, n}$$

$$\textsc{Eq-App}\quad \frac{\Delta \mid \Gamma \vdash e_1 = e_1' : H \xrightarrow{\varepsilon} H' \,!\, n \qquad \Delta \mid \Gamma \vdash e_2 = e_2' : H \,!\, n}{\Delta \mid \Gamma \vdash e_1\ e_2 = e_1'\ e_2' : H' \,!\, \varepsilon}$$

$$\textsc{Eq-Abs}\quad \frac{\Delta \mid \Gamma, x : H \vdash e = e' : H' \,!\, \varepsilon}{\Delta \mid \Gamma \vdash (\lambda x : H.\, e) = (\lambda x : H.\, e') : H \xrightarrow{\varepsilon} H' \,!\, n}$$

$$\textsc{Eq-Refl}\quad \frac{\Delta \mid \Gamma \vdash e : H \,!\, \varepsilon}{\Delta \mid \Gamma \vdash e = e : H \,!\, \varepsilon} \qquad \textsc{Eq-Symm}\quad \frac{\Delta \mid \Gamma \vdash e_1 = e_2 : H \,!\, \varepsilon}{\Delta \mid \Gamma \vdash e_2 = e_1 : H \,!\, \varepsilon}$$

$$\textsc{Eq-Trans}\quad \frac{\Delta \mid \Gamma \vdash e_1 = e_2 : H \,!\, \varepsilon \qquad \Delta \mid \Gamma \vdash e_2 = e_3 : H \,!\, \varepsilon}{\Delta \mid \Gamma \vdash e_1 = e_3 : H \,!\, \varepsilon}$$

$$\textsc{Eq-Pair}\quad \frac{\Delta \mid \Gamma \vdash e : H \times H' \,!\, n}{\Delta \mid \Gamma \vdash (\mathbf{fst}(e), \mathbf{snd}(e)) = e : H \times H' \,!\, n}$$

$$\textsc{Eq-Case}\quad \frac{\Delta \mid \Gamma \vdash e : A + A' \,!\, n}{\Delta \mid \Gamma \vdash \mathbf{case}\ e\ \mathbf{inl}\ x.\mathbf{inl}(x);\ \mathbf{inr}\ x.\mathbf{inr}(x) = e : A + A' \,!\, n}$$

$$\textsc{Eq-M1}\quad \frac{\Delta \mid \Gamma \vdash m : A \,!\, \tau}{\Delta \mid \Gamma \vdash \mathbf{bind}_\tau\ m\ \mathbf{to}\ x\ \mathbf{in}\ (\mathbf{return}_\tau\ x) = m : A \,!\, \tau}$$

$$\textsc{Eq-M2}\quad \frac{\Delta \mid \Gamma \vdash e : A \,!\, n \qquad \Delta \mid \Gamma \vdash f : A \xrightarrow{\tau} A' \,!\, n \qquad x \notin FV(f)}{\Delta \mid \Gamma \vdash \mathbf{bind}_\tau\ (\mathbf{return}_\tau\ e)\ \mathbf{to}\ x\ \mathbf{in}\ f\ x = f\ e : A' \,!\, \tau}$$

$$\textsc{Eq-M3}\quad \frac{\Delta \mid \Gamma \vdash m : A \,!\, \tau \qquad \Delta \mid \Gamma, x : A \vdash e_1 : A' \,!\, \tau \qquad \Delta \mid \Gamma, y : A' \vdash e_2 : A'' \,!\, \tau \qquad x \notin FV(e_2)}{\Delta \mid \Gamma \vdash \mathbf{bind}_\tau\ (\mathbf{bind}_\tau\ m\ \mathbf{to}\ x\ \mathbf{in}\ e_1)\ \mathbf{to}\ y\ \mathbf{in}\ e_2 = \mathbf{bind}_\tau\ m\ \mathbf{to}\ x\ \mathbf{in}\ (\mathbf{bind}_\tau\ e_1\ \mathbf{to}\ y\ \mathbf{in}\ e_2) : A'' \,!\, \tau}$$

Figure 8. Equality rules for DM

Theorem 17 (Monotonicity of $\star$-translation— environments)**.** *For any Δ, Γ, e, H, A one has:*

1. $\Delta \mid \Gamma \vdash e : H \,!\, n \quad \Longrightarrow \quad \underline{\Delta}, \Gamma^{12} \vDash e^{\star 1} \precsim_{H^\star} e^{\star 2}$
2. $\Delta \mid \Gamma \vdash e : A \,!\, \tau \quad \Longrightarrow \quad \underline{\Delta}, \Gamma^{12} \vDash e^{\star 1} \precsim_{(A^\star \to \mathsf{Type}_0) \to \mathsf{Type}_0} e^{\star 2}$

Where we define

$$\cdot^{12} = \cdot$$
$$(\Gamma, x : t)^{12} = \Gamma^{12}, x^1 : t^\star, x_1^2 : t^\star, [x^1 \precsim_{t^\star} x^1 \wedge x^1 \precsim_{t^\star} x^2 \wedge x^2 \precsim_{t^\star} x^2]$$

which essentially duplicates each variable and asserts both monotonicity for each of them and their ordering ($[\phi]$ is notation for $h : \phi$, where h does not appear free anywhere). We then define the $-^1$ substitution as $[x_1^1/x_1, \ldots, x_n^1/x_n]$ and similarly for $-^2$. This trivially implies the previous monotonicity theorem.

Before jumping into the proof, we define and prove the following lemma:

Lemma 18. *For any ϕ, $\Gamma^{12} \vDash \phi$ implies $\Gamma^{12} \vDash \phi[2 \to 1]$. Where $[2 \to 1]$ is the substitution mapping x^2 to x^1 for every x in Γ. Analogously $\Gamma^{12} \vDash \phi[1 \to 2]$.*

Proof. By induction on Γ. This is trivial for an empty gamma. For $\Gamma' = \Gamma, x : t$, assume $(\Gamma, x : t)^{12} \vDash \phi$ holds. By applying (V-$\forall$I) three times, we get:

$$\Gamma^{12} \vDash \forall x^1, x^2. [x^1 \precsim x^1 \wedge x^1 \precsim x^2 \wedge x^2 \precsim x^2] \implies \phi$$

By our IH we get:

$$\Gamma^{12} \vDash \forall x^1, x^2. [x^1 \precsim x^1 \wedge x^1 \precsim x^2 \wedge x^2 \precsim x^2] \implies \phi[2 \to 1]$$

By weakening (note that x^1 and x^2 are not free in the RHS) we get that:

$$(\Gamma, x : t)^{12} \vDash \forall x^1, x^2. [x^1 \precsim x^1 \wedge x^1 \precsim x^2 \wedge x^2 \precsim x^2] \implies \phi[2 \to 1]$$

We can instantiate this (via (V-$\forall$E)) with x^1 on both variables to get:

$$(\Gamma, x : t)^{12} \vDash [x^1 \precsim x^1 \wedge x^1 \precsim x^1 \wedge x^1 \precsim x^1] \implies \phi[2 \to 1][x^1/x^2]$$

Furthermore, it is trivial to show this antecedent in the context $(\Gamma, x : t)^{12}$ and so we get our goal of:

$$(\Gamma, x : t)^{12} \vDash \phi[2 \to 1][x^1/x^2]$$

$\square$

Lemma 19. *For any e, if $\Gamma^{12} \vDash e^{\star 1} \precsim e^{\star 2}$, then $\Gamma^{12} \vDash e^{\star 1} \precsim e^{\star 1} \wedge e^{\star 1} \precsim e^{\star 2} \wedge e^{\star 2} \precsim e^{\star 2}$.*

Proof. Trivial from previous lemma by noting that $e^{\star 1}[1 \to 2] = e^{\star 2}$ and likewise for $[2 \to 1]$, and then using (V-ANDINTRO). $\square$

Proof of Theorem 17 (Monotonicity of $\star$-translation— environments)

Proof. We prove these two propositions by induction on the typing derivation for e. Throughout the proof Δ plays no special role, so we just drop it from the reasoning, keeping in mind that it has to be there for having well-formed types (but nothing else).

Note that during the proof we treat $\precsim_X$ abstractly, so any instantiation with a proper type (not necessarily those where $\precsim$ reduces to equality) would be OK.

Throughout this proof we sometimes skip the subindices for $\precsim$ in favor of compactness. Hopefully, they should be clear from the context.

(1) (ST-VAR)
We need to show $\Gamma^{12} \vDash x_i^1 \precsim_{t_i^\star} x_i^2$. This is trivial from the context and by using the (V-ASSUME) and (V-ANDELIMi) rules.

(2) (ST-CONST)
The constants only deal with base types, so all inductive hypotheses for the arguments reduce to an equality, as does our goal. Our goal is is then trivially provable by applications of (V-EQP).

(3) (ST-ABS)
Say we concluded $\Gamma, x : t \vdash e : s \,!\, \varepsilon$ As IH we have:

$$(\Gamma, x : t)^{12} \vDash e^{\star 1}[x^1/x] \precsim_{s'} e^{\star 2}[x^2/x]$$

Where s' is either $s^\star$ or $(s^\star \to \mathsf{Type}_0) \to \mathsf{Type}_0$ depending on ε. The proof is independent of this. What we need to prove is:

$$\Gamma^{12} \vDash (\lambda x : t^\star . e^{\star 1}) \lesssim_{t^\star \to s'} (\lambda x : t^\star . e^{\star 2})$$

Which by definition is:

$$\Gamma^{12} \vDash \forall x^1, x^2 : t^\star . x^1 \lesssim_{t^\star} x^1 \wedge x^1 \lesssim_{t^\star} x^2 \wedge x^2 \lesssim_{t^\star} x^2 \implies (\lambda x : t^\star . e^{\star 1})\, x^1 \lesssim_{s'} (\lambda x : t^\star . e^{\star 2})\, x^2$$

By reduction ((V-EqRed) + (V-Eq*)), this is equivalent to:

$$\Gamma^{12} \vDash \forall x^1, x^2 : t^\star . x^1 \lesssim_{t^\star} x^1 \wedge x^1 \lesssim_{t^\star} x^2 \wedge x^2 \lesssim_{t^\star} x^2 \implies e^{\star 1}[x^1/x] \lesssim_{s'} e^{\star 2}[x^2/x]$$

Which we can conclude from three applications of (V-∀I), and our IH.

(4) (ST-App)
Say $\Gamma \vdash f : t \xrightarrow{\varepsilon} s \,!\, n$ and $\Gamma \vdash e : a \,!\, n$. As inductive hypothesis we get:

$$\Gamma^{12} \vDash f^{\star 1} \lesssim_{t^\star \to s'} f^{\star 2} \qquad \Gamma^{12} \vDash e^{\star 1} \lesssim_{t^\star} e^{\star 2}$$

Where s' is either $s^\star$ or $(s^\star \to \mathsf{Type}_0) \to \mathsf{Type}_0$ depending on ε. Again, the proof is independent of this. Applying Lemma 19 for our second IH we get that:

$$\Gamma^{12} \vDash e^{\star 1} \lesssim_{t^\star} e^{\star 1} \wedge e^{\star 1} \lesssim_{t^\star} e^{\star 2} \wedge e^{\star 2} \lesssim_{t^\star} e^{\star 2}$$

Expanding the definition of $\lesssim$ on the IH for f we get:

$$\Gamma^{12} \vDash \forall x^1, x^2 : t^\star . x^1 \lesssim_{t^\star} x^1 \wedge x^1 \lesssim_{t^\star} x^2 \wedge x^2 \lesssim_{t^\star} x^2 \implies f^{*1}\, x^1 \lesssim_{s'} f^{*2}\, x^2$$

We instantiate (using (V-∀E)) x^1, x^2 with $e^{\star 1}, e^{\star 2}$, and apply (V-MP) with our proof about $e^{\star 1}$ and $e^{\star 2}$ to get:

$$\Gamma^{12} \vDash f^{*1}\, e^{\star 1} \lesssim_{s'} f^{*2}\, e^{\star 2}$$

Which is exactly our goal in any $(\xrightarrow{n}, \xrightarrow{\tau})$ case.

(5) (ST-Ret)
Say $\Gamma \vdash \mathbf{return}_\tau\, e : t \,!\, \tau$. Our IH gives us:

$$\Gamma^{12} \vDash e^{\star 1} \lesssim_{t^\star} e^{\star 2}$$

And we need to show that:

$$\Gamma^{12} \vDash (\lambda p.\, p\, e^{\star 1}) \lesssim_{(t^\star \to \mathsf{Type}_0) \to \mathsf{Type}_0} (\lambda p.\, p\, e^{\star 2})$$

That is:

$$\Gamma^{12} \vDash \forall p^1, p^2 .\, p^1 \lesssim p^1 \wedge p^1 \lesssim p^2 \wedge p^2 \lesssim p^2 \implies (\lambda p.\, p\, e^{\star 1})\, p_1 \lesssim_{s'} (\lambda p.\, p\, e^{\star 2})\, p_2$$

By reduction this is:

$$\Gamma^{12} \vDash \forall p^1, p^2 .\, p^1 \lesssim p^1 \wedge p^1 \lesssim p^2 \wedge p^2 \lesssim p^2 \implies p_1\, e^{\star 1} \lesssim_{s'} p_2\, e^{\star 2}$$

By applying Lemma 19 to e we get:

$$\Gamma^{12} \vDash e^{\star 1} \lesssim_{t^\star} e^{\star 1} \wedge e^{\star 1} \lesssim_{t^\star} e^{\star 2} \wedge e^{\star 2} \lesssim_{t^\star} e^{\star 2}$$

With this, we can easily prove our goal by applying (V-∀I), and then the assumption of $p_1 \lesssim p_2$ applied to this last proof.

(6) (ST-Bind)
Say $\Gamma \vdash m : a \,!\, \tau$ and $\Gamma, x : a \vdash e : b \,!\, \tau$, so we get $\Gamma \vdash \mathbf{bind}_\tau\, m\, \mathbf{to}\, x\, \mathbf{in}\, e : b \,!\, \tau$. Our IHs are:

$$\Gamma^{12} \vDash m^{\star 1} \lesssim_{(a \to \mathsf{Type}_0) \to \mathsf{Type}_0} m^{\star 2}$$
$$(\Gamma, x : a)^{12} \vDash e^{\star 1}\, [x^1/x] \lesssim_{(b \to \mathsf{Type}_0) \to \mathsf{Type}_0} e^{\star 2}\, [x^2/x]$$

We need to show that:

$$\Gamma^{12} \vDash (\lambda p.\, m^{\star 1}\, (\lambda x.\, e^{\star 1}\, p)) \lesssim_{(b^\star \to \mathsf{Type}_0) \to \mathsf{Type}_0} (\lambda p.\, m^{\star 2}\, (\lambda x.\, e^{\star 2}\, p))$$

Which by expanding the definition, applying (V-∀I), and reducing can be simplified to:

$$\Gamma^{12}, p^1, p^2, [p^1 \lesssim p^1 \wedge p^1 \lesssim p^2 \wedge p^2 \lesssim p^2] \vDash m^{\star 1}\, (\lambda x.\, e^{\star 1}\, p^1) \lesssim m^{\star 2}\, (\lambda x.\, e^{\star 2}\, p^2)$$

By our IH we know that $m^{\star 1} \lesssim m^{\star 2}$, so it would be enough to show that:

$$\Gamma^{12}, p^1, p^2, [p^1 \lesssim p^1 \wedge p^1 \lesssim p^2 \wedge p^2 \lesssim p^2] \vDash (\lambda x.\, e^{\star 1}\, p^1) \lesssim (\lambda x.\, e^{\star 2}\, p^2)$$

and then use Lemma 19. Expanding the definitions, applying (V-∀I) and reducing this can be shown by:

$$\Gamma^{12}, p^1, p^2, [p^1 \lesssim p^1 \wedge p^1 \lesssim p^2 \wedge p^2 \lesssim p^2] \; x^1, x^2, [x^1 \lesssim x^1 \wedge x^1 \lesssim x^2 \wedge x^2 \lesssim x^2] \vDash e^{\star 1}[x^1/x]\, p^1 \lesssim e^{\star 2}[x^2/x]\, p^2$$

Because of our assumptions for p^1 and p^2, this can be shown by proving $e^{\star 1}[x^1/x] \lesssim e^{\star 2}[x^2/x]$. This is trivial by our IH for e, weakening it into this extended environment that includes p^i.

(7) (ST-Pair), (ST-Fst), (ST-Inl)
All trivial from IHs.

(8) (ST-Case)
By case analysis on the IH for the sum type, and reduction.

□

Having this proof implies that any well-typed term will be given a monotonic specification. And, as a consequence, functions preserve monotonicity.

A.7 Conjunctivity

The definition of conjunctivity on EMF* predicate types was given previously. The full theorem which we prove is this:

Theorem 20 (Conjunctivity of $\star$-translation— environments)**.** *For any Δ, Γ, e, H, A one has:*

1. $\Delta \mid \Gamma \vdash e : C \,!\, n \implies \underline{\Delta}, \Gamma_{\mathbb{C}} \vDash \mathbb{C}_{C^\star}(e^\star)$
2. $\Delta \mid \Gamma \vdash e : A \,!\, \tau \implies \underline{\Delta}, \Gamma_{\mathbb{C}} \vDash \mathbb{C}_{(A^\star \to \mathsf{Type}_0) \to \mathsf{Type}_0}(e^\star)$

Where when $\Gamma = x_1 : t_1, \ldots$, we define $\Gamma_{\mathbb{C}} = x_1 : t_1^\star, [\mathbb{C}_{t_1^\star}(x_1)], \ldots$. This trivially implies the previously stated theorem by taking $\Gamma = \cdot$.

Proof. By induction on the typing derivations. Once again, Δ does not play a big role and we omit it.

(1) (ST-Var)
Trivial from context, for any type.

(2) (ST-Const)
Does not apply as no constant gives a type $C \,!\, n$ nor $A \,!\, \tau$

(3) (ST-Abs)
Say we concluded $\Gamma, x : t \vdash e : s \,!\, \varepsilon$ (where that might be $C \,!\, n$ or $A \,!\, \tau$, we treat both cases uniformly). From the IH we get

$$\Gamma_{\mathbb{C}}, x : t^\star, [\mathbb{C}_{t^\star}(x)] \vDash \mathbb{C}_{s'}(e)$$

Where s' is $s^\star$ or $(s^\star \to \mathsf{Type}_0) \to \mathsf{Type}_0$ according to (s, ε). By applying (V-∀I) twice we get:

$$\Gamma_{\mathbb{C}} \vDash \forall x : t^\star . \mathbb{C}_{t^\star}(x) \Rightarrow \mathbb{C}_{s'}(e)$$

Which is the same, by reduction, as:

$$\Gamma_{\mathbb{C}} \vDash \forall x : t^\star . \mathbb{C}_{t^\star}(x) \Rightarrow \mathbb{C}_{s'}((\lambda x.\, e\, x)\, x)$$

Thus by definition of $\mathbb{C}$:

$$\Gamma_{\mathbb{C}} \vDash \mathbb{C}_{t \to s'}(\lambda x.\, e\, x)$$

As required for both cases.

(4) (ST-App)
Trivial by the preservation of $\mathbb{C}$ by application, in both cases (applies (V-MP)).

(5) (ST-RET)
Say we concluded $\Gamma \vdash \textbf{return}_\tau\ e : A \,!\, \tau$. Our goal is then:

$$\Gamma_{\mathbb{C}} \models \mathbb{C}_{(A^\star \to \mathsf{Type}_0) \to \mathsf{Type}_0}(\lambda p.\ p\ e^\star)$$

Which is:

$$\begin{aligned}\Gamma_{\mathbb{C}} \models \forall p_1, p_2.\ (\lambda p.\ p\ e^\star) p_1 \wedge (\lambda p.\ p\ e^\star) p_2 \\ = (\lambda p.\ p\ e^\star)(\lambda x. p_1\ x \wedge p_2\ x)\end{aligned}$$

By reduction that's equivalent to:

$$\Gamma_{\mathbb{C}} \models \forall p_1, p_2.\ p_1\ e^\star \wedge p_2\ e^\star = p_1\ e^\star \wedge p_2\ e^\star$$

Which is trivially true (without use of any IH) by (V-REFL).

(6) (ST-BIND)
Say we concluded $\Gamma \vdash \textbf{bind}_\tau\ e_1\ \textbf{to}\ x\ \textbf{in}\ e_2 : A' \,!\, \tau$, where $e_1 : A \,!\, \tau$. Our IHs are:

$$\begin{aligned}\Gamma_{\mathbb{C}} \models \mathbb{C}_{(A^\star \to \mathsf{Type}_0) \to \mathsf{Type}_0}(e_1^\star) \\ \Gamma_{\mathbb{C}}, x : A^\star, [\mathbb{C}_{A^\star}(x)] \models \mathbb{C}_{(A'^\star \to \mathsf{Type}_0) \to \mathsf{Type}_0}(e_2^\star)\end{aligned}$$

We need to show:

$$\Gamma_{\mathbb{C}} \models \mathbb{C}_{(A'^\star \to \mathsf{Type}_0) \to \mathsf{Type}_0}(\lambda p. e_1^\star(\lambda x. e_2^\star\ p))$$

Expanding the definition, this is:

$$\begin{aligned}\Gamma_{\mathbb{C}} \models \forall p_1, p_2.\ (\lambda p. e_1^\star(\lambda x. e_2^\star\ p))\ p_1 \wedge (\lambda p. e_1^\star(\lambda x. e_2^\star\ p))\ p_2 \\ = (\lambda p. e_1^\star(\lambda x. e_2^\star\ p))(\lambda x. p_1\ x \wedge p_2\ x)\end{aligned}$$

By reduction, this is equivalent to:

$$\begin{aligned}\Gamma_{\mathbb{C}} \models \forall p_1, p_2.\ e_1^\star(\lambda x. e_2^\star\ p_1) \wedge e_1^\star(\lambda x. e_2^\star\ p_2) \\ = e_1^\star(\lambda x. e_2^\star\ (\lambda x. p_1\ x \wedge p_2\ x))\end{aligned}$$

By the IH for e_2 we know $\forall x.\ e_2^\star\ (\lambda x. p_1\ x \wedge p_2\ x) = e_2^\star p_1 \wedge e_2^\star p_2$. By reduction and (V-EXT) this means $(\lambda x. e_2^\star\ (\lambda x. p_1\ x \wedge p_2\ x)) = (\lambda x. e_2^\star p_1 \wedge e_2^\star p_2)$ Thus we replace on the RHS (via (V-SUBST)) and get:

$$\begin{aligned}\Gamma_{\mathbb{C}} \models \forall p_1, p_2.\ e_1^\star(\lambda x. e_2^\star\ p_1) \wedge e_1^\star(\lambda x. e_2^\star\ p_2) \\ = e_1^\star(\lambda x. e_2^\star p_1 \wedge e_2^\star p_2)\end{aligned}$$

By some η-expansion and the IH for e_1 we can turn this to:

$$\begin{aligned}\Gamma_{\mathbb{C}} \models \forall p_1, p_2.\ e_1^\star(\lambda x. e_2^\star\ p_1) \wedge e_1^\star(\lambda x. e_2^\star\ p_2) \\ = e_1^\star(\lambda x. e_2^\star p_1) \wedge e_1^\star(\lambda x. e_2^\star p_2)\end{aligned}$$

Which is trivially provable by (V-REFL).

(7) (ST-PAIR), (ST-FST)
All trivial by IHs.

(8) (ST-INL)
Does not apply for the cases we consider.

(9) (ST-CASE)
Trivial by (V-SUMIND) and the IHs.

□

Thus, any term obtained by the $\star$-translation (return, bind, actions, lifts, ...) will be conjunctive in this sense, which means they also preserve the property through application.

With a completely analogous definition and proof we get the expected result of conjunctivity over (non-empty) universal quantification. The non-empty requirement is not actually stressed during that proof, but it's the wanted result as WPs (which can be taken as arguments) might not distribute over empty universals (in particular, non-satisfiable WPs do not).

References

D. Ahman and T. Uustalu. Update monads: Cointerpreting directed containers. *TYPES*, 2013.

D. Ahman, N. Ghani, and G. D. Plotkin. Dependent types and fibred computational effects. *FOSSACS*, 2016.

R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19:335–376, 2009.

P. Audebaud and E. Zucca. Deriving proof rules from continuation semantics. *Formal Asp. Comput.*, 11(4):426–447, 1999.

G. Barthe and T. Uustalu. CPS translating inductive and coinductive types. *PEPM*. 2002.

G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. *POPL*. 2014.

N. Benton. Simple relational correctness proofs for static analyses and program transformations. *POPL*. 2004.

N. Benton and A. Kennedy. Exceptional syntax. *J. Funct. Program.*, 11(4): 395–410, 2001.

N. Benton, J. Hughes, and E. Moggi. Monads and effects. *APPSEM*. 2000.

E. Brady. Programming and reasoning with algebraic effects and dependent types. *ICFP*, 2013.

E. Brady. Resource-dependent algebraic effects. *TFP*, 2014.

C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. *POPL*, 2014.

A. Charguéraud. Characteristic formulae for the verification of imperative programs. *ICFP*. 2011.

A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. *ICFP*, 2009.

T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988.

L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. *TACAS*. 2008.

E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8):453–457, 1975.

E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

A. Filinski. Representing monads. *POPL*. 1994.

A. Filinski. Representing layered monads. *POPL*. 1999.

A. Filinski. Monads in action. *POPL*. 2010.

J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. *ESOP*. 2013.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. *PLDI*, 1993.

B. Jacobs. Dijkstra and Hoare monads in monadic computation. *Theor. Comput. Sci.*, 604:30–45, 2015.

K. Jensen. Connection between Dijkstra's predicate-transformers and denotational continuation-semantics. DAIMI Report Series 7.86, 1978.

K. Keimel. Healthiness conditions for predicate transformers. *Electr. Notes Theor. Comput. Sci.*, 319:255–270, 2015.

K. Keimel and G. Plotkin. Mixed powerdomains for probability and nondeterminism. submitted to LMCS, 2016.

K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. *LPAR*. 2010.

E. Moggi. Computational lambda-calculus and monads. *LICS*. 1989.

A. Nanevski, J. G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *JFP*, 18(5-6):865–911, 2008.

A. Nogin. Quotient types: A modular approach. *TPHOLs*. 2002.

C. Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In B. W. Paleo and D. Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, 2015.

G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. *ESOP*. 2009.

A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, 2006.

D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in haskell. *SIGPLAN Not.*, 46(12):95–106, 2011.

N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. *ICFP*, 2011.

N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the Dijkstra monad. *PLDI*, 2013.

N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016.

P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. 1990.

P. Wadler. The essence of functional programming. *POPL*. 1992.

P. Wadler. Monads and composable continuations. *Lisp Symb. Comput.*, 7 (1):39–56, 1994.