Verificación de Programas con F*

Clase 13 – 25/06/2024

Misceláneas

- Nuevo link: https://mtzguido.github.io/teaching/vcf24/index.html
- **Empezar a pensar proyectos**
  - Una estructura de datos
  - Algún algoritmo relativamente simple
  - Algún programa real (con IO, etc) y algunas pruebas ligeras sobre el mismo
  - Formalizar alguna estructura matemática (grupo, anillo, etc)
  - Formalizar algún lenguaje de programación
- Ya hay algunos valientes:
  - **Joaquín Arroyo, Francisco Bolzán:** red-black trees
  - **Yamil Saer, Tomás Santucci:** trie
  - **Tomás Castro Rojas:** binomial heaps
  - **Joaquín Caporalini:** heapsort

interfaz similar (esta clase)

Clases de Tipo

- Polimorfismo *ad-hoc*
  - En contraste con polimorfismo *paramétrico* (Sistema F)
  - Esencialmente sobrecarga de operadores, pero extensible y bien definido
  - Wadler, Blott – How to make ad-hoc polymorphism less ad-hoc
- Extensible
  - Agregar tipos

    ```
    instance Eq MyType where { (==) = … }
    ```
  - Agregar operadores:

    ```
    class Monoid a where { mempty :: a; … }
    ```

Clases de Tipo: la alternativa

```
let rec eq_nat (x y : nat) : bool =
  match x, y with
  | Z, Z -> true
  | S x, S y -> eq_nat x y
  | _ -> false

let rec eq_list (#t : Type)
  (eq_t : t -> t -> bool) (xs ys : list t) : bool =
  match xs, ys with
  | [], [] -> true
  | x::xs, y::ys -> eq_t x y && eq_list eq_t xs ys
  | _ -> false

let test = (eq_list eq_nat) [Z] []
```

- Sin clases de tipo, tenemos que usar argumentos auxiliares para pasar las funciones relevantes
  - Definir funciones base (`eq_nat`)
  - Definir combinadores (`eq_list`)
  - Pasar funciones para los tipos asumidos (`eq_t`)
  - Llamar a la función específica de cada tipo (`eq_list eq_nat`)

Clases de Tipo: elaboración

```
let rec eq_nat (x y : nat) : bool =
  match x, y with
  | Z, Z -> true
  | S x, S y -> eq_nat x y
  | _ -> false
```

instance _ : eq nat = { … }

```
let rec eq_list (#t : Type)
  (eq_t : t -> t -> bool) (xs ys : list t) : bool =
  match xs, ys with
  | [], [] -> true
  | x::xs, y::ys -> eq_t x y && eq_list eq_t xs ys
  | _ -> false
```

instance eq_list #t (_ : eq t) :
  eq (list t) = { … }

```
let test = (eq_list eq_nat) [Z] []
```

resuelto automáticamente

Clases de Tipo: canonicidad/coherencia

- Una de las propiedades importantes (al menos en Haskell) es la *coherencia:*
  - Las decisions del *type-checker* no tienen que influir en la semántica del programa
  - Si declaramos `instance Eq [Int]` ¿coincide con la composición de estas?

    ```
    instance Eq a => Eq [a]
    instance Eq Int
    ```

  - Por la misma razón, no se permiten instancias de la forma

    ```
    instance Ord a => Eq a
    ```

  - F* **no** tiene ninguna garantía de canonicidad/coherencia
    - En general los lenguajes con tipos dep. no lo hacen

Improving Typeclass Relations by Being Open

Guido Martínez
CIFASIS-CONICET
Rosario, Argentina
martinez@cifasis-conicet.gov.ar

Mauro Jaskelioff
CIFASIS-CONICET
Rosario, Argentina
jaskelioff@cifasis-conicet.gov.ar

Guido De Luca
Universidad Nacional de Rosario
Rosario, Argentina
gdeluca@dcc.fceia.unr.edu.ar

Clases de Tipo: ¿Curry-Howard?

Clases de Tipo (¿lógica?)

- Clase
- Método
- Instancia base
- Instancia genérica
- Superclase
- Resolución de instancias
- Canonicidad

Tipado dependiente (programa)

- Tipo *record* de operaciones (diccionario)
- Elemento del diccionario
- Valor del tipo diccionario
- Función entre valores del diccionario
- Diccionario que contiene otro como element
- Construcción automática de diccionario
- La automatización siempre construye el mismo diccionario

Clases de Tipo: F*

(demo)

Tareas

- **Proyectos**. Ahora sí, entramos en modo consulta / a pedido.
- En la página listé los proyectos que ya eligieron.