Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencias de la Computación
R-422 — Compiladores

Apunte

Compilando Funciones de Alto Orden

En este apunte estudiaremos una etapa clave de la compilación de un lenguaje funcional de alto orden (como FD4): la "conversión de clausuras" (*closure conversion*). Es la etapa que convierte a las funciones de alto orden a objetos concretos, i.e. clausuras, que las representan. Al terminar la conversión, el programa ya no presenta funciones de alto orden. Es decir, el código resultante es de primer orden (todas las funciones son top-level).

Esta transformación nos acerca mucho a la noción usual de función en código máquina de bajo nivel (x86, MIPS, LLVM, etc) y, por ejemplo, nos permite generar código C.

1. El problema: funciones como valores + anidamiento

Consideremos el siguiente fragmento FD4, donde le damos un nombre a `f` sólo para ser más claros en lo que sigue:

```
let suma : Nat -> (Nat -> Nat) =
  fun (x:Nat) ->
    let f = fun (y:Nat) -> x + y in
    f

let suma5 : Nat -> Nat = suma 5
```

La función `suma` está currificada: al aplicarla a un argumento `x` devuelve una función `f` de tipo `Nat -> Nat`. Esta nueva función toma un valor `y` y le suma `x`. Aquí están ocurriendo dos cosas interesantes: `suma`, al ser llamada, está devolviendo una función, y esa función tiene una variable `x` ligada *fuera de ella*. Esta `x` toma valores potencialmente distintos con cada llamada a `suma`. Por ello, una llamada como `suma 5` no puede simplemente devolver un puntero al código de una función[1], y debe forzosamente devolver un "objeto" distinto al que devuelve (p. ej.) una llamada `suma 9`.

Para solucionar este problema, volveremos a usar la noción de clausura. Una llamada a `suma` devolverá no sólo un puntero a función, sino también una representación concreta de un entorno, dando valores a las variables libres de la función. La evaluación de `suma 5` debe entonces generar una clausura con el entorno `[x->5]` y con el código correspondiente a `(fun (y:Nat) -> x+y)`. Luego, si llamamos a esta clausura, por ejemplo con:

```
let quince : Nat = suma5 10
```

la evaluación llamará al código de la clausura, proveyéndole no sólo su argumento (`10`) sino también su entorno (`[x->5]`). La versión transformada de `(fun (y:Nat) -> x+y)` será tal que tomará el valor de `x` del entorno con el cual está emparejada, y tomará `y` como un argumento normal. Los accesos a variables del entorno involucran extraer el valor del mismo, leyendo el *heap*, mientras que los argumentos normales se pasan según la convención de llamadas de sistema.

[1] Potencialmente se podría generar el código dinámicamente y devolver un puntero a él, pero esto es ineficiente, poco práctico, y potencialmente un riesgo de seguridad.

2. La Conversión de Clasuras

Como primer paso, haremos una pasada sobre el programa para convertir cada función en una creación de clausura. Para ello, usamos la sintaxis `(fun (x:`τ`) -> ...)[`$v_1, \ldots, v_n$`]` para indicar la creación de una clausura para la función dada. Las variables libres de la función son reemplazadas por variables *distinguidas* de la forma e_i, las cuales toman los valores v_i. Contrariamente a la evaluación en la máquina CEK y la máquina virtual, no tomamos todas las variables visibles, sino sólo aquellas que realmente aparecen en el cuerpo de la función. Concretamente, cuando generemos código, esta operación creará (en el heap) un objeto con dos elementos: un puntero a código para la función y un arreglo conteniendo los valores v_i. También usaremos una representación *plana* de clausuras.

El fragmento previo, traducido, debería quedar como:

```
let suma : Nat -> (Nat -> Nat) =
  (fun (x:Nat) -> let f = (fun (y:Nat) -> e0 + y)[x] in f)[]

let suma5 : Nat -> Nat = suma 5

let quince : Nat = suma5 10
```

Ambas funciones dentro de `suma` son transformadas en creaciones de clausuras. La más externa, la definición de `suma`, es una clausura con el entorno vacío `[]` ya que no tiene variables libres. Evaluar `suma` consiste solamente en construir esta clausura y retornarla. Al llamar a `suma`, por ejemplo como en `suma5`, ejecutamos el cuerpo de dicha clausura con el argumento 5. En el cuerpo, llegamos a la creación de la clausura `(fun (y:Nat) ->`e_0 `+ y)[5]`, que se retorna, siendo el valor resultado para `suma5`. Ahora, en la definición de `quince`, llamamos a esta clausura con argumento `10`, con lo cual terminamos ejecutando el código $e_0 + y$ en el entorno `[5]` y con $y = 10$, dando el resultado final de `15`.

Hay que notar dos cosas:

1. Las variables e_i se refieren siempre al entorno *actual*. Los entornos no se relacionan entre ellos.
2. Las clausuras ya no tienen variables libres, con lo cual podemos llevarlas todas a definiciones top-level.

3. Hoisting

Una vez que las funciones locales ya no dependen de su entorno léxico, podemos "elevarlas" a funciones globales. En la etapa de hoisting[2]: recorremos todo el término, reemplazando cada `fun` por un nombre, y generando una definición top-level de ese nombre con el mismo cuerpo.

El ejemplo anterior puede quedar como:

```
let _g0 (y:Nat) : Nat = e0 + y

let _g1 (x:Nat) : Nat -> Nat = _g0[x]

let suma : Nat -> (Nat -> Nat) = _g1[]

let suma5 : Nat -> Nat = suma 5

let quince : Nat = suma5 10
```

Esta etapa no hace más que tomar el resultado de la conversión de clausuras y convertir las funciones lambda dentro de las creaciones de clausuras en definiciones globales, cada una con algún nombre fresco.

[2] También conocido como "Lambda-Lifting".

Una vez que llegamos aquí, las funciones sólo ocurren como definiciones globales, y el camino para compilar a una máquina de registros es mucho más evidente.

A menudo es conveniente hacer closure conversion y hoisting en una misma pasada.

4. Comparación con Otros Lenguajes

No todos los lenguajes con anidamiento o funciones como valores necesitan de la conversión de clausuras. Es la combinación de ambos lo que realmente nos da el poder de un lenguaje funcional *de alto orden*, con el costo de necesitar clausuras en tiempo de ejecución.

Si quitamos el anidamiento de funciones, podemos tener funciones como valores simplemente como punteros. El lenguaje C, básico como es, permite funciones como valores y cae en este campo. Un fragmento FD4 como el siguiente:

```
# Valores función, sin anidamiento
let op (b:Nat) : Nat -> Nat -> Nat =
  ifz b
  then suma
  else resta
```

Podría ser entendido como el siguiente fragmento C, sin mayor problema.

```
typedef int (*binop)(int, int); // binop ≈ ℕ → ℕ → ℕ

binop op (int b) {
    if (b == 0)
        return suma;
    else
        return resta;
}
```

Podemos hacer esto porque `suma` y `resta` no dependen de `b` ni de ninguna otra variable visible. Si dependieran, ¿qué puntero retornaríamos en `op`...? Ninguno servirá: tenemos que crear una estructura con el valor de `x` y un puntero a la función.

Por otro lado, hay lenguajes que permiten funciones anidadas. El siguiente fragmento FD4 define una función anidada pero que es sólo usada localmente:

```
# Anidamiento, sin valores función
let f (x:Nat) : Nat =
  let g (y:Nat) : Nat =
    x + y
  in
  g 17

let _ : Nat = f 42
```

Esto mismo puede hacerse en Pascal (y también en algunas extensiones de C) de la siguiente manera:

```
program Nesting;
  function f(x : integer) : integer;
    function g(y : integer) : integer;
    begin
      g := x + y; // return x+y
```

```
    end;
  begin
    f := g(17); // return g(17)
  end;
begin
  writeln (f(42));
end.
```

¿Por qué puede `g` referirse a las variables de `f` aquí? Porque siempre que `g` está activa al ser llamada, entonces *¡`f` también lo está!* Las llamadas a `g` ocurren sólo dentro de `f`, cuando `f` está viva, y por lo tanto, las variables de `f` son accesibles y tienen valores bien definidos. Como Pascal no tiene valores función, no podemos de ninguna manera devolver `g` sin aplicarla, con lo cual no hacen falta clausuras (todo vive en la pila).

El lenguaje Python combina tanto valores función como anidamiento

```
def f(x):
    def g(y):
        return x + y
    return g;

print(f(42)(17))
```

y, naturalmente, usa clausuras para poder compilar este código (lo que demuestra que las clausuras no son un concepto exclusivo de los lenguajes funcionales).

5. Implementación

Utilizamos un tipo `Ir` para representar a los términos intermedios luego de la conversión y un tipo `IrDecl` para representar las declaraciones globales que se generan. Hay dos tipos de `IrDecl`: las que declaran funciones (estáticas, o sea un fragmento de código) y las que definen valores. Las funciones top-level como `let id (x:Nat) : Nat = x` terminan siendo dos `IrDecl`: una para crear el código de la función y una para asignarle a `id` la clausura correspondiente[3].

El tipo `Ir` es similar al tipo de términos *core*, salvo por el caso de las funciones y las variables. Notar que omitimos los `Pos`).

```
data IrTy = IrInt
          | IrClo
          | IrFunTy

data Ir = IrVar Name
        | IrGlobal Name
        | IrCall Ir [Ir] IrTy
        | IrConst Const
        | IrPrint String Ir
        | IrBinaryOp BinaryOp Ir Ir
        | IrLet Name IrTy Ir Ir
        | IrIfZ Ir Ir Ir
        | MkClosure Name [Ir]
        | IrAccess Ir Int
```

[3] Si bien esta función no tiene ninguna variable libre, usaremos clausuras uniformemente.

Los tipos `Ir` e `IrDecl` se encuentran definidos en el módulo `IR.hs` del repositorio.

Para que el código generado no contenga una cantidad absurda de *casts*, nuestra representación intermedia es tipada. El nodo `IrLet` introduce un let-binding con un tipo explícito para la variable definida, y en `IrCall` se especifica el tipo del resultado. De esta forma, el generador de código C (`C.hs`) puede declarar variables con el tipo esperado, en vez de usar genéricamente `void*` o algo similar. Por otro lado, tener una IR tipada permite escribir un typechecker para la misma, a modo de "seguro" para poder chequear que las optimizaciones no crean términos mal tipados, algo que GHC famosamente hace, pero que nosotros no haremos.

Para las variables en `Ir`, usamos una representación con nombres (fully-named), por lo que hay que generar nombres frescos a medida que es necesario (es útil usar una mónada de estado para llevar un contador de nombres frescos). Para las variables generadas durante la conversión, es ventajoso usar un prefijo que las distinga (por ejemplo '`__`'), para luego poder distinguirlas fácilmente de los nombres globables (que, para mejorar la performance, no deben considerarse como variables libres al armar una clausura).

Para ir recolectando las definiciones es útil utilizar una mónada `Writer` sobre el monoide `[IrDecl]`. Es decir, la función que haga la conversión de clausuras sobre términos podría tener un tipo similar a:

```
closureConvert :: Term -> StateT Int (Writer [IrDecl]) Ir
```

Las funciones serán reemplazadas por declaraciones globales, y en donde ocurrían habrá un nodo `MkClosure` con un nombre global y los valores de su entorno.

La función final `runCC` debe tener tipo `[Decl Term] -> [IrDecl]` (o similar).

Las siguientes fuentes pueden ser útiles: [2, 3] y [1, Capítulo 15].

6. Tareas

a) Implementar la conversión de clausuras y hoisting (sugerimos hacerlo en la misma pasada) en un nuevo módulo `ClosureConvert.hs`. El código convertido debe ser utilizado para producir código C. Agregar una opción al compilador para compilar a código C (`--cc` o `--c`). Para la generación de código C, puede utilizar el módulo `C.hs` que se encuentra en el repositorio. En particular, este módulo provee una función `ir2C :: IrDecls -> String`, que dada una lista de declaraciones devuelve una cadena con el código C correspondiente.

Referencias

[1] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, new ed edition, July 2004.

[2] Matt Might. Closure conversion: How to compile lambda. `http://matt.might.net/articles/closure-conversion/`.

[3] Jeremy Siek. The essence of closure conversion. `https://siek.blogspot.com/2012/07/essence-of-closure-conversion.html`, Jul 2012.