Jose David Fernández Rodríguez, 2005

Implementando lenguajes funcionales puros

La máquina STG

Programación declarativa avanzada,
Profesor Blas Ruiz Jiménez

Introducción

- Lenguajes declarativos:
  - semántica mediante sistemas de reescritura
- Lenguajes funcionales:
  - Gran riqueza sintáctica => expresividad
  - Estamos interesados en lenguajes perezosos
  - Dificultad de compilación/interpretación
  - Basados en el lambda-cálculo
  - Frontera natural entre el front-end y el back-end

Back-end del compilador

- Los lenguajes perezosos tienen un modelo de computación muy alejado del imperativo
  - Pero no es práctico diseñar nuevas arquitecturas para ellos
  - Es necesario trasladar la semántica perezosa a un entorno imperativo
- Estructura tradicional del back-end
  - Recibe como entrada lambda-cálculo enriquecido
    - Ligado a una semántica denotacional (reescritura)
  - Lo traduce a código imperativo de una máquina abstracta
    - Ligado a una semántica operacional (imperativa)
  - Traduce dicho código a lenguaje objeto: ensamblador o C

Máquinas abstractas

- Surgen como medio para formalizar una semántica operacional (imperativa) para lenguajes perezosos
- La comunidad académica ha sido muy productiva:
  - máquina G / Spineless-G / STG
  - máquina SKI
  - máquina <v,G>
  - máquina TIM
  - En teoría son bastante distintas entre sí
  - En el fondo el efecto de las optimizaciones las uniformiza un poco
    - Las diferencias de fondo se refieren al nivel de "interpretetatividad" inherente a cada máquina.

Máquina STG

- Es un diseño relativamente reciente
- Spineless Tagless G machine
- Es usada como base de GHC, el compilador de referencia para Haskell
- Lleva a sus últimas consecuencias el camino recorrido desde la máquina G:
  - uniformización de los objetos del programa en ejecución
    - Minimización del nivel de “interpretetatividad”
  - integración de la aritmética de números enteros
  - ...y multitud de detalles técnicos

Nociones básicas de la STG

- Tipos de datos básicos
  - por valor: primitivos (enteros, caracteres, etc)
  - por referencia: punteros (funciones, LETs, constructores de datos, etc)
- Valores primitivos
  - Incluidos por razones de eficiencia
  - Su presencia explícita en el lenguaje
    - da al front-end más libertad para optimizar
    - se inserta de manera natural con el resto de características
    - permite tipos de datos básicos uniformes respecto a los del usuario, sin necesidad de tratarlos de forma especial

Nociones básicas de la STG

- Valores por referencia (Closures)
  - Unidad fundamental de evaluación de expresiones
  - Representación uniforme para todo objeto no primitivo
    - Inclusive los internos del sistema
  - Idea básica:
    - Cada objeto posee datos dinámicos (determinados en tiempo de ejecución): conjuntos de valores primitivos y referencias
    - Y un comportamiento estático (especificado en tiempo de compilación): piezas de código para la evaluación del programa o de tareas de sistema. Se guardan en la InfoTable
  - Un closure encapsula ambos aspectos
  - Y el diseño de la máquina se dirige a uniformizar al máximo la representación de los distintos closures

Nociones básicas de la STG

- Pilas
  - Se definen cuatro pilas
    - Las pilas V y S para argumentos de tipos primitivo y puntero (referencias a closures), respectivamente
    - La pila de continuaciones (piezas de código)
    - La pila de marcos de actualización (updates), llamados dumps
  - En teoría se podrían mezclar en una sola
  - Aunque en la práctica una implementación seria (la de GHC) usa dos
- Registros
  - Son lugares de comunicación entre piezas de código.
  - En función de la implementación puede haber varios
  - El más importante es NODE

Lambda-cálculo

- Formalismo matemático
  - captura el concepto de función computable
  - plenamente expresivo (por ej. representación de aritmética a la Peano y estructuras de datos)
- Lisp
  - (defun aplica2 (f g x) (let ((h (funcall g x))) (funcall f h h)))
- Haskell
  - aplica2 f g x = f h h where h = g x
- Lambda-cálculo
  - (λf.λg.λx. (λh. f h h) (g x))

Lambda-cálculo

- Por sí sólo no es adecuado como herramienta de implementación
- Hay que añadir
  - Mecanismos de programación estructurada
  - Funciones con número arbitrario de argumentos
  - Funciones predefinidas (por lo menos las aritméticas)
  - Estructuras de datos y construcciones CASE para su evaluación
  - Sintaxis especial para las ligaduras LET (definiciones locales, generalmente sin argumentos)
- El resultado es el lambda-cálculo enriquecido
- Cada máquina impone requisitos adicionales

Lambda-cálculo para la STG

- Las aplicaciones de funciones son planas
  - –No se permiten expresiones como (sum 1 (mul 2 3))
  - –Sino (let n = mul 2 3 in sum 1 n), de forma perezosa
  - –O bien (case mul 2 3 of n -> sum 1 n), de forma estricta

Lambda-cálculo para la STG

- Los constructores de datos también se aplican de forma plana
  - –y además de forma saturada (no currificada)
  - –NO: (cons 1 (cons 2 nil))
  - –TAMPOCO : (cons 1)
  - –SI: (let c2 = cons 2 nil in cons 1 c2)
  - –TAMBIEN: (let cons_fun h t = cons h t in cons_fun 1)

Lambda-cálculo para la STG

- Las funciones se definen como globales (top-level) o locales (ligaduras en LETs)
  - sintaxis uniforme para ambas
  - También para funciones sin argumentos (variables locales o CAFs)
- Cada ligadura tiene un flag que indica si es actualizable con su resultado
  - Se puede decidir mecánicamente, pero la tarea se deja al front-end

Lambda-cálculo para la STG

- Cada expresión tiene un entorno de evaluación para sus ligaduras (variables accesibles)
  - variables globales (funciones y CAFs)
  - argumentos de la función (en las pilas V y S)
  - ligaduras locales: se declaran explícitamente, para cada función o variable local (las globales también pueden tener, por motivos técnicos)
    - El proceso de obtener las ligaduras locales se deja al front-end

Ejemplos

```
twice {} NoUpdate {f, x} = LET
        fx {f, x} DoUpdate {} = f {x}
  IN f {fx}

map {} NoUpdate {f, xs} = CASE xs OF
    Nil -> Nil
    Cons {y, ys} -> LET
        fy {f, y} DoUpdate {} = f {y}
        mapfy {f, ys} DoUpdate {} = map {f, ys}
    IN Cons {fy, mapfy}
```

Ejemplos

```
twice {} NoUpdate {f, x} = LET
        fx {f, x} DoUpdate {} = f {x}
  IN f {fx}

map {} NoUpdate {f, xs} = CASE xs OF
    Nil -> Nil
    Cons {y, ys} -> LET
        fy {f, y} DoUpdate {} = f {y}
        mapfy {f, ys} DoUpdate {} = map {f, ys}
    IN Cons {fy, mapfy}
```

Ligaduras locales

Listas de argumentos

Ejemplos

```
fac {} NoUpdate {num} = LET
    fac_tailcall {fac_tailcall} NoUpdate {n, ac} =
        CASE n - 1 OF
          -1 -> ac
          0 -> ac
          DEFAULT m -> CASE ac * n OF
            ac2 -> fac_tailcall {m, ac2}
    IN fac_tailcall {num, 1}
```

Ejemplos

```
fac {} NoUpdate {num} = LET
    fac_tailcall {fac_tailcall} NoUpdate {n, ac} =
        CASE n - 1 OF
          -1 -> ac
          0 -> ac
          DEFAULT m -> CASE ac * n OF
              ac2 -> fac_tailcall {m, ac2}
    IN fac_tailcall {num, 1}
```

Ligadura local recursiva

valores primitivos

Semántica operacional

- Cada máquina suele tener un conjunto de instrucciones
  - Se realiza una traducción (más o menos compleja) de lambda-cálculo a secuencia de instrucciones
- Con la máquina STG se sigue un enfoque ligeramente diferente
  - Asignamos significado operacional al propio lambda-cálculo STG

Ejemplo

```
map {} NoUpdate {f, xs} = CASE xs OF
   Nil -> Nil
   Cons {y, ys} -> LET
     fy {f, y} DoUpdate {} = f {y}
     mapfy {f, ys} DoUpdate {} = map {f, ys}
   IN Cons {fy, mapfy}
```

-La definición de la semántica operacional puede hacerse muy extensa y prolija

-sólo daremos algunas nociones básicas

Ejemplo

```
map {} NoUpdate {f, xs} = CASE xs OF
```

map es una función que toma dos argumentos

```
    fy {f, y} DoUpdate {} = f {y}
    mapfy {f, ys} DoUpdate {} = map {f, ys}
  IN Cons {fy, mapfy}
```

Ejemplo

```
map {} NoUpdate {f, xs} = CASE xs OF
```

La expresión CASE fuerza la evaluación de la expresión (xs). Se actualiza la pila de continuaciones.

```
  mapfy {f, ys} DoUpdate {} = map {f, ys}
IN Cons {fy, mapfy}
```

Ejemplo

```
map {} NoUpdate {f, xs} = CASE xs OF
  Nil ->
  Cons
   fy {f
   mapf
  IN Co
```

Eventualmente, termina la evaluación de (xs) y se restaura de la pila la continuación del CASE. Supongamos que se continúa por la segunda rama

Ejemplo

La expresión LET reserva memoria para los closures (fy) y (mapfy), llenándolos con sus InfoTables y ligaduras locales

```
Cons {y, ys} -> LET
  fy {f, y} DoUpdate {} = f {y}
  mapfy {f, ys} DoUpdate {} = map {f, ys}
IN Cons {fy, mapfy}
```

Ambos se evalúan de forma semejante:

-Actualizan las pilas V y S para incluir los argumentos de la función

-Asignan a NODE una referencia al closure de la función

-Saltan a la pieza de código de evaluación de la función

Pero este comportamiento queda diferido hasta que sean forzados por una expresión CASE

```
fy {f, y} DoUpdate {} = f {y}
mapfy {f, ys} DoUpdate {} = map {f, ys}
IN Cons {fy, mapfy}
```

Ejemplo

```
map {} NoUpdate {f, xs} = CASE xs OF
```

Se reserva memoria para el closure que representa al constructor CONS. Se rellena con su InfoTable y sus ligaduras

```
IN Cons {fy, mapfy}
```

Ejemplo

Se salta al código de evaluación del constructor. Termina la evaluación de la función. El constructor carga en un registro su etiqueta (tag) y salta a la pieza de código obtenida de la pila de continuaciones

IN Cons {fy, mapfy}

Actualizaciones (updates)

- Hasta ahora se ha descrito una máquina de reducción de árboles
  - Cada vez que se requiere el valor de una variable, vuelve a evaluarse
- Un diseño real requiere reducción de grafos
  - Las funciones **sin argumentos** que devuelven **referencias a closures** *pueden* ser sobreescritas
    - Con el resultado, si su closure es lo bastante pequeño
    - O con una **indirección**
  - Posteriores requerimientos de su valor obtendrán directamente o vía indirección el previamente calculado

Actualizaciones (updates)

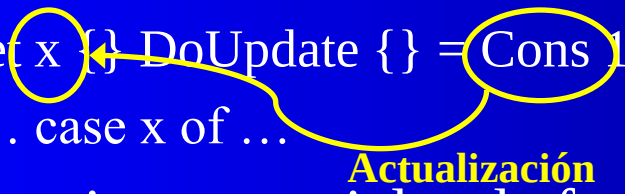
- ¿Qué objetos pueden disparar una actualización al ser evaluados?
  - Constructores de datos
    - let x {} DoUpdate {} = Cons 1 Nil in

      … case x of …
  - Aplicaciones parciales de funciones (PAPs):
    - twice f x = let fx = f x in f fx

      main = let sum3 {} DoUpdate {} = sumar 3 in twice sum3 4
    - Constituyen un tipo especial de closure, al igual que las indirecciones

Actualizaciones (updates)

- ¿Qué objetos pueden disparar una actualización al ser evaluados?
  - Constructores de datos
    - let x {} DoUpdate {} = Cons 1 Nil in
      … case x of …



  - Aplicaciones parciales de funciones (PAPs):
    - twice f x = let fx = f x in f fx
      main = let sum3 {} DoUpdate {} = sumar 3 in twice sum3 4

    - Constituyen un tipo especial de closure, al igual que las indirecciones

Actualizaciones (updates)

- ¿Cómo se prepara una variable para ser sobreescrita cuando sea evaluada?
  - Antes de comenzar la evaluación de su expresión
    - Se salva en la pila de dumps:
      - una referencia al closure de la variable para poder sobreescribirlo
      - Los marcos (punteros base)de las pilas V y S, para las PAPs
    - Se salva en la pila de continuaciones una pseudocontinuación encargada de realizar la sobreescritura, para los constructores
    - Se resetean (igualan a la cima de la pila) los marcos de las pilas V y S, para que la aplicación sea efectivamente parcial

Actualizaciones (updates)

- ¿Cuándo se realiza la actualización?
  - En el caso de los constructores, cuando tras ser evaluados saltan a la pseudocontinuación
  - En el caso de las PAPs, cuando el código que evalúa la función aplicada se da cuenta de que no tiene suficientes argumentos

Actualizaciones (updates)

- ¿Cómo se realiza la actualización?
  - En cualquier caso
    - se restauran de la pila de dumps los marcos de las pilas V y S
    - y además se toma de ella la referencia al closure a sobreescribir
    - se realiza la actualización (con el constructor o el closure PAP)
  - En el caso de los constructores, la pseudocontinuación
    - además, salta a la siguiente pieza de código en la pila de continuaciones
  - En el caso de las PAPs, el código de comprobación de número de argumentos
    - además, construye el closure PAP que guarda los argumentos aplicados (encontrados en las pilas V y S)

Bibliografía adicional

- "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine Version 2.5"
  - Simon Peyton Jones, 1992, web de SPJ:
    http://research.microsoft.com/Users/simonpj
- "The Implementation of functional languages"
  - Simon Peyton Jones, Prentice Hall, 1987
- "Implementing Functional Languages: a Tutorial"
  - Simon Peyton Jones, David R. Lester, 1997, web de SPJ
- "How to make a fast curry: push/enter vs eval/apply"
  - Simon Marlow, Simon Peyton Jones, web de SPJ