EXPERIENCIAS EN IMPLEMENTACIÓN DE LENGUAJES FUNCIONALES: LA MÁQUINA STG

Redactado por:
Jose David
Fernández Rodríguez,
Abril de 2005.

INDICE:

1. Introducción

Empecé a interesarme en este tema hace aproximadamente dos años, leyendo [PEY87]. Este libro, que toma Miranda como sujeto de estudio, analiza desde el principio al fin el proceso de compilación de un lenguaje perezoso de alto nivel. El front-end es interesante en sí mismo, está repleto de subsistemas que merece la pena examinar con profundidad, como el lambda-lifter (necesario en el caso de estudio del libro), el sistema de inferencia/comprobación de tipos, o algunas de las pasadas de compilación de Miranda a lambda-cálculo enriquecido. El bagaje necesario para comprender algunos de los tecnicismos lo tomé de [RUIZ00], así como de textos básicos sobre el lambda-cálculo proporcionados en la asignatura "Ampliación de Programación Declarativa".

Lo que más me llamó la atención fue el back-end: la parte del compilador encargada de inscribir el programa, con sus características de lenguaje funcional puro y perezoso, en el modelo de computación usual de los ordenadores, a través de una máquina abstracta de evaluación. En el libro se describía minuciosamente la máquina G junto a un sistema de compilación lambda-cálculo enriquecido => código G. El modelo de compilación era considerablemente distinto de las gramáticas atribuidas usuales para los lenguajes imperativos: los esquemas de compilación, conceptualmente mucho más sencillos que las gramáticas atribuidas. Los esquemas son básicamente juegos de funciones auto y co-recursivas (reflejando la estructura gramatical del propio lenguaje) que toman expresiones en lambda-cálculo enriquecido y entornos de ligaduras de variables, y devuelven la secuencia de instrucciones que evalúan la expresión.

No obstante, dado lo árido del texto junto con mi dificultoso entendimiento del inglés, no comprendí plenamente el funcionamiento de la máquina G hasta que leí [PEY97]. En este libro, a modo de tutorial, se introduce no sólo la máquina G, sino la Three Instruction Machine (TIM) y la G-concurrente, todas de forma realmente sencilla.

La visión comparativa de la máquina G y la TIM me permitió apreciar la naturaleza de la G, básicamente interpretativa: En memoria se representa de forma literal un grafo que representa la expresión de lambda-cálculo que está siendo reducida (el programa en ejecución). Cada expresión se compila a una pieza de código G que evalúa la expresión, actuando como lo haría un intérprete puro excepto en que los pasos de evaluación están precompilados en vez de ser decididos por un intérprete. De hecho, existen instrucciones G realmente complejas en su comportamiento, como UNWIND, que se corresponde con la búsqueda del siguiente redex a evaluar, tal y como haría un intérprete.

En contraste, la máquina TIM no fue tan sencilla de entender, aunque leer su descripción sirvió mucho para comprender la STG, ya que introduce la operativa de las máquinas "spineless" (o sea, sin representación literal en grafo de la expresión de lambda-cálculo a evaluar), así como la terminología y el concepto de "closure" como objeto que encapsula las ligaduras (variables libres) de una función junto con la secuencia de instrucciones que la evalúan (o sea, realizan la reducción de la expresión que constituye el cuerpo de la función).

En su último capítulo se da una explicación muy práctica sobre lambda-lifting (el proceso de eliminar definiciones locales de funciones con argumentos, flotándolas o subiéndolas hasta el ámbito global), respecto a la eminentemente teórica ofrecida en [PEY87], separando además el concepto de "full-laziness" (flotación de subexpresiones en definiciones locales de funciones a niveles superiores para evitar duplicación de trabajo). En este capítulo de [PEY97] se separan ambas tareas respecto al libro anterior [PEY87]: es deseable que cualquier compilador implemente full-laziness, pero no todas las máquinas abstractas requieren de lambda-lifting (por ejemplo, la máquina G lo necesita, pero la STG no).

Después de leer este libro, conseguí algunos documentos técnicos (el más importante [PEY99]), sobre todo relativos al sistema de apoyo en tiempo de ejecución (RTS) de GHC, ya que los detalles de implementación de más bajo nivel no presentes en el libro eran uno de mis intereses. Incluso intenté comprender el compilador GHC a través de sus fuentes, pero se trata de un sistema extremadamente complejo incluso usando la guía informal que se proporciona. En total, no saqué mucho en claro, así que decidí hacer las cosas por orden y leer [PEY92], la última descripción estable de la máquina STG, que parece líder por el momento respecto a otras máquinas en lo que respecta a eficiencia del código compilado, y es la usada en GHC. De todos modos, los fuentes de GHC y [PEY99] han seguido sin ser de mucha ayuda, pues la implementación ha cambiado mucho desde 1992, con multitud de correcciones, adiciones y eliminaciones de características y optimizaciones. En particular, un cambio particularmente intenso ocurrió al cambiar el modelo de aplicación de funciones de PUSH/ENTER a EVAL/APPLY (básicamente, adelantar el momento en que se considera si el número de argumentos de una aplicación es suficiente para la función aplicada, desde el código de evaluación de la función hasta el código que realiza su aplicación a los argumentos que se le han de pasar), tal y como se describe en [MAR03].

En los apartados que vienen a continuación, el texto [PEY92] es la fuente para resolver cualquier duda.

2. La máquina STG

La máquina STG me resultó más interesante que la G, pues aunque esta última tiene lo que podríamos llamar belleza estética (el modelo es conceptualmente simple y representa de forma muy sencilla y directa la reducción de una lambda-expresión, así como a la propia lambda-expresión en memoria), la STG es muy ingeniosa, su diseño da mucho juego para múltiples optimizaciones y define una semántica operacional para el lambda-cálculo particularmente directa, incluso podríamos usar la imagen "redonda":

- en las representaciones usuales del lambda-cálculo, para manipular valores numéricos (o, en general, entidades que no son funciones o constructores de datos, como los arrays), éstos son envueltos en constructores de datos predefinidos. Para optimizar las operaciones aritméticas (o con cualquier objeto especial) es necesario que el generador de código trabaje duro y sea extremadamente complicado, siendo capaz de realizar transformaciones de expresiones y análisis de comportamiento perezoso-estricto del programa. El problema radica en que la parte del compilador idónea para estas tareas es el

front-end, mientras que el back-end no debería ocuparse de estos asuntos, para los cuales el front-end está enfocado y dispone de una mejor perspectiva. Para solucionarlo, en la máquina STG hay dos tipos básicos de datos: valores primitivos (enteros y en general cualquier valor directo, esto es representado directamente en memoria tal y como sería en un lenguaje imperativo) y valores por referencia o encapsulados, que siempre son un puntero a un closure. Esta separación en dos tipos básicos de datos no es artificial ni difícil de comprender o implementar, sino que se integra de forma natural con el resto de características de la STG. De este modo, el front-end puede llevar a cabo optimizaciones transformando las porciones del programa de alto nivel donde pueda para que manipule directamente valores primitivos.

- un closure representa un objeto del lambda-cálculo o de la máquina STG susceptible de ser evaluado. Consiste en una estructura de datos que sirve para representar casi cualquier cosa: una función, una aplicación parcial, una indirección, un constructor de datos, una computación suspendida (en inglés *thunk*)… Todos los closures tienen una estructura uniforme, con lo que son tratados todos de la misma forma, minimizando enormemente la necesidad de zonas "interpretativas" en el código de evaluación de expresiones y el RTS, al contrario que en otras máquinas (o sea, instrucciones condicionales para discriminar entre posibles opciones u objetos de ejecución). El diseño está especialmente cuidado en este sentido. La noción básica de closure es una estructura de datos, con una serie de datos dinámicos (que pueden ser tanto de tipo primitivo como referencias a otros closures) almacenados, más una referencia a la pieza de código (estática, en función del tipo de closure) que ejecuta la evaluación.

- las expresiones LET sirven para reservar memoria para nuevas funciones con o sin argumentos (cada función se corresponde con un tipo de "closure", y cada vez que es instanciada con un entorno de ligaduras particular, se requiere reservar memoria para la instancia del "closure" que guarda dicho entorno) y nuevas instancias de constructores de datos. Se incluye una forma especial LETPRIM que no sirve para este propósito, sino para efectuar eficientemente operaciones aritméticas sobre valores primitivos sin recurrir a análisis de código en nuestra fase de compilación.

- las expresiones CASE sirven para discriminar alternativas y forzar la evaluación de constructores de datos a partir de expresiones (posiblemente computaciones suspendidas, esto es, funciones sin argumentos globales o definidas mediante LETs, que representan la evaluación diferida de una expresión) a forma normal débil por la cabeza (WHNF). Se incluye una forma especial CASEPRIM para evaluar y discriminar alternativas a partir de expresiones o aplicaciones de funciones que devuelven valores primitivos.

- la aplicación de funciones (o la evaluación de variables, totalmente equivalente a la aplicación de una función sin pasarle argumentos) se interpreta como limpieza del antiguo entorno de pila y construcción del nuevo con los nuevos argumentos de la llamada, y salto al código de evaluación de la función.

- la evaluación de un constructor (o de un valor primitivo) se interpreta como término de una evaluación a WHNF de una expresión y salto al código de evaluación del siguiente redex (esta pieza de código se denomina *continuación*), que siempre es la parte de selección y evaluación de una rama de un CASE. Mención especial merece la evaluación de un valor primitivo, ya sea en forma de variable ligada o valor inmediato, que tiene el mismo efecto aunque operacionalmente se trate de forma ligeramente distinta.

Esto en cuanto a los rasgos generales del lambda-cálculo STG. Veamos ahora algunos de los componentes de la máquina STG:

- pilas para argumentos: son dos separadas para argumentos de las funciones, separados según sus tipos: primitivos y referencias a closures. También pueden incluir ligaduras no albergadas ya en las pilas (o sea, en el closure o en variables temporales) que deban ser salvadas entre las dos partes de la evaluación de una expresión CASE por ser usadas en alguna de las ramas alternativas. Ambas pilas tienen, además de los consabidos punteros de cima, sendos punteros de base (usados en el cálculo de la longitud de la pila), de modo que se pueden apilar múltiples pilas virtuales una encima de otra con el auxilio de la pila de marcos de actualización (descrita más abajo). Esto es útil en la evaluación de aplicaciones parciales.

- pila de continuaciones: lo más parecido a una pila de llamadas a procedimiento en esta máquina. En realidad, en la STG, las piezas de código de evaluación de expresiones no tienen necesidad de estructurarse como procedimientos en el sentido tradicional como en C o Pascal (otra cosa es que se haga así por criterios de conveniencia en implementaciones no optimizadas). En lugar de eso, todas las transiciones entre las piezas de código son interpretables como saltos (en el caso general, indirectos) sin necesidad de mantener una pila de marcos de activación al estilo clásico. Cuando una expresión CASE fuerza una evaluación a WHNF, salva en la pila de continuaciones una referencia a la pieza de código que evalúa las alternativas, denominada continuación. Cuando se termina la reducción a WHNF, se toma la referencia en la cima de la pila de continuaciones y se salta a ella, con lo que se retomará la evaluación de la segunda parte del CASE.

- pila de marcos de actualización: para evitar la reevaluación de closures, al terminar su evaluación a WHNF éstos pueden ser sobreescritos con una indirección al resultado, o bien con una copia del resultado. Esta pila es un registro de los closures que aguardan ser sobreescritos.

- El código del programa se compone de una serie de piezas de código. Cada función local o global se evalúa en una pieza de código, a menos que contenga expresiones CASE, en cuyo caso tiene además tantas piezas de código adicionales (continuaciones) como expresiones CASE anidadas. En éstas, se elige la alternativa a ejecutar del correspondiente CASE tras forzar la evaluación a WHNF de la expresión escrutada por el CASE.

- El registro NODE es una referencia al closure que se está evaluando en cada momento, generalmente una función local o global, con o sin argumentos. No

obstante, en las ramas de una CASE (o sea, en la continuación que elige la alternativa correcta y la evalúa) apunta al constructor de datos al que ha sido reducido la expresión escrutada del CASE, con lo que se hacen accesibles los argumentos del constructor de datos, que están almacenados en su closure. La última acción antes de terminar una evaluación de aplicación de función, devolución de valor no primitivo o generación de un constructor de datos es apuntar NODE a ese closure.

- En el registro CONSTRTAG los constructores de datos cargan su etiqueta identificativa (sirve como número identificativo respecto a todos los constructores de un mismo tipo de datos), que sirve para que la continuación de un CASE elija la alternativa a ejecutar. Hay que mencionar que este es el mecanismo más sencillo de identificación de constructores, pero existen optimizaciones mucho más elaboradas descritas en [PEY92] (el retorno vectorizado, en el que cada rama tiene su propia pieza de código, y la rama a la cual saltar es escogida directamente por el constructor).

- En el registro VRET se depositan los valores primitivos numéricos devueltos por expresiones que se evalúan a este tipo de dato STG. Es el registro escrutado (en vez de CONSTRTAG) en las expresiones CASE para valores primitivos (CASEPRIM).

3. Actualizaciones

Sin actualizaciones, cada vez que una computación suspendida es forzada (o sea, su closure es evaluado a WHNF por la pieza de código asociada a su expresión), volvería a evaluarse desde el principio, malgastando tiempo de cómputo y espacio de memoria, duplicando los resultados obtenidos en evaluaciones anteriores.

Para evitarlo se recurre a la actualización: cada vez que se alcanza la WHNF de una computación suspendida, el closure de la segunda es actualizado, bien con una indirección al resultado, bien con una copia del mismo. La segunda opción ahorra tiempo gastado en seguir indirecciones, pero no está clara su utilidad a menos que se sobreescriba sólo la primera vez, durante la construcción del resultado (pues las indirecciones son eliminadas durante la recolección de basura y la copia indiscriminada aumenta potencial e inútilmente el espacio de memoria necesario). Siguiendo el espíritu de la STG, las indirecciones se modelan como un tipo especial de closure, cuyo código de evaluación simplemente salta al código de evaluación del closure al que apunta.

Existen dos tipos de closures construidos en tiempo de ejecución susceptibles de ser el destino de una actualización al estar en WHNF: los constructores y las aplicaciones parciales. Aunque la STG intenta tratar todo de un modo absolutamente uniforme, la asimetría esencial entre la semántica operacional de ambos (o sea, las funciones son aplicadas, y los constructores de datos son construidos, valga la redundancia) obliga a métodos de actualización diferentes según la WHNF final sea de uno u otro tipo:

- en el primer caso, antes de que el closure que debe ser actualizado comience a ser evaluado, en la pila de continuaciones se introduce una pseudocontinuación,

y en la de marcos de actualización, una referencia al closure que debe ser sobreescrito. Eventualmente, termina evaluándose un constructor de datos, que al evaluarse salta, en vez de a la continuación de un CASE, a la pseudocontinuación. Ésta lleva a cabo la actualización, sobreescribiendo con una indirección a dicho constructor de datos el closure cuya referencia se conserva en la pila de marcos de actualización, y después salta a la siguiente continuación, reanudando el ciclo de operación normal de la STG.

- en el segundo caso, igualmente antes de que el closure a actualizar comience a ser evaluado, se introduce en la pila de marcos de actualización una referencia al closure que debe ser actualizado y los punteros base de las pilas de argumentos, y se actualiza el valor de éstos con el de los respectivos punteros de cima. El efecto de esta operación es que la máquina STG considera las pilas vacías, aunque esto no afecta a la visibilidad de los argumentos en caso de que el closure que va a evaluarse sea una función con argumentos, pues éstos se direccionan a partir de los punteros de cima, que han quedado intactos. Eventualmente, el closure se evalúa a una aplicación parcial. Si el programa está correctamente tipado y no hacemos actualizaciones, jamás se dará el caso de que una función no encuentre en la pila todos los argumentos que necesita para ser evaluada. Pero con las operaciones que se han descrito, efectivamente una función se puede encontrar con que no tiene suficientes argumentos en las pilas para comenzar a evaluarse. En tal caso, lo que hace se hace es generar un closure especial tipo PAP (o sea, aplicación parcial, que recoge los argumentos situados en las pilas en ese momento y una referencia al closure de la función parcialmente aplicada a esos argumentos), y tomar de la cima del marco de actualización la referencia al closure a actualizar, que es sobreescrito con una indirección al PAP, y los punteros de base anteriores, para restaurar las pilas de argumentos anteriores. En un programa correctamente tipado, se garantiza que lo que contengan esas pilas son los argumentos que le faltan a la función para empezar a evaluarse. El closure PAP es de longitud dinámica (todos los demás closures tienen una configuración estática), y al ser evaluado simplemente introduce en las pilas los argumentos que almacena y salta al código de evaluación de la función que fue parcialmente aplicada a dichos argumentos (nuevamente, si el programa está correctamente tipado, antes de comenzar a evaluarse el PAP, se habrán introducido en las pilas los argumentos que faltan). Evidentemente, todo este esquema funciona gracias a que la convención de paso de argumentos estipula que los argumentos dispuestos más a la derecha (o sea, los capturados en una aplicación parcial) deben situarse en las posiciones más altas de las pilas, y el resto en posiciones progresivamente inferiores.

Como se ve, la forma de preparar y llevar a cabo las actualizaciones es muy distinta según el destino sea un constructor de datos o una PAP. No obstante, tenemos el problema de que estamos diseñando un back-end para lenguajes polimórficos en los que las funciones pueden ser manipuladas como datos. Por tanto, frecuentemente nos veremos en el caso de que una computación suspendida es polimórfica y por tanto no sabemos si la WHNF resultante de su evaluación será un constructor o una PAP. La solución más sencilla es mezclar ambos procesos de preparación para actualización: antes de evaluar un closure que debe ser actualizado, se introduce en la pila de marcos de actualización una referencia a dicho closure y los punteros de base de las pilas de argumentos, y además se introduce en la pila de actualizaciones la pseudocontinuación.

Cuando la WHNF finalmente es alcanzada y evaluada, se deben realizar consideraciones adicionales en cada posible tipo de WHNF:

- es un constructor: los punteros de base son restaurados (aunque esto no tenga ningún efecto)
- es una PAP: se saca la pseudocontinuación de la pila (ya que en este caso no tiene efecto, y hay que dejar la pila intacta).

Los closures son marcados como actualizables o no por el front-end. Como apunte, hacer notar que las funciones con argumentos no pueden ser actualizables (evidentemente a causa de que el valor al que son evaluadas depende de sus argumentos), mientras que las funciones sin argumentos (computaciones suspendidas) siempre pueden actualizarse, pero ello no es necesario si ya están en WHNF o el front-end puede probar que sólo son forzadas (evaluadas) una vez. En esto, la STG mejora mucho respecto a la G, pues esta última, salvo algunas optimizaciones, realiza ciegamente una actualización tras cada paso de reducción, con la consiguiente pérdida de rendimiento.

4. El compilador

Para un neófito, la estructura del documento que describe la STG, [PEY92], es un tanto confusa, sobre todo en lo que se refiere a la interpretación detallada de la semántica operacional de la máquina. Para una mejor comprensión del texto, empecé una implementación de un minicompilador, en realidad un back-end enormemente simplificado para una versión restringida de una STG. El compilador acepta lambda-cálculo enriquecido especialmente formateado y estructurado (similar al lenguaje "Core" que representa la entrada del back-end de GHC) y genera un texto en C con funciones y estructuras de datos para la evaluación del programa. Fui añadiendo características poco a poco (casi siempre sin terminar de afinar partes anteriores de la implementación), quedándome finalmente con una máquina STG operativa pero muy simple y sin recolección de basura.

Para escribir el compilador seguí la técnica de los esquemas de compilación, definiéndolos para las distintas estructuras y expresiones de mi versión del lenguaje "Core". El compilador fue creciendo incrementalmente, lo cual fijó algunas decisiones tempranas de diseño bastante malas y produjo en general un código bastante desordenado. La más destacable es que el entorno de compilación (un cajón de sastre que almacena información sobre aspectos del entorno de ejecución como el entorno de ligaduras visibles en cada momento o la disposición en memoria del contenido de cada closure, todo lo cual debe ser tenido en cuenta durante el proceso de compilación; en compiladores imperativos se correspondería con la tabla de símbolos) está fragmentado en múltiples estructuras de datos, que constituyen interminables ristras de argumentos pasados a las funciones que implementan los esquemas de compilación.

Es muy posible que el código hubiera quedado mucho más limpio agrupando tanto las estructuras de datos del entorno de compilación, como los resultados parciales de dicho proceso en una única superestructura de datos, y usar una mónada de transformadores de estado. Con esta solución, se podría tanto extraer datos del entorno monádico justo cuando fueran necesarios, como guardar resultados parciales de

compilación combinándolos con los ya generados, mejorando notablemente con todo ello a legibilidad del código. La conveniencia de usar una notación monádica es mayor por cuanto muchas de las fases de modificación de entornos de ligaduras y generación de código se conciben como una serie de transformaciones o modificaciones sucesivas sobre el entorno original.

Además, las distintas partes del entorno de compilación (disposiciones de "closures" y ligaduras de variables, particularmente) están implementadas como listas de elementos accedidas por identificador nominal, lo cual ralentiza mucho los accesos.

A grandes rasgos, la estructura del compilador es muy sencilla: Reconocimiento sintáctico -> compilación simbólica -> escritura del texto objeto:

- el texto fuente es reconocido por un parser cuasi-LL(1) ad-hoc, bastante tosco, ya que hemos preferido concentrarnos en el proceso de compilación en sí. Este parser no es capaz de proporcionar mensajes de errores léxico-sintácticos, y además su condición LL(1) ha empujado el diseño de la gramática a caracterizarse por gran cantidad de símbolos marcadores.

- El parser genera una representación simbólica del texto fuente, mediante una estructura de datos (tipo "Program") que expresa la colección de definiciones de lambda-cálculo. La estructura de datos más importante es "Expr", que representa una expresión de lambda-cálculo arbitrariamente compleja.

- El esquema de compilación principal ("pscheme" o esquema del programa) toma dicha representación, construye la parte básica del entorno de compilación (para las referencias a constructores, funciones de ámbito global y funciones predefinidas), y aplica el esquema de compilación de funciones ("lscheme" o esquema de lambdas) a cada definición global del programa, y uno especial a cada constructor de datos ("dscheme", o esquema para constructores). Devuelve una estructura de datos simbólica fácilmente traducible a texto en lenguaje C, que contiene las definiciones de estructuras de datos y funciones C definidas por los esquemas de compilación "dscheme" y "lscheme", además de definiciones de variables usadas como flags de depuración: HEAVYDEBUG controla si se ofrece información detallada de depuración antes de cada evaluación de una pieza de código, e IGNOREPAUSES si se permite que el usuario siga el proceso de depuración deteniéndolo hasta que se pulsa una tecla.

- En cada compilación de una función global o constructor de datos, se compilan representaciones simbólicas de las funciones y estructuras de datos C que representan el "closure" de cada función o constructor del lambda-cálculo (que son integradas por el esquema de compilación principal). En el caso de "lscheme", también se tiene que compilar el cuerpo de la función C que realiza la evaluación de la expresión que la define, mediante el esquema de compilación de expresiones ("escheme", o esquema de expresiones).

- El esquema de compilación de expresiones es el más complejo, y el más importante; representa el núcleo de todo el proceso de compilación. Puesto que los LETs pueden albergar definiciones locales de funciones, no sólo produce como resultado de compilación una representación simbólica del cuerpo de una

función C, sino más definiciones simbólicas en C de closures (como “lscheme”) para los LETs.

- Tras pasar por los esquemas de compilación, hemos obtenido la representación simbólica de un archivo fuente C, que transformamos fácilmente en texto C (una cadena de caracteres). En esta fase, las referencias a ligaduras (que han permanecido en forma simbólica durante la aplicación de los esquemas de compilación) son resueltas en referencias a variables, posiciones de pilas o expresiones puntero en lenguaje C.

Ofrecemos aquí la gramática del lenguaje definida en BNF (los símbolos no alfanuméricos usados en las producciones están entrecomillados para evitar confusiones):

```
%programa completo: lista de declaraciones de tipos y de funciones top-level
<prog> ::= {<datatypedef>} {<lambdadef>}

%definición de un tipo de datos: declara el nombre y los constructores
<datatypedef> ::= DATA nombreTipo "{" {<constrdef>}  "}"

%definición del constructor de un tipo de datos: declara el nombre y los argumentos (tipo de cada uno)
<constrdef> ::= CONSTR nombreConstructor {<kind>} ";"

%tipos de la máquina STG: valores primitivos y punteros a closures
<kind> ::= Pr | Pt

%definición de una función o CAF top-level: nombre, lista de ligaduras del closure,
%flag booleano que declara si es actualizable, lista de argumentos de la función, y
%expresión a la cual se evalúa
<lambdadef> ::= LAMBDA "{" nombreFuncion "\" {<var>} <updatable> {<var>} "->" <expr> "}"

%definición de una variable de ligadura STG para ligaduras de closure o argumentos:
%nombre de la variable y su tipo básico
<var> ::= nombreVariable ":" <kind>

%flags de actualización
<updatable> ::= True | False

%expresión STG
<expr> ::= <trace> | <varReturn> | <primitiveReturn> | <functionApplication>
       | <constructorApplication> | <let> | <primitiveLet>
       | <case> | <primitiveCase>

%expresión de traza/debug: muestra la traza por pantalla antes de evaluar la subexpresión
<trace> ::= OUTPUT "(" {<tracetoken>} ")" BEFORE <expr>

%opciones de traceado: pausa, valor de una variable, o cadena de caracteres
<tracetoken> ::= PAUSE | nombreVariable | literalEntrecomillado

%expresión que devuelve el valor de una variable primitiva o referencia a closure
<varReturn> ::= "$" nombreVariable

%expresión que devuelve el valor de un inmediato primitivo: entero o carácter
<primitiveReturn> ::= I entero | C caracterEntrecomilladoSimple

%expresión que devuelve el resultado de aplicar una función a unos argumentos
<functionApplication> ::= "[" nombreFuncion {<arg>} "]"

%argumento de aplicación de función o constructor de datos: variable o valor inmediato
%primitivo, entero o carácter
<arg> ::= <varReturn> | <primitiveReturn>
```

```
%expresión que devuelve el resultado de aplicar un constructor a sus argumentos
<constructorApplication> ::= "[" "%" nombreConstructor {<arg>} "]"

%expresión LET: indicador de LET normal o recursivo, lista de definiciones locales
%y subexpresión
<let> ::= LET {<localdef>} IN <expr>
      | LETREC {<localdef>} IN <expr>

%definición local LET: puede ser una función o variable local (definida del mismo
%modo que una top-level) o un constructor de datos
<localdef> ::= "(" DEF "{" nombreFuncion "\" {<var>} <updatable> {<var>} "->" <expr> "}" ")"
          | "(" DEFCONSTR nombreVariable "=" <constructorApplication> ")"

%definición local LET primitiva: liga una variable primitiva a una expresión aritmético-lógica
<primitiveLet> ::= LETPRIM nombreVariable "=" <primExpr> IN <expr>

%expresión primitiva aritmético-lógica: nombre de variable primitiva, inmediato entero, o
%expresión con operadores binarios o unarios
<primExpr> ::= nombreVariable | entero
          | "(" <primExpr> <binaryOp> <primExpr> ")"
          | "(" <unaryOp> <primExpr> ")"

%operador primitivo binario
<binaryOp> ::= "+" | "-" | "*" | "/" | "&&" | "||" | "==" | "!=" | ">" | ">=" | "<" | "<="
          | "&" | "|" | ">>" | "<<"

%operador primitivo binario
<unaryOp> ::= "!" | "-"

%expresión CASE: tipo de los datos escrutados (puede ser cualquiera si sólo hay rama DEFAULT),
%subexpresión escrutada, posible variable ligada al resultado de la expresión, ramas
%alternativas y posible rama por defecto
<case> ::= CASEDATA nombreTipo "." <expr> ["." BIND nombreVariable] OF "{"
        {<alter>} [<alterDefault>] "}"

%rama alternativa de un CASE: nombre del constructor, lista de ligaduras a los argumentos
%del constructor, y expresión evaluada en la rama
<alter> ::= "(" nombreConstructor {nombreVariable} "->" <expr> ")"

%rama por defecto de un CASE
<alterDefault> ::= "(" "$" DEFAULT "->" <expr> ")"

%expresión CASE primitiva: expresión escrutada, ramas alternativas y posible rama por defecto
<primitiveCase> ::= CASEPRIM <expr> OF "{" {<alterPrimitive>} [<alterPrimitiveDefault>] "}"

%rama alternativa de una CASE primitivo: número entero con el que se compara la expresión
%escrutada y expresión evaluada en la rama
<alterPrimitive> ::= "(" entero "->" <expr> ")"

%rama por defecto de un CASE primitivo: variable primitiva ligada al resultado de la
%expresión escrutada y cuerpo de la rama
<alterPrimitiveDefault> ::= "(" DEFAULT nombreVariable "->" <expr> ")"
```

Veamos un pequeño ejemplo para aclarar la gramática:

```
#Tipo de datos entero, con implementación semejante a la de GHC#
DATA Int { CONSTR MkInt Pr ; }
```

```
#  sumar :: Int -> Int -> Int
   sumar (MkInt a) (MkInt b) = MkInt (a+b)
#
LAMBDA {sumar \ | False x:Pt y:Pt ->
  CASEDATA Int . $ x OF {
    (MkInt xx -> CASEDATA Int . $ y OF {
      (MkInt yy -> LETPRIM res = (xx+yy) IN [ % MkInt $ res]
      )}
    )}
}

#  twice :: (a -> a) -> a -> a
   twice f x = f (f x)
#
LAMBDA {twice \ | False f:Pt x:Pt ->
  LET (DEF {aplicafx \ f:Pt x:Pt | True -> [f $ x] } )
  IN [f $ aplicafx]
}
```

5. Sistema en tiempo de ejecución (RTS)

Como ya se ha dicho, un programa en lambda-cálculo enriquecido para la STG es una colección de definiciones globales de funciones, con una semántica operacional directa en la máquina STG. Simularemos la STG sobre un entorno imperativo traduciendo en programa fuente a código C. Las tareas son múltiples: representar en C los closures, las pilas, las piezas de código, y el heap (la reserva de memoria donde se instancian los closures):

- Un closure se representa como una zona de memoria que contiene un puntero a una InfoTable y una carga que puede ser muchas cosas en función del tipo de closure:
  - -En una PAP, los argumentos de la aplicación parcial (y su cantidad).
  - -En un constructor de datos, sus argumentos.
  - -En una función con o sin argumentos, las variables ligadas a las que se hace referencia desde la expresión que la evalúa. En estos tres casos, por simplicidad de tratamiento, en el closure se disponen primero todos los valores primitivos y a continuación todas las referencias a otros closures.
  - -En una indirección, la referencia al closure destino (constructor o PAP)

  La InfoTable es una estructura de datos compartida por todos los closures de un mismo tipo, que contiene metadatos, información estadística y referencias a las distintas funciones C del closure: punto de entrada general para la evaluación del closure, punto de entrada para sin comprobación de número de argumentos (por eficiencia) funciones de comprobación de número, rutinas de depuración, de recolección de basura, etc.

- Para simular un sistema real (si bien notablemente simplificado), el heap es una zona contigua de memoria, en el cual el espacio se reserva secuencialmente (hay un puntero que indica el comienzo de la zona libre, que avanza conforme se reserva espacio para cada closure). Es el modelo de memoria más efectivo que se ha encontrado, especialmente cuando se tiene en cuenta la necesidad de implementar un algoritmo de recolección de basura eficiente. En nuestro sistema altamente simplificado el heap se organiza en celdas que pueden contener cualquier cosa (son uniones C): referencias a InfoTables, valores primitivos o referencias a closures.

- Cada una de las cuatro pilas es un array junto con una variable índice que hace las veces de puntero de cima (en el caso de las pilas V y S además hay punteros de base cuya función ya ha sido descrita).

- Las piezas de código que evalúan los closures son funciones C. Las funciones, antes de evaluarse, deben comprobar si tienen en las pilas el número requerido de argumentos. Esta comprobación es obviada cuando el compilador puede asegurar que es innecesaria (primordialmente, cuando se efectúa una aplicación de una función conocida), para lo cual se proporciona un segundo punto de entrada al código de evaluación que no realiza dicha comprobación. Ambos puntos de entrada son funciones C. ¿Cómo se reconcilian? Si el primero comprueba que hay suficientes argumentos para evaluar la función, termina saltando al segundo.

- ¿Cómo evitar que las sucesivas llamadas a las funciones C que representan las piezas de código desborden la pila de llamadas C? La solución es muy ingeniosa. La evaluación es controlada desde un bucle principal de la forma:

```
cont = main_evaluate;
while (1) {
        cont = (*cont)();
}
```

  En el que la variable "cont" es una referencia a una pieza de código, representada como una función C que no toma argumentos y devuelve una referencia a la siguiente pieza de código a ejecutar. Su valor inicial es la pieza de código que evalúa la función principal del programa STG, que no debe tener argumentos. De esta forma, la pila de llamadas C no puede desbordarse por esta causa.

El sistema de tiempo de ejecución (RTS) está formado por una serie de fuentes C que definen la infraestructura descrita:

- definiciones, funciones y macros para el tratamiento de cada una de las cuatro pilas y el heap
- definiciones de los registros especiales como variables globales (NODE, VRET y CONSTRTAG)
- definiciones de InfoTables para closures especiales como las indirecciones o las PAPs (implementando la semántica descrita a lo largo del documento)
- funciones aritméticas predefinidas sobre valores primitivos, inclusive la condicional (pero ninguna estrictamente necesaria, ya que todas pueden definirse directamente en nuestro lenguaje STG gracias a la expresión PRIMLET)
- funciones C auxiliares (por ejemplo, makePAP() construye un closure PAP, pues el procedimiento es siempre el mismo)
- la función main de C que ejecuta el programa STG (la que contiene el bucle principal descrito arriba)
- etc.

6. Más allá de esta implementación

Aunque no ha sido implementado, describiremos brevemente las particularidades del sistema de recolección de basura. La máquina STG no está casada

con ningún modelo en particular de recolector de basura, o característica del mismo: incremental, generacional, mark-scan, two-space, etc. El momento de comprobar si hay que recoger la basura es al principio de cada pieza de código en la que se reserve memoria para nuevos closures, como suele ser usual. El diseño de la máquina STG se nota más bien en la organización del recolector de basura: en la mayoría de las máquinas abstractas, se trata de una rutina o conjunto de rutinas bastante complejas, que trata cada bloque de memoria "interpretativamente", decidiendo qué hacer según de qué objeto se trate. En la STG, la función principal de recolección de basura se limita a recoger las raíces y a invocar las funciones de recolección de la InfoTable de cada closure (mediante estas funciones, cada closure "sabe" como salvarse a sí mismo de la recolección de basura). Con esto se consigue un tratamiento totalmente uniforme incluso para los casos especiales, como las indirecciones (que deben desaparecer durante la recolección de basura).

El proceso de recolección de basura puede incluso ser concurrente de forma transparente respecto al resto del sistema. De hecho, el modelo STG es fácilmente extendible (al menos en primera instancia) para incluir concurrencia, esto es, reducción concurrente e incluso paralela de varios redexes. La facilidad con la que el paradigma de reducción de grafos (donde caben la mayoría de las máquinas abstractas, incluyendo la G y la STG) se adapta al paradigma concurrente está muy bien descrito para el caso de la máquina G en el tutorial [PEY97]. A grandes rasgos, se puede decir que el paradigma se adapta fácilmente a la paralelización, ya que el grafo en memoria que representa la expresión a reducir (el programa a ejecutar) sirve a la vez como sistema de compartición de datos entre los procesos y mecanismo de sincronización. En la máquina STG, el grafo no está tan literalmente representado como en el caso de la G, pero aún así es posible usar los mismos métodos de concurrencia, con las ventajas que aporta la STG: para la comunicación entre procesos se pueden usar closures especiales.

Conceptualmente es bastante sencillo: cuando un proceso comienza a evaluar un redex, sobreescribe el closure que deberá actualizarse con el resultado de la reducción con un closure especial de sincronización. Si otro proceso necesita el resultado del primero, al saltar al closure, en realidad ejecuta la pieza de código del closure de sincronización, quedando dormido hasta que el primer proceso finalice y devuelva su resultado, momento en que se reanuda su ejecución. Incluso en el caso de una arquitectura con memoria distribuida (el modelo básico es bueno para memoria compartida, pero escala mal para modelos distribuidos), el diseño de la STG permite una implementación transparente para las referencias a closures en zonas no locales, mediante closures especiales de indirección a zonas de memoria no locales. Todo esto se propone con mayor lujo de detalles en [PEY92].

Asimismo, se ha de recordar que esta implementación es extremadamente primitiva tanto en el RTS como en el refinamiento de los esquemas de compilación y las pocas optimizaciones introducidas. En [PEY92] se proponen muchas optimizaciones. Yo mismo he ponderado dos propias (no observadas en nigún texto), si bien probablemente ambas de eficacia marginal:

- Cada vez que se reserva memoria para un constructor de datos, es casi seguro que una computación suspendida está esperando a ser sobreescrita con una indirección al mismo. Mediante algunos posibles mecanismos, el código que realiza la reserva podría chequear si se da el caso, y si el closure destino es lo

bastante grande, escribir el closure del constructor de datos directamente sobre el mismo en vez de reservar nueva memoria para él. Esta optimización potencialmente ralentiza la frecuencia con la que es necesario recolectar la basura, al reutilizar el espacio. Todo esto se describe muy brevemente en el mensaje de correo [FER04a].

- Una de las razones de ser de los closures en un lenguaje perezoso es que pueden ser actualizados con su resultado. Al introducir valores primitivos y la posibilidad de que haya expresiones (cómputos) que se evalúen a valores primitivos, tenemos el problema de que no los podemos actualizar con su resultado, pues éste no es un closure. ¿Cómo se puede solucionar esto? Estos closures se podrían sobreescribir con un tipo especial de closure, al modo de una indirección, que mantuviera una variable con el resultado primitivo de la evaluación, y cuyo código de evaluación simplemente devolviera dicho valor primitivo. Ahora bien, no está claro que esta optimización sea necesaria, pues puede conseguirse el mismo efecto usando un tipo de datos entero no primitivo (o sea, con un constructor de datos con un único argumento que sea el entero primitivo; o sea, el típico tipo "Int"). Todo depende de qué estilo de programas en lambda-cálculo STG genere el front-end del compilador (si se usan o no indiscriminadamente los tipos primitivos relajando el criterio de semántica totalmente perezosa). Esto se describe brevemente en el mensaje de correo [FER04b].

7. Síntesis: compilación de expresiones

Vamos a resintetizar lo descrito hasta ahora explicando brevemente el funcionamiento de los esquemas de compilación, explicándolos con un poco de más detalle.

El esquema "pscheme" ya ha sido convenientemente explicado: construye el entorno global con las funciones globales, los constructores de datos y las funciones aritméticas predefinidas en el RTS, y aplica los esquemas "dscheme" y "lscheme" a constructores de datos y funciones globales, respectivamente.

El esquema "dscheme" genera macros que declaran y definen los componentes del closure de cada constructor de datos: pieza de código de evaluación, estructura InfoTable del closure, etc.

El esquema "lscheme" compila el closure de una definición de función, ya sea local o global. Para ello, aumenta el entorno global con las ligaduras de variables almacenadas en el closure y de los argumentos de la función, y aplica "escheme" a la expresión que constituye el cuerpo de la función para generar la pieza de código que evalúa la función, más los posibles closures y continuaciones producto de las definiciones locales de funciones introducidas con LET. Si la función está marcada como actualizable, además se incluyen al principio de la pieza de código de evaluación (naturalmente, después de la comprobación del número de argumentos) instrucciones que preparan la actualización del modo descrito anteriormente:

- Se salva en la pila de marcos de actualización una referencia al closure de la función (obtenida del registro NODE) y los punteros base de las pilas de argumentos V y S
- Se salva la pseudocontinuación de actualización en la pila de continuaciones
- Se "dejan vacías" las pilas de argumentos igualando los punteros de base a los de cima, aunque naturalmente el contenido queda intacto por debajo, y sigue siendo accesible pues los punteros de cima no varían.

El esquema "escheme" es el más complejo, pues es el que realiza el "trabajo real", compilando las expresiones en código C que las evalúa. Describiremos someramente el proceso de compilación para cada expresión:

LET {…} IN subexpr

Compilar un LET tiene dos efectos:

- Se compilan las funciones locales y computaciones suspendidas definidas por el LET aplicando recursivamente el esquema "lscheme", y el resultado se añade al resultado del LET. No todas las definiciones locales son de este tipo, también se pueden definir variables que referencian constructores de datos.
- El código de evaluación de un LET es:

  -Para cada variable definida por el LET, se crea un nuevo closure y se rellena adecuadamente: primero una referencia a su InfoTable (que será compilada mediante el paso anteriormente descrito), luego las ligaduras de tipo primitivo y después las ligaduras de tipo puntero.

  -El código que realiza la tarea anterior se coloca delante del código de evaluación de la subexpresión, compilada recursivamente con "escheme". Es importante aplicarle a esta subexpresión un entorno de resolución de referencias a variables modificado que incluya las variables ligadas por el LET.

LETREC {…} IN subexpr

Un LETREC es idéntico a un LET excepto porque la reserva de memoria para los closures se realiza de una sola vez, de modo que en el momento de rellenarlos se puede disponer de sus punteros, de forma que se puede incluir en ellos referencias auto y co-recursivas en dichos closures a todos los definidos en el mismo LETREC. Por ejemplo, una función que devuelva una lista infinita cuyos elementos sean todos el mismo se puede definir así:

```
DATA List { CONSTR Cons Pt Pt ; CONSTR Nil ; }

#repeat :: a -> List a
8585  repeat item = repeat_auto where repeat_auto = Cons item repeat_auto
#
LAMBDA { repeat \ | False item:Pt -> LETREC
  (DEFCONSTR repeat_auto = [ % Cons $ item $ repeat_auto])
  IN $ repeat_auto
}
```

LETPRIM var = primexpr IN subexpr

Un LETPRIM se compila generando una nueva variable local C, a la cual asignamos valor traduciendo directamente la expresión primitiva (consistente en una combinación aritmética de otras variables de tipo primitivo) a una sentencia de asignación C. Después se anexa el código que evalúa la subexpresión, compilada con un entorno aumentado con la nueva ligadura de tipo primitivo.

CASE scrutexpr OF {…}

Compilar un CASE es algo más complicado. Se va a generar una función C adicional que representa la continuación del CASE tras la evaluación de la expresión forzada. En dicha continuación ya no será accesible el closure actual sino el del constructor al que se evalúe la expresión escrutada. Por tanto, se genera código para

- Salvar en las pilas de argumentos las variables residentes en el closure que sean referenciadas desde alguna de las alternativas del CASE
- Salvar en la pila de continuaciones una referencia a la continuación del CASE (la cual será cargada por el código de evaluación del constructor al cual se evalúe la expresión escrutada)
- Y finalmente se anexa el código de evaluación de la expresión escrutada, con el entorno de variables modificado para reflejar los cambios en las pilas de argumentos.

También se genera el código para la continuación. Es una función C con una gran sentencia switch:

- para discriminar las ramas alternativas se usa el registro CONSTRTAG, donde el constructor de datos habrá cargado su identificador.
- en cada alternativa se incluye el código de evaluación de la expresión correspondiente, con el entorno de resolución de variables modificado para tener constancia de la disposición de las pilas y los argumentos del constructor de datos correspondiente a la alternativa.
- la alternativa por defecto puede incluir el código de evaluación de la respectiva definida en el programa STG, o ser una autogenerada que compruebe errores en tiempo de ejecución en caso de CASE no saturado.

CASEPRIM scrutexpr OF {…}

Igual que la CASE excepto porque la expresión escrutada debe devolver un valor primitivo, por convención en el registro VRET, que será el registro comprobado en la sentencia switch de la continuación.

Aplicación de una función: nombreFunción var1 var2 … varn

El código de evaluación debe seguir los siguientes pasos:

- reconfigurar las pilas de argumentos V y S: hay que eliminar el contenido actual e introducir los nuevos argumentos. Puesto que algunos argumentos pueden estar ya en la pila, hay que tener cuidado en este paso, salvándolos temporalmente en variables locales.
- se asigna el registro NODE para que apunte a la pieza de código que evalúa la función especificada.

- Se salta a la pieza de código de evaluación de la próxima función (o sea, esta función C finaliza devolviendo una referencia a la misma). Una optimización consiste en que, si la función es conocida y además se comprueba que se le pasa un número suficiente de argumentos, sabemos que se puede obviar la comprobación de argumentos que realiza toda pieza de código de evaluación de una función al principio, saltando directamente al código de evaluación.

Aplicación de una variable o un valor inmediato

Se distinguen dos casos:

- una variable primitiva o valor inmediato primitivo se tratan exactamente de la misma forma, similarmente al código de evaluación de un constructor de datos:
  -se asigna al registro VRET el valor expresado por la variable o valor inmediato
  -se saca de la pila de continuaciones la referencia que está en la cima
  -se vacían las pilas de argumentos V y S
  -se salta a la continuación obtenida (o sea, esta función C finaliza devolviendo una referencia a la función C que implementa la continuación)

- una variable no primitiva, o sea, referencia a closure, se trata como una aplicación de función sin argumentos:
  -se asigna al registro NODE una referencia a dicho closure
  -se vacían las pilas de argumentos V y S
  -se salta a la pieza de código de evaluación de dicho closure. Si la variable es una referencia inmediata a una función sin argumentos, es posible obviar la comprobación del número de argumentos y se puede saltar directamente al código de evaluación

Aplicación de un constructor de datos: nombreConstructor var1 var2 … varn

Es como un cruce entre la evaluación de un LET y una aplicación de una variable por referencia. De hecho, se trata de azúcar sintáctica, ya que operacionalmente se corresponde precisamente con

```
LET x = nombreConstructor var1 var2 … varn IN x
```

8. Bibliografía

[FER04a] Mensaje de correo en el que expongo una pequeña optimización para la máquina STG
http://www.haskell.org//pipermail/haskell-cafe/2004-July/006460.html

[FER04b] Mensaje de correo en el que propongo otra optimización para la máquina STG, menos trivial que la anterior pero igualmente de eficacia no contrastada
http://www.haskell.org//pipermail/haskell-cafe/2004-July/006507.html

[MAR03] Simon Marlow y Simon Peyton Jones, "How to make a fast curry: push/enter vs eval/apply", 2003
http://research.microsoft.com/Users/simonpj/Papers/eval-apply/eval-apply.ps

[PEY87] Simon Peyton Jones y otros, "The Implementation of Functional Programming Languages", 1987
http://research.microsoft.com/Users/simonpj/Papers/slpj-book-1987/slpj-book-1987.pdf.gz

[PEY92] Simon Peyton Jones, "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine Version 2.5", 1992
http://research.microsoft.com/copyright/accept.asp?path=/users/simonpj/papers/spineless-tagless-gmachine.ps.gz&pub=34

[PEY97] Simon Peyton Jones y David R. Lester, "Implementing Functional Languages: a Tutorial", 1997
http://research.microsoft.com/Users/simonpj/Papers/pj-lester-book/student.pdf.gz

[PEY99] Simon Peyton Jones, Simon Marlow, "The STG runtime system (revised)", 1999
http://www.haskell.org/ghc/docs/papers/run-time-system.ps.gz

[RUIZ00] Blas Ruiz Jiménez y otros, "Razonando con Haskell", 2000, ISBN 84-607-1218-4

Página web de haskell: www.haskell.org

Página web de Simon Peyton Jones: http://research.microsoft.com/Users/simonpj