Facultad de Matemática, Astronomía y Física

Trabajo Especial

Generación de Código Intermedio Usando Semántica Funtorial

Autor:
Leonardo Rodríguez.
Directores:
Daniel Fridlender - Miguel Pagano.

29 de diciembre de 2010

Resumen

Este trabajo consiste en la implementación de un front-end para un lenguaje de programación Algol-like. El front-end es la primera etapa del proceso de compilación; cuyo objetivo es generar código en un lenguaje intermedio a partir del programa fuente.
La generación de código intermedio se realiza a partir de la semántica denotacional del lenguaje, es decir, se elige un modelo que permite pensar las ecuaciones semánticas como traducciones al lenguaje intermedio. El modelo semántico que se elige es una categoría funtorial que permite explicitar en las ecuaciones algunas propiedades deseadas del lenguaje. La implementación se realiza en Agda, un lenguaje funcional con tipos dependientes.

Clasificación:

F.3.2 - Semantics of Programming Languages - Denotational semantics.
F.4.1 - Mathematical Logic - Lambda calculus and related systems.

Palabras clave:

Código intermedio, Semántica denotacional , Categoría funtorial , Agda.

Agradecimientos

A mis directores, Daniel Fridlender y Miguel Pagano, que me han guiado y enseñado con pa*ciencia* durante todo el trabajo.

A mi familia por el apoyo que me brindaron durante toda la carrera.

A mis amigos de la facu, ¡ Por todo !

Índice general

Capítulo 1

Introducción

Un proceso de compilación consta básicamente de dos etapas: el front-end y el back-end. El front-end recibe el código fuente y produce una *representación intermedia* del programa, a partir de la cuál, el back-end produce un conjunto de instrucciones en código máquina.

El hecho de tener una representación intermedia permite independizar (en cierta medida) esas dos etapas de compilación. De esta manera, es posible construir más de un back-end para el mismo *lenguaje intermedio*, y cada uno de ellos puede generar código para diferentes máquinas. A su vez, para distintos lenguajes fuente, se puede construir un front-end que produce código en el mismo lenguaje intermedio.

En este trabajo definiremos un lenguaje de programación Algol-like, que llamamos *Peal* (Pequeño Algol), e implementaremos un front-end apropiado para ese lenguaje. Nos enfocaremos particularmente en la generación de código intermedio a partir de las ecuaciones semánticas del lenguaje.

Cuando se define la semántica denotacional de un lenguaje, lo que se está haciendo básicamente es asignar a cada tipo o frase del lenguaje un objeto (o elemento) en un modelo matemático que describe su significado. El modelo matemático podría ser, por ejemplo, un conjunto, un dominio, o una categoría. Una manera de generar código intermedio es elegir un modelo matemático particular que permita pensar a las ecuaciones semánticas como traducciones al lenguaje intermedio. En esta tesis utilizaremos un método para generar código intermedio, propuesto por Reynolds en [25], que parte de la elección de una *categoría* como modelo semántico y luego define las ecuaciones de traducción a un lenguaje intermedio particular. La generación de código utilizando categorías tuvo antecedentes como el trabajo de F.L Morris [19] que anticipó algunas de las ideas de Reynolds.

Una propiedad que tienen los lenguajes Algol-like es que respetan la *disciplina de pila*. Esta propiedad, que veremos más adelante, tiene que ver con un mecanismo de asignación (y liberación) de memoria que no requiere ninguna forma de recolección de basura (garbage collection). Reynolds [26] y Oles [21, 22] desarrollaron un modelo semántico usando *categorías funtoriales* que permiten hacer explícita la disciplina de pila en las ecuaciones semánticas. El método que aplicaremos en esta tesis es una variante de esa semántica funtorial que permite definir una traducción de un lenguaje Algol-like a un código intermedio.

La implementación del front-end se realizó en Agda [20] que es un lenguaje funcional con tipos dependientes (tipos que dependen de valores). La ventaja de usar tipos dependientes en la implementación, como veremos en el capítulo 6, surge naturalmente de la forma (o el tipo) que tienen las ecuaciones semánticas.

El capítulo 2 es una descripción de un modelo estándar de compilador. Describiremos brevemente el rol de cada fase de compilación mediante un ejemplo y luego presentaremos un entorno de ejecución simple para lenguajes con estructura de bloques.

En el capítulo 3, presentaremos el lenguaje *Peal* y su sistema de tipos. Además definiremos un conjunto de reglas de tipado que nos permitirán decidir cuándo una expresión del lenguaje está bien tipada.

En el capítulo 4 definimos algunos conceptos elementales de teoría de categorías que son relevantes en este trabajo.

En el capítulo 5 presentaremos la semántica denotacional (categórica) del lenguaje *Peal*, definiremos el lenguaje intermedio y mostraremos cómo generar código a partir de la semántica. En ese capítulo también explicaremos cómo es que la parametrización que brindan las categorías funtoriales nos permiten expresar la disciplina de pila en las ecuaciones semánticas.

En el capítulo 6 presentaremos la implementación de un front-end para *Peal*. El parser se implementó en Haskell y el typechecking junto con el generador de código se implementaron en Agda. En ese capítulo también explicaremos un método para representar secuencias infinitas de instrucciones y evitar la duplicación de código.

Capítulo 2

El Proceso de Compilación

En este capítulo presentamos un modelo básico de compilador, y hacemos una breve descripción de cada fase en las que se divide ese modelo. Luego presentamos un entorno de ejecución simple para un lenguaje con estructura de bloques y explicamos el concepto de disciplina de pila.
Para un estudio más detallado sobre compiladores referimos al lector a los libros [1, 2]. En el capítulo 6 se presentará la implementación de un *front end* (que incluye las fases desde el parser hasta el generador de código intermedio) para el lenguaje *Peal* .

2.1. Compiladores

Básicamente, un compilador lee un programa escrito en un lenguaje *fuente* y lo traduce a otro programa escrito en lenguaje *objeto*. El lenguaje objeto puede ser un lenguaje de programación, código máquina o alguna representación intermedia.

2.1.1. Contexto del compilador

El proceso llamado "compilación" usualmente involucra otros programas además del compilador:

- El preprocesador. Realiza una serie de procesamientos al programa fuente antes de ser compilado, por ejemplo

    - Expansión de macros.
    - Inclusión de archivos, si el programa contiene varios módulos en diferentes ubicaciones.
    - Traducción de extensiones del lenguaje fuente. Puede haber por ejemplo facilidades sintácticas (o "syntatic sugar") que deben traducirse a un conjunto de primitivas del lenguaje.

- El ensamblador. Traduce el programa objeto a un programa en código máquina. Este último programa puede requerir librerías externas que todavía no están enlazadas con el código.
  La salida del ensamblador es código máquina *reubicable* : sus instrucciones no hacen referencia a posiciones fijas en memoria, sino que se debe dar un offset o desplazamiento para determinar esas direcciones. Esto es útil si se quiere reutilizar el código en distintos espacios de direcciones de memoria.

- El enlazador o linker : Une las piezas reubicables de código y las librerías externas en un único espacio de direcciones, obteniendo así código máquina ejecutable.

2.2. Fases del compilador

Para describir las fases del compilador consideremos un pequeño lenguaje fuente que sólo permite secuencias de asignaciones, con posibles declaraciones de variables enteras o reales. Supongamos que la gramática del lenguaje es la siguiente:

$$
\begin{aligned}
\langle \text{exp} \rangle \ ::= \ & \langle \text{exp} \rangle \ \mathbf{:=} \ \langle \text{exp} \rangle \\
& \mid \langle \text{exp} \rangle \ ; \ \langle \text{exp} \rangle \\
& \mid \langle \text{exp} \rangle + \langle \text{exp} \rangle \\
& \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle \\
& \mid \langle \text{intconst} \rangle \\
& \mid \langle \text{realconst} \rangle \\
& \mid \langle \text{type} \rangle \ \langle \text{id} \rangle \\
& \mid \langle \text{id} \rangle
\end{aligned}
$$

$$\langle \text{type} \rangle \ ::= \ \mathbf{real} \mid \mathbf{int}$$

donde $\langle \text{id} \rangle$ es un conjunto predefinido de identificadores, $\langle \text{intconst} \rangle$ es el conjunto de constantes enteras y $\langle \text{realconst} \rangle$ es el conjunto de constantes reales.
Asumamos que se quiere compilar la secuencia

$$\mathbf{real}\, x := 1.0 \ ; \ \mathbf{real}\, y := 2.0 \ * \ x \ + \ 1,$$

que será la entrada de la primera fase del compilador.

2.2.1. Analizador Léxico

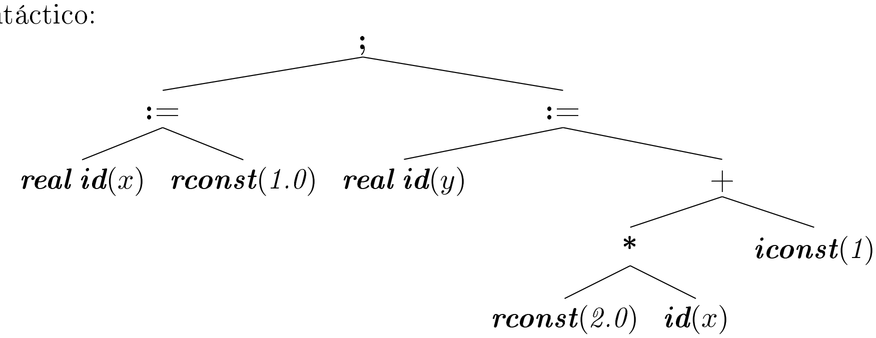
El analizador léxico es la primera fase del compilador. Recibe como entrada el programa fuente y produce una secuencia de *tokens*. Un token representa una secuencia de carácteres, como un identificador, una palabra reservada (como **int** o **real**) o un operador del lenguaje (+ o $*$). La secuencia de carácteres que forma el token se denomina *lexema*. Los siguientes son tokens y lexemas para nuestro lenguaje fuente:

Token	*Lexema*
id	x
:=	:=
+	+
*****	$*$
;	;
iconst	1
rconst	1.2
int	int
real	real

El analizador léxico produce la secuencia de tokens
real id(x) **:=** **rconst**(1.0) **; real id**(y) **:=** **rconst**(2.0) ***** **id**(x) $+$ **iconst**(1). Algunos de los tokens esperan argumentos, como en el caso de **rconst** que tiene asociada la constante correspondiente. En general, el analizador léxico descarta tanto los espacios en blanco que separan los lexemas como también los comentarios escritos en el programa fuente.

2.2.2. Analizador Sintáctico

El analizador sintáctico (o parser) recibe la secuencia de tokens desde el analizador léxico y produce un árbol sintáctico del programa, que se construye utilizando la gramática del lenguaje fuente. Por ejemplo, para la lista de tokens que recibe de la fase anterior, el parser produce el siguiente árbol sintáctico:



El parser debe ser capaz de aplicar "reglas de desambiguación" de la gramática. Por ejemplo, en el árbol anterior, se supone que $*$ tiene mayor precedencia que $+$.

2.2.3. Analizador Semántico

El analizador semántico recorre el árbol generado por el parser y realiza la verificación de tipos (*typechecking*), el análisis del alcance de las variables y la aplicación de coerciones (conversiones implícitas entre tipos).
Para realizar el typechecking es necesario mantener un mapeo de variables a tipos, llamado *contexto*. El dominio del contexto se extiende a medida que se declaran variables.
Utilizando gramáticas de van Wijngaarden podemos definir las construcciones bien tipadas del lenguaje fuente. Por ejemplo, si $\tau \in \{\mathbf{real}, \mathbf{int}\}$ y π es un contexto entonces podemos definir las expresiones de tipo τ como sigue:

$$\begin{aligned}
\langle \pi, \tau \rangle \; ::= \; & \tau\mathbf{const} \\
& \mid \mathbf{id}(x) \text{ si } \{x \mapsto \tau\} \in \pi \\
& \mid \langle \pi, \tau \rangle +_\tau \langle \pi, \tau \rangle \\
& \mid \langle \pi, \tau \rangle \,\mathbf{*}_\tau\, \langle \pi, \tau \rangle
\end{aligned}$$

Notar que los operadores están indexados por τ para indicar el tipo de sus argumentos. Un identificador x es una expresión de tipo τ si se puede encontrar en el contexto un mapeo $x \mapsto \tau$. La conversión de entero a real la agregamos con la producción

$$\langle \pi, \mathbf{real} \rangle \ ::= \ \mathbf{toreal}\ \langle \pi, \mathbf{int} \rangle.$$

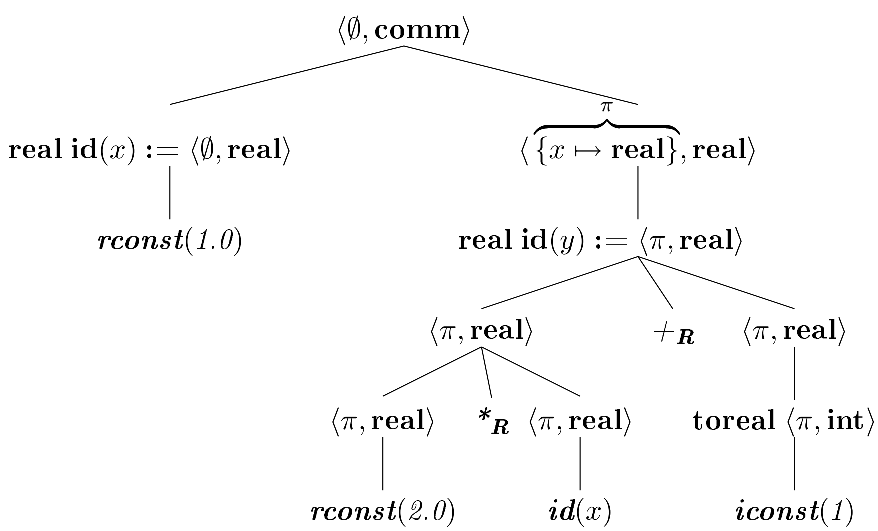
Introducimos el tipo **comm** para denotar asignaciones o declaraciones

$$\begin{aligned} \langle \pi, \mathbf{comm} \rangle \ ::= \ & \tau\ \mathbf{id}(x)\ [\,\mathbf{:=}\ \langle \pi, \tau \rangle\,] \text{ si } x \notin \pi \\ & |\ \mathbf{id}(x)\ \mathbf{:=}\ \langle \pi, \tau \rangle \text{ si } x \in \pi \end{aligned}$$

Con estas producciones estamos asumiendo que el lenguaje fuente no permite declarar variables repetidas, y sólo se puede usar una variable si se ha declarado antes. Los corchetes indican que la inicialización de la variable es opcional en la declaración.
Para el caso del operador "**;**" hay que tener en cuenta la extensión del contexto cuando se declara una variable:

$$\begin{aligned} \langle \pi, \mathbf{comm} \rangle \ ::= \ & \tau\ \mathbf{id}(x)\ [\,\mathbf{:=}\ \langle \pi, \tau \rangle\,]\ \mathbf{;}\ \langle \pi \cup \{x \mapsto \tau\}, \mathbf{comm} \rangle \text{ si } x \notin \pi \\ & |\ \mathbf{id}(x)\ \mathbf{:=}\ \langle \pi, \tau \rangle\ \mathbf{;}\ \langle \pi, \mathbf{comm} \rangle \text{ si } x \in \pi\,, \end{aligned}$$

La salida del analizador semántico es un árbol equipado con la información obtenida en el typechecking:



2.2.4. Generador de Código Intermedio

Luego del análisis semántico, algunos compiladores generan un código intermedio a partir del árbol de derivación. La ventaja de tener una representación intermedia es que entonces todas las fases vistas hasta ahora (incluída ésta) son independientes del código objeto.
El código intermedio puede pensarse como un programa para una máquina abstracta, en el que las variables son posiciones de memoria. El siguiente es el código intermedio generado a partir del árbol que produjo el analizador semántico:

$$
\begin{aligned}
&id1 := 1.0\\
&temp1 := 2.0 *_{\mathbf{R}}\ id1\\
&temp2 := \mathbf{toreal}\ 1\\
&id2 := temp1 +_{\mathbf{R}} temp2
\end{aligned}
$$

Cada asignación del lenguaje intermedio tiene a lo sumo un operador. Cuando se usan más operadores es necesario crear variables temporales para almacenar resultados intermedios.

2.2.5. Optimizador de Código

Antes de generar el código objeto puede ser conveniente mejorar el código intermedio para ahorrar espacio en memoria o aumentar la velocidad de ejecución (no necesariamente ambas).
Hay una gran variedad de optimizaciones de código entre los diferentes compiladores. Algunas de ellas pueden ser:

- Evitar recalcular expresiones comunes. Por ejemplo, el programa $x := 3 * j\ ;\ y := 3 * j$ puede escribirse como $t := 3 * j\ ;\ x := t\ ;\ y := t$

- Eliminación de código inalcanzable, es decir, que nunca llegará a ejecutarse debido a que, por ejemplo, la guarda de un **if** siempre será falsa.

- Cálculo previo de invariantes de ciclo. Si una expresión no cambia su valor durante la ejecución de un ciclo, entonces puede computarse antes del mismo. Por ejemplo, si c no modifica la variable z entonces

  $$\mathbf{while}\ (x \leq z - 2)\ \mathbf{do}\ c$$

  puede reemplazarse por

  $$t := z - 2\ ;\ \mathbf{while}\ (x \leq t)\ \mathbf{do}\ c$$

  siendo t una variable libre para c.

En nuestro caso, la salida del optimizador será la siguiente:

$$\begin{aligned} &id1 := 1.0 \\ &temp1 := 2.0 *_{\mathbf{R}}\ id1 \\ &id2 := temp1 +_{\mathbf{R}} 1.0 \end{aligned}$$

La conversión de entero a real de la constante 1 puede hacerse en tiempo de compilación, entonces la operación **toreal** puede eliminarse.

2.2.6. Generador de Código Objeto

La última fase del compilador es la generación de código objeto. En general, el código objeto consiste de instrucciones en un lenguaje ensamblador. Para cada variable que ocurre en el programa se selecciona una dirección de memoria (esta tarea no es trivial, ver la sección 2.3) y luego se traduce cada instrucción en el lenguaje intermedio a una secuencia de instrucciones en código objeto. En nuestro caso, la salida del generador de código es la siguiente:

$$\begin{aligned} &\text{MOVF } \#1.0, id1 \\ &\text{MOVF } id1, R1 \\ &\text{MULF } \#2.0, R1 \\ &\text{ADDF } \#1.0, R1 \\ &\text{MOVF } R1, id2 \end{aligned}$$

El primer y segundo operando de cada instrucción indican el origen y el destino de los datos, respectivamente. Por ejemplo, la primera instrucción mueve la constante 1.0 a la dirección $id1$. La F al final de cada instrucción indica que estamos operando con números flotantes.

2.2.7. Front-end y Back-end

Las fases que vimos en las subsecciones anteriores se agrupan en dos etapas, el *front-end* y el *back-end*. El front-end consiste de aquellas fases que dependen primordialmente del lenguaje fuente, y muy poco del lenguaje objeto. Normalmente esta estapa incluye las fases desde el análisis léxico hasta la generación de código intermedio. El back-end incluye las fases de optimización y generación de código, que dependen del lenguaje intermedio y del lenguaje objeto pero en general no dependen del lenguaje fuente.
En esta tesis se implementó un front-end para *Peal* donde la generación de código intermedio se realiza partiendo de la semántica denotacional (categórica) del lenguaje (ver capítulo 6).

2.3. Entorno de Ejecución

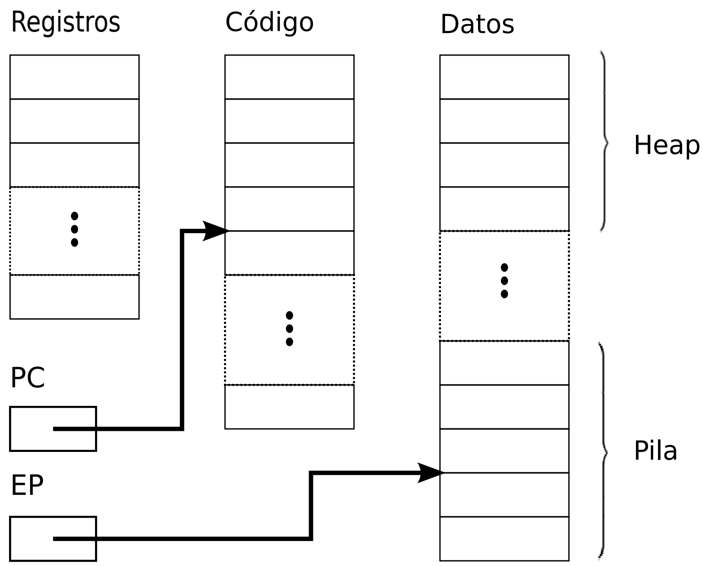
En esta sección presentamos un entorno de ejecución para un lenguaje con estructuras de bloques (basados en [18, cap 7]), que incluye los mecanismos de acceso y almacenamiento de variables en memoria. Utilizando este modelo de ejecución explicamos la disciplina de pila (*stack discipline*), que es una propiedad básica de los lenguajes Algol-like. En el capítulo 5 se utiliza un modelo de ejecución (propuesto en [25]) similar al de esta sección, pero con algunas optimizaciones que se mencionan más adelante.

2.3.1. Lenguajes con estructura de bloques

Algunos lenguajes de programación proveen alguna forma de *bloque*. Un bloque es un segmento de código, identificado con marcadores de inicio y final. Por ejemplo:

```
begin
    integer x;
    integer y;
    ...
    begin
        integer x;
        ...
    end
    ...
end
```

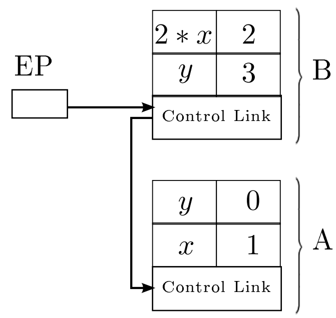
En este caso los bloques se identifican con los marcadores **begin** y **end**. Dos bloques distintos son disjuntos o uno está incluido en el otro (no se superponen parcialmente). Una variable que ocurre dentro de un bloque B es *local* en B si se declara en ese bloque, o *global* en B si se declara en un bloque externo A que contiene a B.
Se puede definir un entorno de ejecución simple para un lenguaje con estructuras de bloques. Consideremos primero un modelo de memoria en el que están separadas la memoria de código de la memoria de datos. Además, en esta última se distingue una región llamada Heap de otra región llamada Pila. La siguiente figura muestra este modelo de memoria:



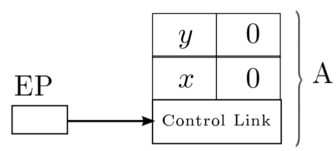
El registro *Program Counter* (PC) guarda la dirección de la instrucción actual, y se incrementa luego de cada instrucción. En este modelo de ejecución, cada vez que el programa entra en un bloque, se agrega a la pila un *stack frame* también llamado registro de activación en el que se guardan los datos de ese bloque, como son sus variables locales y los resultados temporales. El registro *Environment Pointer* (EP) apunta al nuevo stack frame que corresponde al bloque actual. Cuando el programa sale del bloque, su frame se retira de la pila y el EP vuelve a su valor anterior. El heap se utiliza para almacenar datos persistentes, cuya permanencia en memoria no depende del bloque que se está ejecutando.
A continuación ilustramos con un ejemplo cómo se ejecutaría un programa en este entorno. Consideremos el siguiente programa:

$$
\text{bloque A}\left\{\begin{array}{l}
\textbf{begin}\\
\qquad \textbf{integer } x;\\
\qquad \textbf{integer } y;\\
\qquad x := 1;\\
\qquad y := 0;\\
\text{bloque B}\left\{\begin{array}{l}
\textbf{begin}\\
\quad \textbf{integer } y;\\
\quad y := 2 * x + 1;\\
\textbf{end};
\end{array}\right.\\
\qquad x := y\\
\textbf{end}
\end{array}\right.
$$

Justo antes de terminar de ejecutar el bloque más interno B, la pila tiene los siguientes registros de activación:



Cuando el programa sale del bloque interno, el puntero EP se actualiza utilizando el valor de *Control link*, que apunta al frame A. Posteriormente, se elimina el frame B y se continúa con la ejecución del bloque externo A. Luego de ejecutar $x := y$ obtenemos:



Cabe notar que, durante la ejecución del bloque interno, se necesita acceso al frame A para obtener el valor de la variable global x. El mecanismo de acceso debe buscar por toda la lista de frames hasta encontrar la variable (en este caso sólo hay dos elementos en la lista)

2.3.2. Funciones

Las funciones pueden ser llamadas desde varios puntos del programa, por lo que el frame que corresponde a la llamada de una función puede ser más complicado. En principio, necesitamos que el frame tenga espacio para lo siguiente:

- La dirección de la primera instrucción a ejecutar cuando la función termina y retorna el control.
- La dirección donde se guardará el resultado de la función.
- Cada uno de los parámetros de la función.
- Las variables locales y temporales.

El cuerpo de la función puede tener también variables globales, por ejemplo en el siguiente programa (escrito en un lenguaje ilustrativo) la variable x es global en el cuerpo de g:

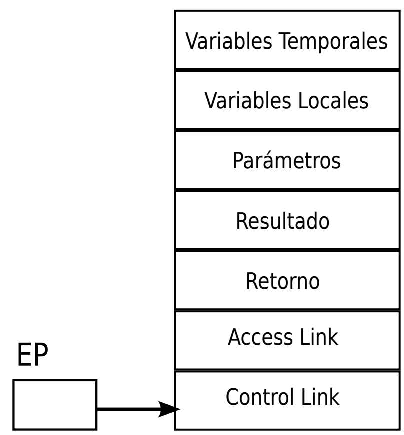
```
begin
    int x := 1;
    let g(z) be x + z in
        let f(y) be
            int x := y + 1;
            g(y * x)
        in
            f(3)
        end
    end
end
```

Notemos que en la ejecución de la llamada $g(y * x)$ hay dos posiciones de memoria que corresponden al identificador x. Esas posiciones están en distintos frames: una está en el frame global mientras que la segunda está en el frame correspondiete al cuerpo de f. De acuerdo a cuál posición se elija para calcular $y * x$ se clasifica el tipo de *alcance* que tiene el lenguaje. En general, para encontrar la declaración de una variable global hay dos opciones:

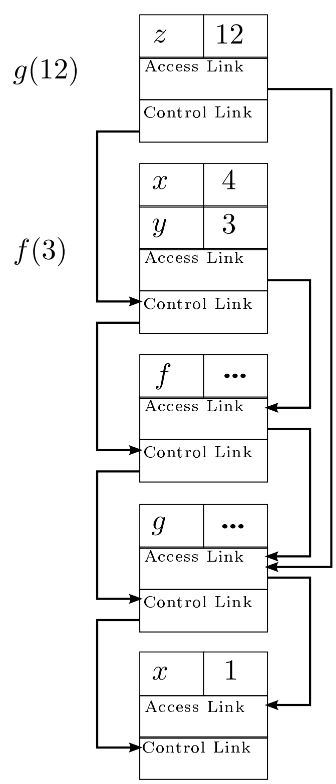
- Alcance estático:
  Un identificador global que ocurre en un bloque A refiere a la variable declarada en el bloque más interno que contiene a A.

- Alcance dinámico:
  El identificador global refiere a la variable asociada con el frame más reciente agregado a la pila.

Para el caso del alcance estático, el frame de una llamada a función necesitará un puntero llamado *Access Link* (enlace de acceso) para guardar la dirección del frame a partir del cuál se encuentran las variables globales que ocurren en esa función. La siguiente figura muestra el frame para el bloque de una llamada a función en un lenguaje con estructura de bloques y alcance estático:



La lista de frames resultante de la ejecución del programa anterior se muestra en la figura a continuación:



Durante la llamada $g(12)$ se crea un nuevo frame con la variable local z y el enlace de acceso apuntando al frame que corresponde a la declaración de g.

Siguiendo los enlaces de acceso, a partir del bloque de la llamada, se obtiene el valor de la variable global x.

2.3.3. Funciones de alto orden

Un lenguaje tiene funciones de alto orden, si las funciones pueden ser pasadas como argumento y devueltas como resultado de otras funciones. Consideramos primero un programa en el que pasamos una función f como argumento de otra función g:

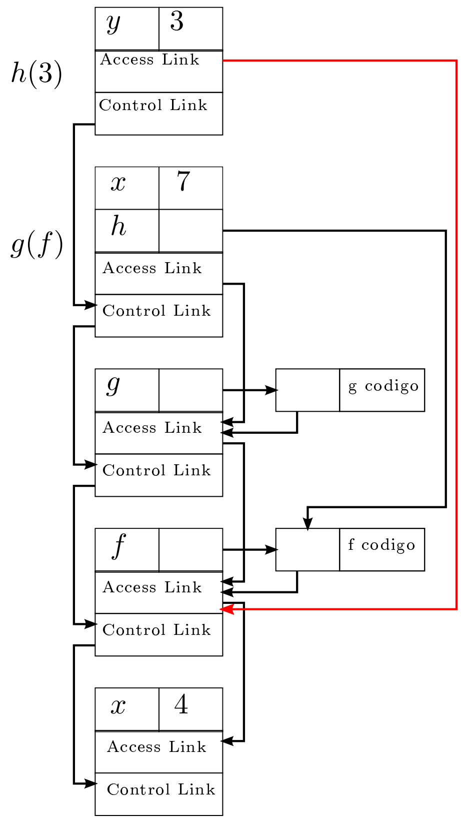
```
begin
    int x = 4;
    let f(y) be x * y in
        let g(h) be int x := 7; h(3) in
            g(f)
        end
    end
end
```

Utilizaremos una estructura de datos llamada *clausura* (closure) para asociar al parámetro h el código de la función f, y además enlazar correctamente los frames para mantener el alcance estático. La clausura consta de dos elementos:

- Un enlace de acceso que apunta a la lista de frames donde se ubican las variables globales de la función.

- Un puntero al código de la función.

A continuación se muestra el estado de la pila hasta la llamada $h(3)$ del programa anterior:



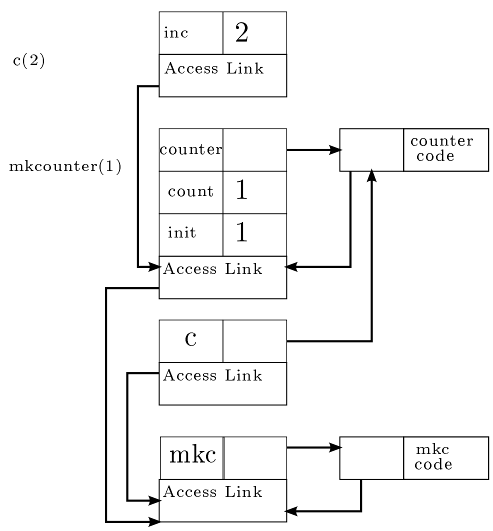
En la llamada a $g(f)$ se agrega el frame con el parámetro h y la variable local x. El parámetro h apunta a la clausura de f obteniendo así acceso al código y las variables globales de f. Luego, cuando se realiza la llamada a $h(3)$, se agrega un frame con el parámetro y y el enlace de acceso apunta al frame correspondiente a la declaración de f (esta información se conoce porque se tiene acceso a la clausura de f). De esta manera, cuando el programa solicita el valor de x en el bloque $h(3)$, se accede a la primer declaración de x y no a la segunda, respetando así el alcance estático.

Disciplina de Pila

En todos los ejemplos vistos hasta ahora, el modelo de ejecución respeta la disciplina de pila (*stack discipline*) : el último frame que se agrega es el primero que se elimina. Esto significa que cuando el programa sale de un bloque, como el frame correspondiente se elimina, el valor de las variables (locales o temporales) declaradas en él ya no tienen efecto sobre el comportamiento del programa. No todos los lenguajes tienen esta propiedad. En particular, si un lenguaje permite devolver funciones como resultado, es posible que no se respete la disciplina de pila. Consideremos por ejemplo el siguiente programa:

let mkcounter(init) **be**
  int count := init;
  let counter(inc) **be**
    count := count + inc; count
  in
    counter
  end
in
  int → **int** c := mkcounter(1);
  $c(2) + c(2)$
end

El estado de la pila en el comienzo de la primer llamada a $c(2)$ es la siguiente:



La llamada $c(2)$ necesita acceder a la variable *count* que fue declarada en el frame de mkcounter (1). Por lo tanto, no es posible eliminar de la pila ese frame, aún cuando la llamada ya ha finalizado. Entonces el frame de la llamada $c(2)$ se eliminará antes que el frame de mkcounter (1) y por lo tanto la disciplina de pila falla.

Nota sobre la implementación

El entorno de ejecución puede tener algunas optimizaciones que se realizan en el capítulo 5:

- Utilizar los registros para almacenar valores, en particular los resultados de funciones.
- No es necesario crear un frame por cada bloque, en particular uno puede agregar frames sólo al llamar funciones.
- No es necesario almacenar el nombre de la variable en el frame. Una forma de indexar las variables es mediante el número de frame y el desplazamiento desde la base.

Capítulo 3

El lenguaje *Peal*

En este capítulo definimos la síntaxis abstracta de *Peal* , y también presentamos el sistema de tipos que posee. Especificando una serie de reglas de inferencia podremos establecer cuáles son las frases del lenguaje que están bien tipadas. Como veremos en el capítulo 6, el typechecker se encarga de construir una "prueba" de que el programa es válido a partir de esas reglas de tipado. Cuando definamos la semántica del lenguaje (en el capítulo 5) veremos que las ecuaciones semánticas se definen sobre esas pruebas y no sobre las frases del lenguaje, lo que permite definir semántica sólo a frases bien tipadas.

3.1. Lenguajes Algol-like

El lenguaje de programación Algol 60 [4] tuvo una fuerte influencia en el diseño de los lenguajes de programación modernos. Una serie de lenguajes posteriores mantuvieron las características más importantes de Algol y dieron lugar al término "Algol-like" para referirse a esta familia de lenguajes. Reynolds caracterizó [26] a los lenguajes Algol-like con las siguientes propiedades:

1. El lenguaje se obtiene básicamente agregando al *lenguaje imperativo simple* [28, capítulo 2] un mecanismo de procedimientos basado en el cálculo lambda tipado con orden de evaluación normal.

2. Se distinguen dos clases de tipos en el lenguaje: los tipos de datos y los tipos de frases. Los tipos de datos (*data types*) denotan básicamente valores "almacenables", es decir, aquellos valores que pueden ser asignados a variables, o que son el resultado de evaluar ciertas expresiones. A las frases del lenguaje se las asocia con tipos (*phrase types*) que denotan valores "no almacenables" o con cierta estructura.

3. Las expresiones del lenguaje no pueden tener efectos secundarios, es decir, no pueden modificar el estado del programa. Se distinguen de los "statements" o comandos que si pueden alterar el estado.

4. El lenguaje respeta la disciplina de pila. Esta propiedad está relacionada con la estructura de bloques del lenguaje, y dice básicamente que las variables que se declaran en el interior de un bloque no tienen efecto una vez que finaliza la ejecución del mismo.

Bajo este criterio, algunos lenguajes como Algol W [5], Simula 97 [10] o Forsythe [27] son "Algol-like" pero Algol 68 [31] y Pascal [14] no lo son (aunque otros autores los han rotulado de esa manera). *Peal* es un lenguaje Algol-like, y es una extensión de un lenguaje definido por Reynolds en [25].

3.2. Tipos

Los sistemas de tipos comenzaron a utilizarse con los primeros lenguajes de programación de alto nivel, como Fortran, Cobol y Algol 60. El cálculo lambda tipado [8] fue fundamental para el desarrollo de los sistemas de tipos, tanto en lenguajes funcionales como imperativos.
Los tipos permiten detectar una amplia variedad de errores de programación en tiempo de compilación. Además, la información que proveen los tipos puede usarse para mejorar la representación o la manipulación de los datos (por ejemplo, el compilador puede optimizar código de acuerdo al tipo de las variables).
La definición del sistema de tipos de *Peal* se realiza en dos etapas. En la primera etapa se definen los tipos de datos (denotados por δ) que representan valores enteros, reales o booleanos:

$$\delta ::= \mathbf{int} \mid \mathbf{real} \mid \mathbf{bool}$$

A veces utilizaremos $\hat{\delta}$ para denotar los tipos de datos numéricos, es decir

$$\hat{\delta} ::= \mathbf{int} \mid \mathbf{real}$$

El tipo **int** puede considerarse un *subtipo* de **real**, ya que existe una manera de convertir valores enteros a valores reales. Podemos definir una relación de subtipado que capture la existencia de estas conversiones. En el caso de los tipos de datos la relación de subtipado es el orden parcial determinado por el siguiente diagrama:

$$\begin{tikzcd}
\mathbf{real} \arrow[d, dash] & \\
& \mathbf{bool} \\
\mathbf{int} &
\end{tikzcd}$$

En la segunda etapa se definen los tipos de frases, denotados por θ:

$$\begin{aligned}\theta ::= &\ \delta\mathbf{exp}\\ |&\ \delta\mathbf{acc}\\ |&\ \delta\mathbf{var}\\ |&\ \mathbf{compl}\\ |&\ \mathbf{comm}\\ |&\ \theta \to \theta\\ |&\ \theta \times \theta\end{aligned}$$

El sistema de tipos distingue entre "aceptores" ($\delta\mathbf{acc}$) y "expresiones" ($\delta\mathbf{exp}$). Típicamente, un aceptor ocurre del lado izquierdo de una asignación mientras que una expresión lo hace del lado derecho. Hay un tercer tipo para las variables ($\delta\mathbf{var}$) que pueden ocurrir en ambos lados de la asignación. Los comandos (**comm**) son frases que pueden "modificar el estado" del programa, como son las asignaciones, la declaración de variables y los ciclos. Las continuaciones (**compl**) son un tipo especial de comandos que "no devuelven el control".
Los tipos de frases también tienen una relación de subtipado. Semánticamente, $\theta \leq \theta'$ significa que hay una conversión de valores denotados por θ a valores denotados por θ'. Pero la relación de subtipado también puede interpretarse sintácticamente : $\theta \leq \theta'$ significa que una frase de tipo θ puede ser usada en cualquier contexto donde se requiera una frase de tipo θ'.
Uno puede considerar las expresiones enteras como un subtipo de las expresiones reales, es decir,

$$\mathbf{intexp} \leq \mathbf{realexp}\ .$$

Los aceptores, al contrario de lo que sucede con las expresiones, pueden ser asignados pero no pueden evaluarse. Si uno tiene un aceptor que recibe números reales, puede utilizarlo también para recibir números enteros, esto sugiere que

$$\mathbf{realacc} \leq \mathbf{intacc}\ .$$

A las variables las podemos ver como aceptores (les podemos asignar valores) y a su vez como expresiones (podemos "leer" el valor que almacenan). Esto se refleja en la relación de subtipado:

$$\begin{aligned}\delta\mathbf{var} &\leq \delta\mathbf{exp}\\ \delta\mathbf{var} &\leq \delta\mathbf{acc}\end{aligned}$$

Hasta aquí, la relación de subtipado entre los tipos de frases se puede graficar de la siguiente manera:

$$
\begin{tikzcd}
\textbf{intacc} \arrow[d, dash] \arrow[ddr, dash] & \textbf{realexp} \arrow[d, dash] \arrow[ddl, dash] & & \\
\textbf{realacc} \arrow[d, dash] & \textbf{intexp} \arrow[d, dash] & \textbf{boolacc} \arrow[dr, dash] & & \textbf{boolexp} \arrow[dl, dash] \\
\textbf{realvar} & \textbf{intvar} & & \textbf{boolvar} &
\end{tikzcd}
$$

Como mencionamos anteriormente, las frases de tipo **compl** son un tipo especial de comandos, lo que sugiere que

$$\textbf{compl} \leq \textbf{comm} \ .$$

El tipo $\theta \rightarrow \theta'$ denota un procedimiento que toma un parámetro de tipo θ y cuya aplicación es una frase de tipo θ'. Para ver la relación de subtipado entre procedimientos, supongamos que $\theta'_1 \leq \theta_1$ y que $\theta_2 \leq \theta'_2$. Entonces un procedimiento de tipo $\theta_1 \rightarrow \theta_2$ puede aceptar un parámetro de tipo θ'_1 ya que ese parámetro se puede convertir a tipo θ_1. Además, una aplicación del procedimiento puede tener tipo θ'_2 ya que el resultado de la aplicación se puede convertir de tipo θ_2 a tipo θ'_2. En otras palabras, respecto a la relación de subtipado, el operador $\rightarrow$ es antimonótono en el primer argumento y monónoto en el segundo. Por ejemplo, como **intexp** $\leq$ **realexp** tenemos:

$$
\begin{tikzcd}
 & \textbf{intexp} \rightarrow \textbf{realexp} \arrow[dl, dash] \arrow[dr, dash] & \\
\textbf{intexp} \rightarrow \textbf{intexp} \arrow[dr, dash] & & \textbf{realexp} \rightarrow \textbf{realexp} \arrow[dl, dash] \\
 & \textbf{realexp} \rightarrow \textbf{intexp} &
\end{tikzcd}
$$

En *Peal* , las frases que tienen tipo $\theta \times \theta'$ serán pares donde la primer componente será de tipo θ y la segunda componente de tipo θ'. En los productos binarios, la relación de subtipado se define componente a componente, es decir

$$\theta_1 \times \theta_2 \leq \theta'_1 \times \theta'_2 \quad \text{si y sólo si} \quad \theta_1 \leq \theta'_1 \ \text{ y } \ \theta_2 \leq \theta'_2$$

3.2.1. Reglas de inferencia de subtipos

A modo de resumen, daremos las reglas de inferencia de la relación de subtipado. El orden parcial $\leqslant$ se define inductivamente por las siguientes reglas:

- Reflexividad

$$\overline{\delta \ \leqslant \ \delta}$$

- Enteros a Reales

$$\overline{\mathbf{int} \ \leqslant \ \mathbf{real}}$$

Notar que en $\leqslant$ las propiedades de transitividad y antisimetría se deducen del hecho de que las anteriores son las dos únicas reglas que determinan el orden. Con $\leq$ denotamos el orden parcial entre los tipos de frases determinado por las siguientes reglas de inferencia:

- Reflexividad

$$\overline{\theta \ \leq \ \theta}$$

- Transitividad

$$\frac{\theta \leq \theta' \qquad \theta' \leq \theta''}{\theta \leq \theta''}$$

- Expresiones

$$\frac{\delta \leqslant \delta'}{\delta\mathbf{exp} \ \leq \ \delta'\mathbf{exp}}$$

- Aceptores

$$\frac{\delta \leqslant \delta'}{\delta'\mathbf{acc} \ \leq \ \delta\mathbf{acc}}$$

- Variables a expresiones

$$\overline{\delta\mathbf{var} \ \leq \ \delta\mathbf{exp}}$$

- Variables a aceptores

$$\overline{\delta\mathbf{var} \ \leq \ \delta\mathbf{acc}}$$

- Continuaciones a comandos

$$\overline{\mathbf{compl} \leq \mathbf{comm}}$$

- Productos

$$\frac{\theta_1 \leq \theta_2 \qquad \theta_1' \leq \theta_2'}{\theta_1 \times \theta_1' \leq \theta_2 \times \theta_2'}$$

- Procedimientos

$$\frac{\theta_1' \leq \theta_1 \qquad \theta_2 \leq \theta_2'}{\theta_1 \to \theta_2 \leq \theta_1' \to \theta_2'}$$

3.3. Sintáxis abstracta

Con la siguiente gramática definimos la sintáxis abstracta de *Peal* :

$$\begin{aligned}
\langle \text{phrase} \rangle ::= \; & \langle \text{const}_\delta \rangle \\
& \mid \langle \text{phrase} \rangle \langle \text{bop}_\delta \rangle \langle \text{phrase} \rangle \\
& \mid \langle \text{uop}_\delta \rangle \langle \text{phrase} \rangle \\
& \mid \langle \text{phrase} \rangle \langle \text{rel} \rangle \langle \text{phrase} \rangle \\
& \mid (\langle \text{phrase} \rangle \, , \, \langle \text{phrase} \rangle) \\
& \mid \langle \text{phrase} \rangle := \langle \text{phrase} \rangle \\
& \mid \langle \text{phrase} \rangle \, ; \, \langle \text{phrase} \rangle \\
& \mid \mathbf{skip} \\
& \mid \mathbf{if} \, \langle \text{phrase} \rangle \, \mathbf{then} \, \langle \text{phrase} \rangle \, \mathbf{else} \, \langle \text{phrase} \rangle \\
& \mid \mathbf{while} \, \langle \text{phrase} \rangle \, \mathbf{do} \, \langle \text{phrase} \rangle \\
& \mid \mathbf{loop} \, \langle \text{phrase} \rangle \\
& \mid \mathbf{newvar} \, \langle \text{id} \rangle \, \delta \, \mathbf{in} \, \langle \text{phrase} \rangle \\
& \mid \mathbf{let} \, \langle \text{id} \rangle \, \mathbf{be} \, \langle \text{phrase} \rangle \, \mathbf{in} \, \langle \text{phrase} \rangle \\
& \mid \mathbf{letrec} \, \langle \text{id} \rangle : \theta \, \mathbf{be} \, \langle \text{phrase} \rangle \, \mathbf{in} \, \langle \text{phrase} \rangle \\
& \mid \mathbf{escape} \, \langle \text{id} \rangle \, \mathbf{in} \, \langle \text{phrase} \rangle \\
& \mid \mathbf{fst} \, \langle \text{phrase} \rangle \\
& \mid \mathbf{snd} \, \langle \text{phrase} \rangle \\
& \mid \lambda \langle \text{id} \rangle : \theta \, . \, \langle \text{phrase} \rangle \\
& \mid \langle \text{phrase} \rangle \langle \text{phrase} \rangle \\
& \mid \langle \text{id} \rangle
\end{aligned}$$

$\langle \text{const}_{\mathbf{int}} \rangle ::= z \quad (z \in \mathbb{Z})$

$\langle \text{const}_{\mathbf{real}} \rangle ::= r \quad (r \in \mathbb{R})$

$\langle \text{const}_{\mathbf{bool}} \rangle ::= \mathbf{true} \mid \mathbf{false}$

$\langle \text{bop}_{\mathbf{int}} \rangle ::= + \mid - \mid * \mid \div \mid \mathbf{rem}$

$\langle \text{bop}_{\mathbf{real}} \rangle ::= + \mid - \mid * \mid /$

$\langle \text{bop}_{\mathbf{bool}} \rangle ::= \Rightarrow \mid \Leftrightarrow \mid \wedge \mid \vee$

$\langle \text{uop}_{\mathbf{int}} \rangle ::= -$

$\langle \text{uop}_{\mathbf{real}} \rangle ::= -$

$\langle \text{uop}_{\mathbf{bool}} \rangle ::= \sim$

$\langle \text{rel} \rangle ::= \leq \mid < \mid \geq \mid > \mid = \mid \neq$

$\langle \text{id} \rangle$ es un conjunto de identificadores

El lenguaje tiene aritmética entera, real y booleana. En el programa 3.1 declaramos tres variables (una para cada tipo de datos) y usamos algunos de los operadores binarios. Notar que el operador + puede utilizarse para sumar enteros o reales, es decir, es un operador polimórfico. En la segunda asignación, la expresión $2 * y$ es de tipo entero; pero por las reglas de subtipado, también es de tipo real.

$$\begin{array}{l}
\textbf{newvar}\ x\ \textbf{real in}\\
\quad \textbf{newvar}\ y\ \textbf{int in}\\
\quad\quad \textbf{newvar}\ b\ \textbf{bool in}\\
\quad\quad\quad y := y + 1\,;\\
\quad\quad\quad x := 2 * y + 8.0\\
\quad\quad\quad b := (x \geq y)\ \vee (y = 2)
\end{array} \tag{3.1}$$

El lenguaje *Peal* incluye los términos habituales del cálculo lambda (identificadores, abstracciones y aplicaciones). En el ejemplo 3.2 usamos **let** para definir un par donde la primer componente es una función (o abtracción) y la segunda es un comando. Luego usamos **fst** y **snd** para obtener las componentes del par.

$$\begin{array}{l}
\textbf{newvar}\ x\ \textbf{int in}\\
\quad\quad \textbf{let}\ p\ \textbf{be}\ (\lambda j\ : \textbf{intexp}.\ j + 1\,,\ x := x + 2)\ \textbf{in}\\
\quad\quad\quad\quad x := (\textbf{fst}\ p)\ x;\\
\quad\quad\quad\quad \textbf{snd}\ p
\end{array} \tag{3.2}$$

El comando "**escape** ι **in** c" crea una continuación ι que al ejecutarse produce una salida del comando interno c (lo interrumpe) y luego prosigue con el resto del programa. En el ejemplo 3.3, la continuación *break* se utiliza para salir del ciclo **loop** cuando se cumpla determinada condición.

$$\begin{array}{l}
\textbf{newvar}\ x\ \textbf{int in}\\
\quad \textbf{newvar}\ y\ \textbf{int in}\\
\quad\quad \textbf{escape}\ \mathit{break}\ \textbf{in}\\
\quad\quad\quad \textbf{loop}\\
\quad\quad\quad\quad \textbf{if}\ (x \leq y)\ \textbf{then}\ x := x + 1;\ y := y - 1\ \textbf{else}\ \mathit{break}\\
\quad\quad x := x + 1.
\end{array} \tag{3.3}$$

Las definiciones recursivas se realizan con **letrec**. En el ejemplo 3.4 definimos la función recursiva g que toma como parámetro una expresión booleana y un comando.

$$\begin{array}{l}
\textbf{letrec}\ g\ :\ \textbf{boolexp}\ \rightarrow\ \textbf{comm} \rightarrow \textbf{comm be}\\
\quad\quad (\lambda b : \textbf{boolexp}.\ \lambda c : \textbf{comm}.\ \textbf{if}\ b\ \textbf{then}\ c\ ;\ g\, b\, c\ \textbf{else skip}\ )\\
\textbf{in newvar}\ x\ \textbf{int in}\ g\ (x \leq 2)\ (x := x + 1)
\end{array} \tag{3.4}$$

3.4. Reglas de Tipado

Como es habitual en los sistemas de tipos, una afirmación de que una frase tiene cierto tipo se realiza bajo un *contexto* en donde se declaran los tipos de sus identificadores libres. Definimos un contexto como una función de dominio finito que asigna a cada identificador un tipo. Si π es un contexto, entonces $[\,\pi \mid \iota : \theta\,]$ es el contexto con dominio $dom(\pi) \cup \{\iota\}$ tal que

$$[\,\pi \mid \iota : \theta\,]\,\iota' = \begin{cases} \theta & \iota' = \iota \\ \pi\iota' & \iota' \neq \iota \ . \end{cases}$$

Si π es un contexto, p es una frase y θ es un tipo, entonces un *juicio*

$$\pi \vdash p : \theta$$

significa que la frase p tiene tipo θ cuando a sus identificadores libres se les asignan tipos según π.
Una prueba (o derivación) de que un juicio es válido se construye con las reglas de inferencia que listamos a continuación:

- Subsunción

$$\frac{\pi \vdash p : \theta \qquad \theta \leq \theta'}{\pi \vdash p : \theta'}$$

  Esta regla formaliza la idea de que si $\theta \leq \theta'$ entonces una frase de tipo θ es también una frase de tipo θ'

- Identificadores

$$\frac{}{\pi \vdash \iota : \pi(\iota)} \qquad \iota \in dom(\pi)$$

- Aplicación

$$\frac{\pi \vdash p_1 : \theta \to \theta' \qquad \pi \vdash p_2 : \theta}{\pi \vdash p_1\, p_2 : \theta'}$$

- Abstracción

$$\frac{[\,\pi \mid \iota : \theta\,] \vdash p : \theta'}{\pi \vdash (\lambda\iota : \theta \,.\, p) : \theta \to \theta'}$$

  El tipo explícito θ en la abstracción es necesario para simplificar el algoritmo de typechecking (verificación de tipos)

- Comandos condicionales

$$\frac{\pi \vdash b : \mathbf{boolexp} \qquad \pi \vdash c : \mathbf{comm} \qquad \pi \vdash c' : \mathbf{comm}}{\pi \vdash \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c' : \mathbf{comm}}$$

- Continuaciones condicionales

$$\frac{\pi \vdash b \,:\, \mathbf{boolexp} \qquad \pi \vdash c \,:\, \mathbf{compl} \qquad \pi \vdash c' \,:\, \mathbf{compl}}{\pi \vdash \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c' \,:\, \mathbf{compl}}$$

- Constantes

$$\frac{}{\pi \vdash \langle \mathrm{const}_\delta \rangle \,:\, \delta\mathbf{exp}} \qquad\qquad \frac{}{\pi \vdash \mathbf{skip} \,:\, \mathbf{comm}}$$

- Operadores Unarios

$$\frac{\pi \vdash e \,:\, \delta\mathbf{exp}}{\pi \vdash \ominus e \,:\, \delta\mathbf{exp}} \quad \ominus \in \langle \mathrm{uop}_\delta \rangle$$

- Operadores Binarios

$$\frac{\pi \vdash e_1 \,:\, \delta\mathbf{exp} \qquad \pi \vdash e_2 \,:\, \delta\mathbf{exp}}{\pi \vdash e_1 \oplus e_2 \,:\, \delta\mathbf{exp}} \quad \oplus \in \langle \mathrm{bop}_\delta \rangle$$

- Relaciones

$$\frac{\pi \vdash e_1 \,:\, \hat{\delta}\mathbf{exp} \qquad \pi \vdash e_2 \,:\, \hat{\delta}\mathbf{exp}}{\pi \vdash e_1 \oslash e_2 \,:\, \mathbf{boolexp}} \quad \oslash \in \langle \mathrm{rel} \rangle$$

- Composición secuencial

$$\frac{\pi \vdash c_1 \,:\, \mathbf{comm} \qquad \pi \vdash c_2 \,:\, \mathbf{comm}}{\pi \vdash c_1 ; c_2 \,:\, \mathbf{comm}}$$

$$\frac{\pi \vdash c_1 \,:\, \mathbf{comm} \qquad \pi \vdash c_2 \,:\, \mathbf{compl}}{\pi \vdash c_1 ; c_2 \,:\, \mathbf{compl}}$$

- Escape

$$\frac{[\pi \,|\, \iota : \mathbf{compl}] \vdash c \,:\, \mathbf{comm}}{\pi \vdash \mathbf{escape}\ \iota\ \mathbf{in}\ c \,:\, \mathbf{comm}}$$

- Comando **while**

$$\frac{\pi \vdash b \,:\, \mathbf{boolexp} \qquad \pi \vdash c \,:\, \mathbf{comm}}{\pi \vdash \mathbf{while}\ b\ \mathbf{do}\ c \,:\, \mathbf{comm}}$$

- Continuación **loop**

$$\frac{\pi \vdash c \,:\, \mathbf{comm}}{\pi \vdash \mathbf{loop}\ c \,:\, \mathbf{compl}}$$

- Definición no recursiva

$$\frac{\pi \vdash p : \theta \qquad [\pi \mid \iota : \theta] \vdash p' : \theta'}{\pi \vdash \mathbf{let}\ \iota\ \mathbf{be}\ p\ \mathbf{in}\ p' : \theta'}$$

- Definición recursiva

$$\frac{[\pi \mid \iota : \theta] \vdash p : \theta \qquad [\pi \mid \iota : \theta] \vdash p' : \theta'}{\pi \vdash \mathbf{letrec}\ \iota\ \mathbf{be}\ p\ \mathbf{in}\ p' : \theta'}$$

- Asignación

$$\frac{\pi \vdash a : \delta\mathbf{acc} \qquad \pi \vdash e : \delta\mathbf{exp}}{\pi \vdash a := e : \mathbf{comm}}$$

- Pares

$$\frac{\pi \vdash e : \theta \qquad \pi \vdash e' : \theta'}{\pi \vdash (e\,,\,e') : \theta \times \theta'}$$

$$\frac{\pi \vdash p : \theta \times \theta'}{\pi \vdash \mathbf{fst}\ p : \theta} \qquad\qquad \frac{\pi \vdash p : \theta \times \theta'}{\pi \vdash \mathbf{snd}\ p : \theta'}$$

- Declaración de variables

$$\frac{[\pi \mid \iota : \delta\mathbf{var}] \vdash c : \mathbf{comm}}{\pi \vdash \mathbf{newvar}\ \iota\ \delta\ \mathbf{in}\ c : \mathbf{comm}}$$

Capítulo 4

Categorias

En esta tesis aplicamos ciertos conceptos de teoría de categorías. En particular, en el capítulo 5 usaremos una categoría para describir la semántica del lenguaje. Básicamente, los tipos se interpretarán como objetos de la categoría, y los programas como morfismos.
En este capítulo hacemos una breve introducción a teoría de categorías, definimos conceptos básicos como *funtor* o *transformación natural* y damos algunos ejemplos concretos.
El propósito de este capítulo es que nuestro trabajo sea autocontenido, por lo que explicamos sólo los conceptos que son relevantes para el mismo. Para un estudio más detallado de teoría de categorías referimos al lector al libro "Category Theory" de Steve Awodey [3].

4.1. Definiciones básicas

Una *categoria* consiste de:

- Objetos : A, B, C, ...
- Morfismos : f, g, ...
- Para cada morfismo f hay objetos:

  $$dom(f),\ cod(f)$$

  llamados dominio y codominio de f. Se escribe:

  $$f : A \to B$$

  para indicar que $A = dom(f)$ y $B = cod(f)$
- Dados morfismos $f : A \to B$ y $g : B \to C$ hay un morfismo:

  $$g \circ f : A \to C$$

  llamado *composición* de f y g

- Para cada objeto A hay un morfismo:

$$1_A : A \rightarrow A$$

llamado *morfismo identidad* de A

y los morfimos deben satisfacer las siguientes propiedades:

- Asociatividad:

$$h \circ (g \circ f) = (h \circ g) \circ f$$

para todo morfismo $f : A \rightarrow B$, $g : B \rightarrow C$, $h : C \rightarrow D$

- Neutro de la composición:

$$f \circ 1_A = f = 1_B \circ f$$

para todo morfismo $f : A \rightarrow B$

Se denota $\mathbf{C}_0$ y $\mathbf{C}_1$ a la colección de objetos y a la colección de morfismos de la categoría $\mathbf{C}$ respectivamente.

Ejemplos. Categorias

- En la categoría **Sets**, los objetos son los conjuntos y los morfismos son las funciones totales entre conjuntos.
- Un *preorden* es un conjunto P equipado con una relación binaria $p \leq q$ que es reflexiva y transitiva. Todo *preorden* puede ser visto como una categoría tomando como objetos a los elementos de P y tomando un único morfismo, $a \rightarrow b$ cada vez que $a \leq b$. Las condiciones de reflexividad y transitividad son necesarias para probar que es una categoría. Un caso particular de preorden es un *poset*, en donde la relación binaria $\leq$ es antisimétrica además de reflexiva y transitiva.

Definición. *Isomorfismos*
En una categoría $\mathbf{C}$, un morfismo $f : A \rightarrow B$ es llamado *isomorfismo* si hay un morfismo $g : B \rightarrow A$ en $\mathbf{C}$ tal que

$$g \circ f = 1_A \text{ y } f \circ g = 1_B$$

Se puede probar que la inversa es única, por lo tanto podemos escribir $g = f^{-1}$. Cuando existe un isomorfismo entre A y B se dice que "A es *isomorfo* a B" o que "A y B son isomorfos"

Definición. *Objeto inicial y terminal*
En una categoría $\mathbf{C}$, un objeto

- 0 es *inicial* si para cualquier objeto C en $\mathbf{C}$ hay un único morfismo:

$$0 \rightarrow C$$

- 1 es *final* si para cualquier objeto C en $\mathbf{C}$ hay un único morfismo:

$$C \to 1$$

Proposición. *Un objeto inicial (final) es único salvo isomorfismo.*

Ejemplo (Objeto inicial y final)**.** En **Sets**, el conjunto vacío $\emptyset$ es inicial ya que para cada conjunto A hay una única función que va de $\emptyset$ en A. Si tomamos un singleton $\{*\}$ tenemos un objeto final ya que para todo conjunto A hay una única función $! : A \to \{*\}$ definida como $!\, x = *$.

4.2. Productos

En una categoría $\mathbf{C}$, un *producto* de objetos A y B consiste de un objeto P y morfismos

$$A \xleftarrow{p_1} P \xrightarrow{p_2} B$$

tales que, para cualquier diagrama de la forma

$$A \xleftarrow{x_1} X \xrightarrow{x_2} B$$

existe un único morfismo $u : X \to P$ tal que $x_1 = p_1 \circ u$ y $x_2 = p_2 \circ u$, es decir, para el cuál el siguiente diagrama conmuta:

$$\begin{tikzcd} & X \arrow[dl, "x_1"'] \arrow[d, dotted, "u"] \arrow[dr, "x_2"] & \\ A & P \arrow[l, "p_1"] \arrow[r, "p_2"'] & B \end{tikzcd}$$

Proposición. *El producto es único salvo isomorfismo*

El objeto P de la definición se escribe $A \times B$ y el morfismo u como $\langle x_1, x_2 \rangle$.

Ejemplos. Producto Cartesiano
En la categoría **Sets** para los conjuntos A y B tenemos el producto:

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$$

donde $A \times B$ es el producto cartesiano, y π_1, π_2 son las proyecciones usuales. Para cualquier conjunto X y funciones $f : X \to A$, $g : X \to B$ tenemos que el siguiente diagrama conmuta:

$$\begin{tikzcd} & X \arrow[dl, "f"'] \arrow[d, dotted, "\langle f, g \rangle"] \arrow[dr, "g"] & \\ A & A \times B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B \end{tikzcd}$$

donde $\langle f, g \rangle x = (f\, x, g\, x)$

4.2.1. Categorias con productos

Una categoría *tiene productos binarios* si tiene producto para cada par de objetos.
Supongamos que tenemos los siguientes objetos y morfismos de una categoría con productos binarios:

$$\begin{tikzcd} A & A \times A' \arrow[l, "p_1"'] \arrow[r, "p_2"] & A' \arrow[d, "f'"] \\ B \arrow[from=u, "f"'] & B \times B' \arrow[l, "q_1"'] \arrow[r, "q_2"] & B' \end{tikzcd}$$

Entonces definimos :

$$f \times f' : A \times A' \to B \times B'$$

como $f \times f' = \langle f \circ p_1, f' \circ p_2 \rangle$. Luego ambos cuadrados del siguiente diagrama conmutan:

$$\begin{tikzcd} A \arrow[d, "f"'] & A \times A' \arrow[l, "p_1"'] \arrow[r, "p_2"] \arrow[d, dotted, "f \times f'"] & A' \arrow[d, "f'"] \\ B & B \times B' \arrow[l, "q_1"'] \arrow[r, "q_2"] & B' \end{tikzcd}$$

La definición de producto dada anteriormente puede generalizarse a productos n-arios. El objeto terminal de una categoría puede pensarse como el producto de 0 objetos; y cualquier objeto A puede verse como el *producto unario* de A con él mismo. Además se puede probar que si una categoría tiene productos binarios, entonces tiene todos los productos n-arios con $n \geq 1$. Una categoría *tiene todos los productos finitos* si tiene objeto terminal y productos binarios.

4.3. Exponenciales

Sea $\mathbf{C}$ una categoría con productos binarios. Un exponencial de los objetos B y C consiste de un objeto

$$C^B$$

y un morfismo

$$\epsilon : C^B \times B \to C$$

tal que para cualquier objeto Z y morfismo

$$f : Z \times B \to C$$

hay un único morfismo

$$\hat{f} : Z \to C^B$$

tal que

$$\epsilon \circ (\hat{f} \times 1_B) = f \quad ,$$

diagramáticamente podemos expresar esto como:

$$
\begin{tikzcd}
C^B & C^B \times B \arrow[r, "\epsilon"] & C \\
Z \arrow[u, dotted, "\hat{f}"] & Z \times B \arrow[u, "\hat{f} \times 1_B"] \arrow[ur, "f"'] &
\end{tikzcd}
$$

Ejemplos. Exponenciales en **Sets**

Tomamos dos conjuntos B y C, y definimos C^B como el conjunto de funciones que van de B a C.

Además definimos

$$\epsilon : C^B \times B \to C$$

como

$$\epsilon(g, b) = g(b)$$

Para cualquier conjunto Z y función $f : Z \times B \to C$ podemos definir

$$\hat{f} : Z \to C^B$$

como

$$\hat{f}(z)(b) = f(z, b)$$

Y entonces vale

$$
\begin{aligned}
\epsilon \circ (\hat{f} \times 1_B)(z, b) &= \epsilon(\hat{f}(z), b) \\
&= \hat{f}(z)(b) \\
&= f(z, b)
\end{aligned}
$$

Definición. *Categorías Cartesianas Cerradas*

Una categoría es *cartesiana cerrada* si posee todos los productos finitos y exponenciales. La categoría **Sets** es un ejemplo de categoría cartesiana cerrada.

4.4. Categorias Funtoriales

Definición. *Funtores*

Un funtor

$$F : \mathbf{C} \to \mathbf{D}$$

entre las categorías **C** y **D** es un mapeo de objetos a objetos y morfismos a morfismos tal que

- $F(f : A \to B) = F(f) : F(A) \to F(B)$
- $F(g \circ f) = F(g) \circ F(f)$
- $F(1_A) = 1_{F(A)}$

Ejemplo. Dada una categoria $\mathbf{C}$, y objetos A, B en $\mathbf{C}$, definimos el conjunto

$$Hom_{\mathbf{C}}(A,B) = \{f \in \mathbf{C}_1 \mid f : A \to B\} \ ,$$

y si $g : B \to B'$ es un morfismo en $\mathbf{C}$ entonces definimos

$$\begin{aligned} &Hom_{\mathbf{C}}(A,g) : Hom_{\mathbf{C}}(A,B) \to Hom_{\mathbf{C}}(A,B') \\ &Hom_{\mathbf{C}}(A,g)\, f = g \circ f \end{aligned}$$

Veamos que $Hom_{\mathbf{C}}(A,-) : \mathbf{C} \to \mathbf{Sets}$ es un funtor. Es sencillo verificar que $Hom(A,1_B) = 1_{Hom(A,B)}$, ya que si $f : A \to B$ entonces tenemos

$$\begin{aligned} Hom_{\mathbf{C}}(A,1_B) &= 1_B \circ f \\ &= f \\ &= 1_{Hom_{\mathbf{C}}(A,B)}\, f \end{aligned}$$

Por otro lado, si $f : B \to C$, $g : C \to D$ y $h : A \to B$ son morfismos de $\mathbf{C}$ entonces tenemos

$$\begin{aligned} Hom_{\mathbf{C}}(A, g \circ f)\, h &= (g \circ f) \circ h \\ &= g \circ (f \circ h) \\ &= g \circ (Hom_{\mathbf{C}}(A,f)\, h) \\ &= Hom_{\mathbf{C}}(A,g)\,(Hom_{\mathbf{C}}(A,f)\, h) \\ &= (Hom_{\mathbf{C}}(A,g) \circ Hom_{\mathbf{C}}(A,f))\, h \end{aligned}$$

El funtor $Hom_{\mathbf{C}}(A,-)$ a veces lo escribiremos como $Hom_{\mathbf{C}}A$. Otra notación que usaremos frecuentemente es $A \underset{\mathbf{C}}{\to} B$ para denotar al conjunto $Hom_{\mathbf{C}}(A,B)$.

Definición. *Transformaciones naturales*
Dadas $\mathbf{C}$, $\mathbf{D}$ categorias, y los funtores

$$F, G : \mathbf{C} \to \mathbf{D}$$

una *transformación natural* $\vartheta : F \to G$ es una familia de morfismos en $\mathbf{D}$

$$(\vartheta_C : FC \to GC)_{C \in \mathbf{C}_0}$$

tal que para cualquier $f : C \to C'$ de $\mathbf{C}$, se tiene $\vartheta_{C'} \circ F(f) = G(f) \circ \vartheta_C$ es decir, el siguiente diagrama conmuta:

$$\begin{tikzcd} FC \arrow[r, "F(f)"] \arrow[d, "\vartheta_C"'] & FC' \arrow[d, "\vartheta_{C'}"] \\ GC \arrow[r, "G(f)"'] & GC' \end{tikzcd}$$

Ejemplo. Transformaciones naturales
Consideremos la categoría $\mathbf{Sets}^{op}$ cuyos objetos son conjuntos y cuyos morfismos $f : X \to Y$ son funciones de Y en X. Podemos definir un funtor $H : \mathbf{Sets}^{op} \to \mathbf{Sets}$ que asigna a un conjunto X el conjunto

$$H(X) = Hom_{\mathbf{Sets}}(X, \{0, 1\})$$

y a cada morfismo $f : X \to Y$ un morfismo $H(f) : H(X) \to H(Y)$ definido de la siguiente manera:

$$H(f)\,\phi = \phi \circ f$$

El funtor $\mathcal{P} : \mathbf{Sets}^{op} \to \mathbf{Sets}$ asigna a cada conjunto X el conjunto

$$\mathcal{P}\,X = \{S \mid S \subseteq X\}$$

y a cada morfismo $f : X \to Y$ un morfismo $\mathcal{P}(f) : \mathcal{P}(X) \to \mathcal{P}(Y)$ de la siguiente forma

$$\begin{aligned}\mathcal{P}\,(f)S &= \bigcup_{x\in S} f^{-1}x \\ &= f^{-1}(S)\end{aligned}$$

Para cada conjunto X sea $\vartheta_X : H(X) \to \mathcal{P}(X)$ definida como sigue:

$$\vartheta_X\,\phi = \{x \in X \mid \phi(x) = 1\} = \phi^{-1}(1)$$

podemos ver que:

$$\begin{aligned}(\mathcal{P}(f) \circ \vartheta_X)(\phi) &= \mathcal{P}(f)(\phi^{-1}(1)) \\ &= f^{-1}(\phi^{-1}(1)) \\ &= (f^{-1} \circ \phi^{-1})(1) \\ &= (\phi \circ f)^{-1}(1) \\ &= \vartheta_Y(\phi \circ f) \\ &= \vartheta_Y(H(f)\,\phi) \\ &= (\vartheta_Y \circ H(f))(\phi)\end{aligned}$$

y entonces el siguiente diagrama conmuta:

$$\begin{tikzcd} H(X) \arrow[r, "H(f)"] \arrow[d, "\vartheta_X"'] & H(Y) \arrow[d, "\vartheta_Y"] \\ \mathcal{P}(X) \arrow[r, "\mathcal{P}(f)"'] & \mathcal{P}(Y) \end{tikzcd}$$

y por lo tanto la familia de funciones ϑ_X es una transformación natural entre los funtores H y $\mathcal{P}$.

Definición. Categorías Funtoriales

Una categoría funtorial $\mathbf{D}^{\mathbf{C}}$ tiene como objetos a los funtores $F : \mathbf{C} \to \mathbf{D}$ y como morfismos a las transformaciones naturales $\vartheta : F \to G$.

Capítulo 5

Semántica y Generación de Código

5.1. Introducción

Definir la semántica denotacional de un lenguaje implica básicamente asignar a cada tipo y a cada frase del lenguaje distintas entidades (o elementos) en un modelo matématico que describa su significado. El modelo puede ser un conjunto, un dominio, o como en nuestro caso, una categoría.
En la primera parte del capítulo se muestra cómo dar semántica eligiendo como modelo una categoría. En particular, mostramos cómo al elegir una categoría funtorial con ciertas propiedades seremos capaces de definir la semántica denotacional de *Peal* de tal manera que se refleje su disciplina de pila explícitamente en las ecuaciones semánticas.
La segunda parte explica cómo generar código intermedio a partir de la semántica categórica de *Peal* . El método que se aplica es el que presenta John Reynolds en [25].

5.2. Semántica basada en categorías funtoriales

Sea Θ el menor conjunto que contiene los tipos de datos y los tipos de frases. Si equipamos a Θ con el orden parcial $\leq$ (relación de subtipado), obtenemos un poset. Por lo tanto podemos ver a Θ como una categoría que tiene como objetos a los tipos y tiene un único morfismo de θ a θ' cada vez que $\theta \leq \theta'$.
Para dar semántica a los tipos, asumimos que hay un funtor $[\![\_]\!]$ de Θ a una categoría cartesiana cerrada $\mathcal{K}$ llamada *categoría semántica*. El funtor asigna a cada tipo un objeto de $\mathcal{K}$ y a cada morfismo $\theta \leq \theta'$ un "morfismo de conversión implícita" denotado $[\![\ \theta \leq \theta'\ ]\!]$.
El funtor $[\![\_]\!]$ interpreta el constructor de tipos $\rightarrow$ como el operador de exponenciación en $\mathcal{K}$ (denotado por $\underset{\mathcal{K}}{\Rightarrow}$). Por otro lado, el constructor $\times$ es interpretado como

el producto en $\mathcal{K}$. Esto queda expresado en las siguientes ecuaciones:

$$\llbracket\, \theta \to \theta' \,\rrbracket = \llbracket\, \theta' \,\rrbracket^{\llbracket\, \theta \,\rrbracket} = \llbracket\, \theta \,\rrbracket \underset{\mathcal{K}}{\Rrightarrow} \llbracket\, \theta' \,\rrbracket$$
$$\llbracket\, \theta \times \theta' \,\rrbracket = \llbracket\, \theta \,\rrbracket \times \llbracket\, \theta' \,\rrbracket \ .$$

Sea Θ^* el conjunto de contextos con codominio incluido en Θ, es decir

$$\Theta^* = \{\pi \mid \pi \text{ es un contexto y para todo } \iota \in dom(\pi) \text{ se cumple } \pi\iota \in \Theta\}$$

Si consideramos el orden parcial sobre contextos

$$\pi \leq \pi' \quad \text{si y sólo si} \quad dom(\pi) = dom(\pi') \ \text{ y } \ \forall \iota \in dom(\pi) : \pi\iota \leq \pi'\iota$$

entonces Θ^* es un poset y puede verse como categoría. La semántica de un contexto es especificada por un funtor $\llbracket\_\rrbracket^*$ de Θ^* en $\mathcal{K}$, que asigna a cada contexto un producto en la categoría semántica:

$$\llbracket\, \pi \,\rrbracket^* = \prod_{\iota \in dom(\pi)}^{\mathcal{K}} \llbracket\, \pi\iota \,\rrbracket \tag{5.1}$$

Por último, consideramos la semántica de un juicio $\pi \vdash p : \theta$ como un morfismo entre la semántica de π y la semántica de θ, es decir

$$\llbracket\, p \,\rrbracket_{\pi,\theta} \ \in \llbracket\, \pi \,\rrbracket^* \underset{\mathcal{K}}{\to} \llbracket\, \theta \,\rrbracket$$

donde utilizamos la notación $A \underset{\mathcal{C}}{\to} B$ para denotar el conjunto de morfismos de A en B en la categoría $\mathcal{C}$.

Ejemplo

Como un ejemplo podemos considerar el caso $\mathcal{K} = \text{PDOM}$, la categoría de predominios y funciones continuas. En ese caso $\llbracket\, \theta \,\rrbracket$ es un predominio, $\llbracket\, \theta \leq \theta' \,\rrbracket$ es una función continua que va de $\llbracket\, \theta \,\rrbracket$ en $\llbracket\, \theta' \,\rrbracket$. El constructor de producto de tipos se interpreta como el producto cartesiano, es decir:

$$\begin{aligned}\llbracket\, \theta \times \theta' \,\rrbracket &= \llbracket\, \theta \,\rrbracket \times \llbracket\, \theta' \,\rrbracket \\ &= \{(x,y) \mid x \in \llbracket\, \theta \,\rrbracket \ \text{ y } \ y \in \llbracket\, \theta' \,\rrbracket\} \ .\end{aligned}$$

Por otro lado, un tipo $\theta \to \theta'$ se interpreta como un predominio de funciones de $\llbracket\, \theta \,\rrbracket$ en $\llbracket\, \theta' \,\rrbracket$, como lo indica la siguiente ecuación:

$$\begin{aligned}\llbracket\, \theta \to \theta' \,\rrbracket &= \llbracket\, \theta' \,\rrbracket^{\llbracket\, \theta \,\rrbracket} \\ &= \{\, f \mid \ dom(f) = \llbracket\, \theta \,\rrbracket \ \text{ y } \ codom(f) = \llbracket\, \theta' \,\rrbracket\} \ .\end{aligned}$$

La semántica de un contexto será el producto cartesiano de la semántica de todos los tipos en su dominio, es decir

$$\llbracket\, \pi \,\rrbracket^* = \prod_{\iota \in dom(\pi)}^{\text{PDOM}} \llbracket\, \pi\iota \,\rrbracket \ .$$

Se puede demostrar que $[\![\ \pi\ ]\!]^*$ es isomorfo al predominio

$$E(\pi) = \{\ \eta \mid dom(\eta) = dom(\pi) \text{ y para todo } \iota \in dom(\pi) \text{ se cumple } \eta\,\iota \in [\![\ \pi\iota\ ]\!]\ \}$$

cuyos elementos se denominan *entornos*. Por ejemplo, para el caso

$$dom(\pi) = \{\ \iota\ ,\ \iota'\ \}$$

se pueden definir las biyecciones

$$\begin{array}{ll} \phi : [\![\ \pi\,\iota\ ]\!] \times [\![\ \pi\,\iota'\ ]\!] \to E(\pi) & \phi^{-1} : E(\pi) \to [\![\ \pi\,\iota\ ]\!] \times [\![\ \pi\,\iota'\ ]\!] \\ \phi\ (x,y)\ \iota = x & \phi^{-1}\ \eta = (\ \eta\,\iota\ ,\ \eta\,\iota'\ ) \\ \phi\ (x,y)\ \iota' = y & \end{array}$$

La semántica de un juicio $\pi \vdash p : \theta$ es una función continua de $[\![\ \pi\ ]\!]^*$ en $[\![\ \theta\ ]\!]$, es decir

$$[\![\ p\ ]\!]_{\pi,\theta} \ \in [\![\ \pi\ ]\!]^* \to [\![\ \theta\ ]\!]\ ,$$

y por lo tanto podemos aplicar $[\![\ p\ ]\!]_{\pi,\theta}$ a un entorno y obtener un elemento en $[\![\ \theta\ ]\!]$, es decir

$$[\![\ p\ ]\!]_{\pi,\theta}\ \eta\ \in\ [\![\ \theta\ ]\!]\ .$$

Aprovechamos el ejemplo para introducir una notación para la extensión de entornos. Si $\eta \in [\![\ \pi\ ]\!]^*$ y $x \in [\![\ \theta\ ]\!]$ entonces $[\,\eta \mid \iota : x\,]$ es el entorno perteneciente a $[\![\ [\,\pi \mid \iota : \theta\,]\ ]\!]^*$ tal que

$$[\,\eta \mid \iota : x\,]\,\iota' = \begin{cases} x & \iota' = \iota \\ \eta\,\iota' & \iota' \neq \iota\ . \end{cases}$$

5.2.1. Categorias funtoriales y disciplina de pila

Habiendo planteado la idea de utilizar categorías como modelo semántico del lenguaje, elegiremos una categoría funtorial particular y luego la usaremos para definir ecuaciones semánticas de algunas frases del lenguaje (aunque la semántica completa la presentamos en la sección 5.3). Como veremos, la parametrización que nos brindan las categorías funtoriales nos permiten hacer explícita la disciplina de pila (cf.sec 2.3.3) en el lenguaje.

Sean $A_1, ..., A_n$ conjuntos. Un *store shape* $\langle A_1, ..., A_n\rangle$ es el conjunto de n-uplas

$$\langle \sigma_1, ..., \sigma_n \rangle$$

donde cada σ_i pertenece a A_i. Los elementos de un store shape se llaman *estados*. Un store shape S puede pensarse como predominio (de orden discreto). Un dominio es un predominio que tiene un elemento mínimo; dado el predominio S, denotamos $S_\perp$ al dominio que se obtiene de agregar un mínimo elemento a S:

$$x\ \leq_{S_\perp} y \text{ si y sólo si } x = y \text{ o } x = \perp\ ,$$

y denotamos $\iota_\uparrow : S \to S_\perp$ a la función de inyección que satisface $\iota_\uparrow\, x = x$.
El operador $+\!\!+$ denotará tanto la concatenación de store shapes como de estados. Sean S, S' store shapes y $\sigma \in S +\!\!+ S'$. Entonces denotamos $head_S\, \sigma \in S$ y $tail_{S'}\, \sigma \in S'$ a los únicos estados tales que

$$(head_S\, \sigma) +\!\!+ (tail_{S'}\, \sigma) = \sigma \;\; .$$

En lo que sigue usaremos la función $last_A : (S +\!\!+ \langle A \rangle) \to A$ que queda descripta por la siguiente ecuación:

$$last_A(\sigma +\!\!+ \langle x \rangle) = x \;\; .$$

Decimos que $S \leq S'$ si S es una subsecuencia de S'. Notemos que $\leq$ es un orden parcial, y por lo tanto el conjunto de store shapes es un poset.
Tomamos Σ como la categoría donde los objetos son los store shapes y cada vez que $S \leq S'$ definimos el morfismo

$$(head_S, G_{SS'}) : S \to S'$$

donde $G_{SS'} : (S \to S_\perp) \to (S' \to S'_\perp)$ se define

$$G_{SS'}\; c = \lambda\sigma'.(\lambda\sigma.\ \sigma +\!\!+ (tail_H\, \sigma'))_\perp(c\,(head_S\, \sigma'))$$

y $S' = S +\!\!+ H$. Definimos la composición de morfismos como sigue:

$$(head_S, G_{SS'}) \circ (head_{S'}, G_{S'S''}) = (head_S \circ head_{S'}, G_{S'S''} \circ G_{SS'})$$

Intuitivamente, el primer componente del morfismo nos dice cómo extraer un estado S de un estado más grande S', y el segundo componente nos dice cómo "simular" una transición entre estados pequeños en una transición entre estados más grandes. Como tenemos un único morfismo cada vez que S es subsecuencia de S', podemos usar también $S \leq S'$ para denotarlo (de hecho, se puede ver que hay un isomorfismo entre Σ y el poset determinado por $\leq$ visto como categoría).
Consideremos la categoría PDOM de predominios y funciones continuas, y elijamos como categoría semántica $\mathcal{K}$ a la categoría funtorial PDOM^Σ , es decir

$$\mathcal{K} = \mathrm{PDOM}^\Sigma \;\; .$$

Si θ es un tipo, $[\![\, \theta \,]\!]$ es un funtor que va de Σ en PDOM. Además, para cada juicio $\pi \vdash p : \theta$, tenemos que $[\![\, p \,]\!]_{\pi,\theta}$ es una transformación natural entre los funtores $[\![\, \pi \,]\!]^*$ y $[\![\, \theta \,]\!]$. Esa transformación natural tiene un componente para cada objeto S de Σ, denotado $[\![\, p \,]\!]_{\pi,\theta}\, S$. Este componente es una función contínua que va del predominio $[\![\, \pi \,]\!]^* S$ al predominio $[\![\, \theta \,]\!] S$, es decir

$$[\![\, p \,]\!]_{\pi,\theta}\, S \in [\![\, \pi \,]\!]^* S \to [\![\, \theta \,]\!] S$$

Para cada morfismo $S \leq S'$ de Σ se debe cumplir la condición de naturalidad,

$$[\![\, p \,]\!]_{\pi,\theta}\, S' \circ [\![\, \pi \,]\!]^*(S \leq S') = [\![\, \theta \,]\!](S \leq S') \circ [\![\, p \,]\!]_{\pi,\theta}\, S \;\; ,$$

es decir, el siguiente diagrama conmuta:

$$\begin{tikzcd}
\llbracket \pi \rrbracket^* S \arrow[r, "\llbracket \pi \rrbracket^*(S \leq S')"] \arrow[d, "\llbracket p \rrbracket_{\pi,\theta}\, S"'] & \llbracket \pi \rrbracket^* S' \arrow[d, "\llbracket p \rrbracket_{\pi,\theta}\, S'"] \\
\llbracket \theta \rrbracket S \arrow[r, "\llbracket \theta \rrbracket(S \leq S')"'] & \llbracket \theta \rrbracket S'
\end{tikzcd} \quad .$$

Definimos ahora la semántica de los tipos como sigue:

$$\begin{gathered}
A_{\mathbf{int}} = \mathbb{Z} \qquad A_{\mathbf{real}} = \mathbb{R} \qquad A_{\mathbf{bool}} = \mathbb{B} \\
\llbracket \mathbf{comm} \rrbracket S = S \to S_\perp \\
\llbracket \delta\mathbf{exp} \rrbracket S = S \to (A_\delta)_\perp \\
\llbracket \delta\mathbf{acc} \rrbracket S = A_\delta \to \llbracket \mathbf{comm} \rrbracket S \\
\llbracket \delta\mathbf{var} \rrbracket = \llbracket \delta\mathbf{acc} \rrbracket \times \llbracket \delta\mathbf{exp} \rrbracket
\end{gathered}$$

Hemos definido la acción del funtor $\llbracket \mathbf{comm} \rrbracket : \Sigma \to \mathrm{PDOM}$ en un store shape $S \in \Sigma$ como el predominio de funciones contínuas (pensadas como transiciones) que van del conjunto de estados S al conjunto $S_\perp$. De la misma manera, $\llbracket \delta\mathbf{exp} \rrbracket S$ es el predominio de funciones contínuas que reciben un estado en S y devuelven un valor en $(A_\delta)_\perp$. En el caso de los aceptores, una función perteneciente al predominio $\llbracket \delta\mathbf{acc} \rrbracket S$ recibe un valor en $(A_\delta)_\perp$ y devuelve una transición de estados pertenecientes a S. El funtor $\llbracket \delta\mathbf{var} \rrbracket$ es el producto entre los funtores $\llbracket \delta\mathbf{acc} \rrbracket$ y $\llbracket \delta\mathbf{exp} \rrbracket$. Más adelante definiremos la acción de estos funtores en los morfismos de la categoría Σ.
Para ver la ventaja de haber elegido PDOM^Σ como categoría semántica, consideremos la ecuación correspondiente al comando **newvar**:

$$\begin{aligned}
&\llbracket \mathbf{newvar}\ \iota\ \mathbf{int\ in}\ c \rrbracket_{\pi,\mathbf{comm}}\ S\, \eta\, \sigma \\
&\qquad = (head_S)_\perp(\llbracket c \rrbracket_{[\pi \mid \iota:\, \mathbf{intvar}],\mathbf{comm}}\ (S + \langle A_\delta \rangle)\ \hat{\eta}\ (\sigma\ +\ \langle 0 \rangle)) \ .
\end{aligned}$$

El comando c se evalúa en un store-shape extendido $S + \langle A_\delta \rangle$ que es el conjunto de estados donde la última componente (es decir, $\langle A_\delta \rangle$) guarda el valor de la variable ι. El entorno $\hat{\eta}$ es básicamente una extensión de η con la nueva variable. Definimos $\hat{\eta}$ en la ecuación 5.2 donde por ahora el lector puede omitir la ocurrencia de la función $\llbracket \pi \rrbracket^*(S \leq S + \langle A_\delta \rangle)$ que explicamos más adelante:

$$\hat{\eta} = [\llbracket \pi \rrbracket^*(S \leq S + \langle A_\delta \rangle)\eta \mid \iota : \langle a, e \rangle] \tag{5.2}$$

El par $\langle a, e \rangle$ asociado al identificador ι está compuesto por un aceptor $a \in \llbracket \mathbf{intacc} \rrbracket(S + \langle A_\delta \rangle)$ y una expresión $e \in \llbracket \mathbf{intexp} \rrbracket(S + \langle A_\delta \rangle)$ que se definen como sigue:

$$\begin{aligned}
a &= \lambda x.\ \lambda \sigma'.\ \iota_\uparrow((head_S\, \sigma') + \langle x \rangle) \\
e &= \lambda \sigma'.\ \iota_\uparrow(last_{A_\delta} \sigma') \ .
\end{aligned}$$

Básicamente, el aceptor a recibe un valor entero y modifica la componente $\langle A_\delta \rangle$ del estado con ese valor. La expresión e, por otro lado, recibe un estado y devuelve el valor de la componente $\langle A_\delta \rangle$ (donde se encuentra el valor de ι).
Al finalizar el comando c, la función $head_S$ se encarga de eliminar la variable y de extraer el store shape S. Es evidente que el valor de la variable ι no tiene efecto una vez finalizado el bloque de **newvar**. Esto nos dice que la disciplina de pila es explícita en la ecuación.
El entorno $\hat{\eta} \in [\![\, [\pi \mid \iota : \theta] \,]\!]^*(S + \langle A_\delta \rangle)$ no puede ser una extensión de $\eta \in [\![\, \pi \,]\!]^* S$ pero sí puede ser una extensión de un entorno perteneciente a $[\![\, \pi \,]\!]^*(S + \langle A_\delta \rangle)$. Pero entonces podemos "traducir" η usando la acción del funtor $[\![\, \pi \,]\!]^*$ en el morfismo $S \leq S + \langle A_\delta \rangle$, que es una función continua que va del predominio $[\![\, \pi \,]\!]^* S$ en el predominio $[\![\, \pi \,]\!]^*(S + \langle A_\delta \rangle)$.
Antes de definir la conversión $[\![\, \pi \,]\!]^*(S \leq S + \langle A_\delta \rangle)$, definimos para cada tipo básico θ la acción del funtor $[\![\, \theta \,]\!]$ en el morfismo $S \leq S'$ de la forma

$$(g : S' \to S \;\; , \;\; G : (S \to S_\perp) \to (S' \to S'_\perp)) \;\; ,$$

con las siguientes ecuaciones:

$$\begin{aligned}
&[\![\, \mathbf{comm} \,]\!] \, (g, G) = G \\
&[\![\, \delta\mathbf{exp} \,]\!] \, (g, G) \, e = e \circ g \\
&[\![\, \delta\mathbf{acc} \,]\!] \, (g, G) \, a = G \circ a \\
&[\![\, \delta\mathbf{var} \,]\!] \, (g, G) \, (a, e) = ([\![\, \delta\mathbf{acc} \,]\!] \, (g, G) \, a, [\![\, \delta\mathbf{exp} \,]\!] \, (g, G) \, e) \;\; ,
\end{aligned}$$

y luego extendemos punto a punto la definición para contextos; sea π un contexto entonces:

$$[\![\, \pi \,]\!]^* \, (g, G) \, \eta \, \iota = [\![\, \pi\iota \,]\!] \, (g, G) \, (\eta \, \iota) \;\; .$$

En [21] se demostró que PDOM^Σ es una categoría cartesiana cerrada (para cualquier categoría Σ). En particular, los exponenciales son funtores cuya acción en objetos de Σ es la siguiente:

$$(F \underset{\mathcal{K}}{\Rrightarrow} G) S = Hom_\Sigma S \times F \underset{\mathcal{K}}{\rightarrow} G$$

Para ver que el exponencial captura la interacción entre los procedimientos y la estructura de bloques, supongamos que

$$p \in [\![\, \theta_1 \to \theta_2 \,]\!] S = ([\![\, \theta_1 \,]\!] \underset{\mathcal{K}}{\Rrightarrow} [\![\, \theta_2 \,]\!]) S = (Hom_\Sigma S \times [\![\, \theta_1 \,]\!]) \underset{\mathcal{K}}{\rightarrow} [\![\, \theta_2 \,]\!]$$

es la semántica de un procedimiento de tipo $\theta_1 \to \theta_2$; entonces p es una transformación natural tal que

$$pS' \in (S \underset{\Sigma}{\rightarrow} S' \times [\![\, \theta_1 \,]\!] S') \to [\![\, \theta_2 \,]\!] S' \;\; .$$

Aquí, S es el store shape apropiado para el momento de la definición del procedimiento y contiene estados especificando los valores de las variables libres que ocurren en él. Por ejemplo, si p es la semántica del procedimiento *silly* en el programa

$$\begin{array}{l}\textbf{newvar}\ x\ \textbf{int in}\\ \quad \textbf{let}\ silly\ \textbf{be}\ \lambda c : \textbf{comm}.\ (c\,;\ x := x+1\,;\ c)\ \textbf{in}\\ \qquad \textbf{newvar}\ y\ \textbf{int in}\ silly(x := x+y)\end{array}$$

entonces S sería el store shape apropiado para un estado que especifique el valor de x. Sin embargo, como se ve en el programa, puede haber llamadas a *silly* desde un bloque más interno, donde el store shape apropiado S' contiene estados "más largos" con las nuevas variables declaradas. Entonces, tanto el parámetro del procedimiento *silly* como el resultado de la aplicación deben ser apropiados a S'. Es por eso que p necesita la información del morfismo en $S \xrightarrow[\Sigma]{} S'$ que indica cómo expandir un estado en S a un estado en S'.
Lo anterior nos sugiere la ecuación semántica de una abstracción,

$$[\![\, \lambda\iota\ :\ \theta.\ e\,]\!]_{\pi,\theta\to\theta'}\ S\,\eta\, S'(\varepsilon, z) = [\![\, e\,]\!]_{[\pi\,|\,\iota:\theta],\theta'}\ S'[[\![\, \pi\,]\!]^*\,\varepsilon\,\eta\,|\,\iota : z]$$

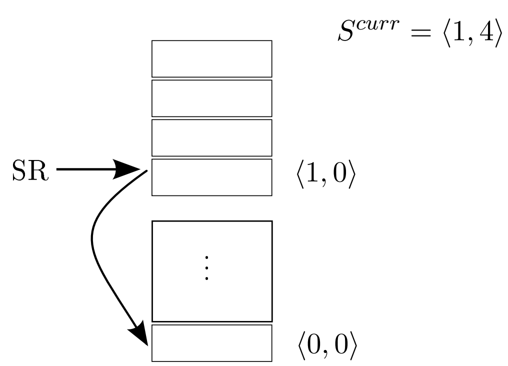
donde $S \leq S'$ y ε es una expansión de S en S'.

5.3. Generación de Código

En esta sección definimos un lenguaje intermedio y luego usamos la noción de semántica funtorial para generar código en ese lenguaje. Como veremos, esto último se logra viendo al lenguaje intermedio como un predominio (o como un objeto de PDOM) y por otro lado reemplazando la categoría Σ de store-shapes por la categoría de descriptores de pila (pares que identifican posiciones de memoria). De esta manera, las ecuaciones semánticas se podrán pensar como traducciones de *Peal* al lenguaje intermedio.

5.3.1. Descriptores de pila

Todas la información necesaria para la ejecución de un programa será almacenada en una secuencia de bloques contiguos llamados *frames* contenidos dentro de una pila.



Cuando esta secuencia tiene longitud n, denotamos a cada frame con su índice entre 0 y $n-1$ de acuerdo a la posición del frame contando desde la base hasta el tope de la pila. Los frames se organizan como una lista enlazada: un registro SR apunta a la base del frame tope y a su vez la primer celda de cada frame apunta al frame anterior.
El compilador conoce para cada variable cuál es el índice del frame S_f en donde se ubica y cuál es el desplazamiento S_d, que es la distancia desde la base del frame hasta la celda que contiene el valor de la variable. Este par de enteros no negativos $S = \langle S_f, S_d \rangle$ se denomina *descriptor de pila*.
El *descriptor de pila actual* denotado S^{curr} consiste en el par $\langle f, d \rangle$ donde f es el índice y d es el tamaño del frame tope. Notar que este descriptor indica cuál es la posición de la primer celda libre, que ocupará la siguiente variable creada por el programa.
Los descriptores de pila se ordenan lexicográficamente:

$$\langle S_f, S_d \rangle \leq \langle S'_f, S'_d \rangle \quad \text{si y sólo si} \quad S_f < S'_f \text{ o bien } (S_f = S'_f \text{ y } S_d \leq S'_d)$$

La pila puede crecer de dos maneras: agregando celdas en el último frame o bien insertando un nuevo frame en el tope de la pila. Estas dos operaciones se reflejan en un incremento del descriptor de pila actual. Cuando se agregar celdas nuevas, uno debe sumar un entero a la segunda componente del descriptor, por lo que definimos la operación suma (o resta) de la siguiente manera:

$$\langle S_f, S_d \rangle \pm \partial = \langle S_f, S_d \pm \partial \rangle$$

El primer paso para describir la generación de código intermedio a partir de la semántica denotacional del lenguaje es reemplazar la categoría Σ de store shapes por el conjunto ordenado de descriptores de pila (visto como una categoría, denotada también por Σ).

5.3.2. Código intermedio

El lenguaje intermedio será descripto por una gramática de van Wijngaarden. Esta gramática consiste de cuatro familias de no terminales indexadas por descriptores de pila: $\langle L_S \rangle$ denota registros que pueden situarse en el lado izquierdo de una asignación, $\langle R_S \rangle$ y $\langle S_S \rangle$ denotan expresiones y "expresiones simples" que pueden ir a la derecha de una asignación y por último $\langle I_S \rangle$ denota secuencias de instrucciones. La intención de indexar por descriptores a los no terminales es que, por ejemplo, un miembro de $\langle I_S \rangle$ es una secuencia de instrucciones que puede ser ejecutada si el descriptor de pila actual es S.

$$\begin{aligned}
\langle L_S \rangle \; &::= \; S^v \text{ cuando } S^v \leq S - 1 \\
&\;\;\;\;\; | \, \mathbf{sbrs} \\
\langle S_S \rangle \; &::= \; \langle L_S \rangle \; | \; \mathbf{lit}_\delta \; A_\delta
\end{aligned}$$

$$
\begin{aligned}
\langle R_S \rangle \;&::= \langle S_S \rangle \\
&\mid \langle \mathrm{uop}_{\hat\delta} \rangle \, \langle S_S \rangle \\
&\mid \langle S_S \rangle \, \langle \mathrm{bop}_{\hat\delta} \rangle \, \langle S_S \rangle \\
&\mid \textbf{toreal} \, \langle S_S \rangle \\
\langle I_S \rangle \;&::= \textbf{stop} \\
&\mid \langle L_{S+\partial} \rangle := \langle R_S \rangle [\partial];\, \langle I_{S+\partial} \rangle \\
&\mid \textbf{if} \, \langle S_S \rangle \, \langle \mathrm{rel} \rangle \, \langle S_S \rangle [\partial] \; \textbf{then} \, \langle I_{S+\partial} \rangle \; \textbf{else} \, \langle I_{S+\partial} \rangle \text{ cuando } S_d + \partial \geq 0 \\
&\mid \textbf{adjustdisp}[\partial];\, \langle I_{S+\partial} \rangle \text{ cuando } S_d + \partial \geq 0 \\
&\mid \textbf{popto} \, S';\, \langle I_{S'} \rangle \text{ cuando } S' \leq S
\end{aligned}
$$

Más adelante en este capítulo introducimos más construcciones del lenguaje intermedio. El registro **sbrs** se utiliza para guardar el resultado de la aplicación de un procedimiento y $\mathbf{lit}_\delta \; A_\delta$ son las constantes (o literales). El operador **toreal** se utiliza para realizar conversiones de valores enteros a reales.
Algunas instrucciones están seguidas de $[\partial]$ indicando que, luego de su ejecución, se debe incrementar (o decrementar) en ∂ el descriptor de pila actual, debido a la asignación o liberación de variables temporales o del programa. Este incremento es el único efecto que tiene la instrucción **adjustdisp**. La instrucción **popto** S' produce que el descriptor de pila actual tome el valor S'. Se utiliza para reducir el número de frames y causa un cambio en el registro SR.
El conjunto de instrucciones $\langle I_S \rangle$ es un dominio que incluye también las secuencias infinitas de instrucciones que serán necesarias para compilar ciclos y recursiones[1]. En el capítulo 6 mostramos una manera de representar estas secuencias infinitas.

5.3.3. De la semántica a la compilación

Para aplicar la semántica funtorial a la generación de código intermedio es necesario cambiar la semántica de los tipos, esta vez usando semántica de continuaciones. La traducción de una continuación será una secuencia de instrucciones; esto se refleja definiendo la acción del funtor $[\![\, \mathbf{compl} \,]\!] : \Sigma \to \mathrm{PDOM}$ en un descriptor S como el predominio de las secuencias de instrucciones:

$$[\![\, \mathbf{compl} \,]\!] \, S = \langle I_S \rangle \ .$$

En lo que sigue usaremos un funtor $\mathcal{R}$ cuya acción en objetos de Σ es el predominio de las expresiones $\langle R_S \rangle$, es decir:

$$\mathcal{R}\, S = \langle R_S \rangle$$

Usaremos además los siguientes tipos auxiliares[2] que denotan continuaciones enteras, reales y booleanas:

$$\theta ::= \delta\mathbf{compl} \ ,$$

[1] Estamos completando el conjunto que describe la gramática en el sentido de [29]

[2] Le llamamos tipos auxiliares debido a que no hay construcciones del lenguaje que puedan tener estos tipos.

y cuya traducción definimos como sigue:

$$[\![\ \hat{\delta}\mathbf{compl}\ ]\!] = \mathcal{R} \underset{\mathcal{K}}{\Rrightarrow} [\![\ \mathbf{compl}\ ]\!]$$
$$[\![\ \mathbf{boolcompl}\ ]\!] = [\![\ \mathbf{compl}\ ]\!] \times [\![\ \mathbf{compl}\ ]\!]$$

Una continuación entera es básicamente una continuación a la que debemos proveerle de una expresión entera; análogamente con las continuaciones reales. Como veremos luego, una continuación booleana es un par de continuaciones donde la primera se utiliza en el caso de que una expresión booleana sea verdadera y la segunda en el caso de que sea falsa.
La traducción del resto de los tipos se define en términos de los anteriores :

$$[\![\ \mathbf{comm}\ ]\!] = [\![\ \mathbf{compl}\ ]\!] \underset{\mathcal{K}}{\Rrightarrow} [\![\ \mathbf{compl}\ ]\!]$$
$$[\![\ \delta\mathbf{exp}\ ]\!] = [\![\ \delta\mathbf{compl}\ ]\!] \underset{\mathcal{K}}{\Rrightarrow} [\![\ \mathbf{compl}\ ]\!]$$
$$[\![\ \delta\mathbf{acc}\ ]\!] = [\![\ \mathbf{compl}\ ]\!] \underset{\mathcal{K}}{\Rrightarrow} [\![\ \delta\mathbf{compl}\ ]\!]$$
$$[\![\ \delta\mathbf{var}\ ]\!] = [\![\ \delta\mathbf{acc}\ ]\!] \times [\![\ \delta\mathbf{exp}\ ]\!]$$
$$[\![\ \theta \to \theta'\ ]\!] = [\![\ \theta\ ]\!] \underset{\mathcal{K}}{\Rrightarrow} [\![\ \theta'\ ]\!]$$
$$[\![\ \theta \times \theta'\ ]\!] = [\![\ \theta\ ]\!] \times [\![\ \theta'\ ]\!]$$

Como Σ es un preorden visto como categoría, se puede simplificar la definición del exponencial en $\mathcal{K}$. Vimos que si

$$p \in (F \underset{\mathcal{K}}{\Rrightarrow} G)S = (Hom_{\Sigma}S \times F) \underset{\mathcal{K}}{\to} G$$

entonces

$$pS' \in (S \underset{\Sigma}{\to} S' \times FS') \to GS'$$

Pero el conjunto $S \underset{\Sigma}{\to} S'$ tiene un único morfimo cuando $S \leq S'$ y es vacío cuando no. Entonces, una condición equivalente es que pS' pertenece a $FS' \to GS'$ cuando $S \leq S'$ y es la función vacía cuando no. Denotamos esto como

$$p(S' \geq S) \in FS' \to GS'$$

o bien

$$p(S' \geq S)(x \in FS') \in GS'$$

Toda frase en el lenguaje fuente se traduce a una secuencia de instrucciones o bien a una función que, aplicada en tiempo de compilación, produce secuencias de instrucciones. Más aún, el tipo de la frase en el lenguaje fuente determina el tipo de su traducción en el compilador.

5.3.4. Comandos

Si una frase c tiene tipo **comm** bajo el contexto π, entonces

$$[\![\ c\ ]\!]_{\pi,\mathbf{comm}} \in [\![\ \pi\ ]\!]^* \underset{\mathcal{K}}{\to} [\![\ \mathbf{comm}\ ]\!]$$

y por lo tanto

$$[\![\, c \,]\!]_{\pi,\mathbf{comm}} \, S(\eta \in [\![\, \pi \,]\!]^* S) \in [\![\, \mathbf{comm} \,]\!] S = ([\![\, \mathbf{compl} \,]\!] \underset{\mathcal{K}}{\Rightarrow} [\![\, \mathbf{compl} \,]\!]) S$$

luego

$$[\![\, c \,]\!]_{\pi,\mathbf{comm}} \, S(\eta \in [\![\, \pi \,]\!]^* S)(S' \geq S)(\kappa \in \langle I_{S'} \rangle) \in \langle I_{S'} \rangle$$

La función de traducción de c acepta un entorno η apropiado al descriptor S y una secuencia de instrucciones κ apropiada a un descriptor posiblemente mayor S' y retorna otra secuencia de instrucciones apropiada a S'. Como κ describe las instrucciones a ejecutar luego de c, es una *continuación* en el sentido usual. La traducción de **skip** retorna su continuación sin cambios:

$$[\![\, \mathbf{skip} \,]\!]_{\pi,\mathbf{comm}} \, S\eta S'\kappa = \kappa$$

Por otro lado, la traducción de $(c_1; c_2)$ es la traducción de c_1 usando una continuación que es la traducción de c_2 usando la continuación κ:

$$[\![\, c_1; c_2 \,]\!]_{\pi,\mathbf{comm}} \, S\eta S'\kappa = [\![\, c_1 \,]\!]_{\pi,\mathbf{comm}} \, S\eta S'([\![\, c_2 \,]\!]_{\pi,\mathbf{comm}} \, S\eta S'\kappa)$$

La traducción de una asignación está dada por la siguiente ecuación:

$$[\![\, a := e \,]\!]_{\pi,\mathbf{comm}} \, S\eta S'\kappa = [\![\, e \,]\!]_{\pi,\delta\mathbf{exp}} \, S\eta S'([\![\, a \,]\!]_{\pi,\delta\mathbf{acc}} \, S\eta S'\kappa)$$

5.3.5. Expresiones enteras y reales

Sea e una frase de tipo **intexp** bajo π, entonces

$$[\![\, e \,]\!]_{\pi,\mathbf{intexp}} \, S(\eta \in [\![\, \pi \,]\!]^* S)(S' \geq S)(\beta \in [\![\, \mathbf{intcompl} \,]\!] S') \in \langle I_{S'} \rangle$$

donde β es una continuación entera tal que

$$\beta(S'' \geq S')(r \in \langle R_{S''} \rangle) \in \langle I_{S''} \rangle$$

El descriptor S indica el segmento de pila con las variables de programa que pueden ser accedidas durante la evaluación de e. Por otro lado, S' incluye además posibles variables temporales creadas antes de comenzar la traducción. Al descriptor S' tenemos que añadirle las variables temporales necesarias para calcular el valor de e; el descriptor resultante será el primer argumento S'' de la continuación entera β.

Se puede pensar entonces a β como una continuación a la que necesitamos "proveerle" de una expresión r (que en este caso denotará el valor de e) y además un descriptor S'' que indica el segmento de la pila que contiene las variables temporales que ocurren en dicha expresión.

Para el caso de las constantes no necesitamos variables temporales, entonces directamente proveemos a β con S' y esa constante

$$[\![\, 42 \,]\!]_{\pi,\mathbf{intexp}} \, S\eta S'\beta = \beta S'(\mathbf{lit}_{\mathbf{int}} 42)$$

Por otro lado, para el caso de los operadores unarios, la traducción de $-e$ se obtiene evaluando e con una continuación modificada β'

$$[\![-e]\!]_{\pi,\mathbf{intexp}}\, S\eta S'\beta = [\![e]\!]_{\pi,\mathbf{intexp}}\, S\eta S'\beta'$$

Si la expresión r que se obtiene al evaluar e no contiene operadores (es una expresión simple) entonces proveemos a β con $-r$ sin crear variables temporales adicionales

$$\beta' S'' r = \beta S''(-r)$$

Si r no es una expresión simple, entonces

1. Computamos el valor de r
2. Descartamos las variables temporales de S'' que están por encima de S', es decir, las que ocurren en r.
3. Guardamos el valor de r en una nueva variable temporal, que se ubicará en S'
4. Finalmente proveemos a β con la expresión $-S'$ y el descriptor $S'+1$.

Esto queda resumido en la ecuación:

$$\beta' S'' r = S' := r[S'_d + 1 - S''_d]\, ;\, \beta(S'+1)(-S')$$

Podemos abstraer el uso de las variables temporales con la siguiente función:

$$\text{usetmp}\, S'\, \beta\, S''\, r = \begin{cases} \beta S'' r & \text{cuando } r \in \langle S_{S''} \rangle \\ S' := r[S'_d + 1 - S''_d]\, ;\, \beta(S'+1)S' & \text{cuando } r \notin \langle S_{S''} \rangle \end{cases}$$

Entonces podemos escribir la ecuación de $-e$ de la siguiente manera:

$$[\![-e]\!]_{\pi,\mathbf{intexp}}\, S\eta S'\beta = [\![e]\!]_{\pi,\mathbf{intexp}}\, S\eta S'(\text{usetmp } S'(\lambda S''.\lambda r.\, \beta S''(-r)))$$

Ahora definimos la traducción de los operadores binarios

$$\begin{aligned} &[\![e_1 \oplus e_2]\!]_{\pi,\hat{\delta}\mathbf{exp}}\, S\eta S'\beta = \\ &\qquad [\![e_1]\!]_{\pi,\hat{\delta}\mathbf{exp}}\, S\eta S'(\text{usetmp } S'(\lambda S''.\lambda r_1. \\ &\qquad\qquad [\![e_2]\!]_{\pi,\hat{\delta}\mathbf{exp}}\, S\eta S''(\text{usetmp } S''(\lambda S'''.\lambda r_2.\, \beta S'''(r_1 \oplus r_2))))) \end{aligned}$$

donde $\oplus \in \langle \text{bop}_{\hat{\delta}} \rangle$

Consideremos la conversión de expresiones enteras a expresiones reales. Tenemos que $[\![\mathbf{intexp} \leq \mathbf{realexp}]\!]$ es una transformación natural del funtor $[\![\mathbf{intexp}]\!]$ en el funtor $[\![\mathbf{realexp}]\!]$. Por lo tanto, cada componente $[\![\mathbf{intexp} \leq \mathbf{realexp}]\!]\, S$ es una función continua del predominio $[\![\mathbf{intexp}]\!]\, S$ al predominio $[\![\mathbf{realexp}]\!] S$, que definimos acontinuación:

$$\begin{aligned} &[\![\mathbf{intexp} \leq \mathbf{realexp}]\!] S\, (e \in [\![\mathbf{intexp}]\!] S) = \\ &\qquad\qquad \lambda S'.\lambda\beta.\, e\, S'\, (\text{usetmp } S'(\lambda S''.\lambda r.\, \beta S''(\mathbf{toreal}\, r)))\ . \end{aligned}$$

De manera similar, la conversión de aceptores reales en aceptores enteros se define como sigue:

$$\llbracket \textbf{realacc} \leq \textbf{intacc} \rrbracket S\,(a \in \llbracket \textbf{realacc} \rrbracket S) = \\ \lambda S'.\lambda\kappa.\ \text{usetmp}\ S'(\lambda S''.\lambda r.\ aS'\kappa S''(\textbf{toreal}\ r))\ .$$

5.3.6. Expresiones Booleanas y Condicionales

La traducción de una expresión booleana b acepta un par de secuencias de instrucciones $\langle\kappa,\hat{\kappa}\rangle$ y elige a κ como continuación si b es **true** o a $\hat{\kappa}$ si b es **false**. La ecuación de las constantes entonces queda muy simple:

$$\llbracket \textbf{true} \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle\kappa,\hat{\kappa}\rangle = \kappa$$
$$\llbracket \textbf{false} \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle\kappa,\hat{\kappa}\rangle = \hat{\kappa}$$

La traducción de los operadores booleanos se logra manipulando el par de continuaciones para finalmente elegir la correcta:

$$\llbracket \sim b \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle\kappa,\hat{\kappa}\rangle = \llbracket b \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle\hat{\kappa},\kappa\rangle$$

$$\llbracket b_1 \vee b_2 \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle\kappa,\hat{\kappa}\rangle = \\ \llbracket b_1 \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle\kappa, \llbracket b_2 \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle\kappa,\hat{\kappa}\rangle\rangle$$

$$\llbracket b_1 \wedge b_2 \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle\kappa,\hat{\kappa}\rangle = \\ \llbracket b_1 \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle \llbracket b_2 \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle\kappa,\hat{\kappa}\rangle,\hat{\kappa}\rangle$$

$$\llbracket b_1 \Rightarrow b_2 \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle\kappa,\hat{\kappa}\rangle = \\ \llbracket b_1 \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle \llbracket b_2 \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle\kappa,\hat{\kappa}\rangle,\kappa\rangle$$

$$\llbracket b_1 \Leftrightarrow b_2 \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle\kappa,\hat{\kappa}\rangle = \\ \llbracket b_1 \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle \llbracket b_2 \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle\kappa,\hat{\kappa}\rangle, \llbracket b_2 \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle\hat{\kappa},\kappa\rangle\rangle$$

En el caso de las relaciones podemos necesitar usar variables temporales para calcular su valor de verdad, para luego elegir la continuación apropiada:

$$\llbracket e_1 \otimes e_2 \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle\kappa,\hat{\kappa}\rangle = \\ \llbracket e_1 \rrbracket_{\pi,\hat{\delta}\textbf{exp}}\ S\eta S'(\text{usetmp}\ S'(\lambda S''.\lambda r_1. \\ \llbracket e_2 \rrbracket_{\pi,\hat{\delta}\textbf{exp}}\ S\eta S''(\text{usetmp}\ S''(\lambda S'''.\lambda r_2. \\ \textbf{if}\ r_1 \otimes r_2[S'_d - S'''_d]\ \textbf{then}\ \kappa\ \textbf{else}\ \hat{\kappa}))))$$

donde $\otimes \in \langle\text{rel}\rangle$

Finalmente consideramos los comandos condicionales:

$$\llbracket \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \rrbracket_{\pi,\textbf{comm}}\ S\eta S'\kappa = \\ \llbracket b \rrbracket_{\pi,\textbf{boolexp}}\ S\eta S'\langle \llbracket c_1 \rrbracket_{\pi,\textbf{comm}}\ S\eta S'\kappa, \llbracket c_2 \rrbracket_{\pi,\textbf{comm}}\ S\eta S'\kappa\rangle$$

En varias de las ecuaciones se hace evidente el problema de la duplicación de código. En particular, en los comandos condicionales la secuencia de instrucciones κ aparece en ambas componentes del par de continuaciones. En el capítulo 6 proponemos una manera de resolver este problema.

5.3.7. Declaración de Variables

La traducción del comando **newvar** $\iota\ \delta$ **in** c es una secuencia de instrucciones que crea una nueva variable, la inicializa, ejecuta c, descarta la variable nueva, y finalmente ejecuta la continuación. Sean

$$A^0_{\mathbf{int}} = 0 \qquad A^0_{\mathbf{real}} = 0.0 \qquad A^0_{\mathbf{bool}} = \mathbf{false}$$

entonces definimos la ecuación

$$\begin{aligned}
&[\![\ \mathbf{newvar}\ \iota\ \delta\ \mathbf{in}\ c\ ]\!]_{\pi,\mathbf{comm}}\ S\,\eta\,(S' \geq S)\,(\kappa \in \langle I_{S'}\rangle) = \\
&\qquad S' := \mathbf{lit}_\delta\ A^0_\delta\ [1]; \\
&\qquad\quad [\![\ c\ ]\!]_{[\pi\,|\,\iota:\delta\mathbf{var}],\mathbf{comm}}\ S''\,\hat{\eta}\,S''\,(\mathbf{adjustdisp}[-1];\kappa)\ ,
\end{aligned}$$

donde $S'' = S' + 1$. La nueva variable se ubica en S' y el comando c se evalua con el descriptor extendido S'' que la contiene. El entorno para la traducción de c es

$$\hat{\eta} = [[\![\ \pi\ ]\!]^*(S \leq S'')\eta\,|\,\iota : \langle a, e\rangle]$$

que es la extensión de η que mapea ι con un par aceptor-expresión que describe la nueva variable. El entorno $\hat{\eta} \in [\![\ [\pi\,|\,\iota : \delta\mathbf{var}]\ ]\!]^* S''$ no puede ser una extensión de $\eta \in [\![\ \pi\ ]\!]^* S$, pero sí puede ser una extensión de un entorno en $[\![\ \pi\ ]\!]^* S''$. Este entorno se obtiene aplicando a η la función $[\![\ \pi\ ]\!]^*(S \leq S'')$ que definiremos luego. Para $\hat{\delta} \in \{\mathbf{int}, \mathbf{real}\}$, la expresión $e \in [\![\ \hat{\delta}\mathbf{exp}\ ]\!] S''$ le provee a la continuación β la nueva variable ubicada en S':

$$eS'''\beta = \beta S''' S'\ ,$$

y el aceptor $a \in [\![\ \hat{\delta}\mathbf{acc}\ ]\!] S''$ antepone a la continuación κ' una asignación que almacena el valor de r en la variable nueva, y descarta las variables temporales que ocurren en r:

$$aS'''\kappa' S''''r = S' := r[S'''_d - S''''_d];\kappa'$$

Para el caso de las expresiones booleanas, la expresión $e \in [\![\ \mathbf{boolexp}\ ]\!] S''$ elige una continuación del par según el valor almacenado en S'.

$$eS'''(\kappa, \hat{\kappa}) = \mathbf{if}\ S'\ \mathbf{then}\ \kappa\ \mathbf{else}\ \hat{\kappa}$$

En la ecuación anterior introducimos un nuevo condicional al lenguaje intermedio:[3]

$$\langle I_S\rangle ::= \mathbf{if}\ \langle S_S\rangle\ \mathbf{then}\ \langle I_S\rangle\ \mathbf{else}\ \langle I_S\rangle\ .$$

[3] Podría ser en realidad un syntax sugar del condicional ya existente

El aceptor $a \in [\![\mathbf{boolacc}]\!] S'''$ toma una continuación κ y devuelve un par de continuaciones. Queremos que κ se ejecute independientemente del valor de verdad de S', por lo que aparece en ambos componentes del par. Anteponemos en cada componente la asignación a S' correspondiente:

$$aS'''\kappa = (S' := \mathbf{lit_{bool}}\ \mathbf{true}[0]\,;\kappa\ \ ,\ \ S' := \mathbf{lit_{bool}}\ \mathbf{false}[0]\,;\kappa)\ \ .$$

Acción de los funtores en morfismos

Cuando F es un funtor, que es la semántica de un tipo o contexto, se escribe $F(S \leq S')$ para la aplicación del funtor en el único morfismo de Σ de S en un descriptor más largo S'. Tal aplicación es una función de conversión entre elementos de los predominios FS y FS'. Para el caso de $[\![\mathbf{compl}]\!]$, la conversión consiste en ajustar la pila para la continuación que toma como argumento

$$[\![\mathbf{compl}]\!](S \leq S')(\kappa \in \langle I_S \rangle) = \begin{cases} \mathbf{adjustdisp}[S_d - S'_d]; \kappa & \text{cuando } S'_f = S_f \\ \mathbf{popto}\ S; \kappa & \text{en otro caso} \end{cases} \tag{5.3}$$

Para los exponenciales, la función $f \in (F \underset{\mathcal{K}}{\Rrightarrow} G)S$ cuyo dominio es el conjunto de descriptores mayores a S, se restringe al conjunto de descriptores mayores a S'

$$(F \underset{\mathcal{K}}{\Rrightarrow} G)(S \leq S')f = f\rceil_{\{S'' \mid S'' \geq S'\}}$$

Para los productos, la aplicación del morfismo se define punto a punto

$$(F \times G)(S \leq S')\langle a, e \rangle = \langle F(S \leq S')a, G(S \leq S')e \rangle$$

Ahora definimos la conversión entre entornos, que se aplica punto a punto

$$[\![\pi]\!]^*(S \leq S')\eta\iota = [\![\pi\iota]\!](S \leq S')(\eta\iota)$$

Por último, consideramos la coerción entre variables y aceptores o expresiones

$$\begin{aligned} [\![\delta\mathbf{var} \leq \delta\mathbf{acc}]\!] S\, \langle a, e \rangle &= a \\ [\![\delta\mathbf{var} \leq \delta\mathbf{exp}]\!] S\, \langle a, e \rangle &= e \end{aligned}$$

5.3.8. Procedimientos

Las siguientes ecuaciones describen la semántica funtorial de las construcciones del cálculo lambda que forman parte de nuestro lenguaje:

$$\begin{gathered} [\![\iota]\!]_{\pi,\pi\iota}\ S\eta = \eta\iota \\ [\![p_1 p_2]\!]_{\pi,\theta'}\ S\eta = [\![p_1]\!]_{\pi,\theta\to\theta'}\ S\eta S([\![p_2]\!]_{\pi,\theta}\ S\eta) \\ [\![\lambda\iota : \theta.p]\!]_{\pi,\theta\to\theta'}\ S\eta S' a = [\![p]\!]_{[\pi \mid \iota:\theta],\theta'}\ S'[[\![\pi]\!]^*(S \leq S')\eta \mid \iota : a] \end{gathered}$$

La traducción de la frase **let** ι **be** p **in** p' se desprende de su definición como syntax sugar de $(\lambda\iota.\, p')\, p$

$$[\![\,\mathbf{let}\ \iota\ \mathbf{be}\ p\ \mathbf{in}\ p'\,]\!]_{\pi,\theta'}\ S\eta = [\![\,p'\,]\!]_{[\pi\,|\,\iota:\theta],\theta'}\ S[\eta\,|\,\iota : [\![\,p\,]\!]_{\pi,\theta}\ S\eta]$$

La ecuación de la conversión implícita entre procedimientos es la siguiente:

$$[\![\,\theta_1 \to \theta_2\ \leq\ \theta_1' \to \theta_2'\,]\!]\, S\,(f \in [\![\,\theta_1 \to \theta_2\,]\!]S)\,(S' \geq S)\,(a \in [\![\,\theta_1'\,]\!]S') =$$

$$[\![\,\theta_2 \leq \theta_2'\,]\!]\, S'\,(fS'([\![\,\theta_1' \leq \theta_1\,]\!]S'a))$$

cuando $\theta_1' \leq \theta_1$ y $\theta_2 \leq \theta_2'$

Estas ecuaciones describen la traducción de procedimientos no recursivos a código "abierto" o "inline". Los procedimentos se reducen completamente en tiempo de compilación, generando código puramente imperativo. Esto difiere de otros métodos de compilación donde la implementación inline de procedimientos es difícil de lograr.

5.3.9. Pares

La traducción de los productos binarios es punto a punto. Las construcciones **fst** y **snd** se traducen a las proyecciones correspondientes

$$\begin{aligned}
[\![\,(p_1\,,\,p_2)\,]\!]_{\pi,\theta\times\theta'} &= [\![\,p_1\,]\!]_{\pi,\theta} \times [\![\,p_2\,]\!]_{\pi,\theta'} \\
[\![\,\mathbf{fst}\ p\,]\!]_{\pi,\theta}\ S &= \pi_1\ ([\![\,p\,]\!]_{\pi,\theta\times\theta'}\ S) \\
[\![\,\mathbf{snd}\ p\,]\!]_{\pi,\theta'}\ S &= \pi_2\ ([\![\,p\,]\!]_{\pi,\theta\times\theta'}\ S)
\end{aligned}$$

La coerción entre un producto y otro también se define punto a punto

$$[\![\,\theta_1 \times \theta_2 \leq \theta_1' \times \theta_2'\,]\!]\, S\,\langle p, q\rangle = \langle [\![\,\theta_1 \leq \theta_1'\,]\!]Sp, [\![\,\theta_2 \leq \theta_2'\,]\!]Sq\rangle$$

5.3.10. Subrutinas

Una subrutina es una secuencia de instrucciones que puede ser invocada (o "llamada") desde distintos puntos de un programa del lenguaje intermedio. Las subrutinas son útiles para implementar procedimientos (y otros tipos de frases) definidos recursivamente.
Una subrutina acepta cero o más argumentos, cada uno de ellos es a su vez una subrutina. Para clasificar una subrutina podemos asignarle un *tipo simple:*

$$\varphi\ ::=\ \mathbf{compl}\,|\,\hat{\delta}\mathbf{compl}\,|\,\varphi \to \varphi\ . \tag{5.4}$$

La invocación de una subrutina se logra utilizando la instrucción

$$\mathbf{call}\ i\ f\ (a_1, \ldots, a_n)$$

donde i es la subrutina llamada, y $a_1, \ldots, a_n$ son sus argumentos. El índice f especifica (el último frame de) la lista donde se insertará un nuevo frame, apropiado para ejecutar la subrutina. Esta lista se denomina lista *global* ya que ahi se encuentran las variables globales del procedimento.

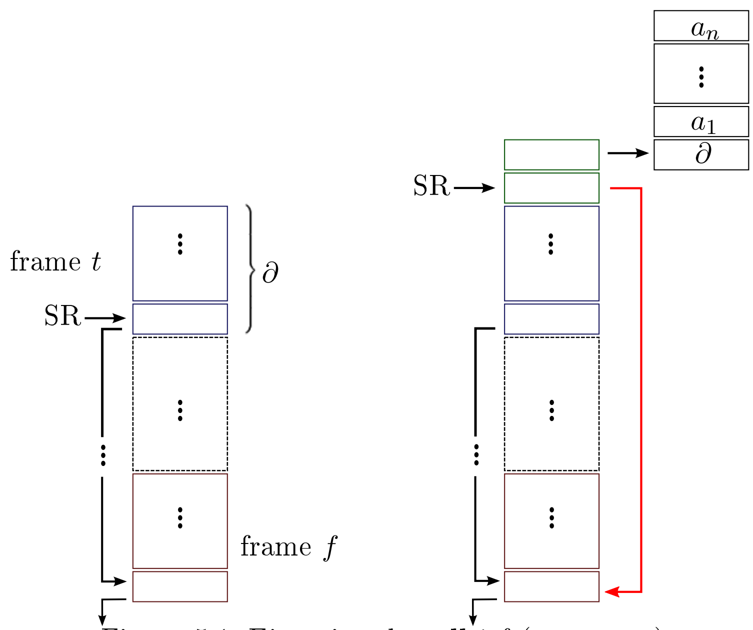


Figura 5.1: Ejecución de **call** i f $(a_1, \ldots, a_n)$

Como se aprecia en la figura 5.1, la ejecución de **call** crea un nuevo frame con dos celdas: la primera apuntando a la base del frame f y la segunda apuntando a un bloque de celdas que contiene la lista de argumentos junto con el entero $\partial = S_d^{curr}$, que corresponde al desplazamiento del frame tope en el momento de la llamada.
En la figura podemos notar que la nueva lista de frames puede tener menos frames que la anterior; el frame tope tiene índice $f+1$ (y ese valor puede ser menor que t).
La instrucción **call** finaliza cediendo el control a i, que se ejecuta con el descriptor $S^{curr} = \langle f+1, 2\rangle$.
En caso de que la subrutina no acepte argumentos, no es eficiente crear un nuevo frame ya que el bloque de argumentos será vacío. Por lo tanto, en lugar de ejecutar **call** i f (), se actualiza el valor de SR de manera que apunte a la base del frame f para luego ceder el control a i.
Ahora consideramos el caso de que la subrutina que se desea invocar es un argumento de otra. Supongamos que ocurre en la posición j del bloque de argumentos apuntado por un frame f. Entonces, si la subrutina acepta al menos un argumento, utilizamos la instrucción

$$\textbf{acall}\ j\ f\ (a_1, \ldots, a_n)$$

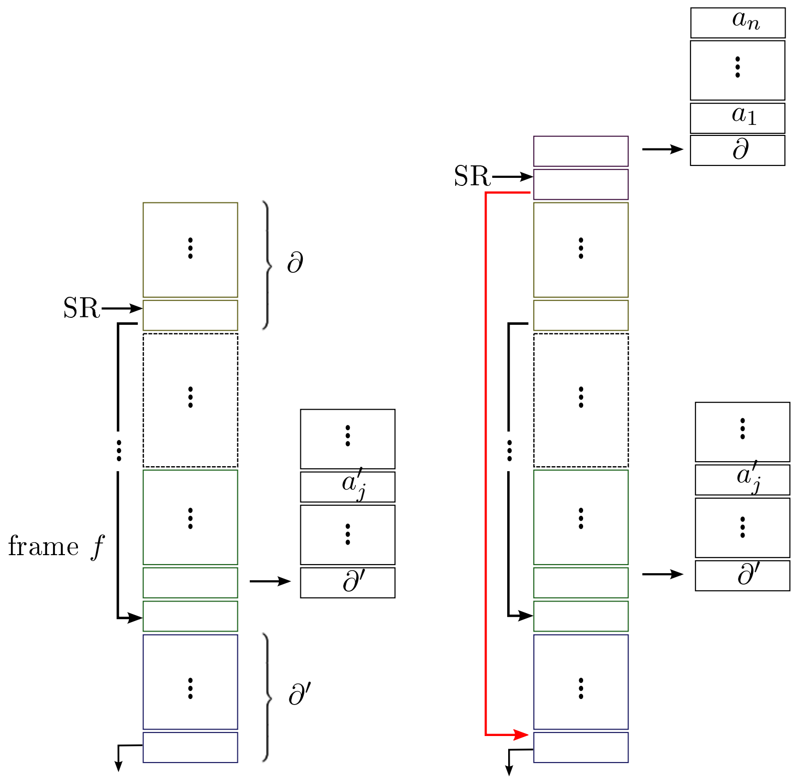


Figura 5.2: Ejecución de **acall** $j\ f\ (a_1, \ldots, a_n)$

que modifica la pila como la figura 5.2, y luego cede el control a a'_j. Si por el contrario, la subrutina no toma argumentos, entonces reseteamos SR para que apunte al frame f y luego ejecutamos **ajump** j,

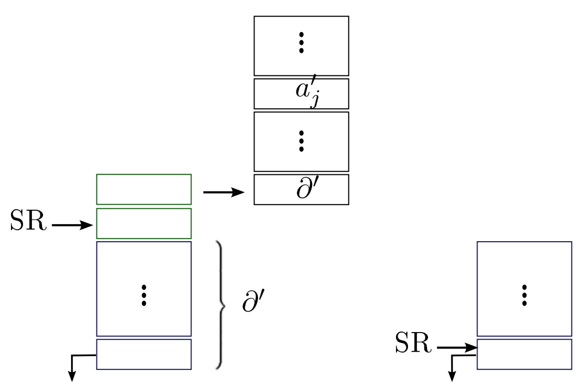


Figura 5.3: Ejecución de **ajump** j

que modifica la pila como en la figura 5.3 y cede el control a a'_j.
Hemos introducido tres nuevos constructores de secuencias de instrucciones para invocar subrutinas:

$$\begin{aligned}\langle I_S\rangle \ ::= \ & \mathbf{call}\ \langle I_{f^+}\rangle\ f\ \langle\langle SR_S^{\varphi_1}\rangle,\ldots,\langle SR_S^{\varphi_n}\rangle\rangle\\ & |\,\mathbf{acall}\ j\ f\ \langle\langle SR_S^{\varphi_1}\rangle,\ldots,\langle SR_S^{\varphi_n}\rangle\rangle\\ & |\,\mathbf{ajump}\ j\end{aligned}$$

donde $f \leq S_f$, $f^+ = \langle f+1, 2\rangle$, $n \geq 1$, $j \geq 1$ y escribimos SR_S^{φ} para denotar una subrutina de tipo φ cuya lista global esta dada por el descritor de pila S.

$$\langle SR_S^{\varphi}\rangle ::= \begin{cases}\langle I_S\rangle & \varphi \in \{\mathbf{compl}, \hat{\delta}\mathbf{compl}\}\\ \langle I_{S^+}\rangle & \text{en otro caso}\end{cases}$$

donde $\langle S_f, S_d\rangle^+ = \langle S_f+1, 2\rangle$.

Generando subrutinas

Definiremos una familia de funciones que son útiles para generar subrutinas y sus correspondientes llamadas. A partir de la traducción de una frase de tipo simple, generaremos una subrutina utilizando la función

$$\text{mk-subr}_{\varphi} S \in [\![\ \varphi\ ]\!] S \to \langle SR_S^{\varphi}\rangle\ ,$$

la llamada a la subrutina la construiremos con la función:

$$\text{mk-call}_{\varphi} S \in \langle SR_S^{\varphi}\rangle \to [\![\ \varphi\ ]\!] S$$

y finalmente la llamada a los argumentos la generamos con la función:

$$\text{mk-argcall}_{\varphi} S \in \mathcal{N} \to [\![\ \varphi\ ]\!] S$$

donde $\mathcal{N}$ denota el conjunto de números positivos. La función $\text{mk-argcall}_{\varphi} S$ devuelve una llamada a un argumento de tipo simple φ que es el j-ésimo argumento en el bloque apuntado por el frame tope de la lista descripta por S.
Esta familia de funciones se define por inducción mutua en los tipos simples:

$$\begin{gathered}\text{mk-subr}_{\mathbf{compl}} S\ \kappa = \kappa\\ \text{mk-call}_{\mathbf{compl}} S\ i = i\\ \text{mk-argcall}_{\mathbf{compl}} S\ j = \mathbf{ajump}\ j\end{gathered}$$

y cuando $\varphi = \varphi_1 \to \ldots \to \varphi_n \to \mathbf{compl}$ para algún $n \geq 1$:

$$\begin{aligned}&\text{mk-subr}_{\varphi}\, S\, (c \in [\![\ \varphi\ ]\!] S) =\\ &\qquad cS^+(\text{mk-argcall}_{\varphi_1} S^+1)\ldots S^+(\text{mk-argcall}_{\varphi_n} S^+n)\end{aligned}$$

$$\text{mk-call}_{\varphi}\, S\, i\, S^1\, (a_1 \in [\![\ \varphi_1\ ]\!] S^1)\ldots S^n(a_n \in [\![\ \varphi_n\ ]\!] S^n) =$$

$$\begin{array}{l}\textbf{call}\ i\ S_f \\ \qquad (\text{mk-subr}_{\varphi_1} S^n([\![\ \varphi_1\ ]\!](S^1 \leq S^n)a_1), \\ \qquad\qquad \vdots \\ \qquad \text{mk-subr}_{\varphi_n} S^n([\![\ \varphi_n\ ]\!](S^n \leq S^n)a_n))\end{array}$$

$$\begin{array}{l}\text{mk-argcall}_{\varphi}\, S\, j\, S^1\,(a_1 \in [\![\ \varphi_1\ ]\!]S^1) \ldots S^n(a_n \in [\![\ \varphi_n\ ]\!]S^n) = \\ \qquad \textbf{acall}\ j\ S_f \\ \qquad\qquad (\text{mk-subr}_{\varphi_1} S^n([\![\ \varphi_1\ ]\!](S^1 \leq S^n)a_1), \\ \qquad\qquad\qquad \vdots \\ \qquad\qquad \text{mk-subr}_{\varphi_n} S^n([\![\ \varphi_n\ ]\!](S^n \leq S^n)a_n))\end{array}$$

Si φ finaliza en $\hat{\delta}\mathbf{compl}$ utilizamos el registro **sbrs** para transmitir un resultado, ya sea entero o real, a una continuación. Sea saveres_δ la siguiente función:

$$\text{saveres}_\delta\, S\, \beta = S := \textbf{sbrs}\ [1]\,;\ \beta\,(S+1)\,S$$

Entonces,

$$\text{mk-subr}_{\hat{\delta}\mathbf{compl}} S\ \beta = \text{saveres}\, S\, \beta$$

$$\text{mk-call}_{\hat{\delta}\mathbf{compl}} S\ i\ S' r = \textbf{sbrs} := r\ [S_d - S'_d]\,;\ i$$

$$\text{mk-argcall}_{\hat{\delta}\mathbf{compl}} S\ j\ S' r = \textbf{sbrs} := r\ [S_d - S'_d]\,;\ \textbf{ajump}\ j$$

y cuando $\varphi = \varphi_1 \to \ldots \to \varphi_n \to \hat{\delta}\mathbf{compl}$ para algún $n \geq 1$:

$$\begin{array}{l}\text{mk-subr}_{\varphi}\, S\,(\beta \in [\![\ \varphi\ ]\!]S) = \text{saveres}\, S^+(\\ \qquad \beta S^+(\text{mk-argcall}_{\varphi_1} S^+ 1) \ldots S^+(\text{mk-argcall}_{\varphi_n} S^+ n))\end{array}$$

$$\begin{array}{l}\text{mk-call}_{\varphi}\, S\, i\, S^1\,(a_1 \in [\![\ \varphi_1\ ]\!]S^1) \ldots S^n(a_n \in [\![\ \varphi_n\ ]\!]S^n)\, S'\, r = \\ \\ \qquad \textbf{sbrs}{:=}\ r\ [S^n_d - S'_d]; \\ \qquad\quad \textbf{call}\ i\ S_f \\ \qquad\qquad (\text{mk-subr}_{\varphi_1} S^n([\![\ \varphi_1\ ]\!](S^1 \leq S^n)a_1), \\ \qquad\qquad\qquad \vdots \\ \qquad\qquad \text{mk-subr}_{\varphi_n} S^n([\![\ \varphi_n\ ]\!](S^n \leq S^n)a_n))\end{array}$$

$$\begin{array}{l}\text{mk-argcall}_{\varphi}\, S\, j\, S^1\,(a_1 \in [\![\ \varphi_1\ ]\!]S^1) \ldots S^n(a_n \in [\![\ \varphi_n\ ]\!]S^n)\, S'(r \in \langle R_{S'} \rangle) = \\ \\ \qquad \textbf{sbrs}{:=}\ r\ [S^n_d - S'_d]; \\ \qquad\quad \textbf{acall}\ j\ S_f \\ \qquad\qquad (\text{mk-subr}_{\varphi_1} S^n([\![\ \varphi_1\ ]\!](S^1 \leq S^n)a_1), \\ \qquad\qquad\qquad \vdots \\ \qquad\qquad \text{mk-subr}_{\varphi_n} S^n([\![\ \varphi_n\ ]\!](S^n \leq S^n)a_n))\end{array}$$

Hasta aquí, sólo podemos generar subrutinas para frases de tipos simples. Para poder generar subutinas para frases de cualquier tipo, el primer paso es definir una función Γ que "simplifique" cada tipo en una secuencia de tipos simples:

$$\begin{gathered}
\Gamma\,\mathbf{compl} = \mathbf{compl} \\
\Gamma\,\hat{\delta}\mathbf{compl} = \hat{\delta}\mathbf{compl} \\
\Gamma\,\mathbf{boolcompl} = \mathbf{compl}, \mathbf{compl} \\
\Gamma\,\mathbf{comm} = \mathbf{compl} \to \mathbf{compl} \\
\Gamma\,\hat{\delta}\mathbf{exp} = \hat{\delta}\mathbf{compl} \to \mathbf{compl} \\
\Gamma\,\mathbf{boolexp} = \mathbf{compl} \to \mathbf{compl} \to \mathbf{compl} \\
\Gamma\,\hat{\delta}\mathbf{acc} = \mathbf{compl} \to \hat{\delta}\mathbf{compl} \\
\Gamma\,\mathbf{boolacc} = \mathbf{compl} \to \mathbf{compl}, \mathbf{compl} \to \mathbf{compl} \\
\Gamma\,\delta\mathbf{var} = \Gamma\,\delta\mathbf{acc} \,+\!\!\!+\, \Gamma\,\delta\mathbf{exp} \\
\Gamma\theta \times \theta' = \Gamma\,\theta \,+\!\!\!+\, \Gamma\,\theta'
\end{gathered}$$

Para el caso de los tipos funcionales, supongamos

$$\Gamma\,\theta = \varphi_1, \ldots, \varphi_m \qquad\qquad \Gamma\,\theta' = \varphi'_1, \ldots, \varphi'_n$$

$$\hat{\varphi} = \varphi_1 \to \ldots \to \varphi_m \ \ ,$$

entonces definimos

$$\Gamma\ \theta \to \theta' = \hat{\varphi} \to \varphi'_1,\ \ldots\ , \hat{\varphi} \to \varphi'_n \ \ .$$

Consideramos el siguiente subconjunto de tipos, denotado por $\hat{\theta}$:

$$\begin{aligned}
\hat{\theta}\ ::=\ & \mathbf{compl} \\
& |\, \hat{\delta}\mathbf{compl} \\
& |\, \hat{\delta}\mathbf{acc} \\
& |\, \delta\mathbf{exp} \\
& |\, \mathbf{comm} \\
& |\, \hat{\theta} \to \hat{\theta}
\end{aligned} \tag{5.5}$$

Notemos que $\hat{\theta}$ contiene los tipos $\tau \in \theta$ tales que $\Gamma\ \tau$ es una secuencia con un único tipo (simple). Podemos llamar $\hat{\Gamma}\ \tau$ a ese tipo.
Se puede demostrar que para cada $\tau \in \hat{\theta}$ se pueden definir funciones $\phi_\tau S$ y $\psi_\tau S$ tales que

$$[\![\ \tau\ ]\!]S \ \underset{\psi_\tau S}{\overset{\phi_\tau S}{\rightleftarrows}}\ [\![\ \hat{\Gamma}\ \tau\ ]\!]S$$

es un isomorfismo. Por lo tanto, podemos compilar cada procedimiento (recursivo) de tipo τ con una subrutina de tipo $\varphi = \hat{\Gamma}\ \tau$ como sigue

$$[\![\ \mathbf{letrec}\ \iota : \tau\ \mathbf{be}\ p\ \mathbf{in}\ p'\ ]\!]_{\pi,\theta'}\ S\,\eta = [\![\ p'\ ]\!]_{[\pi\,|\,\iota:\tau],\theta'}\ S\eta' \tag{5.6}$$

donde

$$\begin{aligned}\eta' &= [\eta \mid \iota : \psi_\tau S\,(\text{mk-call}_\varphi\, S\, i)\,]\\ i &= \text{mk-subr}_\varphi\, S\,(\phi_\tau S\,(\llbracket\, p\,\rrbracket_{[\pi \mid \iota:\tau],\tau}\; S\,\eta'))\end{aligned}$$

Aquí i y η' son puntos fijos calculados simultáneamente, pues existe una una dependencia mutua entre ellos. En la implementación, representaremos esta dependencia mutua utilizando *labels*, como se explica en el capítulo 6.
Podemos generalizar el método para cualquier tipo en θ, ya que si $\Gamma\,\theta = \varphi_1, \ldots, \varphi_n$ entonces hay funciones $\phi_\theta S$ y $\psi_\theta S$ tales que

$$\llbracket\, \theta\,\rrbracket S \;\underset{\psi_\theta S}{\overset{\phi_\theta S}{\rightleftarrows}}\; \llbracket\, \varphi_1\,\rrbracket S \times \ldots \times \llbracket\, \varphi_n\,\rrbracket S \tag{5.7}$$

es un isomorfismo. Entonces la ecuación 5.6 se generaliza como sigue:

$$\llbracket\, \textbf{letrec}\ \iota : \theta\ \textbf{be}\ p\ \textbf{in}\ p'\,\rrbracket_{\pi,\theta'}\; S\,\eta = \llbracket\, p'\,\rrbracket_{[\pi \mid \iota:\theta],\theta'}\; S\eta'$$

donde

$$\begin{aligned}\eta' &= [\,\eta \mid \iota : \psi_\theta S(\text{mk-call}_{\varphi_1} S i_1, \ldots, \text{mk-call}_{\varphi_n} S i_n)\,]\\ i_1 &= \text{mk-subr}_{\varphi_1} S a_1\\ &\;\;\vdots\\ i_n &= \text{mk-subr}_{\varphi_n} S a_n\\ (a_1, \ldots, a_n) &= \phi_\theta S(\llbracket\, p\,\rrbracket_{[\pi \mid \iota:\theta],\theta}\; S\,\eta')\ .\end{aligned}$$

5.3.11. Continuaciones e Iteraciones

Una continuación se puede ver como un un tipo especial de comando que “nunca devuelve el control”, es decir, nunca ejecuta la continuación que toma como argumento. Esto se refleja en la conversión implícita de **compl** a **comm**, que se define como sigue:

$$\llbracket\, \textbf{compl} \le \textbf{comm}\,\rrbracket S\,\kappa\, S'\,\kappa' = \llbracket\, \textbf{compl}\,\rrbracket (S \le S')\kappa\ \ ,$$

(Notar que la continuación κ' nunca se ejecuta). La combinación secuencial de un comando con una continuación es una continuación

$$\llbracket\, c_1; c_2\,\rrbracket_{\pi,\textbf{compl}}\; S\eta = \llbracket\, c_1\,\rrbracket_{\pi,\textbf{comm}}\; S\eta S(\llbracket\, c_2\,\rrbracket_{\pi,\textbf{compl}}\; S\eta)$$

También podemos construir continuaciones usando condicionales:

$$\begin{aligned}&\llbracket\, \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\,\rrbracket_{\pi,\textbf{compl}}\; S\eta =\\ &\qquad \llbracket\, b\,\rrbracket_{\pi,\textbf{boolexp}}\; S\eta S\langle \llbracket\, c_1\,\rrbracket_{\pi,\textbf{compl}}\; S\eta, \llbracket\, c_2\,\rrbracket_{\pi,\textbf{compl}}\; S\eta\rangle\end{aligned}$$

El comando **escape** ι **in** c crea una continuación cuyo efecto es salir de la ejecución del comando c para proseguir con la continuación correspondiente:

$$\begin{aligned}&[\![\, \textbf{escape}\ \iota\ \textbf{in}\ c \,]\!]_{\pi,\textbf{comm}}\ S\eta S'\kappa = \\ &\qquad [\![\, c \,]\!]_{[\pi\,|\,\iota:\textbf{compl}],\textbf{comm}}\ S'[[\![\, \pi \,]\!]^*(S \leq S')\eta \mid \iota : \kappa]S'\kappa\end{aligned}$$

Algunas contrucciones iterarivas pueden definirse como syntax sugar. Un ejemplo es la traducción de **loop**:

$$\textbf{loop}\ c \doteq \textbf{letrec}\ \iota\ \textbf{be}\ c\,;\,\iota\ \textbf{in}\ \iota$$

donde ι es un identificador que no ocurre libre en c. Si aplicamos la ecuación de **letrec** en esa definición sintáctica, llegamos a la traducción

$$[\![\, \textbf{loop}\ c \,]\!]_{\pi,\textbf{compl}}\ S\eta = i$$

donde

$$i = [\![\, c \,]\!]_{\pi,\textbf{comm}}\ S\eta S i$$

Otro ejemplo es el comando **while**,

$$\begin{aligned}&\textbf{while}\ b\ \textbf{do}\ c \doteq \textbf{escape}\ e\ \textbf{in} \\ &\qquad \textbf{letrec}\ \iota\ \textbf{be if}\ b\ \textbf{then}\ (c\,;\,\iota)\ \textbf{else}\ e\ \textbf{in}\ \iota\end{aligned}$$

De esta definición se obtiene la siguiente traducción:

$$[\![\, \textbf{while}\ b\ \textbf{do}\ c \,]\!]_{\pi,\textbf{comm}}\ S\eta S'\kappa = i$$

donde

$$\begin{aligned}i = {}&[\![\, b \,]\!]_{\pi,\textbf{boolexp}}\ S'([\![\, \pi \,]\!]^*(S \leq S')\eta)S' \\ &\qquad \langle [\![\, c \,]\!]_{\pi,\textbf{comm}}\ S'([\![\, \pi \,]\!]^*(S \leq S')\eta)S' i,\ \kappa\rangle\end{aligned}$$

5.3.12. Subsunción

En el capítulo 3 vimos la regla de subsunción, que establece que si $\theta \leq \theta'$ entonces una frase de tipo θ tiene también tipo θ':

$$\frac{\pi \vdash p : \theta \qquad \theta \leq \theta'}{\pi \vdash p : \theta'}$$

La ecuación semántica correspondiente a esa regla es la siguiente:

$$[\![\, p \,]\!]_{\pi,\theta'} = [\![\, \theta \leq \theta' \,]\!] \circ [\![\, p \,]\!]_{\pi,\theta} \quad \text{cuando } \theta \leq \theta'$$

Todas las reglas de tipado, excepto la de subsunción, son dirigidas por síntaxis. Es decir, si la conclusión de la regla es $\pi \vdash p : \theta$ entonces las premisas son juicios $\pi_1 \vdash e_1 : \theta_1, \ldots, \pi_n \vdash e_n : \theta_n$ donde cada e_i es una subfrase (estricta) de p.

En cambio, la regla de subsunción tiene una premisa con la misma frase p. Debido a esta regla, es más difícil probar la propiedad de *coherencia* que nos dice que distintas derivaciones para el mismo juicio deben dar lugar a la misma semántica. No hemos probado al coherencia de *Peal* , pero consideramos que se puede utilizar el hecho de que la semántica funtorial de lenguajes Algol-like con intersección de tipos[4] es coherente [24] para probar que la semántica de *Peal* es coherente, puesto que esos lenguajes son una extensión de *Peal* (en el sentido formal que todo juicio en *Peal* es un juicio en esos lenguajes).

5.3.13. Ejemplo

Consideremos el siguiente programa P

$$
\begin{array}{l}
\mathbf{newvar}\ x\ \mathbf{int\ in} \\
\qquad \mathbf{letrec}\ pr\ \mathbf{be}\ \lambda c : \mathbf{comm}.\ (c\ ;\ x := x + 1; \\
\qquad\qquad \mathbf{if}\ x \leq 1\ \mathbf{then}\ pr(c\ ;\ c)\ \mathbf{else\ skip}) \\
\qquad \mathbf{in\ newvar}\ y\ \mathbf{int\ in}\ pr(y := y + x * x)
\end{array}
$$

Entonces si aplicamos los argumentos adecuados obtenemos la traducción:

$$
\begin{array}{l}
[\![\ P\ ]\!]_{[],\mathbf{comm}}\ \ \langle 0,0\rangle\ []\ \langle 0,0\rangle\ \mathbf{stop} = \\
\quad \langle 0,0\rangle := \mathbf{lit_{int}}0[1]; \langle 0,1\rangle := \mathbf{lit_{int}}0[1]; \\
\quad \overbrace{\mathbf{call}}^{A}\ i\ 0\ (\langle 1,2\rangle := \langle 0,0\rangle * \langle 0,0\rangle\,[1]; \\
\qquad\qquad \langle 0,1\rangle := \langle 0,1\rangle + \langle 1,2\rangle\,[-1];\ \mathbf{ajump}\ 1, \\
\qquad\qquad \mathbf{adjustdisp}\,[-1]; \mathbf{adjustdisp}\,[-1]; \mathbf{stop})
\end{array}
$$

donde i es

$$
\begin{array}{l}
\overbrace{\mathbf{acall}}^{B}\ 1\ 1\ (\langle 0,0\rangle := \langle 0,0\rangle + \mathbf{lit_{int}}1\,[0]; \\
\qquad\quad \mathbf{if}\ \langle 0,0\rangle\ \leq\ \mathbf{lit_{int}}1\,[0]\ \mathbf{then} \\
\qquad\quad\ \ \overbrace{\mathbf{call}}^{C}\ i\ 0\ (\overbrace{\mathbf{acall}}^{D}\ 1\ 1\ (\overbrace{\mathbf{acall}}^{E}\ 1\ 1\ (\mathbf{ajump}\ 1)), \\
\qquad\qquad\qquad \mathbf{ajump}\ 2) \\
\qquad\quad \mathbf{else\ ajump}\ 2)
\end{array}
$$

La ejecución de este programa se muestra en la figura 5.4. En esta figura se utilizan las letras A, B, C, D, E para denotar el bloque de argumentos de la instrucción etiquetada con el mismo nombre en el programa anterior. El color de una flecha indica el frame al que apunta. En la parte superior de cada dibujo se muestra el descriptor de pila actual. Describimos a continuación cada paso de ejecución mostrado en la figura:

[4] La intersección de tipos se describe en [23]

1. $\langle 0,0\rangle := \mathbf{lit_{int}} 0[1]; \langle 0,1\rangle := \mathbf{lit_{int}} 0[1]$
2. **call** i 0 A
3. **acall** 1 1 B
4. $\langle 1,2\rangle := \langle 0,0\rangle * \langle 0,0\rangle\,[1]; \langle 0,1\rangle := \langle 0,1\rangle + \langle 1,2\rangle\,[-1];$
5. **ajump** 1
6. $\langle 0,0\rangle := \langle 0,0\rangle + \mathbf{lit_{int}} 1\,[0]$
7. **call** i 0 C
8. **acall** 1 1 B
9. **acall** 1 1 D
10. $\langle 1,2\rangle := \langle 0,0\rangle * \langle 0,0\rangle\,[1]; \langle 0,1\rangle := \langle 0,1\rangle + \langle 1,2\rangle\,[-1];$
11. **ajump** 1
12. **acall** 1 1 E
13. $\langle 1,2\rangle := \langle 0,0\rangle * \langle 0,0\rangle\,[1]; \langle 0,1\rangle := \langle 0,1\rangle + \langle 1,2\rangle\,[-1];$
14. **ajump** 1
15. **ajump** 1
16. $\langle 0,0\rangle := \langle 0,0\rangle + \mathbf{lit_{int}} 1\,[0]$
17. **ajump** 2
18. **ajump** 2
19. **adjustdisp** $[-1]$
20. **adjustdisp** $[-1]$

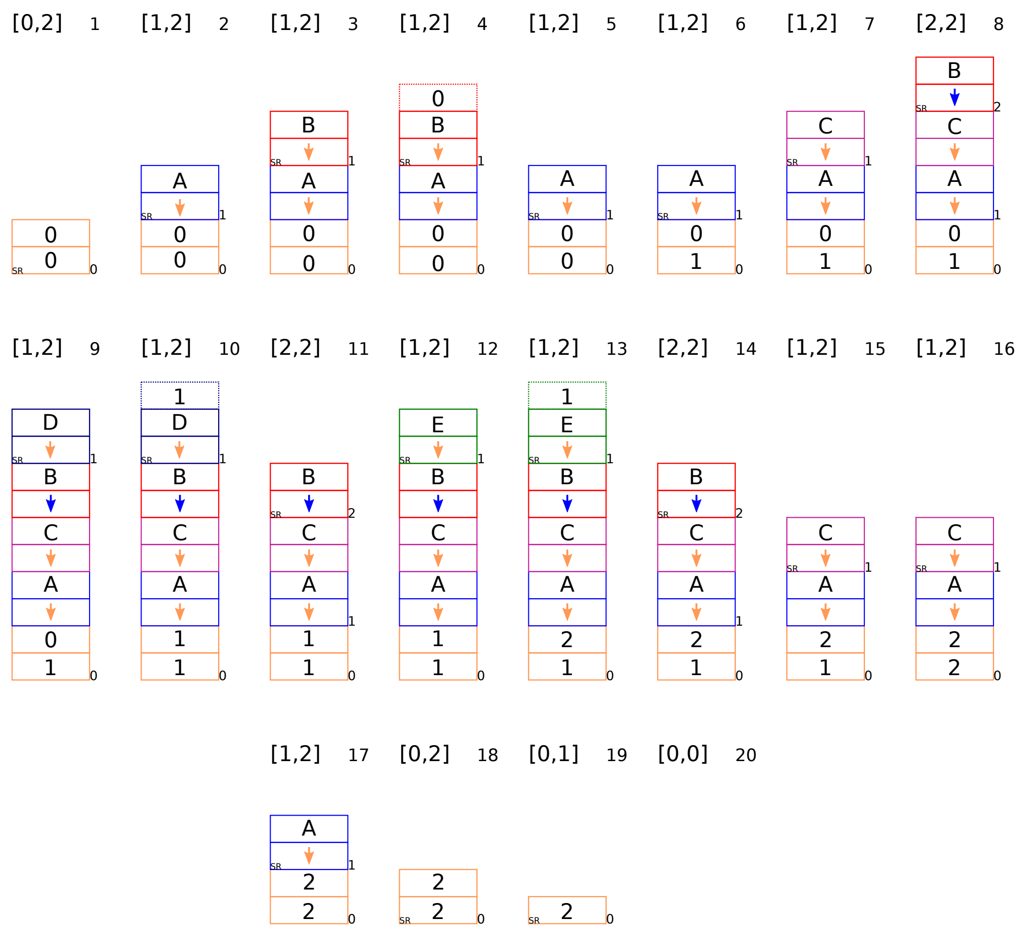


Figura 5.4: Ejemplo de ejecución de un programa

Capítulo 6

Implementación

En este capítulo presentamos la implementación del front-end, y explicamos los segmentos relevantes del código de cada fase. El parser fue programado en Haskell y las funciones de typechecking y generación de código fueron programadas en Agda, un lenguaje funcional con tipos dependientes.

6.1. Elección del lenguaje

Supongamos que $\mathcal{T}$ es el conjunto de tipos del lenguaje en que realizaremos la implementación (como por ejemplo, Haskell). En particular supongamos que $\mathsf{Types} \in \mathcal{T}$ es el tipo que representa a los tipos de *Peal*.
En la implementación, debemos tener tipos en $\mathcal{T}$ que representen a la semántica de cada tipo de *Peal* . Es decir, si θ es un tipo de *Peal* , debemos definir un tipo $\tau_\theta \in \mathcal{T}$ que represente al objeto $[\![\ \theta\ ]\!]$. Esto sugiere que para implementar al funtor $[\![\_]\!]$ necesitamos definir una familia de tipos indexada por elementos de Types, es decir

$$[\![\_]\!] : \mathsf{Types} \ \rightarrow \ \mathcal{T} \ . \tag{6.1}$$

La manera más natural de definir familias de tipos es usando un lenguaje con tipos dependientes. En lenguajes como Haskell y ML hay una separación clara entre los tipos y los valores, pero en los lenguajes con tipos dependientes (Agda [20] , Dependent ML [32] , ATS [9]) los tipos pueden contener (o depender de) valores arbitrarios, y pueden aparecer como argumentos o resultados de funciones.
Las alternativas que surgieron para la elección del lenguaje de implementación fueron las siguientes:

1. No utilizar tipos dependientes. Un lenguaje con tipos dependientes no es absolutamente necesario para la implementación. En particular, podríamos cambiar la definición 6.1 utilizando la unión disjunta de los tipos:

$$[\![\_]\!] : \mathsf{Types} \ \rightarrow \ \biguplus_{\theta \in \Theta} \tau_\theta \ .$$

Pero esta alternativa además de ser artificial puede complicar la implementación (siempre hay que verificar a qué conjunto de la unión disjunta pertenece el resultado). Los tipos de datos y las funciones implementadas podrían tener tipos (no dependientes) más permisivos y devolver error en caso de que aparezcan construcciones sin sentido. Sin embargo, este alternativa oscurecería la relación que hay entre la implementación y las ecuaciones de la teoría.

2. Simular los tipos dependientes en Haskell. En [12, 17, 30] encontramos técnicas para imitar algunos aspectos de los tipos dependientes en Haskell. Algunas de estas técnicas utilizan extensiones de Haskell tales como

   - Clases de tipos, con multiparámetros y dependencias funcionales.
   - Familias de tipos.
   - GADTs

   En [17, sec 6.1] se mencionan algunas limitaciones de esas técnicas respecto a los lenguajes con tipos dependientes. Las familias de tipos que nos permite construir Haskell (extendido[1]) sólo pueden indexarse por tipos y no por términos.

3. Utilizar un lenguaje con tipos dependientes.

La elección fue utilizar tipos dependientes, y elegimos Agda como lenguaje de implementación. El parser, sin embargo, fue implementado en Haskell.

6.1.1. Agda

Agda [7] es un lenguaje de programación funcional con tipos dependientes. Es una implementación de la teoría de tipos intuicionista creada por el lógico sueco Per Martin-Löf. Algunas de las características que posee son:

- Tipos de datos inductivos.
- Pattern matching.
- Modulos parametrizables.
- Analizador de terminación y de cobertura.
- Síntaxis flexible y atractiva (permite usar caracteres unicode)
- Capacidad de interactuar con Haskell.

Los tipos dependientes tienen una carecterística especial : pueden codificar propiedades de valores como tipos cuyos elementos son *pruebas* de que la propiedad es verdadera. En ese sentido, Agda es también un asistente de demostración : permite expresar proposiciones lógicas como tipos, y una prueba de la proposición es un programa del tipo correspondiente.
Referimos al lector a [6, 20] para una explicación más detallada del lenguaje Agda

[1] Ver `http://www.haskell.org/haskellwiki/GHC/Type_families`

y de los tipos dependientes. Podemos dar un ejemplo de la definición en Agda de un tipo de datos: el tipo Vector con longitud

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

El tipo del constructor _::_ es un ejemplo de tipo dependiente. El primero argumento es un número natural n implícito (encerrado entre llaves). Es implícito porque el typechecker de Agda puede inferir su valor a partir del tercer argumento. La función head que devuelve el primer elemento del vector está definida sólo para vectores no vacíos. Podemos expresar este hecho en el tipo de la función:

```
head : {n : ℕ} → {A : Set} → Vec A (suc n) → A
head (x :: xs) = x
```

Si intentamos definir head para el caso [], el typechecker fallará ya que para crear un vector de tipo Vec A (suc n) sólo es posible utilizar el constructor _::_.
En Haskell, al contrario de lo que pasa en Agda, la lista vacía y la lista con al menos un elemento tienen el mismo tipo. Entonces no podemos definir la función head con un dominio total; y por lo tanto quedará indefinida para el caso vacío.

Tipos de datos y notación

En este capítulo utilizaremos algunos tipos de datos definidos en la librería estándar de Agda. Algunos de ellos los listamos a continuación; el lector puede consultar el código fuente [2] para ver las definiciones de los tipos.

- Producto binario dependiente (Σ), y los casos particulares
  - Existencial ($\exists$)
  - Producto binario no dependiente ($\times$)
- Unión Disjunta ($\uplus$)
- Decibilidad (Dec)
- Mónadas (en particular la mónada de estado). Alguna notación sobre mónadas que utilizamos en el código es la siguiente:
  - Denotamos $\gg=$ al binding de la mónada (excepto en el caso de la mónada identidad, donde utilizamos $\ggg$ para denotar el binding). A veces la operación $m \gg= \lambda\_ \rightarrow k$ se escribe $m \gg k$.
  - Usamos $\uparrow$ para denotar la operación *return* de la mónada.

[2]Disponible en `http://wiki.portal.chalmers.se/agda`

6.2. Expresiones primarias

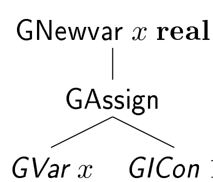
El primer paso para implementar el front-end fue definir en Haskell el tipo de datos GExp que se utiliza como una representación del lenguaje fuente:

```
data GExp = GICon Int
    | GBCon Bool
    | GRCon Double
    | GVar Id
    | GAssign GExp GExp
    | GNewvar Id DType GExp
    | GOp1 Op1 GExp
    | GOp2 Op2 GExp GExp
    | ...
type Id = String
data DType = Int | Real | Bool
```

Como veremos, el parser genera una expresión de tipo GExp, que puede verse como un árbol sintáctico del programa. Por ejemplo, para la expresión que resulta del parseo del programa

$$\begin{array}{l} \textbf{newvar } x \textbf{ real in} \\ \quad x := 1 \\ \textbf{end} \end{array} , \tag{6.2}$$

uno obtiene el siguiente árbol sintáctico:



A las expresiones de tipo GExp le denominamos expresiones primarias puesto que no tienen información sobre el tipo de las frases, y por lo tanto podrían estar mal tipadas. Como veremos, el typechecking recibirá la expresión GExp y verificará si está bien tipada, construyendo un árbol de derivación.

6.3. Parser

El parser construye una una expresión de tipo GExp a partir del código fuente. Implementamos el parser en Haskell, utilizando Parsec : una librería de combinadores de parser monádicos.
En Parsec, un parser es un valor de primer orden, es decir, puede ser pasado como argumento o devuelto por una función. Además, se pueden combinar varios parsers en uno solo, utilizando los combinadores de la librería.

En esta sección se explica brevemente cómo se construyó el parser del lenguaje. Los detalles del uso de la librería se pueden encontrar en la guía de usuario [16].
Excepto para los operadores y las relaciones, cuyo parseo explicamos más adelante, definimos un parser para cada constructor de GExp. Por ejemplo, el parser para el constructor GNewvar es el siguiente:

```
newvar :: Parser (GExp)
newvar = try $ keyword "newvar" ≫ ident ≫= λid →
   dty ≫= λdt → keyword "in" ≫ gexp ≫= λc →
   keyword "end" ≫ return (GNewvar id dt c)
```

Si lo pensamos en forma secuencial, la función newvar parsea:

1. La palabra reservada “newvar”, usando el parser keyword que provee Parsec.
2. Un identificador (el nombre de la variable). El parser ident también está en la librería Parsec.
3. Un tipo de datos, usando el parser dty.
4. La palabra reservada “in”.
5. La expresión interna del comando.
6. La palabra reservada “end”.

Finalmente newvar devuelve una expresión de tipo GExp que representa el segmento de código parseado. Otro ejemplo es el parser de las variables:

```
var :: Parser GExp
var = try $ ident ≫= return ∘ GVar
```

que simplemente parsea el nombre de la variable y la devuelve dentro del constructor GVar. El try inicial es útil para evitar consumir caracteres cuando falla el parseo, por ejemplo, si lee una palabra reservada en vez de un identificador.
Una vez que se tiene un parser por cada constructor se los combina en un solo parser, utilizando el operador ⟨|⟩

```
atom :: Parser GExp
atom = var ⟨|⟩ newvar ⟨|⟩ ...
```

Para construir el parser es necesario también crear una lista de operadores del lenguaje. Para especificar un operador se utilizan los constructores Infix y Prefix [16, sec 2.7] que provee Parsec. Para el caso de los operadores binarios infijos se debe especificar la asociatividad, que puede ser a izquierda, a derecha o ninguna. El orden en que se escribe la lista de operadores es de acuerdo a la precedencia (los primeros elementos son los de mayor precedencia). La lista ops del siguiente código especifica (parte) de los operadores que acepta el parser:

```
binop s f assoc = Infix (oper s ≫ return f) assoc
prefix s f = Prefix (oper s ≫ return f)
op1 = [[prefix "-" (GOp1 Minus), prefix "~" (GOp1 Neg)]]
op2 = [[binop "*"   (GOp2 Times)  AssocLeft,
         binop "/"   (GOp2 RDiv)   AssocLeft,
         binop "%"   (GOp2 IDiv)   AssocLeft,
         binop "rem" (GOp2 Rem)    AssocLeft]
        ,
        [binop "+"   (GOp2 Plus)   AssocLeft,
         binop "-"   (GOp2 Minus2) AssocLeft]
       ]
ops = op1 ++ op2 ++ ...
```

Los operadores $+$ y $-$ tienen la misma precedencia y asocian a izquierda. El operador $*$ tiene mayor precedencia que $+$ y también asocia a izquierda. Los operadores unarios tienen la máxima precedencia.
Por último, el parser para el lenguaje completo se construye usando buildExpressionParser, que toma como argumentos la lista de operadores y el parser de los constructores:

```
gexp = buildExpressionParser ops atom
```

6.4. Typechecking

Una vez que tenemos un término de tipo GExp necesitamos saber si esa expresión corresponde a una expresión bien tipada, para ello necesitamos un algoritmo de inferencia de tipos [11, 13]. Como es habitual, necesitamos considerar el problema más general de tratar de inferir un tipo para una expresión en un contexto dado y definir simultáneamente el algoritmo de chequeo de tipos, i.e. dado un término, un tipo y un contexto debemos poder decidir si en ese contexto, el término tiene ese tipo.
La implementación de las funciones de inferencia y chequeo de tipos se realizó en en Agda. La función de inferencia intenta contruir un árbol de derivación (o una prueba) de que la expresión de tipo GExp está bien tipada; esta prueba se construye a partir de las reglas de inferencia vistas en la sección 3.3.

6.4.1. Tipos

Primero definimos en Agda a los tipos abstractos DType y Type que denotan los tipos de datos y los tipos de frases del lenguaje, respectivamente:

```
data DType : Set where
  int : DType
  real : DType
  bool : DType
infixr 3 _⇾_
data Type : Set where
  δexp   : DType → Type
  δacc   : DType → Type
  δvar   : DType → Type
  comm   : Type
  compl  : Type
  δcompl : DType → Type
  prod   : Type → Type → Type
  _⇾_    : Type → Type → Type
```

El operador _⇾_ es el constructor de tipos funcionales, y prod es el constructor de producto binario de tipos. Podemos definir algunas constantes para dar nombres más cortos a los tipos:

```
intexp : Type
intexp = δexp int
realexp : Type
realexp = δexp real
boolexp : Type
boolexp = δexp bool
```

6.4.2. Subtipos

A continuación, definimos la relación de subtipo entre los tipos de datos y los tipos de frases:

```
data _δ≼_ : DType → DType → Set where
  δrefl : {d : DType} → (d δ≼ d)
  ir : int δ≼ real
data _≼_ : Type → Type → Set where
  τrefl : {τ : Type} → τ ≼ τ
  δexp-sub : {d₁ d₂ : DType} → d₁ δ≼ d₂ → δexp d₁ ≼ δexp d₂
  δacc-sub : {d₁ d₂ : DType} → d₁ δ≼ d₂ → δacc d₂ ≼ δacc d₁
  var≼exp : {d : DType} → δvar d ≼ δexp d
  var≼acc : {d : DType} → δvar d ≼ δacc d
  compl≼comm : compl ≼ comm
  prod-sub : {τ₁ τ₂ τ₃ τ₄ : Type} → τ₁ ≼ τ₃ → τ₂ ≼ τ₄ → (prod τ₁ τ₂) ≼ (prod τ₃ τ₄)
  ⇾-sub : {τ τ′ τ″ τ‴ : Type} → τ″ ≼ τ → τ′ ≼ τ‴ → (τ ⇾ τ′) ≼ (τ″ ⇾ τ‴)
  trans : {τ τ′ τ″ : Type} → τ ≼ τ′ → τ′ ≼ τ″ → τ ≼ τ″
```

Cada constructor representa una regla de subtipado y los argumentos del constructor corresponden a las premisas de la regla, entonces un elemento de tipo $\tau \preccurlyeq \tau'$

puede verse como una prueba de que τ es subtipo de τ'.
Durante el typechecking necesitaremos un procedimiento de decisión para $\preccurlyeq$, es decir, una función que dados tipos τ y τ' nos construya una prueba de $\tau \preccurlyeq \tau'$, o bien una prueba de $\tau \not\preccurlyeq \tau'$. La signatura de esa función es la siguiente:

$$\preccurlyeq\mathsf{dec}\ :\ (\mathsf{t}\ :\ \mathsf{Type}) \to (\mathsf{t'}\ :\ \mathsf{Type}) \to \mathsf{Dec}\ (\mathsf{t} \preccurlyeq \mathsf{t'})$$

Para definir $\preccurlyeq$dec se necesitan demostrar una serie de propiedades, como por ejemplo, que el tipo **comm** es maximal con respecto a la relación de subtipo:

```
comm-max : ∀ {t} → comm ≼ t → t ≡ comm
comm-max τrefl = refl
comm-max {t} (trans {.comm} {t″} {.t} y y') with t″ | comm-max y
... | .comm | refl = comm-max y'
```

La relación $\delta\preccurlyeq$ es un preorden, pero la regla de transitividad no necesita agregarse como constructor porque podemos demostrarla como sigue:

```
δ≼trans : ∀ {d d′ d″} → d δ≼ d′ → d′ δ≼ d″ → d δ≼ d″
δ≼trans δrefl δrefl = δrefl
δ≼trans δrefl ir = ir
δ≼trans ir δrefl = ir
```

6.4.3. Contexto

Definimos el contexto como una lista pares de identificadores y tipos

```
ID  = String
Ctx = List (ID × Type)
```

y definimos también las pruebas de pertenencia al contexto; hd es la prueba de que el identificador está en el primer par del contexto y tl es la prueba de que el identificador se encuentra a partir del segundo elemento:

```
data _∈_ (ι : ID) : Ctx → Set where
  hd : forall {t π} → ι ∈ ((ι, t) :: π)
  tl : forall {z t π} → ι ∈ π → ι ∈ ((z, t) :: π)
```

Si tenemos una prueba de pertenencia de un identificador en el contexto, podemos acceder al tipo asociado con el identificador utilizando la función gettype:

```
gettype : {ι : ID} → {π : Ctx} → ι ∈ π → Type
gettype {_} {[]} ()
gettype {.ι} {(ι, t) :: π} hd = t
gettype {z} {(ι, t) :: π} (tl p) = gettype p
```

Notemos que no es posible construir una prueba de que $\iota \in []$; por lo que el typechecker de Agda detecta esa situación y nos permite no definir ese caso en gettype.

Cuando se analiza el scope de las variables, es necesario verificar si un identificador pertenece o no al contexto. La función lookup devuelve inside y una prueba de pertenencia en caso de encontrar la variable y outside en otro caso:

```
data Lookup (π : Ctx) : ID → Set where
  inside : {ι : ID} → (ι ∈ π) → Lookup π ι
  outside : {ι : ID} → Lookup π ι
lookup : (π : Ctx) → (ι : ID) → Lookup π ι
lookup [] ι = outside
lookup ((ι, t) :: π) ι0 with ι ≟ ι0
lookup ((ι, t) :: π) .ι | yes refl = inside hd
lookup ((ι, t) :: π) ι0 | no _ with lookup π ι0
... | inside p = inside (tl p)
... | outside = outside
```

6.4.4. Reglas de inferencia

Una prueba de que un juicio $\pi \vdash e : \theta$ es válido se representa en Agda con el tipo abstracto

$$\textbf{data}\ \_\vdash\_:\_ \ :\ \mathsf{Ctx} \to \mathsf{GExp} \to \mathsf{Type} \to \mathsf{Set}\ \ \textbf{where}\ ,$$

que tiene un constructor por cada regla de tipado. No presentamos la definición del tipo para ayudar a la legibilidad, pero se encuentra en el código A.1 del apéndice. Como un ejemplo, consideremos la siguiente derivación (correspondiente al programa 6.2):

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\pi \vdash \mathsf{GVar}\ x : \delta\mathsf{var\ real} \\ \delta\mathsf{var\ real} \preccurlyeq \delta\mathsf{acc\ real}\end{array}}{\pi \vdash \mathsf{GVar}\ x : \delta\mathsf{acc\ real}} \qquad \dfrac{\begin{array}{c}\pi \vdash \mathsf{GICon}\ 1 : \delta\mathsf{exp\ int} \\ \delta\mathsf{exp\ int} \preccurlyeq \delta\mathsf{exp\ real}\end{array}}{\pi \vdash \mathsf{GICon}\ 1 : \delta\mathsf{exp\ real}}}{\pi \vdash \mathsf{GAssign}\,(\mathsf{GVar}\,x)\,(\mathsf{GICon}\ 1) : \mathsf{comm}}}{[] \vdash \mathsf{GNewvar}\ x\ \mathbf{real}\,(\mathsf{GAssign}\ (\mathsf{GVar}\ x)\ (\mathsf{GICon}\ 1)) : \mathsf{comm}}$$

donde π es igual a $[(x, \delta\mathsf{var\ real})]$. En nuestra implementación, podemos representar a ese árbol de derivación con la siguiente expresión:

```
Newvar {π x real}
                 Assign
                       Sub
                          Var hd
                          var≼acc
                       Sub
                          ICon {1}
                          δexp-sub ir
```

No siempre podremos construir la derivación, ya que pueden haber errores de tipo en el programa. Por ejemplo, para el programa

$$\textbf{newvar}\ x\ \textbf{real in}\ x := x + \textbf{true}$$

la función de inferencia de tipos devolverá error.

6.4.5. Inferencia de tipos

Las funciones tc (*type checking*) y ti (*type inference*), tienen la siguiente signatura:

$$\begin{aligned}
&\mathsf{tc}\ :\ (\pi\ :\ \mathsf{Ctx}) \rightarrow (\mathsf{e}\ :\ \mathsf{GExp}) \rightarrow (\mathsf{t}\ :\ \mathsf{Type}) \rightarrow (\pi \vdash \mathsf{e} : \mathsf{t}) \uplus \mathsf{Error}\\
&\mathsf{ti}\ :\ (\pi\ :\ \mathsf{Ctx}) \rightarrow (\mathsf{e}\ :\ \mathsf{GExp}) \rightarrow \exists\ (\lambda\ \mathsf{t} \rightarrow \pi \vdash \mathsf{e} : \mathsf{t}) \uplus \mathsf{Error}
\end{aligned}$$

El caso más sencillo es la aplicación de ti a una constante. Por ejemplo, ti π (GICon 1) devuelve el producto dependiente que consta del tipo δexp int y de una prueba de tipo $\pi \vdash$ GICon 1 : δexp int:

$$\mathsf{ti}\ \pi\ (\mathsf{GICon}\ 1) = \mathsf{inj}_1\ (\delta\mathsf{exp\ int},\ \mathsf{ICon}\ \{1\ \pi\})\ \ .$$

Como se ve en el ejemplo anterior, el tipo que infiere ti es siempre el menor posible. Si por ejemplo queremos verificar que GICon 1 tiene tipo δexp real bajo el contexto π, podemos aplicar tc para obtener una derivación de tipo $\pi \vdash$ GICon 1 : δexp real:

$$\mathsf{tc}\ \pi\ (\mathsf{GICon}\ 1)\ (\delta\mathsf{exp\ real}) = \mathsf{inj}_1\ \mathsf{Sub}\ (\mathsf{ICon}\ \{1\ \pi\})\ (\delta\mathsf{exp\text{-}sub\ ir})\ \ .$$

Las funciones de inferencia y chequeo de tipos son mutuamente recursivas, ya que la función ti necesita en algunos casos chequear tipos usando tc y por otro lado, la función tc (como se ve en el ejemplo anterior) devuelve la derivación que obtiene de ti pero extendida (si es necesario) con la regla de subsunción. Esto último se refleja en la definición de tc:

```
tc : (π : Ctx) → (e : GExp) → (t : Type) → (π ⊢ e : t) ⊎ Error
tc π e t with ti π e
tc π e t | inj₁ (t′, π⊢e:t′) with t′ | ≼dec t′ t
... | .t | yes τrefl = inj₁ π⊢e:t′
... | _  | yes t′≼t  = inj₁ (Sub (π⊢e:t′) t′≼t)
... | _  | no _ = inj₂ (coerce-err t′ t)
tc π e t | inj₂ err = inj₂ err
```

Para inferir el tipo de un identificador, ti lo busca en el contexto con la función lookup (ver sección 6.4.3). Si lo encuentra, devuelve el tipo asociado y en otro caso, el identificador está fuera de scope (devuelve error):

```
ti π (GVar ι) with lookup π ι
... | inside ι∈π = inj₁ (gettype ι∈π, Var ι∈π)
... | outside = inj₂ (scope-err ι)
```

En el caso de la abstracción GLam ι t e, la función ti infiere primero el tipo de e extendiendo el contexto con el par (ι , t) (utiliza la anotación t para dar el tipo de ι en lugar de inferirlo):

$$
\begin{array}{l}
\mathsf{ti}\ \pi\ (\mathsf{GLam}\ \iota\ \mathsf{t}\ \mathsf{e})\ \textbf{with}\ \mathsf{ti}\ ((\iota, \mathsf{t}) :: \pi)\ \mathsf{e} \\
\ldots\ |\ \mathsf{inj}_1\ (\mathsf{t}', \pi'{\vdash}\mathsf{e}{:}\mathsf{t}')\ =\ \mathsf{inj}_1\ ((\mathsf{t} \twoheadrightarrow \mathsf{t}'), \mathsf{Lam}\ \pi'{\vdash}\mathsf{e}{:}\mathsf{t}') \\
\ldots\ |\ \mathsf{inj}_2\ \mathsf{err}\ =\ \mathsf{inj}_2\ (\mathsf{lam\text{-}err}\ \mathsf{err})
\end{array}
$$

En la aplicación GApp e e', primero infiere el tipo de e, que debe ser un tipo funcional t'$\twoheadrightarrow$t (en otro caso, devuelve error). Luego verifica usando tc que e' tiene tipo t' (es decir, que puede ser un argumento de la función e).

$$
\begin{array}{l}
\mathsf{ti}\ \pi\ (\mathsf{GApp}\ \mathsf{e}\ \mathsf{e}')\ \textbf{with}\ \mathsf{ti}\ \pi\ \mathsf{e} \\
\mathsf{ti}\ \pi\ (\mathsf{GApp}\ \mathsf{e}\ \mathsf{e}')\ \ |\ \ \mathsf{inj}_1\ ((\mathsf{t} \twoheadrightarrow \mathsf{t}'), \pi{\vdash}\mathsf{e}{:}\mathsf{t}{\twoheadrightarrow}\mathsf{t}')\ \textbf{with}\ \mathsf{tc}\ \pi\ \mathsf{e}'\ \mathsf{t} \\
\ldots\ |\ \mathsf{inj}_1\ (\pi{\vdash}\mathsf{e}'{:}\mathsf{t})\ =\ \mathsf{inj}_1\ (\mathsf{t}', \mathsf{App}\ \pi{\vdash}\mathsf{e}{:}\mathsf{t}{\twoheadrightarrow}\mathsf{t}'\ \pi{\vdash}\mathsf{e}'{:}\mathsf{t}) \\
\ldots\ |\ \mathsf{inj}_2\ \mathsf{err}\ =\ \mathsf{inj}_2\ (\mathsf{argapp\text{-}err}\ \mathsf{err}) \\
\mathsf{ti}\ \pi\ (\mathsf{GApp}\ \mathsf{e}\ \mathsf{e}')\ \ |\ \ \mathsf{inj}_1\ \_\ =\ \mathsf{inj}_2\ \mathsf{notfun\text{-}err} \\
\mathsf{ti}\ \pi\ (\mathsf{GApp}\ \mathsf{e}\ \mathsf{e}')\ \ |\ \ \mathsf{inj}_2\ \mathsf{err}\ =\ \mathsf{inj}_2\ (\mathsf{funapp\text{-}err}\ \mathsf{err})
\end{array}
$$

El código completo de la función ti se encuentra en la figura A.2 del apéndice.

6.4.6. Inferencia en procedimientos recursivos

En el capítulo 5 vimos la definición de tipos simples (5.4). Las frases de tipo simple pueden compilarse a una subrutina. La representación de los tipos simples en Agda se realiza con el tipo Simple t (que puede pensarse como una prueba de que t es simple):

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Simple}\ :\ \mathsf{Type} \rightarrow \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{compl}\ :\ \mathsf{Simple}\ \mathsf{compl} \\
\quad \mathsf{intcompl}\ :\ \mathsf{Simple}\ (\delta\mathsf{compl}\ \mathsf{int}) \\
\quad \mathsf{realcompl}\ :\ \mathsf{Simple}\ (\delta\mathsf{compl}\ \mathsf{real}) \\
\quad \_{\twoheadrightarrow}\_\ :\ \{\mathsf{t}\ \mathsf{t}'\ :\ \mathsf{Type}\} \rightarrow \mathsf{Simple}\ \mathsf{t} \rightarrow \mathsf{Simple}\ \mathsf{t}' \rightarrow \mathsf{Simple}\ (\mathsf{t} \twoheadrightarrow \mathsf{t}')
\end{array}
$$

En la sección 5.3.10 definimos el conjunto $\hat{\theta}$ que contiene a aquellos tipos cuya simplificación via la función Γ daba como resultado un sólo tipo simple y no una tupla de dos o más tipos simples. Nuestra implementación restringe los tipos de los procedimientos recursivos a los tipos en $\hat{\theta}$ (como se explica más adelante). El tipo Type_1 representa al conjunto $\hat{\theta}$; es decir, hay un constructor para cada tipo que pertenece a $\hat{\theta}$.

```
data Type₁ : Type → Set where
  int-exp : Type₁ (δexp int)
  real-exp : Type₁ (δexp real)
  bool-exp : Type₁ (δexp bool)
  compl : Type₁ (compl)
  comm : Type₁ (comm)
  int-acc : Type₁ (δacc int)
  real-acc : Type₁ (δacc real)
  int-compl : Type₁ (δcompl int)
  real-compl : Type₁ (δcompl real)
  _⇾_ : {t t′ : Type} → Type₁ t → Type₁ t′ → Type₁ (t ⇾ t′)
```

En la inferencia de tipos de GLetrec ι t p p', la función ti verficará que se puede construir una prueba Types_1 t (para eso utiliza la función isType_1) y en otro caso devuelve error:

```
ti π (GLetrec ι t e e′) with tc ((ι, t) :: π) e t | ti ((ι, t) :: π) e′
ti π (GLetrec ι t e e′) | inj₁ (π′⊢e:t) | inj₁ (t′, π′⊢e′:t′) with isType₁ t
... | just t₁   = inj₁ (t′, Lrec₁ t₁ π′⊢e:t π′⊢e′:t′)
... | nothing  = inj₂ (type₁-err t)
ti π (GLetrec ι t e e′) | inj₂ err | _ = inj₂ (beletrec-err err)
ti π (GLetrec ι t e e′) | _ | inj₂ err = inj₂ (inletrec-err err)
```

6.5. Generación de Código Intermedio

El generador de código intermedio parte del árbol de derivación y produce código en el lenguaje intermedio. En esta sección mostramos la representación del lenguaje intermedio en Agda, y la implementación de las funciones que llevan a cabo la traducción.

6.5.1. Descriptores de Pila

Representamos los descriptores de pila como un par de números naturales:

$$\mathsf{SD} = \mathbb{N} \times \mathbb{N}$$

Llamaremos fc (*frame count*) y disp (*displacement*) a la primera y segunda proyección del descriptor, respectivamente. Definimos la suma de un descriptor y un número natural de la siguiente manera:

```
_∔_ : SD → ℕ → SD
(f, d) ∔ x = (f, d + x)
```

6.5.2. El lenguaje intermedio

El lenguaje intermedio lo definimos en la sección 5.3.2. En Agda lo representamos definiendo varios tipos abstractos. El primer tipo que definimos es el de los "valores asignables" del lenguaje intermedio, es decir, aquellos que pueden ocurrir en el lado izquierdo (*left hand side*) de una asignación (los descriptores de pila y el registro Sbrs)

```
data Lh (s : SD) : Set where
  S : (s' : SD) → Lh s
  Sbrs : Lh s
```

En el lado derecho de una asignación pueden aparecer valores *simples* (sin operadores) como los descriptores o las constantes:

```
data Sh (s : SD) : Set where
  Lh↑ : Lh s → Sh s
  litδ : (τ : DType) → δ⟦ τ ⟧ → Sh s
```

El constructor litδ toma dos argumentos

- Un tipo de datos τ.
- Una constante de tipo $\delta[\![\ \tau\ ]\!]$, donde la función $\delta[\![\_]\!]$ asigna a cada tipo de datos un tipo de la librería estándar de Agda:

```
δ⟦_⟧ : DType → Set
δ⟦ int ⟧ = Int
δ⟦ real ⟧ = Real
δ⟦ bool ⟧ = Bool
```

Del lado derecho de la asignación (*right hand side*) pueden ocurrir también expresiones con a lo sumo un operador:

```
data Rh (s : SD) : Set where
  Sh↑ : Sh s → Rh s
  I₁  : {op : Op₁} → Iop₁ op → Sh s → Rh s
  R₁  : {op : Op₁} → Rop₁ op → Sh s → Rh s
  I₂  : {op : Op₂} → Iop₂ op → Sh s → Sh s → Rh s
  R₂  : {op : Op₂} → Rop₂ op → Sh s → Sh s → Rh s
  toReal : Sh s → Rh s
```

Los constructores I_1 e I_2 toman operadores enteros unarios y binarios, respectivamente. Un elemento de tipo Iop_1 op es una prueba de que el operador op es entero y unario. Por otro lado, el constructor toReal se utiliza para indicar una coerción entre expresiones enteras y reales.

Por último, describimos el tipo de datos I s que representa secuencias de instrucciones. Consideremos por ejemplo el siguiente constructor:

$$[\_]\text{-}[\_]\_{:=}\_\!\gg\_ \;:\; (\delta \;:\; \mathbb{Z}) \to (\mathsf{s}' \;:\; \mathsf{SD}) \to \mathsf{Lh}\ \mathsf{s}' \to \mathsf{Rh}\ \mathsf{s} \to \mathsf{I}\ \mathsf{s}' \to \mathsf{I}\ \mathsf{s}$$

Este constructor representa la asignación de un registro o descriptor. Los argumentos que toma son los siguientes:

- Un entero δ (desplazamiento del descriptor de pila actual)
- El descriptor de pila s' apropiado para la continuación.
- El lado izquierdo, una expresión de tipo Lh.
- Una expresión de tipo Rh (lado derecho).
- La continuación de tipo I s'.

La siguiente es la representación de las secuencias de instrucciones:

```
mutual
data I (s : SD) : Set where
  Stop : I s
  [_]-[_] _:=_» _ : (δ : ℤ) → (s′ : SD) → Lh s′ → Rh s → I s′ → I s
  if_:_,_[_] ⟨_⟩then_else_ : Rel → Sh s → Sh s → (δ : ℤ) → (s′ : SD)
      → I s′ → I s′ → I s
  adjust[_]-[_] »_ : (δ : ℤ) → (s′ : SD) → I s′ → I s
  popto_»_ : (s′ : SD) → I s′ → I s
  call : (f : ℕ) → I (f ⁺) → ArgList s → I s
  acall : (j : ℕ) → (f : ℕ) → ArgList s → I s
  ajump : (j : ℕ) → I s
  jmp : Label → I s
  if_then»_else»_ : Lh s → I s → I s → I s
ArgList : SD → Set
ArgList s = List (∃₂ (λ t φ → I (sr {t} s φ)))
```

Hay un constructor para cada instrucción del lenguaje intermedio que vimos en el capítulo 5. El segundo constructor if no es realmente necesario ya que puede definirse usando el primero; pero lo incluimos porque facilita la definición de algunas ecuaciones de las expresiones booleanas. Por último, notemos que el tipo de ArgList se utiliza para almacenar los argumentos de la instrucción call.

Nota sobre la representación

Nuestra representación del lenguaje intermedio es más permisiva que la gramática que lo define, es decir, hay elementos de tipo I s que no representan ninguna secuencia de instrucciones posible. Por ejemplo, la restricción $S \leq S^{curr} - 1$ de la definición 5.3.2 se omitió en el constructor S del tipo Lh. El constructor de asignación podría haberse definido como sigue:

$$[\_]\text{-}[\_]\_{:=}\_\!\gg\_ \;:\; (\delta \;:\; \mathbb{Z}) \to \mathsf{Lh}\ (\mathsf{s} \dotplus \delta) \to \mathsf{Rh}\ \mathsf{s} \to \mathsf{I}\ (\mathsf{s} \dotplus \delta) \to \mathsf{I}\ \mathsf{s}$$

pero para simplificar la implementación nos quedamos con la definición presentada anteriormente.

6.5.3. Semántica de Tipos

El funtor $[\![\_]\!] : \Theta \to \mathcal{K}$ asigna a cada tipo θ un objeto de la categoría semántica. Con la elección de $\mathcal{K}$ como PDOM$^\Sigma$ sabemos que $[\![\, \theta \,]\!]$ es un funtor de Σ en PDOM, y que por lo tanto $[\![\, \theta \,]\!]\, S$ es un predominio. Si S y S' son descriptores de pila y $S \leq S'$, entonces $[\![\, \theta \,]\!]\, (S \leq S')$ es una función continua que va de $[\![\, \theta \,]\!]\, S$ en $[\![\, \theta \,]\!]\, S'$.
Para dar la semántica de los tipos en Agda uno puede definir dos funciones:

- $\tau[\![\_]\!]$: $\mathsf{Type} \to \mathsf{SD} \to \mathsf{Set}$
- $[\![\_]\!]\langle\_,\_\rangle$: $(\mathsf{t} : \mathsf{Type}) \to (\mathsf{s} : \mathsf{SD}) \to (\mathsf{s}' : \mathsf{SD}) \to \tau[\![\, \mathsf{t} \,]\!]\, \mathsf{s} \to \tau[\![\, \mathsf{t} \,]\!]\, \mathsf{s}'$

Informalmente[3], podemos pensar que $\tau[\![\ \theta\ ]\!]$ representa al funtor $[\![\ \theta\ ]\!]$ y que $[\![\, \theta \,]\!]\langle S, S'\rangle$ representa la función $[\![\, \theta \,]\!](S \leq S')$.

Consideremos por ejemplo la definición de $\tau[\![\ \mathsf{compl}\ ]\!]$. Una opción sería definirlo como sigue:

$$\tau[\![\,\mathsf{compl}\,]\!]\,\mathsf{s} = \mathsf{I}\,\mathsf{s}$$

pero como veremos más adelante en este capítulo, utilizaremos una mónada de estado para resolver algunos problemas como la duplicación de código. Entonces, "encerrando" a $\mathsf{I}\,\mathsf{s}$ en la mónada $\mathsf{LabelState}$ obtenemos:

$$\tau[\![\,\mathsf{compl}\,]\!]\,\mathsf{s} = \mathsf{LabelState}(\mathsf{I}\,\mathsf{s})$$

La definición de $[\![\ \mathbf{compl}\ ]\!](S \leq S')$ (5.3) se representa en la implementación de la siguiente manera

```
i-cumm : (s s' : SD) → I s → I s'
i-cumm s s' k with s ≟ s'
i-cumm s .s k | yes refl = k
i-cumm s s' k | no _ =
   if (fc s') =n (fc s) then adjust [disp s ⊖ disp s'] - [s] » k else popto s » k
⟦_⟧⟨_,_⟩ : (t : Type) → (s : SD) → (s' : SD) → τ⟦ t ⟧ s → τ⟦ t ⟧ s'
⟦ compl ⟧⟨ s, s' ⟩ κ = κ ≫= λ k → ↑ i-cumm s s' k
```

En el código A.5 del apéndice se encuentra la definición completa de la semántica de tipos.

[3]Estamos utilizando un abuso de notación con las metavariables, por ejemplo, θ denota un tipo y su representante en Types

Entornos

El siguiente código define la representación de los entornos:

```
data Env : Ctx → SD → Set where
  [] : {s : SD} → Env [] s
  _↦_::_ : ∀ {π t s} → (ι : ID) → τ⟦ t ⟧ s → (η : Env π s) → Env ((ι, t) :: π) s

⟦_⟧* : Ctx → SD → Set
⟦ π ⟧* s = Env π s

⟦_⟧*⟨_, _⟩ : (π : Ctx) → (s s′ : SD) → ⟦ π ⟧* s → ⟦ π ⟧* s′
⟦ . [] ⟧*⟨ s, s′ ⟩ [] = []
⟦ (ι, t) :: π ⟧*⟨ s, s′ ⟩ (.ι ↦ x :: η)
  = (ι ↦ ⟦ t ⟧⟨ s, s′ ⟩ x :: ⟦ π ⟧*⟨ s, s′ ⟩ η)
```

Como antes, la acción del funtor $[\![\pi]\!]^*$ a los objetos y a los morfismos se representa en Agda con las dos funciones anteriores. Notar que en el caso los morfismos, el funtor actúa punto a punto, como se definió en la ecuación 5.1.

Coerciones

Si t y t' son tipos, entonces $[\![t \leq t']\!] : [\![t]\!] \to [\![t']\!]$ es una transformación natural, cuyos componentes $[\![t \leq t']\!]S : [\![t]\!]S \to [\![t']\!]S$ son funciones continuas que representan conversiones entre tipos. En Agda, representamos estas conversiones con la función S⟦_⟧ definida en el código A.7 del apéndice. Consideremos por ejemplo, el caso de la conversión entre expresiones, que se define como sigue:

```
S⟦_⟧ : ∀ {t t′} → t ≼ t′ → (s : SD) → τ⟦ t ⟧ s → τ⟦ t′ ⟧ s
S⟦_⟧ (δexp-sub δrefl) s x = x
S⟦_⟧ (δexp-sub ir) s m = m ⋙= λ e →
  ↑ λ s′ m′ → m′ ⋙= λ β →
    e s′ (↑ useTemp s′ (λ s″ r → β s″ (toReal r)))
```

6.5.4. Subrutinas

Recordemos que para generar subrutinas utilizamos la familia de funciones definida en 5.3.10. Como ejemplo consideremos la función:

$$\text{mk-subr}_{\varphi} S \in [\![\varphi]\!] S \to \langle SR^{\varphi}_{S} \rangle$$

La implementación de mk-subr se realizó con la siguiente función en Agda (notar que representamos a $\langle SR^{\varphi}_{S} \rangle$ con el tipo srφ s φ):

$$
\begin{array}{l}
\mathsf{gmk\text{-}subr}\ :\ \{\mathsf{t}\ :\ \mathsf{Type}\} \to \mathbb{N} \to (\varphi\ :\ \mathsf{Simple\ t}) \to (\mathsf{s}\ :\ \mathsf{SD}) \to \tau[\![\ \mathsf{t}\ ]\!]\ \mathsf{s} \to \mathsf{sr}\varphi\ \mathsf{s}\ \varphi \\
\mathsf{gmk\text{-}subr\ n\ compl\ s}\ \kappa\ =\ \kappa \\
\mathsf{gmk\text{-}subr\ n\ intcompl\ s}\ \beta\ =\ \mathsf{saveres\ s}\ \beta \\
\mathsf{gmk\text{-}subr\ n\ realcompl\ s}\ \beta\ =\ \mathsf{saveres\ s}\ \beta \\
\mathsf{gmk\text{-}subr\ n}\ (\varphi \multimap \mathsf{compl})\ \mathsf{s\ m}\ =\ \mathsf{m}\ \ggg\!=\ \lambda\ \mathsf{c} \to \mathsf{c}\ (\mathsf{s}\ _+)\ (\mathsf{gmk\text{-}argcall}\ [\,]\ \varphi\ (\mathsf{s}\ _+)\ \mathsf{n}) \\
\mathsf{gmk\text{-}subr\ n}\ (\varphi \multimap \mathsf{intcompl})\ \mathsf{s\ m} \\
\quad =\ \mathsf{m}\ \ggg\!=\ \lambda\ \beta \to \\
\qquad \mathsf{saveres\ s^+}\ (\beta\ \mathsf{s^+}\ (\mathsf{gmk\text{-}argcall}\ [\,]\ \varphi\ \mathsf{s^+\ n})) \\
\qquad \mathbf{where}\ \mathsf{s^+}\ =\ (\mathsf{s}\ _+) \\
\mathsf{gmk\text{-}subr\ n}\ (\varphi \multimap \mathsf{realcompl})\ \mathsf{s\ m} \\
\quad =\ \mathsf{m}\ \ggg\!=\ \lambda\ \beta \to \\
\qquad \mathsf{saveres\ s^+}\ (\beta\ \mathsf{s^+}\ (\mathsf{gmk\text{-}argcall}\ [\,]\ \varphi\ \mathsf{s^+\ n})) \\
\qquad \mathbf{where}\ \mathsf{s^+}\ =\ (\mathsf{s}\ _+) \\
\mathsf{gmk\text{-}subr\ n}\ (\varphi \multimap (\varphi' \multimap \varphi''))\ \mathsf{s\ m} \\
\quad =\ \mathsf{m}\ \ggg\!=\ \lambda\ \mathsf{c} \to \\
\qquad \mathsf{c}\ (\mathsf{s}\ _+)\ (\mathsf{gmk\text{-}argcall}\ [\,]\ \varphi\ (\mathsf{s}\ _+)\ \mathsf{n}) \ggg \lambda\ \mathsf{c}' \to \\
\qquad \mathsf{gmk\text{-}subr}\ (\mathsf{suc\ n})\ (\varphi' \multimap \varphi'')\ \mathsf{s\ c}'
\end{array}
$$

La función mk-subr es un caso particular de gmk-subr (con $\mathsf{n} = 1$) :

$$
\begin{array}{l}
\mathsf{mk\text{-}subr}\ :\ \{\mathsf{t}\ :\ \mathsf{Type}\} \to (\varphi\ :\ \mathsf{Simple\ t}) \to (\mathsf{s}\ :\ \mathsf{SD}) \to \tau[\![\ \mathsf{t}\ ]\!]\ \mathsf{s} \to \mathsf{sr}\varphi\ \mathsf{s}\ \varphi \\
\mathsf{mk\text{-}subr}\ \varphi\ =\ \mathsf{gmk\text{-}subr}\ 1\ \varphi
\end{array}
$$

El resto de las funciones para generar subrutinas se encuentran en el código A.9 del apéndice.

6.5.5. Traducción

En el capítulo 5 vimos que si $\pi \vdash e : t$ es un juicio, entonces $[\![\ e\ ]\!]_{\pi,t}$ es una transformación natural entre los funtores $[\![\ \pi\ ]\!]^*$ y $[\![\ t\ ]\!]$, es decir:

$$[\![\ e\ ]\!]_{\pi,t}\ \in [\![\ \pi\ ]\!]^* \xrightarrow[\mathcal{K}]{} [\![\ t\ ]\!]$$

Por lo tanto, tenemos que cada componente $[\![\ e\ ]\!]_{\pi,t}\ S$ de la transformación natural es una función continua de $[\![\ \pi\ ]\!]^* S$ en $[\![\ t\ ]\!] S$:

$$[\![\ e\ ]\!]_{\pi,t}\ S \in [\![\ \pi\ ]\!]^* S \to [\![\ t\ ]\!] S$$

En Agda representamos estos componentes con la función eval:

$$\mathsf{eval}\ :\ \forall\ \{\pi\ \mathsf{e\ t}\} \to (\pi \vdash \mathsf{e : t}) \to (\mathsf{s}\ :\ \mathsf{SD}) \to [\![\ \pi\ ]\!]^*\ \mathsf{s} \to \tau[\![\ \mathsf{t}\ ]\!]\ \mathsf{s}$$

Consideremos por ejemplo la ecuación para los operadores enteros

$$
\begin{array}{l}
[\![\ e_1 \oplus e_2\ ]\!]_{\pi,\mathbf{intexp}}\ S\eta S'\beta = \\
\qquad [\![\ e_1\ ]\!]_{\pi,\mathbf{intexp}}\ S\eta S'(\mathrm{usetmp}\ S'(\lambda S''.\lambda r_1. \\
\qquad\qquad [\![\ e_2\ ]\!]_{\pi,\mathbf{intexp}}\ S\eta S''(\mathrm{usetmp}\ S''(\lambda S'''.\lambda r_2.\ \beta S'''(r_1 \oplus r_2)))))\ \ ,
\end{array}
$$

que se representa con la siguiente definición en eval

$$
\begin{array}{l}
\mathsf{eval}\ (\mathsf{Op2I}\ \{\pi\}\ \{\mathsf{e}\}\ \{\mathsf{e}'\}\ \mathsf{iop}\ \mathsf{j}_1\ \mathsf{j}_2)\ \mathsf{s}\ \eta \\
\quad = \ \mathsf{eval}\ \mathsf{j}_1\ \mathsf{s}\ \eta \ \gg\!= \ \lambda\ \mathsf{f} \rightarrow \\
\qquad \mathsf{eval}\ \mathsf{j}_2\ \mathsf{s}\ \eta \quad \gg\!= \ \lambda\ \mathsf{g} \rightarrow \\
\qquad \uparrow \lambda\ \mathsf{s}'\ \mathsf{m} \rightarrow \mathsf{m}\ \gg\!= \ \lambda\ \beta \rightarrow \mathsf{f}\ \mathsf{s}'\ (\uparrow \mathsf{useTemp}\ \mathsf{s}'\ (\lambda\ \mathsf{s}''\ \mathsf{r}_1 \rightarrow \\
\qquad \mathsf{g}\ \mathsf{s}''\ (\uparrow \mathsf{useTemp}\ \mathsf{s}''\ (\lambda\ \mathsf{s}'''\ \mathsf{r}_2 \rightarrow \beta\ \mathsf{s}'''\ (\mathsf{I}_2\ \mathsf{iop}\ (\mathsf{sh\text{-}cumm}\ \mathsf{r}_1)\ \mathsf{r}_2)))))
\end{array}
$$

donde iop es una prueba de que el operador es binario y entero, y j_1 es una derivación de que el primer operando es una expresión entera (análogamente para j_2).
Utilizamos $\mathsf{S}[\![\_]\!]$ para traducir la regla de subsunción que representa la coerción entre tipos:

$$
\mathsf{eval}\ (\mathsf{Sub}\ \mathsf{j}\ \mathsf{p})\ \mathsf{s}\ \eta \ = \ \mathsf{S}[\![\ \mathsf{p}\ ]\!]\ \mathsf{s}\ (\mathsf{eval}\ \mathsf{j}\ \mathsf{s}\ \eta)
$$

La definición completa de eval está en el código A.11 del apéndice.

Duplicación de Código

Consideremos la ecuación para el comando **if** que vimos en el capítulo 5:

$$
\begin{array}{l}
[\![\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ ]\!]_{\pi,\textbf{comm}}\ S\eta S'\kappa = \\
\quad [\![\ b\ ]\!]_{\pi,\textbf{boolexp}}\ S\eta S'\langle [\![\ c_1\ ]\!]_{\pi,\textbf{comm}}\ S\eta S'\kappa, [\![\ c_2\ ]\!]_{\pi,\textbf{comm}}\ S\eta S'\kappa\rangle
\end{array}
\tag{6.3}
$$

Notar que la secuencia de instrucciones κ aparece en ambos componentes de la continuación del comando **if** . Por lo tanto, si representamos la ecuación 6.3 directamente, el código de κ aparecerá al menos dos veces en la traducción. Podemos solucionar este problema manteniendo una lista de asociaciones de *labels* con instrucciones. Por ejemplo, si tenemos la asociación $\ell \mapsto \kappa$ podemos reemplazar cada ocurrencia de κ en la ecuación por un "salto" a la instrucción con label ℓ. Implementamos la lista de asociaciones, que llamamos LabelMap, como sigue:

$$
\mathsf{LabelMap} \ = \ \mathsf{List}\ (\mathsf{Label} \times \exists\ \mathsf{I})
$$

Utilizamos la mónada de estado de la librería estándar de Agda para mantener la información sobre la tabla. Un estado será un par donde la primera componente contiene el próximo label a utilizar, y la segunda componente contiene la lista de asociaciones:

$$
\begin{array}{l}
\mathsf{StateTy} \ = \ \mathsf{Label} \times \mathsf{LabelMap} \\
\mathsf{LabelState} \ = \ \mathsf{State}\ \mathsf{StateTy}
\end{array}
$$

Entonces, la ecuación 6.3 se representa en eval con la definición:

$$
\begin{array}{l}
\mathsf{eval}\ (\mathsf{CommIf}\ \mathsf{j}\ \mathsf{j}'\ \mathsf{j}'')\ \mathsf{s}\ \eta \\
=\ \mathsf{eval}\ \mathsf{j}\ \mathsf{s}\ \eta\ \ggg\!=\ \lambda\ \mathsf{b} \rightarrow \\
\quad \mathsf{eval}\ \mathsf{j}'\ \mathsf{s}\ \eta\ \ggg\!=\ \lambda\ \mathsf{c}_1 \rightarrow \\
\quad \mathsf{eval}\ \mathsf{j}''\ \mathsf{s}\ \eta\ \ggg\!=\ \lambda\ \mathsf{c}_2 \rightarrow \\
\quad \uparrow \lambda\ \mathsf{s}'\ \kappa \rightarrow \kappa\ \ggg\!=\ \lambda\ \mathsf{k} \rightarrow \mathsf{withNewLabel}\ \mathsf{k}\ \ggg\!=\ \lambda\ \mathsf{st} \rightarrow \\
\quad \mathsf{jmp}\ (\mathsf{proj}_1\ \mathsf{st}) \ggg \lambda\ \mathsf{kjmp} \rightarrow \\
\quad \mathsf{b}\ \mathsf{s}'\ (\uparrow (\mathsf{c}_1\ \mathsf{s}'\ (\uparrow \mathsf{kjmp}), \mathsf{c}_2\ \mathsf{s}'\ (\uparrow \mathsf{kjmp})))
\end{array}
$$

donde la función withNewLabel (definida en el código A.8 del apéndice) modifica el estado asignando un nuevo label a la secuencia de instrucciones que toma como argumento.

Procedimientos recursivos

Recordemos la ecuación 5.6 de las definiciones recursivas:

$$
\begin{array}{c}
[\![\ \mathbf{letrec}\ \iota : \tau\ \mathbf{be}\ p\ \mathbf{in}\ p'\ ]\!]_{\pi,\theta'}\ S\,\eta = [\![\ p'\ ]\!]_{[\pi\,|\,\iota:\tau],\theta'}\ S\eta' \\
\eta' = [\eta\,|\,\iota : \psi_\tau S\,(\text{mk-subr}_\varphi\,S\,i)\,] \\
i = \text{mk-subr}_\varphi\,S\,(\phi_\tau S\,([\![\ p\ ]\!]_{[\pi\,|\,\iota:\tau],\tau}\ S\,\eta'))
\end{array}
$$

Como η' e i son mutuamente dependientes, esta ecuación puede generar una secuencia infinita de instrucciones. Si asignamos a i un label ℓ y luego reemplazamos i por jmp ℓ en η' entonces eliminamos esa dependencia mutua. La representación de esa ecuación en eval queda como sigue:

$$
\begin{array}{l}
\mathsf{eval}\ (\mathsf{Lrec}_1\ \{\pi\}\ \{\iota\}\ \{\mathsf{e}\}\ \{\mathsf{e}'\}\ \{\_\}\ \{\_\}\ \mathsf{t}_1\ \mathsf{p}\ \mathsf{p}')\ \mathsf{s}\ \eta \\
\quad =\ \mathsf{reserveLabel}\ \ggg\!=\ \lambda\ \mathsf{st} \rightarrow \\
\quad\quad \mathsf{mk\text{-}call}_1\ \mathsf{t}_1\ \mathsf{s}\ (\uparrow \mathsf{jmp}\ (\mathsf{proj}_1\ \mathsf{st})) \ggg \lambda\ \mathsf{scall} \rightarrow \\
\quad\quad (\iota \mapsto \mathsf{scall} :: \eta) \ggg \lambda\ \eta' \rightarrow \\
\quad\quad \mathsf{eval}\ \mathsf{p}\ \mathsf{s}\ \eta' \ggg \lambda\ \mathsf{k} \rightarrow \\
\quad\quad \mathsf{mk\text{-}subr}_1\ \mathsf{t}_1\ \mathsf{s}\ \mathsf{k}\ \ggg\!= \\
\quad\quad \mathsf{assignLabel}\ (\mathsf{proj}_1\ \mathsf{st})\ \gg \\
\quad\quad \mathsf{eval}\ \mathsf{p}'\ \mathsf{s}\ \eta'
\end{array}
$$

Las funciones reserveLabel y assignLabel se definen en el código A.8 del apéndice.

Capítulo 7

Conclusiones

En esta tesis implementamos un front-end para el lenguaje *Peal* , aplicando un método para generar código intermedio a partir de la semántica funtorial del lenguaje. La categoría funtorial nos permitió explicitar la disciplina de pila en las ecuaciones semánticas.
Lo que implementamos finalmente para *Peal* fue:

- El parser, utilizando la librería Parsec.
- Las funciones de chequeo e inferencia de tipos, que se encargan de construir un árbol de derivación utilizando las reglas de tipado.
- La función de traducción, que toma el árbol de derivación y produce (en escencia) una secuencia de instrucciones en código intermedio. El hecho de que las traducciones se definen sobre el árbol de derivación, nos dice que sólo estamos definiendo semántica para expresiones bien tipadas.

Una limitación de nuestra implementación es la restricción de los tipos de los procedimientos recursivos que se definen con **letrec** a aquellos tipos cuya simplificación vía la función Γ da como resultado un único tipo simple y no una tupla de tipos simples (llamamos a esos tipos $\hat{\theta}$, ver sección 6.4.6). Esto puede solucionarse implementando los isomorfimos de tipos a tuplas de tipos simples definidos en la sección 5.7.
Una mejora al trabajo sería formalizar el agregado de referencias a instrucciones. Nosotros utilizamos mónadas de estado para poder representar las secuencias infinitas de instrucciones y evitar duplicación de código, pero es un agregado artificial que no guarda relación con las ecuaciones semánticas. Tal vez eligiendo una categoría diferente como modelo semántico se evita el uso de las mónadas y se formaliza el uso de *labels* en las ecuaciones de la teoría.
El método de generación de código podría extenderse para lenguajes y sistemas de tipos más complejos. El lenguaje Forsythe [27], diseñado por Reynolds, es una especie de "Algol ideal" que tiene un sistema de tipos más general y más flexible que el de *Peal* . Un primer objetivo que nos planteamos al comienzo del trabajo

fue construir el front-end para Forsythe, siendo el front-end de *Peal* el primer paso para conseguir ese objetivo.
John Reynolds menciona en [25, sec 17] que dada la fuerte relación que hay entre la semántica funtorial y el método de generación de código, uno podría utilizar este enfoque para construir pruebas de correctitud del compilador.
Por último, la construcción de un back-end para *Peal* es claramente una manera de continuar este trabajo. Como una opción, mencionamos la posibilidad de traducir el lenguaje intermedio a código LLVM [15].

Apéndice A

Segmentos de Código

Código A.1 Reglas de inferencia

```
data _⊢_:_ : Ctx → GExp → Type → Set where
  ICon : {x : Int} → {π : Ctx} → π ⊢ GICon x : intexp
  BCon : {b : Bool} → {π : Ctx} → π ⊢ GBCon b : boolexp
  RCon : {r : Real} → {π : Ctx} → π ⊢ GRCon r : realexp
  Op1I : ∀ {π e op} → Iop₁ op → π ⊢ e : intexp → π ⊢ GOp1 op e : intexp
  Op1R : ∀ {π e op} → Rop₁ op → π ⊢ e : realexp → π ⊢ GOp1 op e : realexp
  Op2I : ∀ {π e e′ op} → Iop₂ op → π ⊢ e : intexp → π ⊢ e′ : intexp
        → π ⊢ GOp2 op e e′ : intexp
  Op2R : ∀ {π e e′ op} → Rop₂ op → π ⊢ e : realexp → π ⊢ e′ : realexp
        → π ⊢ GOp2 op e e′ : realexp
  RelBR : ∀ {π e e′ r} → π ⊢ e : realexp → π ⊢ e′ : realexp
        → π ⊢ GRel r e e′ : boolexp
  RelBI : ∀ {π e e′ r} → π ⊢ e : intexp → π ⊢ e′ : intexp
        → π ⊢ GRel r e e′ : boolexp
  Op2B : ∀ {π e e′ op} → Bop₂ op → π ⊢ e : boolexp → π ⊢ e′ : boolexp
        → π ⊢ GOp2 op e e′ : boolexp
  Op1B : ∀ {π e op} → Bop₁ op → π ⊢ e : boolexp → π ⊢ GOp1 op e : boolexp
  CommIf : ∀ {π b e e′} → π ⊢ b : boolexp → π ⊢ e : comm → π ⊢ e′ : comm
    → π ⊢ GIf b e e′ : comm
  ComplIf : ∀ {π b e e′} → π ⊢ b : boolexp → π ⊢ e : compl → π ⊢ e′ : compl
    → π ⊢ GIf b e e′ : compl
  Var  : ∀ {π ι} → (p : ι ∈ π) → π ⊢ (GVar ι) : gettype p
  Lam  : ∀ {π ι e t t′} → ((ι, t) :: π) ⊢ e : t′ → π ⊢ (GLam ι t e) : (t ⇾ t′)
  App  : ∀ {π e e′ t t′} → π ⊢ e : (t ⇾ t′) → π ⊢ e′ : t → π ⊢ (GApp e e′) : t′
  Pair : ∀ {π e e′ t t′} → π ⊢ e : t → π ⊢ e′ : t′ → π ⊢ (GPair e e′) : prod t t′
  Fst  : ∀ {π e t t′} → π ⊢ e : prod t t′ → π ⊢ (GFst e) : t
  Snd  : ∀ {π e t t′} → π ⊢ e : prod t t′ → π ⊢ (GSnd e) : t′
  Esc  : ∀ {π ι e} → ((ι, compl) :: π) ⊢ e : comm → π ⊢ (GEscape ι e) : comm
  Lrec₁ : ∀ {π ι e e′ t t′} → Type₁ t → ((ι, t) :: π) ⊢ e : t
        → ((ι, t) :: π) ⊢ e′ : t′ → π ⊢ (GLetrec ι t e e′) : t′
  Newvar : ∀ {π ι d c} → ((ι, δvar d) :: π) ⊢ c : comm
        → π ⊢ GNewvar ι d c : comm
  ComplSeq : ∀ {π c c′} → π ⊢ c : comm → π ⊢ c′ : compl → π ⊢ GSeq c c′ : compl
  CommSeq : ∀ {π c c′} → π ⊢ c : comm → π ⊢ c′ : comm → π ⊢ GSeq c c′ : comm
  Assign : ∀ {π a e d} → π ⊢ a : δacc d → π ⊢ e : δexp d → π ⊢ GAssign a e : comm
  Skip : ∀ {π} → π ⊢ GSkip : comm
  Loop : ∀ {π c} → π ⊢ c : comm → π ⊢ (GLoop c) : compl
  While : ∀ {π b c} → π ⊢ b : δexp bool → π ⊢ c : comm → π ⊢ (GWhile b c) : comm
  Let  : ∀ {π ι p p′ t t′} → π ⊢ p : t → ((ι, t) :: π) ⊢ p′ : t′
        → π ⊢ (GLet ι p p′) : t′
  Sub  : ∀ {π e t t′} → π ⊢ e : t → t ≼ t′ → π ⊢ e : t′
```

Código A.2 Inferencia de tipos

```
ti : (π : Ctx) → (e : GExp) → ∃ (λ t → π ⊢ e : t) ⊎ Error
ti π (GICon _) = inj1 (intexp, ICon)
ti π (GBCon _) = inj1 (boolexp, BCon)
ti π (GRCon _) = inj1 (realexp, RCon)
ti π (GOp1 op e) with tc π e (δexp int) | isiop1 op
... | inj1 (π⊢e:i) | just iop = inj1 (intexp, Op1I iop π⊢e:i)
... | _ | _ with tc π e (δexp real) | isrop1 op
... | inj1 (π⊢e:r) | just rop = inj1 (realexp, Op1R rop π⊢e:r)
... | _ | _ with tc π e (δexp bool) | isbop1 op
... | inj1 (π⊢e:b) | just bop = inj1 (boolexp, Op1B bop π⊢e:b)
... | _ | _ = inj2 (badop1-err op)
ti π (GOp2 op e e′) with tc π e (δexp int) | tc π e′ (δexp int) | isiop2 op
... | inj1 π⊢e:i | inj1 π⊢e′:i | just iop = inj1 (intexp, Op2I iop π⊢e:i π⊢e′:i)
... | _ | _ | _ with tc π e (δexp real) | tc π e′ (δexp real) | isrop2 op
... | inj1 π⊢e:r | inj1 π⊢e′:r | just rop = inj1 (realexp, Op2R rop π⊢e:r π⊢e′:r)
... | _ | _ | _ with tc π e (δexp bool) | tc π e′ (δexp bool) | isbop2 op
... | inj1 π⊢e:b | inj1 π⊢e′:b | just bop = inj1 (boolexp, Op2B bop π⊢e:b π⊢e′:b)
... | _ | _ | _ = inj2 (badop2-err op)
ti π (GRel rel e e′) with tc π e (δexp int) | tc π e′ (δexp int)
... | inj1 (π⊢e:i) | inj1 (π⊢e′:i) = inj1 (boolexp, RelBI π⊢e:i π⊢e′:i)
... | _ | _ with tc π e (δexp real) | tc π e′ (δexp real)
... | inj1 (π⊢e:r) | inj1 (π⊢e′:r) = inj1 (boolexp, RelBR π⊢e:r π⊢e′:r)
... | inj2 err | _ = inj2 (frel-err rel err)
... | _ | inj2 err = inj2 (srel-err rel err)
ti π (GIf b e e′) with tc π b (δexp bool)
ti π (GIf b e e′) | inj2 err = inj2 (bif-err err)
ti π (GIf b e e′) | inj1 (π⊢b:bool) with tc π e compl | tc π e′ compl
... | inj1 (π⊢e:compl) | inj1 (π⊢e′:compl)
   = inj1 (compl, ComplIf π⊢b:bool π⊢e:compl π⊢e′:compl)
... | _ | _ with tc π e comm | tc π e′ comm
... | inj1 (π⊢e:comm) | inj1 (π⊢e′:comm)
   = inj1 (comm, CommIf π⊢b:bool π⊢e:comm π⊢e′:comm)
... | inj2 err | _ = inj2 (tif-err err)
... | _ | inj2 err = inj2 (eif-err err)
ti π (GVar ι) with lookup π ι
... | inside ι∈π = inj1 (gettype ι∈π, Var ι∈π)
... | outside = inj2 (scope-err ι)
ti π (GLam ι t e) with ti ((ι, t) :: π) e
... | inj1 (t′, π′⊢e:t′) = inj1 ((t ⇾ t′), Lam π′⊢e:t′)
... | inj2 err = inj2 (lam-err err)
ti π (GApp e e′) with ti π e
ti π (GApp e e′) | inj1 ((t ⇾ t′), π⊢e:t⇾t′) with tc π e′ t
... | inj1 (π⊢e′:t) = inj1 (t′, App π⊢e:t⇾t′ π⊢e′:t)
... | inj2 err = inj2 (argapp-err err)
ti π (GApp e e′) | inj1 _ = inj2 notfun-err
ti π (GApp e e′) | inj2 err = inj2 (funapp-err err)
```

Código A.3 Inferencia de tipos (continúa)

```
ti π (GPair e e′) with ti π e | ti π e′
... | inj₁ (t, π⊢e:t) | inj₁ (t′, π⊢e′:t′)
                = inj₁ (prod t t′, Pair π⊢e:t π⊢e′:t′)
... | inj₂ err | _ = inj₂ (fpair-err err)
... | _ | inj₂ err = inj₂ (spair-err err)
ti π (GFst p) with ti π p
... | inj₁ (prod t t′, π⊢p:prodtt′) = inj₁ (t, Fst π⊢p:prodtt′)
... | inj₁ _   = inj₂ notpair-err
... | inj₂ err = inj₂ (pair-err err)
ti π (GSnd p) with ti π p
... | inj₁ (prod t t′, π⊢p:prodtt′) = inj₁ (t′, Snd π⊢p:prodtt′)
... | inj₁ _   = inj₂ notpair-err
... | inj₂ err = inj₂ (pair-err err)
ti π (GLetrec ι t e e′) with tc ((ι, t) :: π) e t | ti ((ι, t) :: π) e′
ti π (GLetrec ι t e e′) | inj₁ (π′⊢e:t) | inj₁ (t′, π′⊢e′:t′) with isType₁ t
... | just t₁  = inj₁ (t′, Lrec₁ t₁ π′⊢e:t π′⊢e′:t′)
... | nothing = inj₂ (type₁-err t)
ti π (GLetrec ι t e e′) | inj₂ err | _ = inj₂ (beletrec-err err)
ti π (GLetrec ι t e e′) | _ | inj₂ err = inj₂ (inletrec-err err)
ti π (GEscape ι e) with tc ((ι, compl) :: π) e comm
... | inj₁ (π′⊢e:comm) = inj₁ (comm, Esc π′⊢e:comm)
... | inj₂ err = inj₂ (escape-err err)
ti π (GNewvar ι d c) with tc ((ι, δvar d) :: π) c comm
... | inj₁ (π′⊢c:comm) = inj₁ (comm, Newvar π′⊢c:comm)
... | inj₂ err = inj₂ (newvar-err err)
ti π (GSeq c c′) with tc π c comm
ti π (GSeq c c′) | inj₂ err = inj₂ (fseq-err err)
ti π (GSeq c c′) | inj₁ (π⊢c:comm) with tc π c′ compl
... | inj₁ (π⊢c′:compl) = inj₁ (compl, ComplSeq π⊢c:comm π⊢c′:compl)
... | _ with tc π c′ comm
... | inj₁ (π⊢c′:comm) = inj₁ (comm, CommSeq π⊢c:comm π⊢c′:comm)
... | inj₂ err = inj₂ (sseq-err err)
ti π (GAssign a e) with ti π a
ti π (GAssign a e) | inj₁ (δacc d, π⊢a:dacc) with tc π e (δexp d)
... | inj₁ (π⊢e:dexp) = inj₁ (comm, Assign π⊢a:dacc π⊢e:dexp)
... | inj₂ err = inj₂ (rassign-err err)
ti π (GAssign a e) | inj₁ (δvar d, π⊢a:dvar) with tc π e (δexp d)
... | inj₁ (π⊢e:dexp) = inj₁ (comm, Assign (Sub (π⊢a:dvar) var≼acc) π⊢e:dexp)
... | inj₂ err = inj₂ (rassign-err err)
ti π (GAssign a e) | inj₁ _ = inj₂ notacc-err
ti π (GAssign a e) | inj₂ err = inj₂ (lassign-err err)
ti π (GSkip) = inj₁ (comm, Skip)
ti π (GLoop c) with tc π c comm
... | inj₁ (π⊢c:comm) = inj₁ (compl, Loop π⊢c:comm)
... | inj₂ err = inj₂ (loop-err err)
```

Código A.4 Inferencia de tipos (continúa)

```
ti π (GWhile b c) with tc π b (δexp bool) | tc π c comm
... | inj₁ (π⊢b:bool) | inj₁ (π⊢c:comm) = inj₁ (comm, While π⊢b:bool π⊢c:comm)
... | inj₂ err | _ = inj₂ (bwhile-err err)
... | _ | inj₂ err = inj₂ (cwhile-err err)
ti π (GLet ι p p′) with ti π p
... | inj₂ err = inj₂ (belet-err err)
... | inj₁ (t, π⊢p:t) with ti ((ι, t) :: π) p′
... | inj₂ err = inj₂ (inlet-err err)
... | inj₁ (t′, π′⊢p′:t′) = inj₁ (t′, Let π⊢p:t π′⊢p′:t′)
```

Código A.5 Semántica de tipos

$$
\begin{array}{l}
\mathsf{compl\text{-}m}\ :\ \mathsf{SD} \rightarrow \mathsf{Set}\\
\mathsf{compl\text{-}m}\ \mathsf{s}\ =\ \mathsf{LabelState}\ (\mathsf{l}\ \mathsf{s})\\
\delta\mathsf{compl\text{-}m}\ :\ (\mathsf{d}\ :\ \mathsf{DType}) \rightarrow \mathsf{SD} \rightarrow \mathsf{Set}\\
\delta\mathsf{compl\text{-}m}\ \mathsf{bool}\ \mathsf{s}\ =\ \mathsf{LabelState}\ (\mathsf{compl\text{-}m}\ \mathsf{s} \times \mathsf{compl\text{-}m}\ \mathsf{s})\\
\delta\mathsf{compl\text{-}m}\ \_\ \mathsf{s}\ =\ \mathsf{LabelState}\ ((\mathsf{s}'\ :\ \mathsf{SD}) \rightarrow \mathsf{Rh}\ \mathsf{s}' \rightarrow \mathsf{compl\text{-}m}\ \mathsf{s}')\\
\delta\mathsf{exp\text{-}m}\ :\ (\mathsf{d}\ :\ \mathsf{DType}) \rightarrow \mathsf{SD} \rightarrow \mathsf{Set}\\
\delta\mathsf{exp\text{-}m}\ \mathsf{d}\ \mathsf{s}\ =\ \mathsf{LabelState}\ ((\mathsf{s}'\ :\ \mathsf{SD}) \rightarrow \delta\mathsf{compl\text{-}m}\ \mathsf{d}\ \mathsf{s}' \rightarrow \mathsf{compl\text{-}m}\ \mathsf{s}')\\
\delta\mathsf{acc\text{-}m}\ :\ (\mathsf{d}\ :\ \mathsf{DType}) \rightarrow \mathsf{SD} \rightarrow \mathsf{Set}\\
\delta\mathsf{acc\text{-}m}\ \mathsf{d}\ \mathsf{s}\ =\ \mathsf{LabelState}\ ((\mathsf{s}'\ :\ \mathsf{SD}) \rightarrow \mathsf{compl\text{-}m}\ \mathsf{s}' \rightarrow \delta\mathsf{compl\text{-}m}\ \mathsf{d}\ \mathsf{s}')\\
\mathsf{comm\text{-}m}\ :\ \mathsf{SD} \rightarrow \mathsf{Set}\\
\mathsf{comm\text{-}m}\ \mathsf{s}\ =\ \mathsf{LabelState}\ ((\mathsf{s}'\ :\ \mathsf{SD}) \rightarrow \mathsf{compl\text{-}m}\ \mathsf{s}' \rightarrow \mathsf{compl\text{-}m}\ \mathsf{s}')\\
\delta\mathsf{compl\text{-}m'}\ :\ (\mathsf{d}\ :\ \mathsf{DType}) \rightarrow \mathsf{SD} \rightarrow \mathsf{Set}\\
\delta\mathsf{compl\text{-}m'}\ \mathsf{bool}\ \mathsf{s}\ =\ \mathsf{compl\text{-}m}\ \mathsf{s} \times \mathsf{compl\text{-}m}\ \mathsf{s}\\
\delta\mathsf{compl\text{-}m'}\ \_\ \mathsf{s}\ =\ (\mathsf{s}'\ :\ \mathsf{SD}) \rightarrow \mathsf{Rh}\ \mathsf{s}' \rightarrow \mathsf{compl\text{-}m}\ \mathsf{s}'\\
\tau[\![\_]\!]\ :\ \mathsf{Type} \rightarrow \mathsf{SD} \rightarrow \mathsf{Set}\\
\tau[\![\ \delta\mathsf{exp}\ \mathsf{d}\ ]\!]\ \mathsf{s}\ =\ \delta\mathsf{exp\text{-}m}\ \mathsf{d}\ \mathsf{s}\\
\tau[\![\ \delta\mathsf{acc}\ \mathsf{d}\ ]\!]\ \mathsf{s}\ =\ \delta\mathsf{acc\text{-}m}\ \mathsf{d}\ \mathsf{s}\\
\tau[\![\ \delta\mathsf{var}\ \mathsf{d}\ ]\!]\ \mathsf{s}\ =\ \mathsf{LabelState}\ (\delta\mathsf{acc\text{-}m}\ \mathsf{d}\ \mathsf{s} \times \delta\mathsf{exp\text{-}m}\ \mathsf{d}\ \mathsf{s})\\
\tau[\![\ \mathsf{comm}\ ]\!]\ \mathsf{s}\ =\ \mathsf{comm\text{-}m}\ \mathsf{s}\\
\tau[\![\ \mathsf{compl}\ ]\!]\ \mathsf{s}\ =\ \mathsf{compl\text{-}m}\ \mathsf{s}\\
\tau[\![\ \delta\mathsf{compl}\ \mathsf{d}\ ]\!]\ \mathsf{s}\ =\ \mathsf{LabelState}\ (\delta\mathsf{compl\text{-}m'}\ \mathsf{d}\ \mathsf{s})\\
\tau[\![\ \mathsf{prod}\ \mathsf{t}\ \mathsf{t}'\ ]\!]\ \mathsf{s}\ =\ \mathsf{LabelState}\ (\tau[\![\ \mathsf{t}\ ]\!]\ \mathsf{s} \times \tau[\![\ \mathsf{t}'\ ]\!]\ \mathsf{s})\\
\tau[\![\ \mathsf{t} \rightarrowtriangle \mathsf{t}'\ ]\!]\ \mathsf{s}\ =\ \mathsf{LabelState}\ ((\mathsf{s}'\ :\ \mathsf{SD}) \rightarrow \tau[\![\ \mathsf{t}\ ]\!]\ \mathsf{s}' \rightarrow \tau[\![\ \mathsf{t}'\ ]\!]\ \mathsf{s}')
\end{array}
$$

Código A.6 Semántica de tipos (continúa)

```
i-cumm : (s s′ : SD) → I s → I s′
i-cumm s s′ k with s ≐ s′
i-cumm s .s k | yes refl = k
i-cumm s s′ k | no _ =
  if (fc s′) =n (fc s) then adjust [disp s ⊖ disp s′] - [s] » k else popto s » k
⟦_⟧⟨_,_⟩_ : (t : Type) → (s : SD) → (s′ : SD) → τ⟦ t ⟧ s → τ⟦ t ⟧ s′
⟦ compl ⟧⟨ s, s′ ⟩ κ = κ ≫= λ k → ↑ i-cumm s s′ k
⟦ δexp bool ⟧⟨ s, s′ ⟩ f = f
⟦ δexp int ⟧⟨ s, s′ ⟩ f = f
⟦ δexp real ⟧⟨ s, s′ ⟩ f = f
⟦ δacc bool ⟧⟨ s, s′ ⟩ f = f
⟦ δacc int ⟧⟨ s, s′ ⟩ f = f
⟦ δacc real ⟧⟨ s, s′ ⟩ f = f
⟦ δvar bool ⟧⟨ s, s′ ⟩ f = f
⟦ δvar int ⟧⟨ s, s′ ⟩ f = f
⟦ δvar real ⟧⟨ s, s′ ⟩ f = f
⟦ comm ⟧⟨ s, s′ ⟩ f = f
⟦ δcompl bool ⟧⟨ s, s′ ⟩ m
   = m ≫= λ p →
     proj₁ p ⋙ λ κ → κ ≫= λ k →
     proj₂ p ⋙ λ κ′ → κ′ ≫= λ k′ →
     ↑ (↑ i-cumm s s′ k, ↑ i-cumm s s′ k′)
⟦ δcompl int ⟧⟨ s, s′ ⟩ f = f
⟦ δcompl real ⟧⟨ s, s′ ⟩ f = f
⟦ prod t t′ ⟧⟨ s, s′ ⟩ m
   = m ≫= λ p →
     proj₁ p ⋙ λ f →
     proj₂ p ⋙ λ g →
     ↑ (⟦ t ⟧⟨ s, s′ ⟩ f, ⟦ t′ ⟧⟨ s, s′ ⟩ g)
⟦ t ⇾ t′ ⟧⟨ s, s′ ⟩ f = f
```

Código A.7 Coerciones

```
S⟦_⟧ : ∀ {t t′} → t ≼ t′ → (s : SD) → τ⟦ t ⟧ s → τ⟦ t′ ⟧ s
S⟦_⟧ τrefl s x = x
S⟦_⟧ (δexp-sub δrefl) s x = x
S⟦_⟧ (δexp-sub ir) s m = m ≫= λ e →
  ↑ λ s′ m′ → m′ ≫= λ β →
    e s′ (↑ useTemp s′ (λ s″ r → β s″ (toReal r)))
S⟦_⟧ (δacc-sub δrefl) s x = x
S⟦_⟧ (δacc-sub ir) s m = m ≫= λ a →
  ↑ λ s′ κ →
    a s′ κ ≫= λ β →
    ↑ useTemp s′ (λ s″ r → β s″ (toReal r))
S⟦_⟧ var≼exp s m = m ≫= proj₂
S⟦_⟧ var≼acc s m = m ≫= proj₁
S⟦_⟧ compl≼comm s κ = κ ≫= λ k → ↑ λ s′ κ′ → ⟦ compl ⟧⟨ s, s′ ⟩ κ
S⟦_⟧ (prod-sub p q) s m = m ≫= λ xy →
  ↑ (S⟦ p ⟧ s (proj₁ xy), S⟦ q ⟧ s (proj₂ xy))
S⟦_⟧ (⇾-sub p q) s m = m ≫= λ f → ↑ λ s′ a → S⟦ q ⟧ s′ (f s′ (S⟦ p ⟧ s′ a))
S⟦_⟧ (trans p q) s x = (S⟦ q ⟧ s ∘ S⟦ p ⟧ s) x
```

Código A.8 Funciones sobre mónada de estado

```
private
  newLabel : {s : SD} → StateTy → I s → StateTy
  newLabel {s} (ℓ, ins) c = suc ℓ, ((ℓ, s, c) :: ins)
  reserveLabel' : StateTy → LabelState StateTy
  reserveLabel' (ℓ, ins) = put (suc ℓ, ins) ≫ return (ℓ, ins)
reserveLabel : LabelState StateTy
reserveLabel = get ≫= reserveLabel'
assignLabel : {s : SD} → Label → I s → LabelState ⊤
assignLabel {s} ℓ k = get ≫= λ st → put (proj₁ st, (ℓ, s, k) :: proj₂ st)
withNewLabel : {s : SD} → I s → LabelState StateTy
withNewLabel is = get ≫= λ st → put (newLabel st is) ≫ return st
```

Código A.9 Subrutinas

```
saveres : (s : SD) → (β : δcompl-m int s) → compl-m s
saveres s m = m ≫= λ β →
  β s′ (RhLh (S s)) ≫= λ βs′s →
  ↑ [disp s′ ⊖ disp s] - [s′] S s := RhLh Sbrs » βs′s
  where s′ = s ∔ 1
Acumm = List $ ∃₂ (λ s t → τ⟦ t ⟧ s × Simple t)
fromAcumm : Acumm → (s : SD) → LabelState (ArgList s)
fromAcumm [] sⁿ = return []
fromAcumm ((s, t, a, φ) :: ps) sⁿ
   = gmk-subr 1 φ sⁿ (⟦ t ⟧⟨ s, sⁿ ⟩ a) ≫= λ k →
     fromAcumm ps sⁿ ≫= λ l →
     return ((t, φ, k) :: l)
gmk-argcall : {t : Type} → Acumm → (φ : Simple t) → (s : SD) → ℕ
  → τ⟦ t ⟧ s
gmk-argcall l compl s j = ↑ ajump j
gmk-argcall l intcompl s j
                            = ↑ λ s′ r →
                              ↑ [disp s ⊖ disp s′] - [s] Sbrs := r » ajump j
gmk-argcall l realcompl s j
                            = ↑ λ s′ r →
                              ↑ [disp s ⊖ disp s′] - [s] Sbrs := r » ajump j
gmk-argcall {t ⇀ .compl} l (φ ⇀ compl) s j
   = ↑ λ sⁿ a →
     fromAcumm (l ++ [(sⁿ, t, a, φ)]) sⁿ ≫= λ l →
     ↑ acall j (fc s) l
gmk-argcall {t ⇀ . (δcompl int)} l (φ ⇀ intcompl) s j
   = ↑ λ sⁿ a → ↑ λ s′ r →
     fromAcumm (l ++ [(sⁿ, t, a, φ)]) sⁿ ≫= λ l →
     ↑ [disp sⁿ ⊖ disp s′] - [sⁿ] Sbrs := r »
     acall j (fc s) l
gmk-argcall {t ⇀ . (δcompl real)} l (φ ⇀ realcompl) s j
   = ↑ λ sⁿ a → ↑ λ s′ r →
     fromAcumm (l ++ [(sⁿ, t, a, φ)]) sⁿ ≫= λ l →
     ↑ [disp sⁿ ⊖ disp s′] - [sⁿ] Sbrs := r »
     acall j (fc s) l
gmk-argcall {t ⇀ (t′ ⇀ t″)} l (φ ⇀ (φ′ ⇀ φ″)) s i
   = ↑ λ s′ a →
     gmk-argcall (l ++ [(s′, t, a, φ)]) (φ′ ⇀ φ″) s i
```

Código A.10 Subrutinas (continúa)

```
gmk-call : {t : Type} → Acumm → (φ : Simple t) → (s : SD) → srφ s φ
  → τ⟦ t ⟧ s
gmk-call l compl s i = i
gmk-call l intcompl s i = ↑ λ s' r → i ≫= λ k →
                          ↑ [disp s ⊖ disp s'] - [s] Sbrs := r » k
gmk-call l realcompl s i = ↑ λ s' r → i ≫= λ k →
                          ↑ [disp s ⊖ disp s'] - [s] Sbrs := r » k
gmk-call {t ⇾ .compl} l (φ ⇾ compl) s i
  = ↑ λ s^n a → i ≫= λ k →
    fromAcumm (l ++ [(s^n, t, a, φ)]) s^n ≫= λ l →
    ↑ call (fc s) k l
gmk-call {t ⇾ . (δcompl int)} l (φ ⇾ intcompl) s i
  = ↑ λ s^n a → ↑ λ s' r → i ≫= λ k →
    fromAcumm (l ++ [(s^n, t, a, φ)]) s^n ≫= λ l →
    ↑ [disp s^n ⊖ disp s'] - [s^n] Sbrs := r »
    call (fc s) k l
gmk-call {t ⇾ . (δcompl real)} l (φ ⇾ realcompl) s i
  = ↑ λ s^n a → ↑ λ s' r → i ≫= λ k →
    fromAcumm (l ++ [(s^n, t, a, φ)]) s^n ≫= λ l →
    ↑ [disp s^n ⊖ disp s'] - [s^n] Sbrs := r »
    call (fc s) k l
gmk-call {t ⇾ (t' ⇾ t'')} l (φ ⇾ (φ' ⇾ φ'')) s i
  = ↑ λ s' a →
    gmk-call (l ++ [(s', t, a, φ)]) (φ' ⇾ φ'') s i

mk-subr : {t : Type} → (φ : Simple t) → (s : SD) → τ⟦ t ⟧ s → srφ s φ
mk-subr φ = gmk-subr 1 φ

mk-call : {t : Type} → (φ : Simple t) → (s : SD) → srφ s φ → τ⟦ t ⟧ s
mk-call φ = gmk-call [] φ

mk-argcall : {t : Type} → Simple t → (s : SD) → ℕ → τ⟦ t ⟧ s
mk-argcall φ = gmk-argcall [] φ

mk-call_1 : {t : Type} → (t_1 : Type_1 t) → (s : SD) → srφ s (issimple_1 t_1)
  → τ⟦ t ⟧ s
mk-call_1 t_1 s k = fromsimple-m s t_1 (mk-call (issimple_1 t_1) s k)

mk-subr_1 : {t : Type} → (t_1 : Type_1 t) → (s : SD) → τ⟦ t ⟧ s
  → srφ s (issimple_1 t_1)
mk-subr_1 t_1 s x = mk-subr (issimple_1 t_1) s (tosimple-m s t_1 x)
```

Código A.11 Traducción

```
eval : ∀ {π e t} → (π ⊢ e : t) → (s : SD) → ⟦ π ⟧* s → τ⟦ t ⟧ s
eval {π} {GICon x} {δexp int} ICon s η
  = ↑ λ s′ m → m ≫= λ β → β s′ (Sh↑ $ litδ int x)
eval {π} {GBCon true} {δexp bool} BCon s η
  = ↑ λ s′ m → m ≫= λ kk′ → proj₁ kk′
eval {π} {GBCon false} {δexp bool} BCon s η
  = ↑ λ s′ m → m ≫= λ kk′ → proj₂ kk′
eval {π} {GRCon x} {δexp real} RCon s η
  = ↑ λ s′ m → m ≫= λ β → β s′ (Sh↑ $ litδ real x)
eval (Op1I {π} {e} iop j) s η
  = eval j s η ≫= λ f →
    ↑ λ s′ m → m ≫= λ β →
    f s′ (↑ useTemp s′ (λ s″ r → β s″ (I₁ iop r)))
eval (Op1R {π} {e} rop j) s η
  = eval j s η ≫= λ f →
    ↑ λ s′ m → m ≫= λ β →
    f s′ (↑ useTemp s′ (λ s″ r → β s″ (R₁ rop r)))
eval (Op2I {π} {e} {e′} iop j₁ j₂) s η
  = eval j₁ s η ≫= λ f →
    eval j₂ s η   ≫= λ g →
    ↑ λ s′ m → m ≫= λ β → f s′ (↑ useTemp s′ (λ s″ r₁ →
    g s″ (↑ useTemp s″ (λ s‴ r₂ → β s‴ (I₂ iop (sh-cumm r₁) r₂)))))
eval (Op2R {π} {e} {e′} rop j₁ j₂) s η
  = eval j₁ s η ≫= λ f →
    eval j₂ s η   ≫= λ g →
    ↑ λ s′ m → m ≫= λ β → f s′ (↑ useTemp s′ (λ s″ r₁ →
    g s″ (↑ useTemp s″ (λ s‴ r₂ → β s‴ (R₂ rop (sh-cumm r₁) r₂)))))
eval (RelBI {π} {e} {e′} {rel} j₁ j₂) s η
  = eval j₁ s η ≫= λ f →
    eval j₂ s η   ≫= λ g →
    ↑ λ s′ m → m ≫= λ kk′ → proj₁ kk′ ≫= λ k → proj₂ kk′ ≫= λ k′ →
    f s′ (↑ useTemp s′ (λ s″ r₁ →
    g s″ (↑ useTemp s″ (λ s‴ r₂ →
    ↑ (if rel : (sh-cumm r₁), r₂ [disp s′ ⊖ disp s‴] ⟨ s′ ⟩then k else k′)))))
eval (RelBR {π} {e} {e′} {rel} j₁ j₂) s η
  = eval j₁ s η ≫= λ f →
    eval j₂ s η   ≫= λ g →
    ↑ λ s′ m → m ≫= λ kk′ → proj₁ kk′ ≫= λ k → proj₂ kk′ ≫= λ k′ →
    f s′ (↑ useTemp s′ (λ s″ r₁ →
    g s″ (↑ useTemp s″ (λ s‴ r₂ →
    ↑ (if rel : (sh-cumm r₁), r₂ [disp s′ ⊖ disp s‴] ⟨ s′ ⟩then k else k′)))))
```

Código A.12 Traducción (continúa)

```
eval (Op2B {π} {e} {e′} or j j′) s η
   = eval j s η      ≫= λ f →
     eval j′ s η     ≫= λ g →
     ↑ λ s′ m → m  ≫= λ kk′ → proj₁ kk′  ≫= λ k → proj₂ kk′  ≫= λ k′ →
     withNewLabel k  ≫= λ st →
     jmp (proj₁ st) ⋙ λ kjmp →
     f s′ (↑ (↑ kjmp, g s′ (↑ (↑ kjmp, ↑ k′))))
eval (Op2B {π} {e} {e′} and j j′) s η
   = eval j s η        ≫= λ f →
     eval j′ s η       ≫= λ g →
     ↑ λ s′ m → m      ≫= λ kk′ → proj₁ kk′  ≫= λ k → proj₂ kk′  ≫= λ k′ →
     withNewLabel k′  ≫= λ st →
     jmp (proj₁ st)    ⋙ λ k′jmp →
     f s′ (↑ (g s′ (↑ (↑ k, ↑ k′jmp)), ↑ k′jmp))
eval (Op2B {π} {e} {e′} imply j j′) s η
   = eval j s η        ≫= λ f →
     eval j′ s η       ≫= λ g →
     ↑ λ s′ m → m      ≫= λ kk′ → proj₁ kk′  ≫= λ k → proj₂ kk′  ≫= λ k′ →
     withNewLabel k  ≫= λ st →
     jmp (proj₁ st) ⋙ λ kjmp →
     f s′ (↑ (g s′ (↑ (↑ kjmp, ↑ k′)), ↑ kjmp))
eval (Op2B {π} {e} {e′} iff j j′) s η
   = eval j s η  ≫= λ f →
     eval j′ s η  ≫= λ g →
     ↑ λ s′ m → m  ≫= λ kk′ → proj₁ kk′  ≫= λ k → proj₂ kk′  ≫= λ k′ →
     withNewLabel k  ≫= λ st →
     withNewLabel k′  ≫= λ st' →
     jmp (proj₁ st) ⋙ λ kjmp →
     jmp (proj₁ st') ⋙ λ k′jmp →
     f s′ (↑ (g s′ (↑ (↑ kjmp, ↑ k′jmp)),
        g s′ (↑ (↑ k′jmp, ↑ kjmp))))
eval (Op1B {π} {e} neg j) s η
   = eval j s η  ≫= λ f →
     ↑ λ s′ m → m  ≫= λ kk′ → f s′ (↑ swap kk′)
eval (CommIf j j′ j″) s η
   = eval j s η  ≫= λ b →
     eval j′ s η  ≫= λ c₁ →
     eval j″ s η  ≫= λ c₂ →
     ↑ λ s′ κ → κ  ≫= λ k → withNewLabel k  ≫= λ st →
     jmp (proj₁ st) ⋙ λ kjmp →
     b s′ (↑ (c₁ s′ (↑ kjmp), c₂ s′ (↑ kjmp)))
eval (ComplIf j j′ j″) s η
   = eval j s η  ≫= λ b →
     b s (↑ (eval j′ s η, eval j″ s η))
```

Código A.13 Traducción (continúa)

$$
\begin{array}{l}
\mathsf{eval}\ (\mathsf{Var}\ \mathsf{p})\ \mathsf{s}\ \eta\ =\ \eta\ !\ \mathsf{p} \\
\mathsf{eval}\ (\mathsf{Lam}\ \{\pi\}\ \{\iota\}\ \{\mathsf{e}\}\ \{\mathsf{t}\}\ \{\mathsf{t}'\}\ \mathsf{j})\ \mathsf{s}\ \eta \\
\quad =\ \uparrow \lambda\ \mathsf{s}'\ \mathsf{a} \rightarrow \mathsf{eval}\ \mathsf{j}\ \mathsf{s}'\ (\iota \mapsto \mathsf{a} :: ([\![\ \pi\ ]\!]^{*}\langle\ \mathsf{s}, \mathsf{s}'\ \rangle\ \eta)) \\
\mathsf{eval}\ (\mathsf{App}\ \mathsf{j}\ \mathsf{j}')\ \mathsf{s}\ \eta \\
\quad =\ \mathsf{eval}\ \mathsf{j}\ \mathsf{s}\ \eta\ \gg\!=\ \lambda\ \mathsf{f} \rightarrow \mathsf{f}\ \mathsf{s}\ (\mathsf{eval}\ \mathsf{j}'\ \mathsf{s}\ \eta) \\
\mathsf{eval}\ (\mathsf{Pair}\ \mathsf{j}\ \mathsf{j}')\ \mathsf{s}\ \eta\ =\ \uparrow ((\mathsf{eval}\ \mathsf{j}\ \mathsf{s}\ \eta), (\mathsf{eval}\ \mathsf{j}'\ \mathsf{s}\ \eta)) \\
\mathsf{eval}\ (\mathsf{Fst}\ \mathsf{j})\ \mathsf{s}\ \eta\ =\ \mathsf{eval}\ \mathsf{j}\ \mathsf{s}\ \eta\ \gg\!=\ \mathsf{proj}_1 \\
\mathsf{eval}\ (\mathsf{Snd}\ \mathsf{j})\ \mathsf{s}\ \eta\ =\ \mathsf{eval}\ \mathsf{j}\ \mathsf{s}\ \eta\ \gg\!=\ \mathsf{proj}_2 \\
\mathsf{eval}\ (\mathsf{Esc}\ \{\pi\}\ \{\iota\}\ \{\mathsf{e}\}\ \mathsf{j})\ \mathsf{s}\ \eta \\
\quad =\ \uparrow \lambda\ \mathsf{s}'\ \kappa \rightarrow \kappa\ \gg\!=\ \lambda\ \mathsf{k} \rightarrow \\
\qquad \mathsf{withNewLabel}\ \mathsf{k}\ \gg\!=\ \lambda\ \mathsf{st} \rightarrow \\
\qquad \mathsf{jmp}\ (\mathsf{proj}_1\ \mathsf{st}) \ggg \lambda\ \mathsf{kjmp} \rightarrow \\
\qquad \mathsf{eval}\ \mathsf{j}\ \mathsf{s}'\ (\iota \mapsto (\uparrow \mathsf{kjmp}) :: [\![\ \pi\ ]\!]^{*}\langle\ \mathsf{s}, \mathsf{s}'\ \rangle\ \eta)\ \gg\!=\ \lambda\ \mathsf{f} \rightarrow \\
\qquad \mathsf{f}\ \mathsf{s}'\ (\uparrow \mathsf{kjmp}) \\
\mathsf{eval}\ (\mathsf{Lrec}_1\ \{\pi\}\ \{\iota\}\ \{\mathsf{e}\}\ \{\mathsf{e}'\}\ \{\_\}\ \{\_\}\ \mathsf{t}_1\ \mathsf{p}\ \mathsf{p}')\ \mathsf{s}\ \eta \\
\quad =\ \mathsf{reserveLabel}\ \gg\!=\ \lambda\ \mathsf{st} \rightarrow \\
\qquad \mathsf{mk\text{-}call}_1\ \mathsf{t}_1\ \mathsf{s}\ (\uparrow \mathsf{jmp}\ (\mathsf{proj}_1\ \mathsf{st})) \ggg \lambda\ \mathsf{scall} \rightarrow \\
\qquad (\iota \mapsto \mathsf{scall} :: \eta) \ggg \lambda\ \eta' \rightarrow \\
\qquad \mathsf{eval}\ \mathsf{p}\ \mathsf{s}\ \eta' \ggg \lambda\ \mathsf{k} \rightarrow \\
\qquad \mathsf{mk\text{-}subr}_1\ \mathsf{t}_1\ \mathsf{s}\ \mathsf{k}\ \gg\!= \\
\qquad \mathsf{assignLabel}\ (\mathsf{proj}_1\ \mathsf{st})\ \gg \\
\qquad \mathsf{eval}\ \mathsf{p}'\ \mathsf{s}\ \eta'
\end{array}
$$

Código A.14 Traducción (continúa)

```
eval {π} {GNewvar ι int c} {.comm} (Newvar j) s η
   = ↑ λ s′ κ → (λ s‴ m → m ≫= λ β → β s‴ (RhLh $ S s′)) ⋙ λ e →
     (λ s‴ κ′ → κ′ ≫= λ k′ →
       ↑ λ s^4 r → ↑ [disp s‴ ⊖ disp s^4] - [s‴] S s′ := r » k′) ⋙ λ a →
     s′ ∔ 1 ⋙ λ s″ →
     ι ↦ ↑ (↑ a, ↑ e) :: ⟦ π ⟧*⟨ s, s″ ⟩ η ⋙ λ η′ →
     κ ≫= λ k →
     eval j s″ η′ ≫= λ f →
     f s″ (↑ adjust [- (+ 1)] - [s′] » k) ≫= λ k_1 →
     ↑ [(+ 1)] - [s″] S s′ := Sh↑ $ litδ int #0 » k_1
eval {π} {GNewvar ι bool c} {.comm} (Newvar j) s η
     = ↑ λ s′ κ →
       (λ s‴ m → m ≫= λ kk′ → proj_1 kk′ ≫= λ k →
         proj_2 kk′ ≫= λ k′ →
         ↑ if S s′ then» k else» k′) ⋙ λ e →
       (λ s‴ κ′ → κ′ ≫= λ k′ →
         withNewLabel k′ ≫= λ st →
         jmp (proj_1 st) ⋙ λ k′jmp →
         ↑ (↑ [+ 0] - [s‴] S s′ := litTrue » k′jmp,
            ↑ [+ 0] - [s‴] S s′ := litFalse » k′jmp)
         ) ⋙ λ a →
       s′ ∔ 1 ⋙ λ s″ →
       ι ↦ ↑ (↑ a, ↑ e) :: ⟦ π ⟧*⟨ s, s″ ⟩ η ⋙ λ η′ →
       κ ≫= λ k →
       eval j s″ η′ ≫= λ f →
       f s″ (↑ adjust [- (+ 1)] - [s′] » k) ≫= λ k_1 →
       ↑ [+ 1] - [s″] S s′ := litFalse » k_1
eval {π} {GNewvar ι real c} {.comm} (Newvar j) s η
     = ↑ λ s′ κ → (λ s‴ m → m ≫= λ β → β s‴ (RhLh $ S s′)) ⋙ λ e →
       (λ s‴ κ′ → κ′ ≫= λ k′ →
       ↑ λ s^4 r → ↑ [disp s‴ ⊖ disp s^4] - [s‴] S s′ := r » k′) ⋙ λ a →
       s′ ∔ 1 ⋙ λ s″ →
       ι ↦ ↑ (↑ a, ↑ e) :: ⟦ π ⟧*⟨ s, s″ ⟩ η ⋙ λ η′ →
       κ ≫= λ k →
       eval j s″ η′ ≫= λ f →
       f s″ (↑ adjust [- (+ 1)] - [s′] » k) ≫= λ k_1 →
       ↑ [(+ 1)] - [s″] S s′ := Sh↑ $ litδ real (int-to-real #0) » k_1
```

Código A.15 Traducción (continúa)

```
eval (ComplSeq j j′) s η
   = eval j s η ≫= λ f →
     f s (eval j′ s η)
eval (CommSeq j j′) s η
   = eval j s η ≫= λ f →
     eval j′ s η ≫= λ g →
     ↑ λ s′ κ → f s′ (g s′ κ)
eval (Assign {π} {a} {e} {d} j j′) s η
   = eval j s η ≫= λ f →
     eval j′ s η ≫= λ g →
     ↑ λ s′ κ → g s′ (f s′ κ)
eval Skip s η = ↑ λ s′ κ → κ
eval (Loop j) s η = reserveLabel ≫= λ st →
  eval j s η ≫= λ c →
  proj₁ st ⋙ λ ℓ →
  c s (↑ jmp ℓ) ≫=
  assignLabel ℓ ≫
  ↑ jmp ℓ
eval {π} (While j j′) s η = ↑ λ s′ κ → κ ≫= λ k →
  reserveLabel ≫= λ st →
  ⟦ π ⟧*⟨ s, s′ ⟩ η ⋙ λ η′ →
  eval j s′ η′ ≫= λ b →
  eval j′ s′ η′ ≫= λ c →
  proj₁ st ⋙ λ ℓ →
  b s′ (↑ (c s′ (↑ jmp ℓ), ↑ k)) ≫=
  assignLabel ℓ ≫
  ↑ jmp ℓ
eval (Let {π} {ι} j j′) s η = eval j′ s (ι ↦ eval j s η :: η)
eval (Sub j p) s η = S⟦ p ⟧ s (eval j s η)
```

Apéndice B

Ejemplos de traducción

En este apéndice presentamos algunos ejemplos de traducción que llevamos a cabo con nuestro front-end. Elegimos ejemplos que ilustran algunas de las características de la traducción, como es la reducción inline de procedimientos, las coerciones y la generación de subrutinas.

Traducción B.1 Procedimientos y pares reducidos a código inline

Programa fuente

$$\begin{array}{l} \mathbf{newvar}\ x\ \mathbf{int\ in} \\ \quad \mathbf{let}\ p\ \mathbf{be}\ (\lambda j\ : \mathbf{intexp}.\ j+1\,,\ x := x+2)\ \mathbf{in} \\ \quad\quad x := (\mathbf{fst}\ p)\ x; \\ \quad\quad \mathbf{snd}\ p \end{array}$$

Código intermedio

```
(0 , 0) := lit 0 [1] ;
(0 , 0) := (0 , 0) + lit 1 [0] ;
(0 , 0) := (0 , 0) + lit 2 [0] ;
adjust [-1] ;
stop
```

Traducción B.2 Coerciones

Programa fuente

newvar x **real in**
  let f **be** λx : **realexp**. $x \,/\, 3.5$ **in**
    let g **be** λh : **intexp** $\rightarrow$ **realexp**. h 9 **in**
      $x := (g\, f) + 1$

Código intermedio

```
(0 , 0) := lit 0.0 [1] ;
(0 , 1) := toReal lit 9 [1] ;
(0 , 1) := (0 , 1) / lit 3.5 [0] ;
(0 , 2) := toReal lit 1 [1] ;
(0 , 0) := (0 , 1) + (0 , 2) [-2] ;
adjust [-1] ;
stop
```

Traducción B.3 Subrutina para una constante

Programa fuente

letrec c **be** 7 **in**
  newvar x **int in** $x := c$

Código intermedio

```
(0 , 0) := lit 0 [1] ;
call 0 (jmp 0)
( (0 , 1) := sbrs [1] ;
  (0 , 0) := (0 , 1) [-1] ;
  adjust [-1] ;
  stop
)
0 -> sbrs := lit 7 [0] ;
     ajump 1
```

Traducción B.4 Ciclos

Programa fuente

$$
\begin{array}{l}
\textbf{newvar}\ x\ \textbf{int in} \\
\quad x := 1; \\
\quad \textbf{newvar}\ i\ \textbf{real in} \\
\quad\quad i := 2; \\
\quad\quad \textbf{while}\ (i \leq 10)\ \textbf{do} \\
\quad\quad\quad x := x * i;\ i := i + 1
\end{array}
$$

Código intermedio

```
(0 , 0) := lit 0 [1] ;
(0 , 0) := lit 1 [0] ;
(0 , 1) := lit 0 [1] ;
(0 , 1) := lit 2 [0] ;
jmp 0

0 ->
if (0 , 1) <= lit 10 [0] then
 (0 , 0) := (0 , 0) * (0 , 1) [0] ;
 (0 , 1) := (0 , 1) + lit 1 [0] ;
 jmp 0
else
 adjust [-1] ;
 adjust [-1] ;
 stop
```

Traducción B.5 Subrutina para un procedimiento recursivo

Programa fuente

$$\begin{array}{l}\textbf{letrec}\ g\ :\ \textbf{boolexp}\ \rightarrow\ \textbf{comm} \rightarrow \textbf{comm be} \\ \qquad (\lambda b : \textbf{boolexp}.\ \lambda c : \textbf{comm}.\ \textbf{if}\ b\ \textbf{then}\ c\ ;\ g\,b\,c\ \textbf{else skip}\ ) \\ \textbf{in newvar}\ x\ \textbf{int in}\ g\ (x \leq 2)\ (x := x + 1)\end{array}$$

Código intermedio

```
(0 , 0) := lit 0 [1] ;
call 0 (jmp 0)
(
 if (0 , 0) <= lit 2 [0] then
  ajump 1
 else
  ajump 2
 ,
 (0 , 0) := (0 , 0) + lit 1 [0] ;
 ajump 1
 ,
 adjust [-1] ;
 stop
)
0 -> acall 1 1
     (
       acall 2 1
       (
        call 0 (jmp 0)
        (
         acall 1 1 (ajump 1 , ajump 2 )
         ,
         acall 2 1 (ajump 1)
         ,
         jmp 1
        )
       )
       ,
       jmp 1
     )
1 -> ajump 3
```

Traducción B.6 Instrucción **popto**

Programa fuente

$$
\begin{array}{l}
\textbf{newvar}\ x\ \textbf{int in} \\
\quad \textbf{escape}\ \mathit{break}\ \textbf{in} \\
\quad\quad (\textbf{letrec}\ c\ :\ \textbf{comm be} \\
\quad\quad\quad \textbf{newvar}\ y\ \textbf{int in} \\
\quad\quad\quad\quad y := 3; \\
\quad\quad\quad\quad x := y; \\
\quad\quad\quad\quad \mathit{break}; \\
\quad\quad\quad\quad c \\
\quad\quad \textbf{in}\ c\ ); \\
\quad x := x + 1
\end{array}
$$

Código intermedio

```
(0 , 0) := lit 0 [1] ;
call 0 (jmp 1) (jmp 0)

1 -> (1 , 2) := lit 0 [1] ;
     (1 , 2) := lit 3 [0] ;
     (0 , 0) := (1 , 2) [0] ;
     popto (0 , 1) ;
     jmp 0

0 ->  (0 , 0) := (0 , 0) + lit 1 [0] ;
      adjust [-1] ;
      stop
```

Bibliografía

[1] Aho, A. V., Sethi, R., Ullman, J. D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986)

[2] Appel, A. W.: Modern Compiler Implementation in ML: Basic Techniques. Cambridge University Press, New York, NY, USA (1997)

[3] Awodey, S.: Category Theory. Oxford University Press, Oxford (2006)

[4] Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., van Wijngaarden, A., Woodger, M.: Report on the Algorithmic Language Algol 60. Commun. ACM **3** (1960) 299–314

[5] Bauer, H. R., Becker, S. I., Graham, S. L.: Algol W. Technical report, Stanford University, Stanford, CA, USA (1968)

[6] Bove, A., Dybjer, P.: Dependent Types at Work. In: Bove, A., Barbosa, L. S., Pardo, A., Pinto, J. S., eds., LerNet ALFA Summer School, volume 5520 of Lecture Notes in Computer Science. Springer (2008), 57–99

[7] Bove, A., Dybjer, P., Norell, U.: A Brief Overview of Agda - A Functional Language with Dependent Types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M., eds., TPHOLs, volume 5674 of Lecture Notes in Computer Science. Springer (2009), 73–78

[8] Church, A.: A Formulation of the Simple Theory of Types. In: The Journal of Symbolic Logic, Vol. 5, No. 2. (1940), 56–68

[9] Cui, S., Donnelly, K., Xi, H.: ATS: A Language that Combines Programming with Theorem Proving. In: Gramlich, B., ed., FroCos, volume 3717 of Lecture Notes in Computer Science. Springer (2005), 310–320

[10] Dahl, O.-J., Nygaard, K.: SIMULA: an Algol-based Simulation Language. Commun. ACM **9** (1966) 671–678

[11] Damas, L., Milner, R.: Principal Type-schemes for Functional Programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '82, ACM, New York, NY, USA (1982), 207–212

[12] Fridlender, D., Indrika, M.: Do We Need Dependent Types? Journal of Functional Programming **10** (2001) 409–415

[13] Hindley, R.: The Principal Type-Scheme of an Object in Combinatory Logic. Transactions of the American Mathematical Society **146** (1969) 29–60

[14] Jensen, K., Wirth, N.: PASCAL User Manual and Report. Springer-Verlag New York, Inc., New York, NY, USA (1974)

[15] Lattner, C., Adve, V. S.: LLVM: a Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO. IEEE Computer Society (2004), 75–88

[16] Leijen, D.: Parsec : A Fast Combinator Parser. University of Utrecht Dept. of Computer Science. `http://legacy.cs.uu.nl/daan/parsec.html`

[17] McBride, C.: Faking It: Simulating Dependent Types in Haskell. J. Funct. Program. **12** (2002) 375–392

[18] Mitchell, J. C.: Concepts in Programming Languages. Cambridge University Press (2003)

[19] Morris, F. L.: Correctness of Translations of Programming Languages – an Algebraic Approach. Ph.D. thesis, Stanford University, Stanford, CA, USA (1972)

[20] Norell, U.: Dependently Typed Programming in Agda. In: Koopman, P. W. M., Plasmeijer, R., Swierstra, S. D., eds., Advanced Functional Programming, volume 5832 of Lecture Notes in Computer Science. Springer (2008), 230–266

[21] Oles, F. J.: A Category-theoretic Approach to the Semantics of Programming Languages. Ph.D. thesis, Syracuse University, Syracuse, NY, USA (1982)

[22] Oles, F. J.: Algebraic methods in semantics, chapter Type algebras, functor categories and block structure. Cambridge University Press, New York, NY, USA (1986), 543–573

[23] Reynolds, J. C.: Conjunctive types and algol-like languages. In: Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science (LICS 1987). IEEE Computer Society Press (1987), 119–119. Invited Talk

[24] Reynolds, J. C.: The Coherence of Languages with Intersection Types. In: Proceedings of the International Conference on Theoretical Aspects of Computer Software. TACS '91, Springer-Verlag, London, UK (1991), 675–700

[25] Reynolds, J. C.: Using Functor Categories to Generate Intermediate Code. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '95, ACM, New York, NY, USA (1995), 25–36

[26] Reynolds, J. C.: ALGOL-like Languages, Volume 1, chapter The Essence of ALGOL. Birkhauser Boston Inc., Cambridge, MA, USA (1997), 67–88

[27] Reynolds, J. C.: ALGOL-like Languages, Volume 1, chapter Design of the Programming Language FORSYTHE. Birkhauser Boston Inc., Cambridge, MA, USA (1997), 173–233

[28] Reynolds, J. C.: Theories of Programming Languages. Cambridge University Press, New York, NY, USA, 1st edition (1998)

[29] Scott, D.: The Lattice of Flow Diagrams. In: Engeler, E., ed., Symposium on Semantics of Algorithmic Languages, volume 188 of Lecture Notes in Mathematics. Springer Berlin / Heidelberg (1971), 311–366

[30] Sheard, T., Hook, J., Linger, N.: GADTs + Extensible Kind System = Dependent Programming. Technical report, Portland State University (2005). `http://www.cs.pdx.edu/~sheard`

[31] van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, L. G. L. T., Fisker, R. G.: Revised Report on the Algorithmic Language Algol 68. Acta Inf. **5** (1975) 1–236

[32] Xi, H.: Dependent ML an Approach to Practical Programming with Dependent Types. J. Funct. Program. **17** (2007) 215–286