Técnicas de análisis de programas en lenguajes funcionales

Clara María Segura Díaz

Departamento de Sistemas Informáticos y Programación
Facultad de Ciencias Matemáticas
Universidad Complutense de Madrid
24 de junio de 1999

Trabajo de Tercer Ciclo

Director del trabajo: Ricardo Peña Marí

Nº de créditos solicitados: 5

Índice General

Agradecimientos: Agradezco a Olga Marroquín y Alberto Verdejo su información sobre paquetes LATEXque me han permitido hacer este trabajo más presentable, a Luis Antonio Galán por su paciencia con mis problemas software, a Cristóbal Pareja por introducirme en el mundo del análisis y de las transformaciones de programas. Y por supuesto a mi director de trabajo, Ricardo Peña, por toda su colaboración e intensa dedicación.

Capítulo 1

Introducción

1.1 Transformación de programas funcionales

La programación funcional es un paradigma que posee una serie de ventajas [Hug90] de las que otro tipo de paradigmas, como el imperativo, carece. La **ausencia de efectos laterales** en los lenguajes funcionales permite razonar con mayor facilidad sobre ellos. Además, la presencia de **pereza** permite el uso de estructuras de datos infinitas. El **orden superior** y el **polimorfismo** proporcionan mayor modularidad y un nivel de abstracción elevado de programación.

Sin embargo, el alejamiento entre los conceptos de la programación funcional de orden superior y la arquitectura concreta, esencialmente imperativa, de las máquinas reales sobre las que se ejecutan los programas, provoca que la implementación de dichos lenguajes sea ineficiente. Una forma de mejorar la eficiencia en estos lenguajes es la **transformación de programas**.

La transformación de programas pretende la reducción de su tiempo de ejecución o de su consumo de espacio de almacenamiento.

Muchas transformaciones requieren un conocimiento previo de información sobre los programas a transformar. Esta información puede obtenerse mediante un **análisis estático** de los programas, es decir, un análisis en tiempo de compilación.

El análisis y la transformación de programas consisten en el procesamiento automático del texto de un programa con el objetivo de obtener, o bien información sobre dicho programa (análisis), o bien un nuevo programa (transformación) con nuevas características, generalmente una mayor eficiencia.

Normalmente la información obtenida mediante el análisis de un programa se emplea en la consiguiente transformación del programa. Otras veces se usa para soportar otros análisis que a su vez se emplean en transformaciones. Por ejemplo, el análisis de clausuras (ver Sección 1.2 para más detalles) permite la extensión de análisis de primer orden a análisis de orden superior.

Las transformaciones se pueden llevar a cabo **a nivel de código fuente** o bien a nivel de **generación de código objeto**.

En el primer caso [San95], determinadas construcciones sintácticas se convierten en otras dentro del mismo lenguaje fuente bajo ciertas condiciones que dependen de la información producida por un análisis estático del programa. El análisis de programas en el segundo caso pretende identificar situaciones de usos restringidos de determinadas construcciones que permitan una implementación menos general pero más eficiente de las mismas. Un ejemplo de ello es el análisis de globalización en lenguajes funcionales estrictos (ver Sección 1.2 para más detalles), que encuentra parámetros de las funciones que pueden ser alojados globalmente en lugar de en la pila (éste es el tratamiento general que se da a los parámetros de una función). En los lenguajes funcionales existen algunas transformaciones que pretenden reducir el tiempo de ejecución mediante la disminución del número de acciones requeridas para manejar la compartición en los lenguajes perezosos, como por ejemplo la tranformación del paso de parámetros por necesidad al paso por nombre (ver Sección 1.2 para más detalles).

Otras pretenden reducir el consumo de espacio de almacenamiento, como por ejemplo la globalización de parámetros.

En los lenguajes funcionales se pueden distinguir dos tipos de consumo de espacio de almacenamiento: el consumo total (cantidad de acciones de reserva de espacio durante el cómputo), y la residencia (número de celdas vivas en un determinado momento). Un consumo total alto de celdas significa que la recolección de basura será frecuente puesto que las celdas libres serán usadas rápidamente, pero cada recolección será rápida si la residencia es baja. Sin embargo, una residencia alta significa que la recolección de basura será frecuente, porque el número de celdas libres es pequeño, pero cada una será lenta porque el número de celdas a marcar o copiar será elevado. Reducir ambas cosas también ahorrará tiempo de ejecución, ya que la reserva de una celda conlleva gasto de tiempo en comprobación de rangos, incremento de punteros y otras tareas; y una recolección frecuente y lenta supone una gran carga en el tiempo de ejecución.

1.2 Análisis y problemas que resuelven

Algunos análisis llevados a cabo en el ámbito de la programación funcional son los presentados a continuación. Sobre la mayoría de ellos hablaremos en distintos puntos de este trabajo.

Análisis de estrictez y compartición

Los lenguajes funcionales están basados en el λ–cálculo. En él, la semántica del paso de parámetros a las funciones recibe el nombre de **paso por nombre**, es decir, se sustituye cada aparición del parámetro formal en el cuerpo de la función por el parámetro real al que se aplica la función.

La implementación ingenua del paso de parámetros por nombre sería muy costosa, ya que un mismo argumento se evalúa tantas veces como aparezca en el cuerpo de la función, sin aprovechar los resultados ya obtenidos en apariciones previas del parámetro.

En los primeros lenguajes funcionales el paso de parámetros se implementaba **por valor**, es decir, en la llamada a una función primero se evaluaban los parámetros y una vez evaluados se pasaba al cuerpo de la función. Esta implementación impedía la creación de estructuras infinitas y llevaba a cabo evaluaciones no necesarias. Estos lenguajes reciben el nombre de estrictos.

Posteriormente, en la máquina–G [AJ89], se implementó por primera vez la pereza mediante la introducción del **paso de parámetros por necesidad**.

Tanto en el paso de parámetros por nombre como en el paso por necesidad, un parámetro solamente se evalúa si se utiliza en el cuerpo de la función. Pero ahora, en el segundo caso, una vez evaluado a forma normal débil[1], el parámetro se actualiza con ese valor, por lo que sucesivos usos de ese parámetro no necesitan evaluarlo de nuevo. Es decir, existen partes del árbol abstracto que están siendo compartidas. Por ello a las máquinas que implementan este tipo de paso de parámetros se las llama de **reducción de grafos**

En el primer caso, sin embargo, no existe tal actualización, por lo que cada vez que se usa el parámetro es necesario evaluarlo de nuevo. Por ello se habla de máquinas de **reducción de árboles**.

La implementación del paso por necesidad crea una clausura o **suspensión** para la expresión sin evaluar que corresponde al parámetro real. El paso del parámetro se lleva a cabo mediante el paso de un puntero a dicha clausura, la cual se actualiza una vez evaluada hasta forma normal débil.

La llamada por necesidad permite la existencia de estructuras infinitas cuando se aplica a estructuras de datos. Sin embargo, requiere mecanismos de actualización que encarecen la implementación. Existen algunos casos en los que la llamada por valor o por nombre es más rentable que la llamada por necesidad:

- Saber que una función va a usar con seguridad un parámetro permite una generación de código más eficiente evitando la construcción de clausuras. Si una función es estricta en un argumento, se puede evitar la construcción de una clausura para dicho argumento, y llevar a cabo su evaluación inmediata. Es decir, se usa el paso de parámetros por valor en lugar del paso por necesidad. El análisis que pretende detectar esta situación recibe el nombre de **análisis de estrictez**.

- Cuando sabemos que un parámetro se va a usar a lo sumo una vez en el cuerpo de una función, entonces podemos utilizar el paso por nombre, puesto que una vez usado no se va a volver a usar y por tanto no se pierde nada evitando la actualización. El análisis que pretende detectar esta situación recibe el nombre de **análisis de compartición**.

El análisis de estrictez tiene otras aplicaciones en **máquinas paralelas**. Permite determinar qué partes de la evaluación de los programas se pueden llevar a cabo de forma paralela. Los lenguajes funcionales son ideales para la evaluación paralela de programas debido a la ausencia de efectos laterales. Esta evaluación paralela comienza cuando aplicamos una función a un argumento: el argumento puede evaluarse en paralelo con la llamada a la función y con la evaluación de otros parámetros. Sin embargo, en presencia de la evaluación perezosa, que retarda la evaluación de un argumento hasta que éste se necesita, esto supone un riesgo de creación de procesos paralelos que a continuación van a ser descartados por no ser necesaria su evaluación, con el consiguiente malgasto de trabajo. Sin embargo, si sabemos que un parámetro se va a evaluar con seguridad, entonces este riesgo desaparece.

Análisis del orden de evaluación

El objetivo principal de este análisis es la optimización de suspensiones mediante el empleo de información sobre el uso pasado o futuro de las variables.

[1] En inglés *weak head normal form*

El código usual generado para una variable en ciertas máquinas, como la máquina–G con marcas, es el siguiente:

```
If estado = ya evaluada
 then
      return valor
 else
      evaluar;
      sobreescribir;
      return valor
```

Con la información adecuada podríamos llevar a cabo las siguientes optimizaciones de dicho código:

1. Si sabemos que la variable ya ha sido evaluada no es necesario chequear el estado, ni evaluar de nuevo, ni sobreescribir.

2. Si no ha sido evaluada no hace falta chequear el estado, pero si además sabemos que tampoco se va a evaluar más, entonces no necesitamos sobreescribir.

3. Si no sabemos nada sobre si se ha evaluado o no, pero sabemos que no se va a evaluar más, entonces no necesitamos sobreescribir.

El análisis del orden de evaluación pretende proporcionar dicha información. Subsume tanto al análisis de estrictez como al de compartición.

Globalización de variables

Un problema de ineficiencia en los lenguajes funcionales estrictos es que incluso estructuras de datos esencialmente globales deben pasarse como parámetros, produciéndose la copia de estructuras de datos o de punteros a ellas. Es interesante por lo tanto, detectar automáticamente tales parámetros globales y sustituirlos por variables globales con asignación imperativa.

Este análisis, por tanto, pretende detectar cuándo un parámetro de una función se puede sustituir por una variable global y encontrar grupos de variables que se pueden asociar con la misma variable global.

Análisis de clausuras

Muchos de los análisis que se presentan se aplican fácilmente a lenguajes de primer orden. Cuando queremos extenderlos a lenguajes con orden superior se hace necesario conocer:

- A qué funciones están ligadas los parámetros funcionales.

- A qué función se evalúa realmente la expresión que se aplica como función.

Por ejemplo, sea el programa:

$$
\begin{array}{ll}
\textbf{letrec} & \\
 & f\ x\ y = (x+y) \\
 & \mathit{twice}\ g\ z = g\ (g\ z) \\
\textbf{in} & \\
 & \mathit{twice}\ (f\ 7)\ 9
\end{array}
$$

Para conocer el número de veces que se usa z necesitamos saber a qué funciones puede estar ligada g. Sabiendo que g está ligada a f con un parámetro evaluado y que el otro parámetro sólo aparece una vez en su cuerpo, podemos saber que z se evaluará una vez. En el Ejemplo 73 de la Sección 7.2.2 se estudiarán los resultados para este ejemplo.

El objetivo de este análisis es, por tanto, encontrar un superconjunto del conjunto de clausuras a las que se puede evaluar una expresión y una aproximación al conjunto de argumentos a los que una determinada lambda se puede aplicar.

Análisis del momento de vinculación

En la técnica de *evaluación parcial*, dada una descripción de los parámetros de un programa que se conocerán en momento de especialización, un análisis del momento de vinculación debe determinar qué partes de un programa dependen solamente de estas partes conocidas (y por tanto son también conocidas en momento de especialización). El objetivo de obtener esta información es construir versiones especializadas de las funciones.

Un análisis del momento de vinculación previo al proceso de especialización tiene beneficios prácticos y es esencial en algunas aproximaciones a la generación automática de compiladores eficientes a partir de intérpretes. Cada vez que el compilador se aplica a un programa p, produce como salida una versión del intérprete especializada a p. Supongamos que tenemos un intérprete $\mathit{eval} : \mathit{code} \times \mathit{input} \rightarrow \mathit{output}$ definido en términos de un conjunto de funciones auxiliares $f_1 \ldots f_n$. Para generar un compilador a partir de él, es necesario saber, para cada uso de f_i en el intérprete, cuánto de cada argumento de f_i quedará fijado cuando el código a compilar se conozca. Esta información la proporciona el análisis del momento de vinculación. Puesto que las f_i se definen unas en función de otras, el corazón del análisis debe ser un método que determine para cada f definida en un programa cuánto quedará fijado de $f\ x$ supuesto conocido cuánto queda fijado de x. Este método no debería necesitar saber el valor de x. En la literatura el hecho de que x quedará fijado en momento de evaluación parcial se describe diciendo que x es *estático*, mientras que el hecho contrario se describe diciendo que x es *dinámico*.

Propagación de constantes

En este análisis, el objetivo es determinar si un argumento de una función en un programa se llama siempre con el mismo valor, es decir, con un valor constante. En tal caso el compilador puede explotar dicha información calculando el valor de la constante en tiempo de compilación. Para obtener esta información es necesario saber con qué valores se llama a la función durante la ejecución del programa.

Ausencia

Mientras que en el análisis de estrictez deseamos saber si una función utiliza un parámetro, el análisis de ausencia pretende averiguar cuándo una función no lo utiliza, es decir "ignora" dicho argumento. En tal caso se puede generar para la expresión argumento un código que devuelva cualquier valor.

Divergencia

Este análisis pretende detectar cuándo la evaluación de una expresión no termina, de forma que para una expresión etiquetada como divergente se puede generar un código que aborte directamente el programa.

1.3 Objetivo y contenido del trabajo

Este trabajo de investigación pretende hacer un recorrido de las distintas técnicas de análisis de programas funcionales, así como del tipo de problemas que se resuelve con ellas.

Se persigue que el trabajo pueda servir como guía para quien esté interesado en introducirse en este campo. Por ello, se presentan los fundamentos de las técnicas más relevantes ilustrándolas con ejemplos que faciliten su comprensión. Además se compara la potencia y el coste de cada una de ellas, para que el lector que pretenda aplicar alguna de estas técnicas pueda decidir cual es la más conveniente a sus objetivos. No se han incluido la mayoría de las demostraciones de teoremas, proposiciones y lemas por razones de espacio, pero en cada caso se indica la bibliografía donde pueden encontrarse.

El Capítulo 2 introduce algunos conceptos básicos que se van a usar en el resto del trabajo. Incluye conceptos de teoría de dominios, teoría de categorías, topologías de Scott y dominios potencia. A continuación se expone en el Capítulo 3 una breve historia del análisis de programas funcionales. En ella, además de las técnicas desarrolladas en este trabajo, se nombran otros enfoques de menor relevancia junto con la bibliografía correspondiente.

La técnica más destacada en análisis de lenguajes funcionales es la interpretación abstracta. No solamente se aplica en este tipo de lenguajes, por lo que en el Capítulo 4 se presentan algunos conceptos generales sobre interpretación abstracta, independientes del tipo de lenguaje particular utilizado.

A la hora de particularizar estos conceptos para aplicarlos a los lenguajes funcionales surgen distintos enfoques. La primera clasificación se puede hacer entre análisis hacia delante y análisis hacia atrás, dependiendo de la forma en que se propaga la información a lo largo del árbol de sintaxis abstracta. Normalmente se suele llamar interpretación abstracta propiamente dicha al análisis hacia delante. Una segunda clasificación se realiza en función de la forma en que se representan las propiedades de los objetos: tenemos interpretación abstracta basada en conjuntos Scott–cerrados o en relaciones de equivalencia parcial. Y finalmente se puede realizar una clasificación entre interpretación abstracta basada en semántica denotacional, y la basada en semántica operacional.

En los siguientes capítulos se intenta seguir un orden cronológico en las distintas aproximaciones de la interpretación abstracta, aunque algunas de ellas se superponen. En el Capítulo 5 se presenta con detalle una interpretación abstracta hacia delante basada en semántica denotacional que utiliza conjuntos Scott–cerrados para representar las propiedades. Se hace de una forma incremental para facilitar la comprensión comenzando por un lenguaje de primer orden y extendiéndolo posteriormente a orden superior, estructuras de datos y polimorfismo. Se presentan además algunas técnicas que pretenden calcular de forma eficiente o de forma aproximada puntos fijos de funciones. En el Capítulo 6 se presenta la técnica del análisis hacia atrás. A continuación, en el Capítulo 7 se presentan las técnicas de interpretación abstracta basada en relaciones de equivalencia parcial e interpretación abstracta basada en semántica operacional. Finalmente, en el Capítulo 8 se presentan los análisis basados en tipos.

La mayoría de los problemas de la Sección 1.2 se presentan como ejemplo de alguna de las técnicas. No todos los análisis se pueden llevar a cabo con todas las técnicas, en cuyo caso se exponen las razones por las que se pueden usar unas y no otras. Finalmente, se incluye una sección de conclusiones, donde la autora enjuicia los logros y debilidades de las técnicas presentadas en el trabajo y su evolución futura. En dicha sección también se avanzan posibles líneas de investigación de la autora en este campo.

En la bibliografía se resaltan en negrita aquellas referencias que la autora ha considerado claves en el área sobre la que se desarrolla este trabajo.

Capítulo 2

Conceptos básicos

En este capítulo se presentan algunos conceptos utilizados en el resto del trabajo. Con ello se pretende que este trabajo sea lo más autocontenido posible. Se presentan algunos conceptos de teoría de dominios [Rea89] [AJ94], teoría de categorías [Pie91] [Mac71] [Bar93], topologías de Scott y dominios potencia [Bur91].

2.1 Teoría básica de dominios

Los análisis de programas basados en interpretación abstracta requieren cierto conocimiento de la teoría de dominios, la cual estudia conjuntos con cierta estructura. Por ello a continuación proporcionamos los conceptos necesarios en el entorno que vamos a desarrollar.

2.1.1 Órdenes parciales

Dado un conjunto P, una **relación binaria** sobre P es un conjunto de pares de elementos de P, es decir un subconjunto de $P \times P$.

Un **preorden** sobre P es una relación binaria $\sqsubseteq$ sobre P con las propiedades reflexiva y transitiva:

- **Reflexiva**: $\forall x \in P. x \sqsubseteq x$.
- **Transitiva**: $\forall x, y, z \in P. (x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow x \sqsubseteq z$.

Si ademas posee la propiedad **antisimétrica**

$$\forall x, y \in P. (x \sqsubseteq y \wedge y \sqsubseteq x) \Rightarrow x = y$$

entonces decimos que $\sqsubseteq$ es un **orden parcial** sobre P. En tal caso, decimos que P es un **conjunto parcialmente ordenado**, y lo denotamos por $< P, \sqsubseteq >$. Para abreviar llamaremos **poset** a un conjunto parcialmente ordenado.

Los conjuntos parcialmente ordenados finitos de pequeño tamaño se pueden representar mediante **diagramas de Hasse**, ver ejemplos de la Figura 2.1. Los elementos se representan mediante puntos en el plano, y la relación de orden mediante líneas que unen dichos puntos, donde el elemento situado más arriba es mayor que el elemento situado más abajo. No es necesario pintar todas las líneas gracias a la transitividad.

Algunos conjuntos parcialmente ordenados infinitos también se pueden representar de esta forma mostrando una parte finita que revela la estructura con la que están construidos.

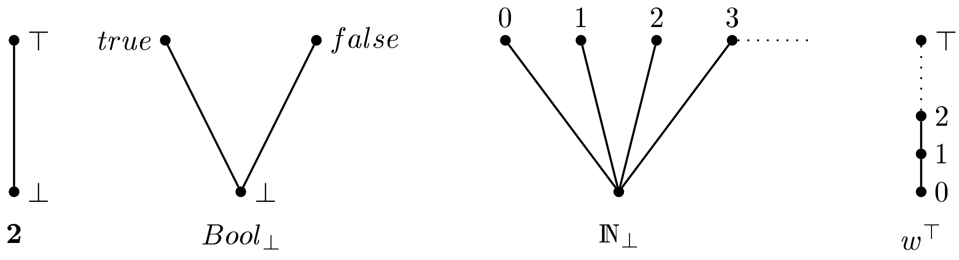


Figura 2.1: Algunos ejemplos de posets que también son cpos

Un orden parcial es **total** si todos los elementos son comparables, es decir,

$$(\forall x, y. x \sqsubseteq y \vee y \sqsubseteq x).$$

Una **cadena** es un subconjunto X de P en el que $\sqsubseteq$ es un orden total.

Dado un poset $< P, \sqsubseteq >$, decimos que x es una **cota superior** de un subconjunto A de P si x es mayor o igual que todos los elementos de A, es decir,

$$(\forall a \in A. a \sqsubseteq x).$$

Al conjunto de todas las cotas superiores de A lo denotamos por $ub(A)$.

Llamamos **mínima cota superior** o **supremo** a la menor de las cotas superiores. Es decir, x es supremo de A si:

$$(\forall a \in A. a \sqsubseteq x \ \wedge \ \forall x' \in P.[(\forall a \in A. a \sqsubseteq x') \Rightarrow (x \sqsubseteq x')])$$

Se puede demostrar que si existe, es único.

Decimos que x es una **cota inferior** de A si y sólo si x es menor o igual que todos los elementos de A, es decir,

$$(\forall a \in A. x \sqsubseteq a)$$

Al conjunto de todas las cotas inferiores de A lo denotamos por $lb(A)$.

Llamamos **máxima cota inferior** o **ínfimo** a la mayor de las cotas inferiores. Es decir, $y \in P$ es ínfimo de A si:

$$(\forall a \in A. y \sqsubseteq a \ \wedge \ \forall y' \in P.(\forall a \in A. y' \sqsubseteq a) \Rightarrow (y' \sqsubseteq y))$$

Si existe, es único.

Dado un poset P, un subconjunto A de P es **dirigido** si es no vacío y para cada par de elementos de A existe una cota superior en A.

Un **orden parcial completo** es un poset D en el que cada subconjunto dirigido tiene un supremo, o lo que es equivalente, en el que cada cadena tiene un supremo. Para abreviar llamamos **cpo** a un poset completo.

Un cpo con ínfimo $\perp$ se llama cpo **apuntado** o **estricto**. En adelante, cuando hablemos de cpos los supondremos apuntados.

Todo orden parcial finito es un orden parcial completo.

Algunos ejemplos de cpos son:

- El cpo de un único elemento I.
- El cpo **2**, que posee dos elementos $\perp$ y $\top$ (también representados a veces por 0 y 1) con $\perp \sqsubseteq \top$, ver Figura 2.1.
- El cpo de los booleanos, ver Figura 2.1, representado por $Bool_\perp$ y formado por los elementos $\{\perp, true, false\}$ tales que $\perp \sqsubseteq true$ y $\perp \sqsubseteq false$.
- El cpo plano de los números naturales con $\perp$, $\mathbb{N}_\perp$, ver Figura 2.1, donde

$$\forall n \in \mathbb{N}_\perp . \perp \sqsubseteq n \wedge \forall n, m \in \mathbb{N}_\perp . n \sqsubseteq m \text{ si y solo si } n = m$$

- Los ordinales w no constituyen un cpo, ver Figura 2.3. Para que formen un cpo es necesario añadir un elemento supremo $\top$, $w^\top$, ver Figura 2.1.
- Dado un conjunto S, el conjunto de todos los subconjuntos de S ordenados por inclusión de conjuntos $P(S)$ forman un cpo. Como ejemplo, en la Figura 2.2 se muestra el cpo $P(\{0,1,2\})$.

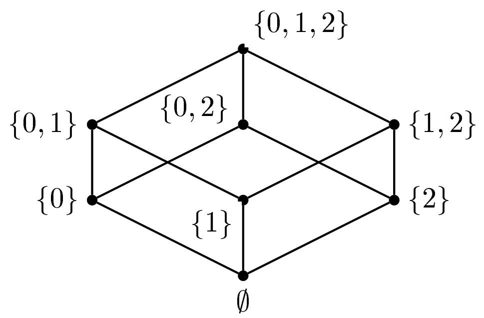


Figura 2.2: El cpo de las partes de $S = \{0,1,2\}$

- Dado un conjunto cualquiera S, se puede definir un cpo a partir de él, $(S_\perp, \sqsubseteq)$, añadiendo un elemento nuevo, $\perp$, a S donde:

$$s_1 \sqsubseteq s_2 \text{ si y solo si } s_1 = \perp \text{ ó } s_1 = s_2$$

Este tipo de cpos reciben el nombre de **planos** puesto que todos los elementos distintos de $\perp$ se encuentran al mismo nivel. Ejemplos de cpos planos son el cpo de los booleanos y el de los naturales de la Figura 2.1.

Figura 2.3: Un ejemplo de poset no completo: w

2.1.2 Retículos

Un conjunto parcialmente ordenado en el que para cada par de elementos existe el supremo (ínfimo) es un $\sqcup(\sqcap)$–**semiretículo**. Si existen ambos entonces es un **retículo**. Un ejemplo es el mostrado en la Figura 2.4, formado por cuatro elementos:

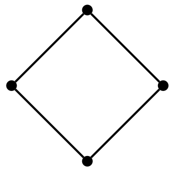


Figura 2.4: Un retículo completo de cuatro puntos

Un retículo L es **completo** si todo subconjunto A de L tiene supremo e ínfimo. Los retículos completos se suelen denotar de la siguiente manera:

$$L(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$$

donde $\top$ y $\perp$ denotan respectivamente el supremo e ínfimo de L, y $\sqcup$ y $\sqcap$ representan los operadores de supremo e ínfimo. Todo retículo finito es completo.

Todo retículo completo es también un orden parcial completo.

Un ejemplo de retículo no completo es la cadena de los números naturales con el orden habitual $0 \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq \ldots$, puesto que para cada dos elementos existe el supremo, pero para un subconjunto infinito, como por ejemplo el subconjunto de los números pares, no existe.

2.1.3 Funciones y puntos fijos

Dados dos cpos D y E, denotemos por $(D \to E)$ al conjunto de funciones de D en E. Dados dos cpos D y E, una función $f : D \to E$ es **monótona** si preserva el orden, es decir,

$$\forall x \in D,\ \forall y \in E. x \sqsubseteq y \text{ en D } \Rightarrow f(x) \sqsubseteq f(y) \text{ en E}$$

Dada una función $f : D \to D$, decimos que $x \in D$ es un **punto fijo** de f si $f(x) = x$. Es el **mínimo punto fijo** si para cualquier otro punto fijo y de f, se cumple $x \sqsubseteq y$. Es el **máximo punto fijo** si para cualquier otro punto fijo y de f, se cumple $x \sqsupseteq y$.

Teorema 1 *(del punto fijo de Tarski)*
Los puntos fijos de una función monótona f sobre un retículo completo $L(\sqsubseteq, \perp, \top, \sqcup, \sqcap)$ forman un retículo completo con orden $\sqsubseteq$, ínfimo el mínimo punto fijo $lfp\ f$, supremo el máximo punto fijo $gfp\ f$ y operadores de supremo e ínfimo los mismos que L, donde:

- $lfp\ f = \sqcap\{x \in L \mid f(x) \sqsubseteq x\}$ *y*
- $gfp\ f = \sqcup\{x \in L \mid x \sqsubseteq f(x)\}$

Una función f entre cpos D y E es **continua** si para cada subconjunto dirigido A de D tenemos $f(\bigsqcup A) = \bigsqcup f(A)$. Continuidad implica monotonía. Denotamos por $[D \to E]$ al conjunto de funciones continuas de D en E.

Además, si una función $f : D \to E$ es monótona y D es finito, entonces f es continua. De hecho, esto es cierto siempre que no existan cadenas ascendentes infinitas en D. Por ejemplo, cualquier función monótona $f : \mathbb{N}_\perp \to E$ es continua.

Una función continua entre cpos apuntados D y E es **estricta** si $f(\perp_D) = \perp_E$. Se dice que **refleja el ínfimo** si $f(d) = \perp_E \Rightarrow d = \perp_D$.

El conjunto de todas las funciones continuas $[D \to E]$ entre dos cpos D y E es otro cpo, donde si f y g son funciones continuas de D en E, entonces $f \sqsubseteq g$ se define como $\forall d \in D :\ f(d) \sqsubseteq g(d)$. El supremo se calcula punto a punto.

Si E es apuntado, es decir, tiene un ínfimo $\perp_E$, entonces el cpo de las funciones continuas es también apuntado donde $\forall d \in D. \perp_{[D \to E]}(d) = \perp_E$

Teorema 2 *Dado un cpo apuntado D:*

- *Cada función continua f de D en D tiene un mínimo punto fijo dado por $\bigsqcup_{n \in \mathbb{N}} f^n(\perp)$, donde f^0 es la función identidad y $f^{n+1} = f \circ f^n$.*
- *La función fix que a cada función continua le hace corresponder su mínimo punto fijo es también continua.*

Se dice que x es un **punto posfijo** de f si $f(x) \sqsubseteq x$.

Si x es un punto posfijo de f mayor que el mínimo punto fijo de f entonces $\{f^n(x) \mid n \geq 0\}$ es una cadena descendente y para todo n se cumple que $f^n(x) \sqsupseteq fix(f)$.

El límite de esta cadena no tiene por qué ser necesariamente el mínimo punto fijo.

Se dice que x es un **punto prefijo** de f si $x \sqsubseteq f(x)$.

Si x es un punto prefijo de f menor que el mínimo punto fijo de f entonces $\{f^n(x) \mid n \geq 0\}$ es una cadena ascendente y para todo n se cumple que $f^n(x) \sqsubseteq fix(f)$.

Si trabajamos sobre retículos finitos además el límite de esta cadena será el mínimo punto fijo.

Una función $f : L \to M$ es **aditiva** o **lineal**, siendo L y M dos retículos, si:

$$f(l_1 \sqcup l_2) = f(l_1) \sqcup f(l_2)$$

para todo $l_1, l_2 \in L$, es decir, si es un homomorfismo respecto a $\sqcup$.

Una función $f : L \times L' \to M$ es **separadamente aditiva**, donde $L \times L'$ es el producto cartesiano de los retículos L y L' y M es un retículo, si:

$$\begin{array}{rcl} f(l_1 \sqcup l_2, l') & = & f(l_1, l') \sqcup f(l_2, l') \\ f(l, l'_1 \sqcup l'_2) & = & f(l, l'_1) \sqcup f(l, l'_2) \end{array}$$

para todo $l, l_1, l_2 \in L$ y $l', l'_1, l'_2 \in L'$.

Si f es aditiva sobre $L \times L' \to M$ entonces es separadamente aditiva pero no viceversa.

2.1.4 Dominios

Un dominio es un cpo con algunas propiedades adicionales que garantizan la existencia de soluciones a las ecuaciones recursivas de dominios.

Sea D un cpo. Un elemento $x \in D$ es **compacto** o **finito** si, para todo subconjunto dirigido M de D tal que $x \sqsubseteq \bigsqcup M$, existe un $y \in M$ tal que $x \sqsubseteq y$. Llamemos $K(D)$ al conjunto de los elementos compactos de D.

Sea el cpo de los números naturales ordenados de la forma habitual, con supremo $\top$ ($w^\top$). En dicho cpo todos los elementos excepto $\top$ son compactos.

El cpo D es **algebraico** si para todo $x \in D$, el conjunto

$$M = \{x_0 \in K(D) \mid x_0 \sqsubseteq x\}$$

es dirigido y $\bigsqcup M = x$. En otras palabras, cada elemento es el límite de sus aproximaciones finitas.

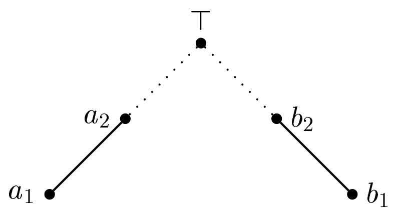


Figura 2.5: Ejemplo de cpo no algebraico

Un ejemplo de cpo no algebraico es el de la Figura 2.5. Este cpo no es algebraico porque carece de elementos compactos. Ninguno de los a_i puede ser compacto puesto que tomando en la definición de elemento compacto $M = \{b_n\}$, se cumple que $a_i \sqsubseteq \bigsqcup M = \top$ y sin embargo $\forall j . a_i \not\sqsubseteq b_j$. De forma similar se puede razonar con los b_j. $\top$ tampoco es compacto, y por tanto ningún elemento lo es.

Existen cpos algebraicos D que cumplen que $[D \to D]$ no es algebraico. Para lograr que $[D \to D]$ pertenezca a la misma categoría de objetos, es necesario imponer algunas condiciones adicionales.

Un conjunto $X \subseteq D$ es **consistente** si todo subconjunto suyo tiene cota superior. Luego, todo conjunto dirigido es consistente.

Se dice que un cpo D es **consistentemente completo** o **coherente** si todos sus subconjuntos consistentes tienen supremo en D:

$$\forall X \subseteq D . (X \text{ consistente } \Rightarrow \bigsqcup X \text{ existe en } D)$$

En la Figura 2.6 se muestran ejemplos de cpos algebraicos, uno de los cuales es coherente y el otro no lo es.

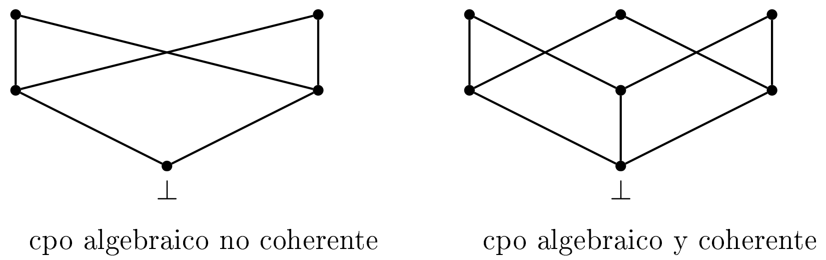


Figura 2.6: Ejemplos de cpos algebraicos coherentes y no coherentes

Un **dominio de Scott** es un cpo algebraico consistentemente completo. Por abreviar los llamaremos dominios. Todos los cpos planos son dominios de Scott y todos los retículos completos son dominios de Scott.

Construcciones sobre dominios

Producto

Dados dos posets $(D, \sqsubseteq_1)$ y $(E, \sqsubseteq_2)$, el producto $(D \times E, \sqsubseteq)$ es el conjunto de pares (x, y) donde $x \in D$ e $y \in E$.

El orden se define componente a componente:

$$(x, y) \sqsubseteq (x', y') \text{ si y solo si } x \sqsubseteq_1 y \text{ y } x' \sqsubseteq_2 y'$$

En caso de ser ambos cpos, el producto es un cpo. El ínfimo es $\perp_{D\times E} = (\perp_D, \perp_E)$ y el operador de supremo $\bigsqcup L = (\bigsqcup M, \bigsqcup N)$ donde:

$$M \quad = \quad \{x \mid \exists y \in E . (x, y) \in L\}$$

$$N \quad = \quad \{y \mid \exists x \in D.(x,y) \in L\}$$

Si ambos son dominios, entonces el producto es dominio donde

$$K(D \times E) = \{(u,v) \mid u \in K(D), v \in K(E)\}$$

En semántica estricta de programas funcionales necesitamos identificar todas las tuplas con una componente indefinida con el propio indefinido, por ello surge otro producto (versión colapsada):

$$D_1 \otimes D_2 = \{(x,y) \in D_1 \times D_2 \mid x \neq \perp \text{ y } y \neq \perp\} \cup \{\perp_{D_1 \otimes D_2}\}$$

donde $\perp_{D_1 \otimes D_2}$ es un nuevo elemento tal que

$$\forall p \in D_1 \otimes D_2 . \perp_{D_1 \otimes D_2} \sqsubseteq p$$

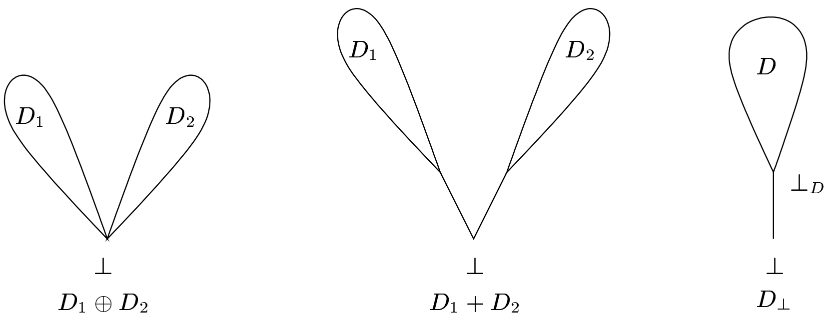
Unión de dominios

La unión de dominios es útil para dotar de semántica a los tipos de datos algebraicos de los lenguajes funcionales. Dados dos cpos D_1 y D_2, su unión o **suma coalescente**, $D_1 \oplus D_2$, es un nuevo cpo formado por la unión disjunta de los elementos de D_1 y D_2, donde los elementos $\perp_{D_1}$ y $\perp_{D_2}$ se identifican en un único elemento $\perp_{D_1 \oplus D_2}$, ver Figura 2.7, donde $d \sqsubseteq d'$ en $D_1 \oplus D_2$ si:

- O bien $d = \perp_{D_1 \oplus D_2}$,
- o bien d y d' pertenecen al mismo D_i $(i = 1, 2)$ y $d \sqsubseteq d'$ en D_i.

Normalmente se define extendiendo los elementos de cada D_i con una etiqueta para distinguirlos.

Otro tipo de unión es la **suma separada** $D_1 + D_2$, formada por la unión disjunta de los elementos de D_1 y D_2 junto con un nuevo elemento $\perp_{D_1+D_2}$ distinto de los demás y por debajo de ellos, ver Figura 2.7.



(a) (b) (c)

Figura 2.7: (a) Suma coalescente de D_1 y D_2; (b) suma separada de D_1 y D_2; (c) elevación de D

Si D_1 y D_2 son dominios, $D_1 + D_2$ es también un dominio donde $K(D_1 + D_2)$ es la unión disjunta de los elementos de $K(D_1)$ y $K(D_2)$, junto con el elementos nuevo $\perp_{D_1+D_2}$.

Elevación de dominios

Una construcción útil en semántica perezosa es la **elevación de dominios**. Dado un constructor de tipos C, en semántica perezosa queremos distinguir entre una expresión de dicho tipo totalmente indefinida y otra que se evalua a $C\ E$ donde E es una expresión indefinida. Es decir, queremos un nuevo elemento distinto de $C\bot$ que represente la indefinición total, el cual se consigue elevando el dominio con un $\bot$ nuevo por debajo de $C\ \bot$.

Dado un cpo D, llamamos $D_\bot$ al cpo elevado, cuyos elementos son los de D junto con un ínfimo añadido $\bot$ por debajo de $\bot_D$, ver Figura 2.7.

A veces se representan los elementos de la siguiente forma:

- $\bot \in D_\bot$
- $<0, d> \in D_\bot$ si $d \in D$

Usaremos esta notación en la Sección 6.3. Si D es un dominio, entonces $D_\bot$ es un dominio, donde $K(D_\bot) = \{\bot\} \cup \{<0, d> \mid d \in K(D)\}$.

Se cumple que:

$$D + E = D_\bot \oplus E_\bot$$

Dada una función $f : D \to E$, se define su **promoción** $f_\bot : D_\bot \to E_\bot$:

$$\begin{array}{l} f_\bot(\bot) = \bot \\ f_\bot(<0, d>) = f(d) \end{array}$$

Dominio de funciones continuas

Dados dos dominios D_1 y D_2, llamamos $[D_1 \to D_2]$ al dominio de las funciones continuas entre dichos dominios, donde

$$\forall d_1 \in D_1.f \sqsubseteq g \text{ si y solo si } f(d_1) \sqsubseteq g(d_1)$$

Producto tensorial

En el caso particular de los retículos completos y funciones continuas, dados dos retículos completos L y L', un **producto tensorial** es un par $(L \otimes L', cross)$ donde:

- $L \otimes L'$ es un retículo completo.
- $cross : L \times L' \to L \otimes L'$ es una función continua separadamente aditiva.
- Para cada retículo completo M y función continua $f : L \times L' \to M$ separadamente aditiva existe una única función continua $f^\otimes : L \otimes L' \to M$ aditiva tal que $f^\otimes \circ cross = f$. Es decir, el siguiente diagrama es conmutativo:

$$\begin{tikzcd} L \times L' \arrow[d, "cross"'] \arrow[dr, "f"] & \\ L \otimes L' \arrow[r, "f^\otimes"'] & M \end{tikzcd}$$

Un caso particular de producto tensorial se utilizará en la Sección 5.4.1 para combinar propiedades de las componentes de un par de forma disyuntiva.

Ecuaciones de dominio recursivas

Cuando se intenta proporcionar una semántica denotacional al λ–cálculo sin tipos surge la necesidad de resolver ecuaciones recursivas de dominios. Para poder expresar el orden superior es necesario resolver la ecuación entre dominios $D = A + [D \to D]$ donde A es un dominio base.

Es bien conocido que los dominios con las funciones continuas entre ellos forman una categoría (ver Sección 2.2) cartesianamente cerrada (cerrada bajo producto y espacio de funciones) en la que esta ecuación recursiva tiene solución.

La solución a esta ecuación recursiva es un modelo del λ–cálculo sin tipos, y la semántica denotacional estándar hace corresponder a cada construcción sintáctica un valor en dicho dominio, ver Sección 5.3.2.

Los tipos de datos definidos de forma recursiva también dan lugar a ecuaciones de dominios recursivas. Por ejemplo, el dominio de las listas es solucion de la ecuacion recursiva $D_{list(\tau)} = I + (D_\tau \times D_{list(\tau)})$.

2.2 Teoría básica de categorías

Esta sección introduce conceptos básicos de la teoría de categorías que junto con los conceptos de la Sección 2.4 serán necesarios en la Sección 5.5.3, para estudiar el polimorfismo en la interpretación abstracta.

Una **categoría** K es una estructura definida por los siguientes conjuntos de datos y propiedades:

1. Una colección de objetos $Obj(K)$.
2. Para cada par de objetos A y B en $Obj(K)$, existe un conjunto $Morf_K(A, B)$ llamado el conjunto de los morfismos de A a B.
3. Dados $f \in Morf_K(A, B)$ y $g \in Morf_K(B, C)$, se puede formar la composición $g \circ f \in Morf_K(A, C)$.
4. Para cada objeto A existe un morfismo $id_A \in Morf_K(A, A)$, llamado el morfismo identidad. Para cada $f \in Morf_K(A, B)$, se cumple que $f \circ id_A = f$ y $id_B \circ f = f$.
5. La composición definida en el tercer apartado es asociativa.

Algunos ejemplos de categorías son los siguientes:

1. Los conjuntos forman una categoría, donde los objetos son conjuntos, los morfismos son funciones entre conjuntos, la composición de morfismos es la composición ordinaria de funciones y el morfismo identidad es la función identidad.
2. Los conjuntos parcialmente ordenados forman una categoría, donde los objetos son conjuntos parcialmente ordenados y los morfismos funciones continuas (el resto igual que en el punto anterior).

3. Los dominios también forman una categoría donde los objetos son dominios, los morfismos son las funciones continuas, la composición de morfismos es la composición ordinaria de funciones y el morfismo identidad es la función identidad.

Dada una categoría K, su **categoría opuesta** K^{op} es la formada por los mismos objetos que K, y tal que para cada par de objetos A y B, $f \in Morf_{K^{op}}(B, A)$ si y solo si $f \in Morf_K(A, B)$.

Sean $\mathcal{C}$ y $\mathcal{D}$ dos categorías. La **categoría producto** $\mathcal{C} \times \mathcal{D}$, es la categoría cuyos objetos son pares de la forma (A, B) donde A y B son objetos de $\mathcal{C}$ y $\mathcal{D}$ respectivamente. Un morfismo entre (A, B) y (C, D), donde $A, C \in \mathcal{C}$ y $B, D \in \mathcal{D}$, será de la forma (f, g) donde f y g son morfismos de A a C y de B a D respectivamente. La composición de morfismos se hace componente a componente. Esta definición se puede generalizar a productos de n categorías.

Un **funtor** F entre dos categorías $\mathcal{C}$ y $\mathcal{D}$ es una aplicación que a cada objeto de $\mathcal{C}$ le hace corresponder un objeto de $\mathcal{D}$, y para cada par de objetos A y B en $\mathcal{C}$, F hace corresponder a cada morfismo $f \in Morf_{\mathcal{C}}(A, B)$ un morfismo $F(f) \in Morf_{\mathcal{D}}(F(A), F(B))$ satisfaciendo las siguientes condiciones:

- $F(id_A) = id_{F(A)}$ para cada $A \in Obj(\mathcal{C})$.
- $F(f \circ g) = F(f) \circ F(g)$ siempre que $f \circ g$ esté definida en $\mathcal{C}$.

Tales funtores reciben el nombre de **covariantes**. Si en lugar de cumplirse la segunda condición, se cumpliera $F(f \circ g) = F(g) \circ F(f)$, entonces decimos que F es **contravariante**.

Sean ahora F y G dos funtores entre las categorías $\mathcal{C}$ y $\mathcal{D}$. Y sea $\{f_A\}$ una colección de morfismos indexada por los objetos de la categoría $\mathcal{C}$. donde cada $f_A \in Morf_{\mathcal{D}}(F(A), G(A))$. Llamamos a esta colección **transformación natural** de F a G si para cada par de objetos A y B en la categoría $\mathcal{C}$, y para cada morfismo $h \in Morf_{\mathcal{C}}(A, B)$ se cumple:

$$f_B \circ F(h) = G(h) \circ f_A$$

es decir, el siguiente diagrama es conmutativo:

$$\begin{tikzcd}
F(A) \arrow[r, "f_A"] \arrow[d, "F(h)"'] & G(A) \arrow[d, "G(h)"] \\
F(B) \arrow[r, "f_B"'] & G(B)
\end{tikzcd}$$

2.3 Topología de Scott y dominios potencia

Los conceptos introducidos en esta sección serán usados posteriormente en la Sección 5.3.2.

2.3.1 Topologías

Sea D un conjunto no vacío. Una colección, Ω, de subconjuntos de D que contiene tanto a D como al conjunto vacío $\emptyset$, y que es cerrado bajo uniones arbitrarias y número finito de intersecciones recibe el nombre de **topología sobre D**. También se dice que (D, Ω) es un **espacio topológico**.

Los elementos de un espacio topológico reciben el nombre de **conjuntos abiertos** (y sus complementarios el de **conjuntos cerrados**).

Una colección B de conjuntos abiertos de D es una **base** de la topología Ω si cada elemento de Ω es unión de algunos de los elementos de B. Ω siempre es base, pero suelen interesar aquellas que son mínimas en algún sentido.

Dados dos espacios topológicos (D_1, Ω_1) y (D_2, Ω_2) y una función $f : D_1 \to D_2$, decimos que f es **continua** (en el sentido topológico) si para cada conjunto abierto $\mathcal{O}$ en D_2, $f^{-1}(\mathcal{O})$ es abierto en D_1. Es decir, las imágenes inversas de conjuntos abiertos también son abiertos.

2.3.2 Topología de Scott

De entre las topologías nos interesan las que surgen de los dominios. Sea D un dominio y Ω la colección de subconjuntos $\mathcal{O}$ de D **cerrados superiormente**, es decir,

$$\forall x \in \mathcal{O}. \forall y \in D : x \sqsubseteq y \Rightarrow y \in \mathcal{O}$$

Cuando S es un subconjunto dirigido de D con $\bigsqcup S \in \mathcal{O}$ entonces $S \cap \mathcal{O} \neq \emptyset$.

No es difícil ver que Ω así construido es una topología, que recibe el nombre de **topología de Scott** sobre D. Además las funciones continuas entre dominios son también continuas en el sentido topológico y viceversa.

Si tomamos, por ejemplo, el dominio de los números naturales con el supremo $\top$ y el orden habitual, los conjuntos Scott–cerrados serán de la forma $S_n = \{m \mid m \sqsubseteq n\}$ para cada n en dicho conjunto.

Los conjuntos cerrados S en esta topología, llamados conjuntos **Scott–cerrados** son subconjuntos de D tales que:

1. Si $x \sqsubseteq s$ donde $s \in S$ entonces $x \in S$
2. $X \subseteq S$ con X dirigido, entonces $\bigsqcup X \in S$

En la Sección 5.3.2 se presenta una interpretación abstracta en la que las propiedades se representan mediante conjuntos Scott–cerrados.

Si D es un dominio , sean:

$$\begin{array}{rcl} a^{\uparrow} & = & \{x \in D \mid a \sqsubseteq x\} \\ a^{\downarrow} & = & \{x \in D \mid x \sqsubseteq a\} \end{array}$$

los cuales reciben respectivamente los nombres de **cierre superior** y **cierre inferior** de $\{a\}$. A veces, el cierre inferior de un conjunto S, que se forma uniendo todas los cierres inferiores de sus elementos, recibe también el nombre de **cierre por la izquierda** y se escribe $LC(S)$.

Si $\{a_1, a_2, \ldots\}$ son los elementos de D, entonces cada $a_i^{\uparrow}$ es abierto y cada conjunto abierto $\mathcal{O}$ es una unión de algunos de dichos conjuntos. Por tanto, estos conjuntos forman una base de la topología de Scott. Reciben el nombre de conjuntos abiertos básicos.

Además los elementos finitos de D son aquellos $a \in D$ tales que $a^{\uparrow}$ es abierto. Sin embargo, no se pueden describir los conjuntos cerrados en función de los $a_i^{\downarrow}$.

2.3.3 Dominios potencia

Dado un dominio D, existen varias formas de definir órdenes sobre el **conjunto potencia de** D (conjunto de los subconjuntos de D). Introduciendo distintas identificaciones entre elementos del conjunto potencia se obtienen los **dominios potencia**. Qué conjuntos se identifican depende de la construcción particular que se use. Sean $X, Y \in P(D)$:

- **Dominio potencia de Smyth o superior:**
  El orden establecido en este dominio potencia es el siguiente:

  $$X \sqsubseteq^{\#} Y \text{ si y solo si } \forall x \in X.\exists y \in Y.x \sqsupseteq y$$

  Sus elementos finitos son uniones finitas de conjuntos abiertos básicos de D. Elementos típicos son las intersecciones de cadenas decrecientes de tales elementos finitos.

- **Dominio potencia de Hoare o inferior:**
  El orden es el siguiente:

  $$X \sqsubseteq^{b} Y \text{ si y solo si } \forall y \in Y.\exists x \in X.x \sqsupseteq y$$

  Sus elementos finitos son uniones finitas de los cierres inferiores de los elementos finitos de D. Los elementos de este dominio potencia son uniones de cadenas crecientes de tales conjuntos.

- **Dominio potencia de Plotkin o convexo:**
  El orden es el siguiente:

  $$X \sqsubseteq^{\natural} Y \text{ si y solo si } X \sqsubseteq^{b} Y \text{ y } X \sqsubseteq^{\#} Y$$

Dado un dominio D, el dominio potencia de Hoare, denotado por $\mathcal{P}(D)$ se forma tomando como elementos los conjuntos Scott–cerrados no vacíos, ordenados por inclusión de conjuntos.

Por ello nos interesa el dominio potencia de Hoare. En la Sección 5.3.2 se utilizará para representar el conjunto de todas las propiedades (los conjuntos Scott–cerrados) definidas sobre un cierto dominio D.

En términos de teoría de categorías $\mathcal{P}$ es un funtor, y por tanto se puede aplicar a funciones continuas entre dominios para dar funciones continuas entre dominios potencia. Es decir, si $f : D \to E$, entonces $\mathcal{P}(f) : \mathcal{P}(D) \to \mathcal{P}(E)$ se define:

$$\forall X \in \mathcal{P}(D).(\mathcal{P}(f))\ (X) = \{f\ x \mid x \in X\}^*$$

donde X^* representa el menor conjunto Scott–cerrado que contiene a X.

Se pueden definir las siguientes funciones:

- $\{\{\bullet\}\} : D \to \mathcal{P}(D)$
  $\{\{d\}\} = \{d' \in D \mid d' \sqsubseteq d\}$

- $\biguplus_D : \mathcal{P}(\mathcal{P}(D)) \to \mathcal{P}(D)$
  $\biguplus_D \theta = \{x \mid \text{ para algún } X \in \theta, x \in X\} = \bigcup \theta$

- $\biguplus_D : D \to \mathcal{P}(D)$
  $\biguplus_D = \bigsqcup$

Se cumplen algunas propiedades:

1. Si D es un dominio entonces $\mathcal{P}(D)$ también es un dominio.
2. Si $f : D \to E$, $\mathcal{P}(f) : \mathcal{P}(D) \to \mathcal{P}(E)$ es continua.
3. $\mathcal{P}(f \circ g) = (\mathcal{P}(f)) \circ (\mathcal{P}(g))$.
4. $\mathcal{P}(id_D) = id_{\mathcal{P}(D)}$.
5. $\{\{\bullet\}\}_D$ es continua.
6. Para $f : D \to E$, $\mathcal{P}(f) \circ \{\{\bullet\}\}_D = \{\{\bullet\}\}_E \circ f$.
7. $\biguplus_D$ es continua.
8. Para $f : D \to E$, $\bigcup_E \circ \mathcal{P}(\mathcal{P}(f)) = \mathcal{P}(f) \circ \bigcup_D$.
9. $\bigsqcup$ es continua.
10. $\bigsqcup \circ \{\{\bullet\}\}_D = id_D$.
11. Cuando X es subconjuntos de un retículo finito, $\bigsqcup X^* = \bigsqcup X$.

que nos llevan a concluir que $\mathcal{P}$ es un funtor sobre la categoría de los dominios, que $\{\{\bullet\}\}$ es una transformación natural entre el funtor identidad en la categoría de dominios y $\mathcal{P}$, que $\bigcup$ es una transformación natural de $\mathcal{P}^2$ a $\mathcal{P}$.

Estas propiedades se ven involucradas en las demostraciones de los teoremas de la Sección 5.3.2, necesarios para demostrar la corrección del análisis allí presentado.

2.4 Transformaciones naturales laxas

A continuación se presentan algunos conceptos sobre una categoría particular que se utilizará más adelante en la Sección 5.5.3. Las demostraciones de los teoremas y proposiciones en esta sección se pueden encontrar en [Bar93].

2.4.1 Pares inmersión–clausura

Dada la categoría C de dominios y funciones continuas, sea la categoría C^{ec} cuyos objetos son los mismos de C y cuyos morfismos son de la forma $(e, c) : A \to^{ec} B$, donde $e : A \to B$ y $c : B \to A$ son morfismos en C que satisfacen:

$$c \circ e = id \text{ y } e \circ c \sqsupseteq id$$

Llamamos a este par de morfismos par inmersión–clausura, donde e es la inmersión, y c es la clausura. C^{ec} recibe el nombre de **categoría de dominios y pares inmersión–clausura**.

La composición de morfismos en C^{ec} se define:

$$(e_2, c_2) \circ (e_1, c_1) = (e_2 \circ e_1, c_1 \circ c_2)$$

Dado un morfismo $f : A \to^{ec} B$ escribimos $f = (f^e, f^c)$ donde f^e es la inmersión y f^c la clausura.

Si h y k son morfismos $A \to^{ec} B$ se cumplen las siguientes propiedades:

- $h^e \circ h^c$ es idempotente y $\sqsupseteq id$.
- h^e es inyectiva, y h^c es estricta y sobreyectiva.
- h^e refleja el $\bot$.
- $h^e \sqsubseteq k^e$ si y solo si $h^c \sqsupseteq k^c$; h^e está determinada únicamente por h^c y viceversa.

La categoría de dominios y pares inmersión–clausura nos permitirá definir en la Sección 5.5.3 una semántica para un lenguaje polimórfico que permite expresar relaciones entre los valores semánticos de los distintos ejemplares de una función polimórfica.

2.4.2 Funtores en $(C^{ec})^n$

Algunos ejemplos de funtores son:

1.

$$\begin{array}{lcl} \times : C^{ec} \times C^{ec} \to C^{ec} & & \\ \times(A, B) & = & A \times B \\ \times(f, g) & = & (f^e \times g^e, f^c \times g^c) \end{array}$$

donde, si $f : A \to B$ y $g : C \to D$, entonces $f \times g : A \times C \to B \times D$ se define:

$$(f \times g)(a, b) = (f(a), g(b))$$

2.

$$\begin{array}{lcl} \to: C^{ec} \times C^{ec} \to C^{ec} & & \\ \to (A, B) & = & A \to B \\ \to (f, g) & = & (\lambda h. g^e \circ h \circ f^c, \lambda h'. g^c \circ h' \circ f^e) \end{array}$$

$$\begin{tikzcd} A \arrow[dd, "f"] & B \arrow[dd, "g"] & A \arrow[dd, "h"] & A' \arrow[dd, "h'"] & A' \arrow[r, "f^c"] \arrow[rrr, bend right, "g^e \circ h \circ f^c"'] & A \arrow[r, "h"] & B \arrow[r, "g^e"] & B' \\ & & & & & & & \\ A' & B' & B & B' & A \arrow[r, "f^e"] \arrow[rrr, bend right, "g^c \circ h' \circ f^e"'] & A' \arrow[r, "h'"] & B' \arrow[r, "g^c"] & B \end{tikzcd}$$

Para cualquier funtor F sobre $(C^{ec})^n$ hay dos funtores: $F^e : (C^{ec})^n \to C^{ec}$ y $F^c : C^{ec} \to (C^{ec})^n$ que actúan igual que F sobre los objetos, pero que sobre un morfismo h seleccionan la componente e ó c de $F(h)$ respectivamente.

Dados dos funtores F y G se pueden definir los funtores:

1.

$$
\begin{array}{rcl}
(F \times G)(A) & = & F(A) \times G(A) \\
(F \times G)(h) & = & ((F^e \times G^e)(h), (F^c \times G^c)(h))
\end{array}
$$

$$
\begin{tikzcd}
A \arrow[d, "h"] & & F(A) \times G(A) \arrow[d, shift left, "F^e \times G^e(h)"] \\
B & & F(B) \times G(B) \arrow[u, shift left, "F^c \times G^c(h)"]
\end{tikzcd}
$$

2.

$$
\begin{array}{rcl}
(F \to G)(A) & = & F(A) \to G(A) \\
(F \to G)(h) & = & (\lambda k.G^e(h) \circ k \circ F^c(h), \lambda l.G^c(h) \circ l \circ F^e(h))
\end{array}
$$

$$
\begin{tikzcd}
A \arrow[d, "h"] & F(A) \arrow[d, "k"] & F(B) \arrow[d, "l"] & G(A) \arrow[d, shift right, "G^e(h)"'] & F(A) \arrow[d, shift right, "F^e(h)"'] \\
B & G(A) & G(B) & G(B) \arrow[u, shift right, "G^c(h)"'] & F(B) \arrow[u, shift right, "F^c(h)"']
\end{tikzcd}
$$

$$
\begin{tikzcd}
F(A) \arrow[r, "F^e(h)"] \arrow[rrr, bend right, "G^c(h) \circ l \circ F^e(h)"'] & F(B) \arrow[r, "l"] & G(B) \arrow[r, "G^c(h)"] & G(A)
\end{tikzcd}
\qquad
\begin{tikzcd}
F(B) \arrow[r, "F^c(h)"] \arrow[rrr, bend right, "G^e(h) \circ k \circ F^c(h)"'] & F(A) \arrow[r, "k"] & G(A) \arrow[r, "G^e(h)"] & G(B)
\end{tikzcd}
$$

Como vemos los funtores se pueden usar para modelizar los constructores de datos. En la Sección 5.5.3 proporcionaremos un modelo categórico, en concreto C^{ec}, al λ–cálculo con tipos, en el que los constructores de tipos se modelizan mediantes los funtores de dicha categoría.

2.4.3 Transformaciones naturales laxas

Sean dos funtores F y G sobre C^{ec}, y sea una familia $F = \{f_A\}_{A \in Obj(C^{ec})}$ de funciones continuas indexadas por dominios de C^{ec}, y $f_A : F(A) \to G(A)$. Decimos que f es una **transformación natural laxa** de F en G, si para cualquier morfismo $h : A \to^{ec} B$ se cumple:

$$G^c(h) \circ f_B \sqsubseteq f_A \circ F^c(h)$$

es decir, el siguiente diagrama es conmutativo:

$$\begin{tikzcd}
A \arrow[dd, "h"'] & F(A) \arrow[r, "f_A"] & G(A) \\
& & \\
B & F(B) \arrow[uu, "F^c(h)"] \arrow[r, "f_B"'] \arrow[uur, phantom, "\sqsupseteq"] & G(B) \arrow[uu, "G^c(h)"']
\end{tikzcd}$$

(o equivalentemente si $f_B \circ F^e(h) \sqsubseteq G^e(h) \circ f_A$).

Llamamos a f_A componente o ejemplar de la transformación natural laxa.

Es importante observar que cada f_A es una función continua, no un par inmersión–clausura. Las transformaciones naturales laxas son importantes a la hora de extender la técnica de la interpretación abstracta para tratar con el polimorfismo, ver Sección 5.5.3.

2.4.4 Retículos finitos y pares inmersión–clausura

Estudiamos el caso particular en que tratamos con retículos finitos, puesto que como veremos más adelante, la interpretación abstracta trata con retículos finitos (ver el ejemplo de la Sección 5.5.3).

La categoría A^{ec} de retículos finitos y pares inmersión–clausura es una subcategoría de C^{ec}.

Definición 3 *Sea A un retículo finito y $a \in A$ distinto de $\top_A$. Sean las funciones $h_a^e : 2 \to A$ y $h_a^c : A \to 2$ definidas por:*

$$\begin{aligned}
h_a^e(x) &= \begin{cases} a & si\ x = 0 \\ \top_A & si\ x = 1 \end{cases} \\
h_a^c(x) &= \begin{cases} 0 & si\ x \sqsubseteq a \\ 1 & e.o.c. \end{cases}
\end{aligned}$$

Estas funciones forman un par inmersión–clausura $h = (h_a^e, h_a^c) : 2 \to^{ec} A$. Además se cumple que cualquier morfismo en $2 \to^{ec} A$ es de esta forma. Estos pares inmersión–clausura se utilizarán en la Sección 5.5.3 para obtener una aproximación de la semántica de cualquier ejemplar de una función polimórfica en función de la semántica del ejemplar más pequeño en el caso particular de retículos finitos. Además se usarán en el Ejemplo 43.

Un caso particular de estos pares inmersión–clausura se usará también en la Sección 5.5.4 para el cálculo de una aproximación del punto fijo de una función.

Teorema 4 *Si f es una transformación natural laxa del funtor F al funtor G, y A es un retículo finito, entonces tenemos:*

$$f_A \sqsubseteq \sqcap_a G^e(h_a) \circ f_2 \circ F^c(h_a)$$

donde a recorre todos los elementos distintos de $\top_A$ de A.

Lo que intuitivamente significa que a partir de f_2 podemos obtener una aproximación a f_A. Este resultado se usará en la Sección 5.5.3 en el Ejemplo 43. Para n parámetros:

$$f_{A_1 \ldots A_n} \sqsubseteq \sqcap G^e(h_1, \ldots, h_n) \circ f_{2 \ldots 2} \circ F^c(h_1, \ldots, h_n)$$

donde $\sqcap$ se toma sobre todas las tuplas posibles de n morfismos $(h_1, \ldots, h_n)$ con $h_i : 2 \to^{ec} A_i$ para cada i.

Funtores de primer orden

Los funtores de primer orden son funtores que no incluyen el funtor de espacio de funciones $\to$.

En este caso la desigualdad del Teorema 4 se convierte en una igualdad si f es una transformación natural de F a G.

Además se puede reducir el número de funciones implicadas en el cálculo de la máxima cota inferior.

Proposición 5 *Sean F y G funtores de primer orden. Sea A un retículo finito con $a, b \in A$ y sea f una transformación natural laxa de F a G Entonces:*

$$G^e(h_a) \circ f_{\mathbf{2}} \circ F^c(h_a) \sqcap G^e(h_b) \circ f_{\mathbf{2}} \circ F^c(h_b) \sqsubseteq G^e(h_{a \sqcap b}) \circ f_{\mathbf{2}} \circ F^c(h_{a \sqcap b})$$

Este teorema tiene como consecuencia el que podamos calcular $\sqcap_a G^e(h_a) \circ f_2 \circ F^c(h_a)$ utilizando solamente algunos de los elementos a. Si hay dos elementos a y b en A tales $a \sqcap b = c$ es distinto de a y de b, entonces podemos eliminar del cálculo de $\sqcap$ el operando $G^e(h_c) \circ f_2 \circ F^c(h_c)$, puesto que sabemos que es mayor, según la proposición anterior. Por tanto, del cálculo de $\sqcap$ podemos eliminar aquellos elementos que resultan del ínfimo de dos elementos distintos de ellos. A un elemento que no resulta del ínfimo de dos elementos distintos de él le llamamo $\sqcap$–irreducibles. Esta idea la recoge el siguiente corolario.

Corolario 6 *Sean F, G, f y A como en el teorema anterior. Entonces:*

$$\sqcap_a G^e(h_a) \circ f_2 \circ F^c(h_a) = \sqcap_t G^e(h_t) \circ f_2 \circ F^c(h_t)$$

donde t está restringido a los elementos de A que son $\sqcap$–irreducibles.

2.4.5 Pares inmersión–proyección

Coquand et al [CGW89] proporcionaron una semántica a las funciones polimórficas utilizando la categoría de los dominios y los pares inmersión–proyección. En la Sección 5.5.3 optaremos, sin embargo, por usar C^{ec} porque en C^{ep} obtenemos aproximaciones a f_A por arriba y no por abajo, como veremos es deseable. La **categoría de dominios y pares inmersión–proyección**, C^{ep} es la categoría cuyos objetos son dominios y sus morfismos son de la forma $(e, p) : A \to B$ donde $e : A \to B$ y $p : B \to A$ son funciones continuas que cumplen:

$$p \circ e = id \text{ y } e \circ p \sqsubseteq id$$

Llamamos inmersión a e y proyección a p.

La composición de pares inmersión–proyección se define de la siguiente manera:

$$(e_2, p_2) \circ (e_1, p_1) = (e_2 \circ e_1, p_1 \circ p_2)$$

Para cualquier morfismo escribimos $f = (f^e, f^p)$ donde f^e es la inmersión y f^p es la proyección.

Si k y l son morfismos $A \to^{ep} B$ se cumplen las siguientes propiedades:

- $k^e \circ k^p$ es una proyección sobre B (es decir, es idempotente y $\sqsubseteq id$).
- k^e es inyectiva y k^p es sobreyectiva.
- Ambas k^e y k^p son estrictas.
- k^e refleja el $\perp$.
- $k^e \sqsupseteq l^e$ si y solo si $k^p \sqsubseteq l^p$, y por tanto k^e está determinada únicamente por k^p y viceversa.

2.4.6 Funtores sobre $(C^{ep})^n$ y transformaciones naturales laxas

De la misma forma que para $(C^{ec})^n$, los constructores de tipos con n variables de tipo se pueden modelizar mediante funtores de $(C^{ep})^n$ en C^{ep}. La definición de los funtores $\times$ y $\to$ se pueden dar de forma similar; basta con sustituir la c por p cuando ocurre como superíndice.

Sean dos funtores F y G, y sea una familia $\{f_A\}$ de funciones continuas indexadas por dominios, y $f_A : F(A) \to G(A)$. Decimos que f es una **transformación natural laxa** de F a G, si para cualquier morfismo $k : A \to^{ep} B$ tenemos:

$$G^p(k) \circ f_B \sqsupseteq f_A \circ F^p(k)$$

es decir, el siguiente diagrama es conmutativo:

$$\begin{tikzcd}
A \arrow[d, "k"'] & F(A) \arrow[r, "f_A"] & G(A) \\
B & F(B) \arrow[u, "F^p(h)"] \arrow[r, "f_B"'] \arrow[ur, phantom, "\sqsubseteq"] & G(B) \arrow[u, "G^p(h)"']
\end{tikzcd}$$

(o equivalentemente si $f_B \circ F^e(k) \sqsupseteq G^e(k) \circ f_A$).

2.4.7 Retículos finitos y pares inmersión–proyección

La categoría A^{ep} de retículos finitos y pares inmersión–proyección es una subcategoría de C^{ep}.

Definición 7 *Sea A un retículo finito y $a \in A$ distinto de $\top_A$. Sean las funciones $k_a^e : 2 \to A$ y $k_a^p : A \to 2$ definidas por:*

$$\begin{aligned} k_a^e(x) &= \begin{cases} a & si\ x = 1 \\ \perp_A & si\ x = 0 \end{cases} \\ k_a^p(x) &= \begin{cases} 1 & si\ x \sqsupseteq a \\ 0 & e.o.c. \end{cases} \end{aligned}$$

Estas funciones forman un par inmersión–proyección $k = (k_a^e, k_a^p) : 2 \to^{ec} A$. Además se cumple que cualquier morfismo en $2 \to^{ep} A$ es de esta forma. Un caso particular de estos pares inmersión–proyección se usarán en la Sección 5.5.4 para calcular una aproximación al punto fijo de una función. Aunque aquí no se presentan, se cumplen teoremas similares a los de C^{ec}, ver [Bar93].

Capítulo 3

Historia del análisis de programas funcionales

La interpretación abstracta se aplicó por primera vez a los lenguajes funcionales por parte de **Mycroft** [Myc81]. En 1980 publicó un análisis de estrictez para programas de primer orden con tipos de datos monomórficos y planos (ver Sección 5.2).

En 1985 **Burn, Hankin y Abramsky** [BHA86] presentaron una forma de extender la interpretación abstracta a programas de orden superior, todavía monomórficos, y presentaron un análisis de estrictez de orden superior. Posteriormente, Burn extendió estas ideas a cualquier análisis [Bur91] (ver Sección 5.3).

Esta fue una de las primeras aproximaciones al orden superior, que abstrae las funciones como **funciones sobre valores abstractos**. Existen otras aproximaciones para extender los análisis a orden superior como el análisis de clausuras de Sestoft [Ses88] (ver Sección 7.2).

Es un método basado en semántica operacional. Produce información del estilo: una expresión e se puede evaluar a una f–clausura con un argumento evaluado o a una g–clausura con dos argumentos evaluados. Cada función se describe mediante una tabla global de descripción de funciones que contiene descripciones de todos los argumentos con los que puede ser llamada. Una aproximación a un valor se representa mediante una clausura aproximada.

El problema es que en una expresión de la forma $e_1 e_2$ no sabemos qué función se va a aplicar a e_2 realmente. Y por tanto no sabemos qué valor abstracto hacerle corresponder a esta expresión. Sestoft usa un análisis auxiliar para encontrar una aproximación al conjunto de funciones a las que se puede evaluar la expresión e_1. A continuación, dispone de una función que asigna a cada función un valor abstracto. Finalmente solamente hay que combinar el valor abstracto de e_1 obtenido de esta manera al de e_2.

Un análisis de clausuras semejante fue desarrollado en paralelo por Shivers [Shi88] para el lenguaje Scheme.

Esta técnica se utiliza en la implementación eficiente de la evaluación perezosa [SA89] y la evaluación parcial [Bon90].

En 1986 **Hudak y Young** [HY86] propusieron una forma alternativa de interpretación abstracta para orden superior, que en este trabajo no se describe con detalle. Recibe el nombre de análisis de pares de estrictez. En él, el valor abstracto de una expresión e que se evalua a una función es un par (v, f), donde v representa la estrictez de la expresión en sí (estrictez directa) y f representa la estrictez de la función a la que se evalúa (estrictez retardada). Es un análisis más preciso, pero muy costoso (exponencial). No trata explícitamente las estructuras de datos, aunque se podrían codificar como funciones con un coste excesivo. Además no existe una seguridad de terminación del análisis en lenguajes no tipados. No se han encontrado condiciones necesarias pero sí algunas suficientes para asegurar la terminación.

En 1985 **Abramsky** [Abr86] encontró la forma de aplicar la interpretación abstracta a programas polimórficos. Introduce la idea de **invarianza polimórfica**, la cual comprueba el hecho de que el análisis en cuestión dé los mismos resultados para cada ejemplar de una función polimórfica.

Posteriormente, en 1989 Hughes [Hug89] encontró una forma de aproximar la interpretación abstracta de cualquier ejemplar de una función polimórfica de primer orden a partir de la de su ejemplar más pequeño, lo cual evitaba recalcular completamente la interpretación abstracta de los ejemplares no mínimos usados en distintos puntos del programa.

Sus ideas fueron extendidas a orden superior y a funciones sobre listas en 1993 por Baraki [Bar93] (Sección 5.5.3).

En 1987 **Wadler** [Wad87] resolvió el problema de analizar estructuras de datos (es decir, tipos de datos no planos). Encontró dominios abstractos finitos para listas mediante la aproximación de cada elemento por el mismo valor abstracto (Sección 5.4.2).

Mientras, **Clack y Peyton Jones** [CP85], en 1985, se enfrentaron a la ineficiencia de la interpretación abstracta. Utilizando fronteras para representar funciones, intentaron mejorar al coste en los casos más comunes (Sección 5.2.5).

Esta idea fue extendida a orden superior por Martin y Hankin [MH87] (Sección 5.3.3). Y posteriormente Hankin y Hunt [HH92] renunciaron a la exactitud y presentaron una forma de aproximar puntos fijos. La idea consiste en trasladar el cálculo del punto fijo en dominios grandes a otros más pequeños (Sección 5.5.4) .

En programas funcionales, el **análisis hacia atrás** está descrito por el concepto de **contexto** de un valor, el cual describe la forma en que el valor va a ser usado en el resto del cómputo. Esta información de contexto se propaga hacia atrás: del contexto de una expresión a los de sus subexpresiones. Es decir la información se propaga desde la raíz hacia las hojas del árbol abstracto.

El concepto de contexto fue formalizado inicialmente mediante **continuaciones** [Hug87], el cual no se trata en este trabajo, y posteriormente mediante la noción de **proyección** [WH87], la cual especifica qué partes de un valor se usarán posteriormente (Capítulo 6).

Los análisis hacia atrás se han usado en:

- **Análisis de estrictez** [Hug86] [Hug87] [WH87] [Joh81] [Wra85] [HW87]

- **Recuperación de espacio**: Jensen y Mogensen [JM90] desarrollaron un análisis para recogida de basura en tiempo de compilación. Se aplicó en un lenguaje de primer orden con estructuras de datos y posteriormente se extendió a orden superior usando análisis de clausuras. Cuando se sabe que una celda no se va a usar nunca más se puede reutilizar.

- **Evaluación parcial**: transforma programas generales en versiones especializadas a entradas parcialmente conocidas

En 1981 Jonhsson [Joh81] desarrolló una analizador de estrictez hacia atrás. En 1985 Wray [Wra85] implementó un analizador de estrictez–ausencia como parte del compilador de Ponder. Destaca por su velocidad y por su capacidad de analizar funciones de segundo orden.

En el mismo año, Hughes [Hug86] publicó un analizador de estrictez capaz de analizar estructuras de datos. Al utilizar dominios finitos se vió obligado a resolver sistemas de ecuaciones de contextos simbólicamente.

Dybjer [Dyb87], en 1987, proporcionó una fundación teórica del análisis hacia atrás basada en imágenes inversas de conjuntos abiertos, que en este trabajo no se desarrolla.

Posteriormente Wadler y Hughes [WH87] modelizaron los contextos mediante proyecciones, aplicando las ideas previas de Wadler para encontrar dominios finitos de contextos para estructuras de datos (Sección 6.2). Kei Davis y Wadler [DW90] mejoraron este análisis haciéndolo más exacto (Sección 6.4).

Hughes y Launchbury [HL92] demostraron que las interpretaciones abstractas para las que existe un análisis hacia atras son aquellas para las que las funciones abstractas (las abstracciones de funciones concretas) son conexiones de Galois.

Paralelamente, Hall y Wise [HW87] desarrollaron un analizador de estrictez hacia atrás que generaba versiones múltiples de cada función, una para cada llamada en un contexto distinto.

Hughes extendió el análisis hacia atrás [Hug88] para funciones de cualquier orden generalizando las ideas de Wray. Juntos desarrollaron un análisis para estimar el número de veces que se evaluará una expresión bajo llamada por nombre. Incluía orden superior y estructuras de datos. Estaba basado en el entorno del análisis hacia atrás desarrollado por Hughes.

El análisis de estrictez no es la única aplicación de la interpretación abstracta en lenguajes funcionales durante la década de los 80.

Hudak y Bloss desarrollaron un análisis para evitar copias innecesarias de arrays funcionales [HB85] [Hud86] y un análisis de caminos (como refinamiento del análisis de estrictez) que proporcionaba información sobre el orden entre eventos [BH86] [BH88] respondiendo a las preguntas qué se evaluará y no se evaluará con seguridad tras la evaluación de una exprersión e, y qué se evaluó y no se evaluó con seguridad antes de la evaluación de una expresión e.

Goldberg desarrolló un análisis de compartición para un lenguaje de orden superior sin tipos con estructuras de datos para optimizar supercombinadores [Gol87]. Utilizó para ello pares de estrictez.

En lenguajes estrictos el análisis de globalización fue tratado por Schmidt [Sch85] y Sestoft [Ses89] (Sección 7.2).

Ya hemos visto que la forma más común de aproximar una función es mediante una función sobre valores abstractos. Pero esta aproximación no es siempre suficiente para los análisis de programas usados en la práctica. Un ejemplo es el análisis de **propagación de constantes** en un lenguaje funcional. El objetivo es determinar si un argumento de una función de un programa se llama siempre con el mismo valor. En tal caso el compilador puede explotar esta información para generar código optimizado.

Para este análisis, no es suficiente con conocer cómo está definida la función, sino que es esencial saber con qué valores puede ser llamada la función durante la ejecución del programa.

La solución propuesta por Jones y Mycroft [JM86] es la de los **grafos mínimos de función**. Un grafo mínimo sobre unos datos de entrada dados se define como el conjunto más pequeño de pares (argumento, valor de función) suficientes para llevar a cabo la ejecución del programa.

Ejemplo 8

$$f(x) = if\ (x = 1)\ 1\ (if\ even(x)\ f(x/2)\ f(3 * x + 1))$$

Su grafo mínimo para el dato de entrada 3 es:

$$\{(3,1),(10,1),(5,1),(16,1),(8,1),(4,1),(2,1),(1,1)\}$$

□

La semántica de grafos mínimos asocia a cada función definida por el programador una información más detallada que la tradicional función argumento–resultado utilizada en semántica estándar. Es un ejemplo de semántica instrumentada.

Jones y Mycroft [JM86] muestran cómo el análisis de propagación de constantes se puede hacer mediante aproximación de esta semántica de grafos mínimos.

Hasta este momento, los análisis basados en interpretación abstracta representaban las propiedades de los valores mediante conjuntos Scott–cerrados. Algunas propiedades, sin embargo, como la propiedad de ser una función constante, no se pueden representar mediante conjuntos Scott–cerrados. Hunt [HS91] [Hun91] [Hun91] desarrolló una interpretación abstracta en la que las propiedades se representaban mediante relaciones de equivalencia parcial (Sección 7.1).

Posteriormente los esfuerzos se centraron en la eficiencia de los análisis. Los análisis basados en tipos pretenden ser una salida de la misma potencia que la interpretación abstracta pero de mayor eficiencia.

El primer uso explícito de los tipos en el análisis de programas lo llevaron a cabo Kuo y Mishra [KM89]. Su desventaja era que el análisis era más débil, es decir, proporcionaba información menos útil que la interpretación abstracta. Posteriormente Jensen [Jen91] [Jen92] incorporó la conjunción de tipos y demostró la equivalencia entre los análisis basados en inferencia de tipos y la interpretación abstracta con respecto al análisis de estrictez. También Benton [Ben92] [Ben93] trabajó en la recuperación de la potencia de la interpretación abstracta. Ambos se centraron en el análisis de estrictez. En este proceso olvidaron la conexión con el aspecto algorítmico y la eficiencia. Burn [Bur92] generalizó su enfoque, pero tampoco se fijó en cuestiones algorítmicas. Más recientemente Hankin y Le Métayer [HL94b] [HL94a] [HM94] han introducido una nueva idea, los tipos perezosos, que permite mantener la potencia de la interpretación abstracta y a la vez desarrollar algoritmos eficientes (Capítulo 8).

Capítulo 4

Conceptos generales de interpretación abstracta

La semántica de los programas describe el conjunto de todos los posibles comportamientos del programa cuando se ejecuta para todas las entradas posibles. La interpretación abstracta es una teoría de aproximación de la semántica de los programas usada para la construcción de algoritmos de análisis de programas basados en la semántica, la comparación de semánticas formales (por ejemplo, para la construcción de una semántica denotacional a partir de una operacional), diseño de métodos de prueba etc. Los analizadores automáticos de programas se utilizan para determinar estáticamente aproximaciones de propiedades dinámicas de los programas. Estas propiedades pueden ser útiles para llevar a cabo transformaciones en los programas (ej. paralelización, evaluacion parcial), optimizaciones del código (ej. recolección de basura en tiempo de compilación, eliminación del occur–check innecesario), depuración (ej. inferencia de tipos) e incluso pruebas de correción (ej. pruebas de terminación).

Normalmente, la interpretación abstracta se entiende en el sentido de **pseudo–evaluación**, es decir, la ejecución no estándar del programa sobre valores abstractos, en lugar de valores reales, para obtener información sobre todas las posibles ejecuciones del programa sobre los valores reales, a partir del resultado abstracto obtenido. Los valores abstractos representan aproximaciones de aquellas propiedades de los valores reales que nos interesan en el análisis concreto que estemos realizando.

4.1 Introducción a la interpretación abstracta

La semántica S de un lenguaje de programación asocia a cada programa del lenguaje p un valor $S[\![p]\!] \in D$ en un dominio semántico D. Estos dominios semánticos D pueden ser sistemas de transiciones (para semántica operacional de cómputo), pomsets, trazas, relaciones (para semántica operacional de evaluación o natural), funciones de orden superior (para semántica denotacional) etc.

Normalmente D está definido composicionalmente por inducción sobre la estructura de los objetos en tiempo de ejecución (cómputos, datos etc.) y S se define composicionalmente por inducción sobre la estructura sintáctica de los programas, usando, por ejemplo, puntos fijos para manejar recursión.

El marco clásico de la interpretación abstracta, introducido por Cousot, [CC77] [Cou81] [CC92b] [CC92a] parte de una semántica estándar del lenguaje de programación, dada en forma de semántica operacional o denotacional. Esta semántica estándar describe los posibles comportamientos de los programas durante su ejecución.

A continuación, se diseña una **semántica estática** o de recolección de propiedades, que se centra en una clase de propiedades de la ejecución de los programas. Esta semántica estática, puede ser una versión instrumentada de la semántica estándar (una ampliación de la semántica denotacional con información adicional de tipo operacional, por ejemplo), o bien una versión reducida a lo esencial de la misma, con el objetivo de ignorar los detalles irrelevantes sobre la ejecución de los programas. Normalmente se describe usando puntos fijos sobre estructuras ordenadas. Esta semántica es la más precisa de las semánticas que se pueden definir para describir una cierta clase de propiedades de programa, sin referirse a otras propiedades fuera de nuestro interés. Es decir, proporciona un método de prueba correcto y relativamente completo de la clase de propiedades que queremos estudiar. Por ello es una referencia semántica para demostrar la corrección de todas las demás semánticas aproximadas definidas para esa clase de propiedades.

La interpretación abstracta consiste en definir semánticas que aproximan la semántica de recolección. Considera propiedades efectivamente computables de los programas. Dichas semánticas deben ser correctas respecto a la semántica de recolección en función de una determinada relación de correción.

Y vienen determinadas por el tipo de propiedades que deseamos considerar y por la correspondencia entre las propiedades concretas y las abstractas. Dicha correspondencia viene dada de formas distintas, pero la más usual es la utilización de **conexiones de Galois**. La razón de ello es que entonces se asegura la existencia de una aproximación mejor a las propiedades concretas de entre todas las abstractas que son correctas.

4.1.1 Un ejemplo sencillo de interpretación abstracta: la regla de los signos

Una interpretación abstracta se puede entender como una semántica no estándar en la que el dominio de valores se reemplaza por un dominio de descripciones de valores, y en la que a cada operador se le da una interpretación no estándar.

Por ejemplo, en lugar de usar enteros podríamos usar los valores abstractos -1 y $+1$ para describir los enteros negativos y positivos respectivamente. A continuación, reinterpretando los operadores de suma y producto de acuerdo con la regla de los signos, la interpretación abstracta puede determinar ciertas propiedades de programas como por ejemplo: "al entrar en este bucle la variable x tiene un valor positivo".

La regla de los signos nos dice que:

$$\begin{array}{rcl} -1 +^{\#} -1 & = & -1 \\ +1 +^{\#} +1 & = & +1 \end{array} \qquad \begin{array}{rcl} +1 \times^{\#} +1 & = & +1 \\ +1 \times^{\#} -1 & = & -1 \\ -1 \times^{\#} +1 & = & -1 \\ -1 \times^{\#} -1 & = & +1 \end{array}$$

donde $+^{\#}$ y $\times^{\#}$ representan en estas igualdades la abstracción de la suma y el producto de enteros.

Sin embargo, existen ciertos casos no especificados, como $+1 +^{\#} -1$, puesto que el resultado depende de los valores numéricos de los datos. Para evitar esto, se introduce un valor abstracto $\top$, que representa el hecho de que no sabemos nada acerca del resultado:

$$\begin{array}{ccccc}
+1 & +^{\#} & -1 & = & \top \\
-1 & +^{\#} & +1 & = & \top \\
\top & +^{\#} & +1 & = & \top \\
\top & +^{\#} & -1 & = & \top \\
+1 & +^{\#} & \top & = & \top \\
-1 & +^{\#} & \top & = & -1
\end{array}
\qquad
\begin{array}{ccccc}
\top & \times^{\#} & +1 & = & \top \\
\top & \times^{\#} & -1 & = & \top \\
+1 & \times^{\#} & \top & = & \top \\
-1 & \times^{\#} & \top & = & \top \\
\top & \times^{\#} & \top & = & \top
\end{array}$$

Se pueden usar varios valores abstractos para aproximar un mismo valor concreto. Por ejemplo el número **6** se puede aproximar por $+1$ y por $\top$. Sin embargo, de $+1$ podemos obtener conclusiones más precisas que de $\top$, puesto que proporciona menos información. A partir de $+1$ podemos concluir que el valor es positivo o cero, mientras que a partir de $\top$ no podemos concluir nada acerca del valor al que aproxima, puesto que contiene toda la información posible.

Para representar el grado de aproximación de los distintos valores abstractos se introduce un orden entre ellos $\leq$. Por ejemplo, $-1 \leq \top$ y $+1 \leq \top$, siendo $+1$ y -1 no comparables.

Existen aun así, valores que podrían ser aproximados por distintos valores abstractos mínimos. En este caso tenemos el 0, que puede ser aproximado tanto por $+1$ como por -1. En estos casos, o bien se hace una elección dependiendo de la expresión a analizar, o bien se introduce de forma explícita enriqueciendo el dominio abstracto.

$$\begin{array}{ccccc}
0 & +^{\#} & +1 & = & +1 \\
0 & +^{\#} & -1 & = & -1 \\
0 & +^{\#} & \top & = & \top \\
0 & +^{\#} & 0 & = & 0 \\
+1 & +^{\#} & 0 & = & +1 \\
-1 & +^{\#} & 0 & = & -1 \\
\top & +^{\#} & 0 & = & \top
\end{array}
\qquad
\begin{array}{ccccc}
0 & \times^{\#} & +1 & = & 0 \\
0 & \times^{\#} & -1 & = & 0 \\
0 & \times^{\#} & \top & = & 0 \\
0 & \times^{\#} & 0 & = & 0 \\
+1 & \times^{\#} & 0 & = & 0 \\
-1 & \times^{\#} & 0 & = & 0 \\
\top & \times^{\#} & 0 & = & 0
\end{array}$$

4.2 Aproximación de las propiedades de los programas

Como hemos dicho anteriormente, se define una semántica de recolección, que representa las propiedades concretas de los programas que queremos estudiar y, a continuación otra abstracta que pretende aproximar dichas propiedades concretas mediante propiedades abstractas .

En general, las **propiedades concretas** de los programas se describen mediante elementos de un conjunto P^b llamado dominio de las propiedades concretas.

La **semántica concreta** asocia con cada programa un elemento del dominio de propiedades concretas P^b. Esta semántica concreta suele estar definida como punto fijo de una función F^b sobre P^b, llamada función de semántica concreta.

Lo más usual es que el dominio de las propiedades concretas sea un orden parcial $P^b(\leq^b)$, donde la relación de orden representa la precisión relativa de las propiedades concretas: $p_1^b \leq^b p_2^b$ significa que p_1^b y p_2^b son propiedades comparables siendo p_1^b más precisa que p_2^b.

Si se trata de un orden parcial completo y F^b es monótona entonces se asegura la existencia de dicho punto fijo (aunque la recursión puede ser transfinita). Si además es continua se puede calcular como $\sqcup_{n\geq 0} F^{b^n}(\bot)$. Si se trata de un retículo finito y F^b es monótona también se puede calcular de esta forma.

El primer paso que debe darse en una interpretación abstracta es el diseño de un dominio de **propiedades abstractas** $P^{\#}$, que pretende ser una versión aproximada del dominio de propiedades concretas P^b.

De forma análoga, las propiedades abstractas se suelen representar mediante elementos de un orden parcial $P^{\#}(\sqsubseteq^{\#})$ donde, de nuevo, el orden $\sqsubseteq^{\#}$ representa la precisión relativa de las propiedades abstractas.

Ejemplo 9 (regla de los signos):

Para el ejemplo anterior de los signos, podemos elegir como dominio de propiedades concretas $P^b = \{false, <0, =0, >0, \leq 0, \neq 0, \geq 0, true\}$, donde cada uno de los elementos representa un conjunto de valores que puede tomar una variable. Así,

$$\begin{array}{rcl} false & = & \emptyset \\ <0 & = & \{x \in \mathbb{Z} \mid x < 0\} \\ \leq 0 & = & \{x \in \mathbb{Z} \mid x \leq 0\} \\ =0 & = & \{0\} \\ \neq 0 & = & \{x \in \mathbb{Z} \mid x \neq 0\} \\ >0 & = & \{x \in \mathbb{Z} \mid x > 0\} \\ \geq & = & \{x \in \mathbb{Z} \mid x \geq 0\} \\ true & = & \mathbb{Z} \end{array}$$

El orden entre los elementos (ver Figura 4.1) viene determinado por tanto por el $\subseteq$ entre los conjuntos de enteros que representan. En cuanto a las propiedades abstractas, una posible elección sería $P^{\#} = \{f^{\#}, -1, 0, +1, t^{\#}\}$, ordenados sus elementos según se muestra en la Figura 4.1.

□

El segundo paso es la definición de una **semántica abstracta**. Su objetivo es encontrar para cada programa una propiedad abstracta a en el dominio de propiedades abstractas $P^{\#}$, que sea una aproximación correcta (ver siguiente sección) de la semántica concreta $c \in P^b$ del programa. Igual que en la semántica concreta, la semántica abstracta se define como punto fijo de una función $F^{\#}$ sobre $P^{\#}$.

4.3 Relación de corrección entre la semántica concreta y la semántica abstracta

El tercer paso que debe darse en el diseño de una interpretación abstracta es la especificación de la correspondencia entre entre las propiedades concretas y abstractas. Ésta se puede definir

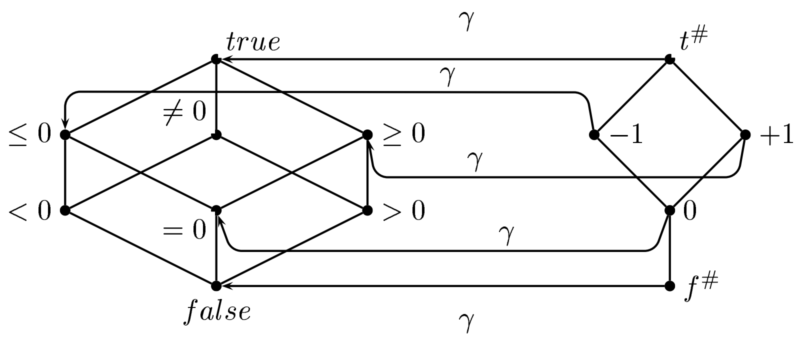


Figura 4.1: Dominios de propiedades concretas y abstractas de los enteros, y función de concreción

mediante una relación de corrección $\sigma \in P(P^b \times P^{\#})$, donde $< c, a > \in \sigma$ significa que la semántica concreta c del programa tiene la propiedad abstracta a.

Una suposición habitual es la **existencia de aproximación abstracta**, es decir, toda propiedad concreta tiene una aproximación abstracta:

$$\forall c \in P^b . \exists a \in P^{\#} . < c, a > \in \sigma$$

Para demostrar la corrección de la semántica abstracta se suele proceder por inducción.

El significado de las propiedades abstractas respecto a las concretas viene dado mediante una **función de concreción** $\gamma \in (P^{\#} \to P^b)$ donde $\gamma(p^{\#})$ es la propiedad concreta que le corresponde a la propiedad abstracta $p^{\#}$.

La noción de aproximación está determinada por una **función de abstracción**[1]

$\alpha \in (P^b \to P^{\#})$, que da la mejor de las aproximaciones abstractas, $\alpha(p^b)$, de la propiedad concreta p^b.

Ejemplo 10 (regla de los signos)

Continuando con el ejemplo anterior, la función de concreción viene representada en el grafo de la Figura 4.1. Y la función de abstracción por la siguiente tabla:

p^b	$false$	< 0	$= 0$	> 0	≤ 0	$\neq 0$	≥ 0	$true$
$\alpha(p^b)$	$f^{\#}$	-1	0	$+1$	-1	$t^{\#}$	$+1$	$t^{\#}$

□

[1] La forma más habitual de proporcionar la corrección de un análisis es a través de una función de abstracción que junto con la función de concreción forma una conexión de Galois. En este capítulo vamos a estudiar esta forma general de proporcionar la corrección. En el siguiente capítulo, estos conceptos se particularizarán para los lenguajes funcionales. También se estudiará una forma alternativa de dar la corrección a través de relaciones lógicas.

La noción de **corrección de la aproximación** se define de la siguiente manera:

$$\forall c \in P^b.\forall a \in P^{\#}.\alpha(c) \leq^{\#} a \Leftrightarrow c \leq^b \gamma(a)$$

.

El hecho de que a es una aproximación válida de la información dada por c se puede expresar mediante $\alpha(c) \leq^{\#} p^{\#}$, es decir $\alpha(c)$ es la menor aproximación de c, y todos los valores abstractos por encima de ella, son también aproximaciones válidas de c.

También se puede expresar mediante $c \leq^b \gamma(a)$, es decir a es aproximación de c, pero también de todas aquellas propiedades concretas de mayor precisión que c.

Definir la corrección exigiendo que el par de funciones abstracción/concreción formen una conexión de Galois asegura que cada propiedad concreta tiene una única mejor aproximación abstracta.

4.3.1 Conexiones de Galois

Definición 11 *Dados dos órdenes parciales $P^b(\leq^b)$ y $P^{\#}(\leq^{\#})$, una **conexión de Galois** es un par de funciones tales que:*

$$\begin{array}{l} \alpha \in (P^b \to P^{\#}) \\ \gamma \in (P^{\#} \to P^b) \\ \forall c \in P^b.\forall a \in P^{\#}.\alpha(c) \leq^{\#} a \Leftrightarrow c \leq^b \gamma(a) \end{array}$$

Cousot lo denota por $P^b(\leq^b) \underset{\gamma}{\overset{\alpha}{\leftrightharpoons}} P^{\#}(\leq^{\#})$.

Las conexiones de Galois tienen varias propiedades, entre ellas la monotonía y la preservación de cotas (α preserva mínimas cotas superiores y γ máximas cotas inferiores).

El hecho de que $\gamma \circ \alpha$ es **extensiva** ($\forall p^b \in P^b.p^b \leq^b \gamma \circ \alpha(p^b)$) se interpreta como el hecho de que la pérdida de información en el proceso de abstracción es segura.

Y el hecho de que $\alpha \circ \gamma$ es **reductiva** ($\forall p^{\#} \in P^{\#}.\alpha \circ \gamma(p^{\#}) \leq^{\#} p^{\#}$) se interpreta como el hecho de que el proceso de concreción no introduce pérdida de información.

Una definición equivalente de una conexión de Galois es la siguiente: α y γ forman una conexión de Galois si son funciones monótonas tales que $\gamma \circ \alpha$ es extensiva y $\alpha \circ \gamma$ es reductiva (también se dice que forman una adjunción). Esta definición alternativa se usará en la Sección 5.3.2 tras la proposición 24.

En una conexión de Galois, una de las funciones determina de forma única a la otra:

$$\forall a \in P^{\#}.\gamma(a) = \sqcup^b\{c|\alpha(c) \leq^{\#} a\}$$

y

$$\forall c \in P^b.\alpha(c) = \sqcap^{\#}\{a|c \leq^b \gamma(a)\}$$

Además, α es sobreyectiva si y solo si γ es inyectiva, en cuyo caso se habla de una **sobreyección de Galois**. Análogamente α es inyectiva si y solo si γ es sobreyectiva, en cuyo caso se habla de una **inyección de Galois**.

4.4 Aproximación del punto fijo

Suponemos que la semántica concreta de un programa viene dada como un punto fijo $lfp\ F^b = \bigsqcup^b_{n\geq 0} F^{b^n}(\perp^b)$ donde $X = F^b(X)$ es la ecuación (o sistema de ecuaciones) asociada al programa, $P^b(\sqsubseteq^b, \perp^b, \sqcup^b)$ es un orden parcial de propiedades concretas de programa y $F^b : P^b(\sqsubseteq^b) \to P^b(\sqsubseteq^b)$ es continua. Suponemos que tal cota superior de los $F^{b^n}(\perp^b)$ existe por ejemplo porque $P^b(\sqsubseteq^b, \perp^b, \sqcup^b)$ es un orden parcial completo apuntado.

Es importante notar que tanto en P^b como en $P^\#$ están involucrados dos órdenes parciales. En general son distintos pero suelen coincidir. $\sqsubseteq^b$ ($\sqsubseteq^\#$) recibe el nombre de **orden de cómputo o computacional**. Es el orden que relaciona las distintas iteraciones $F^{b^n}(\perp^b)$ ($F^{\#^n}(\perp^\#)$) durante el cálculo del punto fijo. Y $\leq^b$ ($\leq^\#$) recibe el nombre de **orden de aproximación**. Especifica la precisión relativa de las propiedades concretas (abstractas).

Examinamos ahora el problema de calcular y aproximar después desde arriba para $\leq^\#$ la semántica abstracta $\alpha\ (lfp\ F^b)$ del programa.

Inducción del punto fijo usando conexiones de Galois

Supongamos que $P^b(\sqsubseteq^b, \sqcup^b)$ y $P^\#(\sqsubseteq^\#, \sqcup^\#)$ son ordenes parciales, que $F^b : P^b \to P^b$ proporciona la semántica concreta $lfp\ F^b$ de un programa, y que estamos interesados en su abstracción $\alpha\ (lfp\ F^b)$, donde $P^b(\sqsubseteq^b) \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} P^\#(\sqsubseteq^\#)$.

Si suponemos que el punto fijo viene dado por $lfp\ F^b = \bigsqcup^b_{n\geq 0} F^{b^n}(\perp^b)$ (por tratarse por ejemplo de una función continua sobre un cpo), es natural intentar calcular $\alpha\ (lfp\ F^b)$ mediante la abstracción de dicha cadena de iteraciones.

Puesto que queremos hacer los cálculos en el dominio abstracto $P^\#$, nos gustaría obtener esta secuencia usando un ínfimo abstracto $\perp^\#$, un operador abstracto $F^\#$ y una mínima cota superior abstracta $\sqcup^\#$ sobre $P^\#$, de la forma $\bigsqcup^\#_{n\geq 0} F^{\#^n}(\perp^\#)$.

Esto es posible si $\forall n. F^{\#^n}(\perp^\#) = \alpha(F^{b^n}(\perp^b))$. Para encontrar hipótesis que impliquen la propiedad deseada podemos razonar por inducción sobre n:

- Para $n = 0$ hay que demostrar que $\alpha(\perp^b) = \perp^\#$.

- Para $n \geq 0$ hay que demostrar que si $\alpha(F^{b^n}(\perp^b)) = F^{\#^n}(\perp^\#)$ entonces $\alpha(F^{b^{n+1}}(\perp^b)) = F^{\#^{n+1}}(\perp^\#)$, lo que usando la hipótesis de inducción se reduce a demostrar que

$$\forall p^b \in P^b.\ \alpha(F^b(p^b)) = F^\#(\alpha(p^b))$$

(ó $F^\# = \alpha \circ F^b \circ \gamma$ y $\forall p^b \in P^b. \gamma \circ \alpha(p^b) = p^b$).

Además si p^b es un punto fijo de F^b entonces, puesto que $\alpha \circ F^b = F^\# \circ \alpha$ entonces $\alpha(p^b)$ es un punto fijo de $F^\#$ y en particular $\alpha(\bigsqcup^b_{n\geq 0} F^{b^n}(\perp^b)) = \bigsqcup^\#_{n\geq 0} F^{\#^n}(\perp^\#)$ es un punto fijo de $F^\#$.

Cuando $F^\#$ es monótona, es además el mínimo punto fijo de $F^\#$ mayor o igual a $\perp^\#$.

Esta situación no es la usual. En general, uno tiene que conformarse con aproximaciones superiores de la abstracción del punto fijo.

4.4.1 Abstracción del punto fijo usando conexiones de Galois

En general uno debe conformarse con una aproximación abstracta $p^\#$ desde arriba. Es decir $\alpha(lfp\ F^b) \leq^\# p^\#$ (o equivalentemente $lfp\ F^b \leq^b \gamma(p^\#)$).

Proposición 12 *Dados dos retículos completos* $P^b(\leq^b, f^b, t^b, \wedge^b, \vee^b)$ *y* $P^\#(\leq^\#, f^\#, t^\#, \wedge^\#, \vee^\#)$, *una conexión de Galois* $P^b(\leq^b) \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} P^\#(\leq^\#)$ *y* $F^b : P^b \to P^b$ *monótona entonces*

$$\alpha(lfp\ F^b) \leq^\# lfp\ (\alpha \circ F^b \circ \gamma)$$

Una consecuencia de esto es que la elección de la semántica concreta y de la conexión de Galois determina completamente la semántica abstracta $lfp\ (\alpha \circ F^b \circ \gamma)$. Es decir, la semántica abstracta se puede derivar de forma constructiva para la semántica concreta mediante un cálculo formal que simplifique $\alpha \circ F^b \circ \gamma$ para expresarlo usando operadores sobre propiedades abstractas.

Veamos en un ejemplo como se puede diseñar de forma sistemática un intérprete abstracto:

Ejemplo 13 (de los restos módulo nueve)

Supongamos que tenemos un lenguaje de expresiones aritméticas, cuya sintaxis viene dada de la siguiente manera:

$$\begin{array}{lcl} E & ::= & P = N \\ P & ::= & N_1 \times N_2 \\ N & ::= & D \mid ND \\ D & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

donde E es una expresión, P un producto, N un número y D un dígito. El dominio estándar es el de los enteros.

La regla del nueve proporciona una condición suficiente para detectar un error en el cálculo del producto de dos números. Se suman las cifras de cada factor y se obtiene su resto módulo nueve. Ambos resultados se multiplican y se calcula el resto módulo nueve produciendo un valor f. Se suman las cifras del resultado y se calcula el resto módulo nueve, produciendo un valor r. La condición dice que si $f \neq r$ entonces el producto es erróneo, y en caso contrario no sabemos nada.

La semántica estándar de estas expresiones viene dada por:

$$\begin{array}{rcl} S[\![P = N]\!] & = & \mathit{true} \text{ si } S[\![P]\!] = S[\![N]\!] \\ S[\![P = N]\!] & = & \mathit{false} \text{ si } S[\![P]\!] \neq S[\![N]\!] \\ S[\![N_1 \times N_2]\!] & = & S[\![N_1]\!] \times S[\![N_2]\!] \\ S[\![ND]\!] & = & (10 \times S[\![N]\!]) + S[\![D]\!] \\ S[\![0]\!] & = & 0 \\ \dots & & \\ S[\![9]\!] & = & 9 \end{array}$$

El dominio abstracto es $\mathbb{Z}_9$. La función de abstracción consiste en cálcular el resto módulo 9 ($[x]_9$ denota el resto del entero x al dividirlo por 9) del valor correspondiente en la semántica estándar:

$$\begin{array}{rcl} \alpha(0) & = & 0 \\ \ldots & & \\ \alpha(8) & = & 8 \\ \alpha(9) & = & 0 \end{array}$$

A continuación, para cada término $f(t_1,\ldots,t_n)$, calculamos un operador abstracto $f^{\#} = \alpha \circ f \circ \gamma$ (se cumple entonces que $\alpha(f(t_1,\ldots,t_n)) = f^{\#}(\alpha(t_1),\ldots,\alpha(t_n))))$. En este caso son $+^{\#}$, $\times^{\#}$ y $=^{\#}$, definidos de la siguiente manera:

$$\begin{array}{rcl} x +^{\#} y & = & [x+y]_9 \\ x \times^{\#} y & = & [x \times y]_9 \\ x =^{\#} y & = & \mathit{error} \text{ si } x = y \\ & = & \mathit{desconocido} \text{ e.o.c} \end{array}$$

donde los valores *error* y *desconocido* se suelen representar con nuevos valores que se añaden al dominio abstracto. El valor *error* proporciona la mínima información posible, aunque de él se puede concluir que la multiplicación es errónea, mientras que *desconocido* proporciona la máxima información, aunque de él no se puede concluir nada acerca de la corrección de la multiplicación realizada. Estos valores suelen representarse por $\perp$ y $\top$ respectivamente. El analizador lleva a cabo dos pasos:

- Un cierto compilador abstracto convierte las expresiones del lenguaje en expresiones abstractas:

$$\begin{array}{rcl} C[\![P = N]\!] & = & (C[\![P]\!] =^{\#} C[\![N]\!]) \\ C[\![N_1 \times N_2]\!] & = & C[\![N_1]\!] \times^{\#} C[\![N_2]\!] \\ C[\![ND]\!] & = & (C[\![N]\!] +^{\#} C[\![D]\!]) \\ C[\![0]\!] & = & 0 \\ \ldots & & \\ C[\![9]\!] & = & 0 \end{array}$$

- Un intérprete evalua las expresiones abstractas:

$$\begin{array}{rcl} I[\![v_1 =^{\#} v_2]\!] & = & \mathit{error} \text{ si } I[\![v_1]\!] \neq I[\![v_2]\!] \\ & = & \mathit{desconocido} \text{ si } I[\![v_1]\!] \neq I[\![v_2]\!] \end{array}$$

$$
\begin{aligned}
I[\![v_1 \times^{\#} v_2]\!] &= [I[\![v_1]\!] \times I[\![v_2]\!]]_9 \\
I[\![v_1 +^{\#} v_2]\!] &= [I[\![v_1]\!] + I[\![v_2]\!]]_9 \\
I[\![0]\!] &= 0 \\
\ldots & \\
I[\![8]\!] &= 8
\end{aligned}
$$

La corrección del analizador se obtiene del hecho:

$$\alpha(S[\![E]\!]) = I[\![C[\![E]\!]]\!]$$

Sea, por ejemplo la expresión $E = (217 \times 38 = 8256)$. Obtenemos que: $\alpha(S[\![E]\!]) = I[\![C[\![E]\!]]\!] = error$, lo cual quiere decir que no se cumple la igualdad.

□

La mayoría de las veces no se pueden obtener los operadores abstractos correspondientes de forma que $\alpha \circ F^b \circ \gamma = F^{\#}$ sino que $\alpha \circ F^b \circ \gamma$ se aproxima desde arriba por una $F^{\#}$ tal que $\forall p^{\#} \in P^{\#}.\alpha \circ F^b \circ \gamma(p^{\#}) \leq^{\#} F^{\#}(p^{\#})$. En ese caso se cumple:

Proposición 14 *Si* $P^b(\sqsubseteq^b, f^b, \sqcup^b)$ *y* $P^{\#}(\sqsubseteq^{\#}, f^{\#}, \sqcup^{\#})$ *son cpos,* $F^b : P^b(\sqsubseteq^b, \sqcup^b) \rightarrow P^b(\sqsubseteq^b, \sqcup^b)$ *y* $F^{\#} : P^{\#}(\sqsubseteq^{\#}, \sqcup^{\#}) \rightarrow P^{\#}(\sqsubseteq^{\#}, \sqcup^{\#})$ *son continuas,* $P^b(\sqsubseteq^b) \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} P^{\#}(\sqsubseteq^{\#})$ *con:*

- $\alpha(\perp^b) = \perp^{\#}$,
- $\forall p^{\#} \in P^{\#}.\alpha \circ F^b \circ \gamma(p^{\#}) \sqsubseteq^{\#} F^{\#}(p^{\#})$

entonces $\alpha(lfp\ F^b) \sqsubseteq^{\#} lfp\ F^{\#}$.

Las condiciones de continuidad se pueden evitar sustituyéndolas por monotonía y secuencias transfinitas de iteraciones.

Capítulo 5

La interpretación abstracta en los lenguajes funcionales

5.1 Introducción

A la hora de aplicar la interpretación abstracta en los lenguajes funcionales, los conceptos generales vistos en la sección anterior se particularizan en ciertos aspectos: el tipo de dominios abstractos, la forma de representar las propiedades, el concepto de corrección y seguridad etc.

Dos preguntas importantes a las que se debe responder con respecto a un marco en el que se desarrolla una interpretación abstracta son:

- Qué propiedades se pueden representar y cómo (es decir, cual es la semántica de recolección).
- Cómo se pueden combinar dichas propiedades para representar propiedades de los tipos construidos.

A la primera pregunta la respondemos aquí de forma general, y a la segunda la responderemos de forma particular en la Sección 5.4.1 proporcionando diversas formas de combinar propiedades de productos cartesianos.

5.1.1 Las propiedades

Las propiedades de las funciones se pueden representar de formas distintas. Una de ellas consiste en utilizar conjuntos **Scott–cerrados**. En estos casos, se utilizan retículos finitos como dominios abstractos que representan a dichas propiedades. Cada punto del dominio abstracto es la interpretación abstracta de un conjunto Scott–cerrado, es decir, representa una propiedad.

Por ejemplo, en el análisis de estrictez se manejan las propiedades representadas por los conjuntos Scott–cerrados $\{\perp_D\}$ y D, las cuales son a su vez interpretados por los valores abstractos 0 y 1 respectivamente.

Las interpretaciones abstractas presentadas en este capítulo, basadas fundamentalmente en el marco presentado por Burn, Hankin y Abramsky [BHA86], representan las propiedades de esta forma.

Sin embargo, algunas propiedades no se pueden codificar mediante conjuntos Scott–cerrados. Dos ejemplos clásicos de ello son la propiedad de que una función es constante (es decir, $\forall x, y.f\ x = f\ y$) y la estrictez de cabeza, ver Sección 5.4.2.

El hecho de que la estrictez de cabeza no se puede expresar mediante conjuntos Scott–cerrados fue demostrado por Kamin [Kam92].

El caso de la propiedad de ser constante de una función se ve más fácilmente. Las propiedades que se pueden expresar mediante conjuntos Scott–cerrados son de la forma $\forall x.f\ x \in S$ para algún conjunto Scott–cerrado S, lo cual no es suficiente para expresar la constancia en general, pues todo conjunto Scott–cerrado contiene el elemento ínfimo. La clave se encuentra en el hecho de que esta propiedad está expresada como una relación. En concreto, si llamamos ALL a la relación satisfecha por todos los pares, e ID a aquella tal que $x\ ID\ y$ si y solo si $x = y$, entonces f es constante si y solo si $\forall x, y.x\ ALL\ y \Rightarrow (f\ x)\ ID\ (f\ y)$.

Por tanto parece necesario que las propiedades representadas en estos casos sean relaciones. Si nos mantenemos en primer orden podemos utilizar **relaciones de equivalencia total**, es decir, con las propiedades reflexiva, simétrica y transitiva.

Una forma de representar las relaciones de equivalencia total es mediante el uso de **proyecciones**, ver Sección 6.2. Por ello el análisis mediante proyecciones es capaz de detectar la estrictez de cabeza en funciones de primer orden.

Sin embargo, al tratar el orden superior se hace necesario considerar las **relaciones de equivalencia parcial** (no necesariamente reflexivas), debido a que incluso aunque S y T sean relaciones de equivalencia total sobre σ y τ, la relación $(S \Rightarrow T)$ sobre $\sigma \rightarrow \tau$ definida por

$$f\ (S \Rightarrow\ T)f' \text{ si y solo si } [s\ S\ s' \text{ implica } (f\ s)T(f'\ s')]$$

no lo es necesariamente (por ejemplo $ALL \Rightarrow ID$ solamente contiene a las funciones constantes).

Este enfoque fue estudiado por Hunt [Hun91] [HS91] y se presentará más adelente, en la Sección 7.1.

5.1.2 Corrección

Las dos formas más usuales de relacionar la interpretación estándar y abstracta de un lenguaje funcional son:

- La definición de una función de **concreción** γ de los valores abstractos a las propiedades concretas, demostrando que si la semántica abstracta de una expresión produce un valor abstracto p, entonces la semántica estándar de dicha expresión tiene la propiedad $\gamma\ p$. Es decir, que si:

  $$(A[\![f]\!]\ \rho^A)\ s^\# = t^\#$$

  entonces se debe cumplir que:

  $$\forall s \in \gamma(s^\#).(S[\![f]\!]\ \rho^S)\ s \in \gamma(t^\#)$$

Para demostrar esto, se proporciona la **función de abstracción** correspondiente, y se demuestra que los resultados obtenidos mediante la interpretación abstracta son seguros, es decir, se encuentran siempre por encima de la abstracción de la interpretación estándar (ver Proposición 30 en la Sección 5.3.2). Dicho resultado se apoya en la propiedad de semihomomorfismo de la función de abstracción (ver Proposición 25 de la Sección 5.3.2). Esta es la opción tomada en este capítulo.

- La definición de una **relación** entre valores abstractos y propiedades, y demostración de que las interpretaciones estándar y abstracta de cada término pertenecen a dicha relación. En este caso, se suele definir la relación sobre los tipos base y extenderla despues a los tipos de orden superior como una relación lógica con ciertas propiedades (por ejemplo, ser estricta, preservar los ínfimos, ser inductiva etc). Este es el enfoque utilizado en interpretación abstracta usando relaciones de equivalencia parcial. Para más detalles, vease la Sección 7.1.4.

5.2 Funciones de primer orden

5.2.1 Introducción

Una de las primeras aplicaciones de la interpretación abstracta en lenguajes funcionales fue el análisis de estrictez. Es el análisis sobre el que más estudios se han desarrollado.

El primero en utilizar la interpretación abstracta para analizar la estrictez de un lenguaje funcional monomórfico de primer orden con recursión sobre dominios planos fue Mycroft [Myc80] [Myc81].

En esta sección presentamos este análisis con la mayor sencillez posible para extenderlo posteriormente a orden superior en la siguiente sección.

Una definición intuitiva de **estrictez** es la siguiente: una función es estricta en un argumento si siempre necesita el valor de dicho argumento. Mycroft formaliza esta idea diciendo que una función es estricta si $f\bot = \bot$.

Es decir, dado un argumento cuya evaluación no termina, f no termina. Podría no terminar por alguna otra razón pero el resultado final de $f\bot$ sería el mismo.

Esta noción se extiende de forma natural a funciones con varios argumentos. Por ejemplo, si f tiene tres argumentos x, y y z, decimos que es estricta en y si:

$$\forall x, z. f\ x\ \bot\ z = \bot$$

5.2.2 Semántica estándar

La semántica estándar del lenguaje sobre el que trabaja Mycroft es una semántica denotacional donde la interpretación de una función f de k parámetros viene dada por una función continua $f^b : D^k \to D$, donde D es un dominio plano, que puede consisitir en la unión disjunta de enteros, booleanos y otros tipos primitivos. Dichas funciones están definidas en el lenguaje mediante ecuaciones mutuamente recursivas de la forma:

$$\begin{array}{l} \textbf{let} \\ \qquad f_1\ (x_1, \ldots, x_k) \quad = E_1 \\ \qquad \vdots \\ \qquad f_n\ (x_1, \ldots, x_k) \quad = E_n \\ \textbf{in } E_0 \end{array}$$

donde la sintaxis de las expresiones es la siguiente:

$$\begin{array}{lcl} E & ::= & x_i \\ & & \mid f_k(E_1, \ldots, E_k) \\ & & \mid \mathbf{A_j}(E_1, \ldots, E_k) \end{array}$$

donde las $\mathbf{A_j}$ son funciones predefinidas, entre las que se encuentra, por ejemplo, el condicional **if**, funciones aritméticas como $+$, $*$ etc.

Por simplicidad de notación, Mycroft supone que todas las funciones tienen el mismo número de parámetros k. En los ejemplos no usa sin embargo tal restricción.

Su interpretación se obtiene mediante el mínimo punto fijo de una función continua de iteración T, donde $T : L \to L$ y $L = (D^k \to D) \times \ldots \times (D^k \to D)$.

Por ser T continua el punto fijo existe y viene dado por $\sqcup_{Dn \geq 0} T^n(\bot_L)$, donde

$$\bot_L = (\lambda x_1 \ldots x_k.\bot_D, \ldots, \lambda x_1, \ldots, x_k.\bot_D)$$

Es decir, el dominio semántico estándar es el cpo de funciones continuas $< D^k \to D, \sqsubseteq, \bot, \sqcup >$ donde $\bot = \lambda x_1 \ldots x_k.\bot_D$, y $\sqsubseteq$, $\sqcup$ se definen punto a punto.

Mycroft proporcionó una semántica de recolección, utilizando dominios potencia para especificar el conjunto de los resultados posibles de una función dados los posibles valores para los argumentos, incluyendo los indefinidos. En la Sección 5.3.2 se estudiará con detalle el papel que juega el dominio potencia de Hoare para definir las propiedades representadas por los valores del dominio abstracto y en la demostración de corrección del análisis, por lo que no se presenta aquí para evitar redundancia.

Mycroft define dos análisis para estudiar las propiedades de estrictez y terminación de las funciones. El análisis de estrictez, como ya se ha señalado, pretende averiguar cuándo una función es estricta en su argumento. El análisis de terminación pretende determinar cuándo una función termina con seguridad.

5.2.3 Análisis de estrictez y terminación

El dominio abstracto que representa las propiedades que nos interesan es el dominio plano $D^\# = \mathbf{2} = \{0, 1\}$, ordenado por $0 \sqsubseteq_\mathbf{2} 0 \sqsubseteq_\mathbf{2} 1 \sqsubseteq_\mathbf{2} 1$. El significado intuitivo de sus elementos es que 0 representa la no terminación y 1 la posible terminación de la evaluación de los términos.

Aprovechando el aspecto booleano de este dominio se pueden utilizar los operadores habituales de conjunción y disyunción como operadores de máxima cota inferior y mínima cota superior respectivamente.

Este dominio se extiende punto a punto a las funciones. Es decir, la interpretación abstracta de una función $f : D^k \to D$ es una función continua $f^{\#} : \mathbf{2}^k \to \mathbf{2}$.

Luego el dominio semántico abstracto es el cpo de funciones continuas $< \mathbf{2}^k \to \mathbf{2}, \sqsubseteq^{\#}, \perp^{\#}, \sqcup^{\#} >$ donde $\perp^{\#} = \lambda x_1 \ldots x_k.0$, y $\sqsubseteq^{\#}$, $\sqcup^{\#}$ se definen punto a punto.

Las funciones abstractas se definen a su vez mediante ecuaciones recursivas. Su interpretación se obtiene mediante el mínimo punto fijo de una función continua de iteración $T^{\#}$, donde $T^{\#} : L^{\#} \to L^{\#}$ y $L^{\#} = (\mathbf{2}^k \to \mathbf{2}) \times \ldots \times (\mathbf{2}^k \to \mathbf{2})$.

Por ser $T^{\#}$ continua el punto fijo existe y viene dado por $\sqcup_{\mathbf{2}n \geq 0} T^{\#^n}(\perp_L)$, donde $\perp_{L^{\#}} = (\lambda x_1 \ldots x_k.0, \ldots, \lambda x_1 \ldots x_k.0)$. Además se calcula en un número finito de iteraciones por ser el retículo $L^{\#}$ finito.

Recordemos que el valor 0 representa a los términos que con toda seguridad no terminan y 1 representa a todos los términos.

La interpretación abstracta de un término e es $e^{\#}$, definido por inducción de la siguiente forma:

- Si es una constante del dominio plano (ej. enteros, booleanos) entonces $e^{\#} = 1$. Es decir, la evaluación de las constantes siempre termina.

- Si e es de la forma e_1 **op** e_2 donde **op** es uno de los operadores binarios predefinidos usuales $+$, $-$, $*$ etc, entonces $e^{\#} = e_1^{\#} \wedge e_2^{\#}$. Es decir, son estrictos en ambos argumentos. Basta con que la evaluación de uno de ellos no termine para que la evaluación de la expresión completa no termine. Por ello Mycroft expresa la interpretación mediante la conjunción. Podemos expresar este hecho definiendo funciones abstractas $op^{\#}$ que se comportan de esta manera. Por ejemplo, $x +^{\#} y = x \wedge y$ donde $x, y \in \mathbf{2}$.

- Si e es **if** b e_2 e_3 entonces $e^{\#} = b^{\#} \wedge (e_2^{\#} \vee e_3^{\#})$. Es decir, la evaluación de un condicional implica la evaluación de la condición y de una de las dos ramas. Podemos expresar este hecho definiendo una función abstracta **if**$^{\#}$ que se comporta de esta manera.

- En cuanto a las funciones definidas por el usuario, se definen en función de las básicas, por lo que basta con proporcionar las versiones abstractas de todas las funciones básicas (aritméticas, **if**...) para tener definidas todas las demás .

La función de abstracción $abs : D \to \mathbf{2}$ es la siguiente:

$$abs(x) = \begin{cases} 0 \text{ si } x = \perp_D \\ 1 \quad \text{e.o.c} \end{cases}$$

La condición de corrección que relaciona ambas intrepretaciones en este análisis es la siguiente: $f^{\#}$ es una abstracción segura de f^b si para todo i $(1 \leq i \leq k)$ se cumple que

$$\begin{gathered} f^{\#}(1/x_1, \ldots 1/x_{i-1}, 0/x_i, 1/x_{i+1}, \ldots 1/x_k) = 0 \\ \Rightarrow \\ \forall d_1, \ldots d_{i-1}, d_{i+1}, \ldots d_k \in D. f^b(d_1/x_1 \ldots d_{i-1}/x_{i-1}, \perp_D/x_i, d_{i+1}/x_{i+1}, \ldots, d_k/x_k) = \perp_D \end{gathered}$$

Es decir, si la función abstracta es estricta en un argumento, entonces también lo es la función original.

Sin embargo, es importante notar que existen funciones estrictas para las que no se cumple que $f^{\#}(1, \ldots 0, \ldots, 1) = 0$, es decir, existen funciones cuya estrictez no se puede detectar. Lo importante es que en aquellas que se detecta, la conclusión obtenida sea correcta. Un ejemplo presentado por Mycroft es el siguiente:

Ejemplo 15 Sea la función:

$$f(x, y) = \textbf{if } P(x) \ y \ 0$$

donde $P(x)$ es un predicado cierto para todo valor de x, por ejemplo $x = x$. En este caso, nosotros nos damos cuenta de que y debería ser pasado por valor, pero el análisis no puede detectarlo.

□

La corrección viene dada por el hecho de que se cumple

$$abs(f^b(x_1, \ldots, x_k)) = f^{\#}(abs(x_1), \ldots, abs(x_k))$$

Pues entonces, si $f^{\#}(1, \ldots, 0, \ldots, 1) = 0$, tomando $x_i = \bot_D$ se cumple que

$$abs(f^b(d_1, \ldots, \bot_D, \ldots, d_k)) = f^{\#}(abs(d_1), \ldots, 0, \ldots, abs(d_k))$$

lo que es igual a 0 por hipótesis (por ser $f^{\#}$ continua, entonces es monótona y por tanto, como $abs(d_i) \sqsubseteq_{\mathbf{2}} 1$ se cumple que $f^{\#}(abs(d_1), \ldots, 0, \ldots, abs(d_k)) \sqsubseteq_{\mathbf{2}} f^{\#}(1, \ldots, 0, \ldots, 1) = 0$, y como consecuencia también vale 0) .

Luego $abs(f^b(d_1, \ldots, \bot_D, \ldots, d_k)) = 0$ lo cual implica que $f^b(d_1, \ldots, \bot_D, \ldots d_k) = \bot_D$ (por definición de abs).

En cuanto al **análisis de terminación**, pretende detectar cuándo la evaluación de una expresión termina. Por ello, en este caso la interpretación del condicional es distinto:

- Si e es **if** $b \ e_2 \ e_3$ entonces $e^{\#} = b^{\#} \wedge e_2^{\#} \wedge e_3^{\#}$. Es decir, la evaluación del condicional termina con seguridad solamente cuando, si tanto la evaluación de la condición como la de las ramas, termina.

Si el análisis de estrictez nos permitía detectar si es seguro evaluar el parámetro de una función antes del cuerpo de la misma, el análisis de terminación nos permite evaluar un parámetro incluso antes de llamar a la función.

Ejemplo 16 Un ejemplo de aplicación que expone Mycroft [Myc80] es el siguiente. Sea $f(x, y) = g(x, y+1) + y$. Debido a que interpretación de la suma es $x +^{\#} y = x \wedge y$ donde $x, y \in \mathbf{2}$, sabemos que y se puede pasar por valor a f. Esto significa que podemos suponer que en el cuerpo de f, $y \neq \bot$, puesto que al pasarlo por valor, si es $\bot$ el resultado final va a seguir siendo $\bot$ en cualquier caso.

Por tanto, $y^{\#} = 1$ y $(y + 1)^{\#} = 1$. Es decir, el segundo parámetro en la llamada a g siempre se le pasa evaluado antes siquiera de llamarla. Esto nos permite evaluar directamente $y + 1$ antes de llamar a g.

Si se cumpliera esto en todas las llamadas a g entonces se le podría pasar a g el segundo parámetro por valor.

□

5.2.4 Ejemplos de análisis de estrictez

Un ejemplo presentado por Clack y Peyton Jones [CP85] es el siguiente. Sea la función $g\ (p,q,r) = \textbf{if}\ (p=0)\ (q+r)\ (q+p)$. Entonces:

$$g^{\#}\ (abs(p), abs(q), abs(r)) = \textbf{if}^{\#}\ (abs(p) =^{\#} 1)\ (abs(q) +^{\#} abs(r))\ (abs(q) +^{\#} abs(p))$$

Renombrando los parámetros de la forma $p^{\#} = abs(p)$ etc., tenemos que:

$$\begin{aligned} g^{\#}\ (p^{\#}, q^{\#}, r^{\#}) &= (p^{\#} \wedge 1) \wedge ((q^{\#} \wedge r^{\#}) \vee (q^{\#} \wedge p^{\#})) \\ &= p^{\#} \wedge q^{\#} \wedge (p^{\#} \vee r^{\#}) \end{aligned}$$

Así, si queremos saber si g es estricta en sus parámetros, calculamos:

$$\begin{aligned} &g^{\#}\ (0,1,1) = 0 \wedge 1 \wedge (0 \vee 1) = 0 \\ &g^{\#}\ (1,0,1) = 1 \wedge 0 \wedge (1 \vee 1) = 0 \\ &g^{\#}\ (1,1,0) = 1 \wedge 1 \wedge (1 \vee 0) = 1 \end{aligned}$$

Estos resultados nos indican que g es estricta en p y q pero no sabemos nada acerca de r.

Un ejemplo con recursión

Sea la función recursiva:

$$f(x,y,z) = \textbf{if}\ (x=0)\ (y-1)\ f((x-1),z,y)$$

La interpretación abstracta de dicha función es la función abstracta $f^{\#} : \mathbf{2}^3 \to \mathbf{2}$, donde:

$$f^{\#}(x,y,z) = x \wedge (y \vee f^{\#}(x,z,y))$$

La solución a esta ecuación recursiva es el punto fijo del operador $T^{\#}$ definido de la siguiente manera:

$$T^{\#} f = \lambda(x,y,z).x \wedge (y \vee f(x,z,y))$$

Dicho punto fijo se calcula como $\sqcup_{\mathbf{2}} T^{\#^n}(\perp)$, donde en este caso $\perp = \lambda(x,y,z).0$, es decir, es la función constante 0.

Esto es equivalente al cálculo de la siguiente cadena de aproximaciones, llamada **cadena de Kleene**:

- $f_0^{\#}$ es la función constante que siempre devuelve 0 (llamada 0–ésima aproximación).
- $f_{i+1}^{\#}$ es la función obtenida usando $f_i^{\#}$ en el lado derecho de la definición (llamada $(i+1)$–ésima aproximación).

Esta secuencia de aproximaciones crece monótonamente. Si $f_n^{\#} = f_{n+1}^{\#}$, entonces $f_n^{\#}$ es el límite de la secuencia y por tanto el punto fijo de $T^{\#}$.

Por estar las funciones interpretadas sobre un dominio finito existe un i a partir del cual no se obtienen nuevas funciones (cualquier cadena creciente de funciones debe ser de longitud finita), es decir, se llega a un punto fijo en un número finito de pasos. En el ejemplo:

	$f_0^\#$	$f_1^\#$	$f_2^\#$	$f_3^\#$
$(0,0,0)$	0	0	0	0
$(0,0,1)$	0	0	0	0
$(0,1,0)$	0	0	0	0
$(0,1,1)$	0	0	0	0
$(1,0,0)$	0	0	0	0
$(1,0,1)$	0	0	1	1
$(1,1,0)$	0	1	1	1
$(1,1,1)$	0	1	1	1

El punto fijo se alcanza en $f_2^\#$. Luego $f^\# = f_2^\#$. Puesto que $f^\#(0,1,1) = 0$ podemos concluir que f es estricta en su primer argumento, aunque no se puede concluir nada sobre los otros dos pues $f^\#(1,0,1)$ y $f^\#(1,1,0)$ son distintos de 0.

Funciones mutuamente recursivas

Normalmente un programa tiene un conjunto de definiciones de funciones mutuamente recursivas $\{f_j\}$. En este caso la solución al sistema de ecuaciones se calcula comenzando por una familia $\{f_{j,0}^\#\}$, donde cada una de ellas es la función constante 0. La colección $\{f_{j,i+1}^\#\}$ se define en términos de las $\{f_{j,i}^\#\}$ de forma similar al caso anterior.

Por ejemplo, si tuviéramos dos funciones f y g, calcularíamos el punto fijo:

$$\begin{array}{ll} f_0^\#\ x{=}0 & g_0^\#\ y{=}0 \\ f_1^\#\ x{=}\ldots g_0^\# \ldots f_0^\# \ldots & g_1^\#\ y{=}\ldots f_0^\# \ldots g_0^\# \ldots \\ f_2^\#\ x{=}\ldots g_1^\# \ldots f_1^\# \ldots & g_2^\#\ y{=}\ldots f_1^\# \ldots g_1^\# \ldots \end{array}$$

A veces resulta más eficiente usar $f_n^\#$ en $g_n^\#$ en lugar de $f_{n-1}^\#$, puesto que ya disponemos de ella.

En cada iteración debe hacerse una comprobación de igualdad. Por tanto si el programa consta de n funciones, es necesario hacer n comparacioness en cada iteración, es decir, hay que comprobar si para todo j, $f_{j,i+1}^\# = f_{j,i}^\#$. Esto resulta ser muy costoso.

Una forma de solucionar esto consistiría en llevar a cabo inicialmente algun tipo de análisis de dependencias para separar las definiciones en grupos mutuamente recursivos mínimos. Esto lo suelen hacer los compiladores funcionales; es un problema de detección de bucles en un grafo.

5.2.5 Cálculo eficiente de puntos fijos

Introducción

Una forma de calcular el punto fijo consiste en iterar sucesivamente un número suficiente de veces para estar seguros de haber alcanzado el punto fijo.

El número de iteraciones necesarias para calcular el punto fijo viene dado por la profundidad del dominio de funciones $\mathbf{2}^k \to \mathbf{2}$, la cual está definida como la longitud de la cadena más larga estrictamente creciente de valores de dicho dominio. Por ejemplo, si $k = 2$ entonces bastan 4 iteraciones. Es un buen método cuando el número de argumentos es pequeño. Desafortunadamente la profundidad del dominio crece exponencialmente con el número de argumentos, con lo que esta aproximación se hace insostenile cuando k es grande. Además es necesario hacer una comparación de igualdad en cada iteración. La eficiencia de la comparación de igualdad influye, por tanto, en la eficiencia del algoritmo completo. En el peor de los casos, el coste de la comparación es exponencial en el numero de argumentos. El problema de detectar la igualdad de funciones de N argumentos con valores en **2** es NP–completo, y por tanto no hay algoritmo polinomial que pueda resolverlo [HY85]. Puesto que no se puede determinar la igualdad de dos funciones en base a su definicion, se hace necesaria alguna representación de las funciones $f_i^{\#}$ que funcione de forma eficiente en los casos más comunes. Chris Clack y Peyton Jones [CP85] [JC87] examinaron tres formas distintas de representación, las **tablas de verdad**, las **expresiones booleanas** y las **fronteras**.

Tablas de verdad

Una función se puede caracterizar completamente mediante pares argumento–resultado. En nuestro caso, los argumentos son tuplas de elementos de **2** y el resultado un elemento de **2**. Por tanto, podemos representar las funciones abstractas mediante tablas de verdad, como la mostrada en el ejemplo con recursión de la Sección 5.2.4. El punto fijo se alcanza cuando la tabla para $f_n^{\#}$ es igual que para $f_{n+1}^{\#}$

Es una representación con coste siempre exponencial en el número de argumentos de la función, independientemente de lo bien que se comporte ésta.

Expresiones booleanas

Debido al aspecto del dominio elegido para el análisis de estrictez, se pueden utilizar expresiones booleanas simbólicas para representar las funciones abstractas. Esta representación es mucho más compacta que la anterior. Por ejemplo:

$$f^{\#}\ (a, b, c, d) = a \wedge d$$

es una representación más reducida que la tabla de 16 filas que necesitaríamos si usáramos tablas de verdad.

Sin embargo, las cosas se complican a la hora de comparar dos expresiones booleanas, para determinar cuándo se alcanza el punto fijo. Éstas pueden parecer muy diferentes simbólicamente y, sin embargo, representar la misma función booleana.

Para poder compararlas, debemos manipularlas hasta obtener una forma canónica. Las formas más comunes son la **forma normal conjuntiva** y la **forma normal disyuntiva**. Ninguna es mejor que la otra en general, ya que algunas expresiones tienen una representación más compacta en forma conjuntiva y otras en forma disyuntiva.

Por otra parte, al pasar a orden superior el uso de expresiones booleanas se hace inmanejable. Por ejemplo, si tenemos la función:

$$f\ g = g\ (f\ g)$$

entonces

$$\begin{array}{rcl} f^{\#}\ g^{\#} & = & g^{\#}(f^{\#}\ g^{\#}) \\ f_0^{\#}\ g^{\#} & = & 0 \\ f_1^{\#}\ g^{\#} & = & g^{\#}\ 0 \\ f_2^{\#}\ g^{\#} & = & g^{\#}(g^{\#}\ 0) \end{array}$$

expresión que crece en cada iteración. Desde $f_1^{\#}$ se cumple que $f_n^{\#} = f_{n+1}^{\#}$ para cualquier $g^{\#}$ (por monotonía), pero no se puede detectar examinando las expresiones booleanas.

Fronteras

Los posibles argumentos de una función $f^{\#}$ forman un retículo finito, el cual se puede representar mediante un grafo no dirigido. Cada uno de los nodos representa una combinación de argumentos. Por ejemplo, si tenemos una función $f^{\#}$ de primer orden de tres argumentos, entonces el retículo tiene el aspecto que muestra la Figura 5.1.

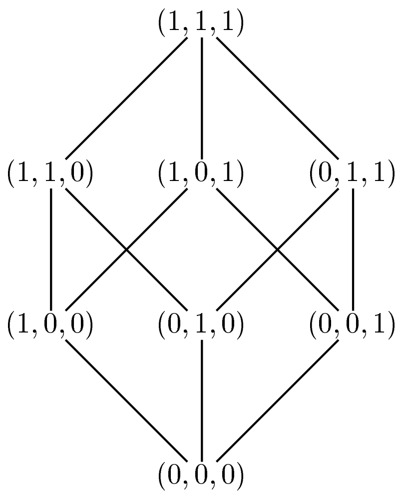


Figura 5.1: Retículo de los posibles argumentos de una función $f^{\#} : \mathbf{2}^3 \rightarrow \mathbf{2}$

Si el resultado de aplicar $f^{\#}$ a los argumentos especificados en un nodo es 0 (1), le llamaremos **0–nodo** (**1–nodo**).

De esta forma podríamos etiquetar los nodos del grafo con el valor correspondiente de $f^{\#}$ en ellos.

Hemos interpretado las funciones como funciones continuas abstractas, y por tanto monótonas. Esto quiere decir que si un nodo es un 0–nodo, entonces, cualquiera que le aproxime, es decir, que esté por debajo de él en el retículo, debe ser también un 0–nodo.

De la misma forma, si un nodo es un 1–nodo, entonces cualquier nodo al que aproxime, es decir, que esté por encima de él en el retículo, debe ser también un 1–nodo.

Esto significa que para representar una función, basta con guardar los valores de aquellos nodos que constituyen la barrera de separación entre los 0–nodos y los 1–nodos.

Este conjunto de nodos reciben el nombre de **frontera**. La **0–frontera** es el conjunto de 0–nodos que están en la frontera, y de forma análoga se define la **1–frontera**.

Ejemplo 17 Sea la función:

$$f(x, y, z) = \textbf{if}\ (x < 0)\ y\ (z + f(x - 1, 0, z))$$

cuya interpretación abstracta es:

$$f^{\#}(x, y, z) = x \wedge (y \vee (z \wedge f^{\#}(x, 1, z)))$$

La secuencia de aproximaciones es la siguiente:

$$\begin{array}{rcl} f_0^{\#}(x, y, z) & = & 0 \\ f_1^{\#}(x, y, z) & = & x \wedge (y \vee (z \wedge f_0^{\#}(x, 1, z))) \\ & = & x \wedge y \\ f_2^{\#}(x, y, z) & = & x \wedge (y \vee (z \wedge f_1^{\#}(x, 1, z))) \\ & = & x \wedge (y \vee z) \end{array}$$

En este ejemplo, la 0–frontera de $f_1^{\#}$ está formada por los nodos $(0, 1, 1)$ y $(1, 0, 1)$, pues todos los nodos con un 0 en la x o en la y son 0–nodos, y de ellos estos dos son los mayores en el retículo. La 1–frontera está formada por el nodo $(1, 1, 0)$, ya que todos los nodos con $x = y = 1$ son 1–nodos, y éste es el menor de ellos. En la Figura 5.2 se muestra el retículo de posibles argumentos, y cada una de las fronteras. Junto a cada punto del retículo figura el valor de $f_1^{\#}$ en ese punto.

La 0–frontera de $f_2^{\#}$ está formada por los punto $(0, 1, 1)$ y $(1, 0, 0)$, y la 1–frontera por los puntos $(1, 1, 0)$ y $(1, 0, 1)$. En la Figura 5.3 se muestra el retículo de posibles argumentos y cada una de las fronteras. Junto a cada punto del retículo figura el valor de $f_2^{\#}$ en ese punto.

□

Formalmente, una **0–frontera** de una función $f^{\#} : \alpha \to \mathbf{2}$ es un conjunto de valores $F0$ de tipo α tal que:

- Por encima de $F0$ hay 1–nodos, y por debajo 0–nodos:

$$\forall v : \alpha, f^{\#} v = 0 \text{ si y solo si } \exists c \in F0 \text{ tal que } v \leq c$$

- $F0$ no es redundante:

$$\forall c_1, c_2 \in F0, c_1 \leq c_2 \Rightarrow c_1 = c_2$$

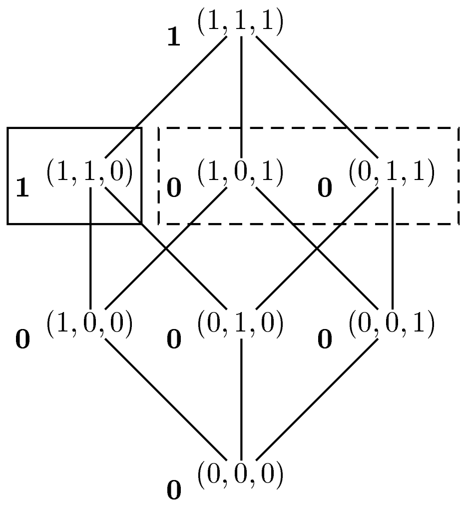


Figura 5.2: Valores de $f_1^{\#}$ y fronteras (0–frontera en caja punteada y 1–frontera en caja continua)

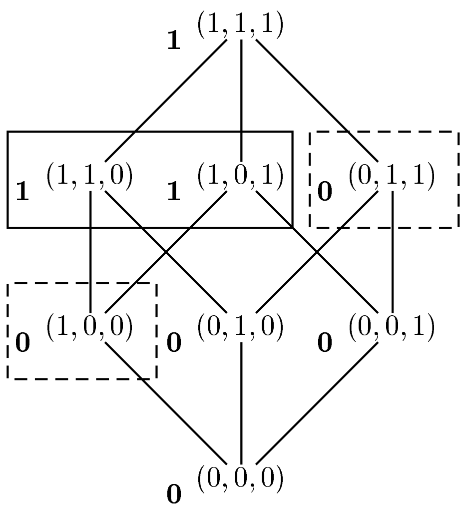


Figura 5.3: Valores de $f_2^{\#}$ y fronteras (0–frontera en caja punteada y 1–frontera en caja continua)

De forma similar se define la 1–frontera.

Se demuestra que la 0–frontera es **única**, y que **dos funciones con las mismas 0–fronteras son iguales**. Una frontera, es por tanto una forma **canónica y compacta** de representar una función.

Las fronteras solamente representan funciones monótonas, y por tanto se corresponden solamente con las expresiones booleanas sin negación, ya que la negación se corresponde con la función que hace corresponder $\perp$ a $\top$ y $\top$ a $\perp$ y esta no es monótona.

Monotonía entre aproximaciones:

Hemos visto que la secuencia de aproximaciones crece de forma monótona, por lo que de una aproximación a la siguiente, la 0–frontera se moverá desde el elemento supremo (en el ejemplo $(1,1,1)$) hacia abajo. Esto sugiere que la 0–frontera es la más adecuada, ya que cuando los puntos de la 0–frontera no varíen de una aproximación a la siguiente, se ha alcanzado el punto fijo. Esto significa que para calcular la 0–frontera de una aproximación podemos partir de la 0–frontera de la anterior explorando hacia abajo.

El algoritmo tendría el siguiente aspecto:

$i = 0$
Fr_i = supremo del retículo de argumentos
cambia = true
mientras cambia
    Calcular Fr_{i+1} a partir de Fr_i
    $cambia = not\ (Fr_{i+1} = Fr_i)$
    $i = i + 1$
fmientras

Cálculo de la 0–frontera:

En cada paso de iteración es necesario calcular la 0–frontera de la función correspondiente. Según hemos indicado antes, debido a la monotonía se puede calcular la 0–frontera de la función $f^{\#}_{i+1}$ partiendo de la 0–frontera de la iteración anterior $f^{\#}_i$.

Sin embargo, la experiencia dice que, a menudo, en sucesivas iteraciones las fronteras se mueven rápidamente desde las posiciones cercanas al supremo hasta las más cercanas al ínfimo (en nuestro ejemplo $(0,0,0)$). Esto hace que el cálculo de la frontera a partir de la anterior resulte costosa.

En el peor de los casos el coste de cálculo de la 0–frontera es proporcional a la profudidad del retículo de argumentos, es decir a n, donde n es el número de argumentos.

Por otro lado, los puntos cercanos al supremo e ínfimo del retículo aportan gran cantidad de información al cálculo de la frontera: si sabemos que un punto cercano al ínfimo del retículo es un 1–nodo, ya sabemos que todos los nodos por encima de él son también 1–nodos recortándose de forma considerable el número de nodos a investigar; y de la misma manera si cerca del supremo del retículo hallamos un 0–nodo, ya sabemos que todos los que están por debajo de él serán también 0–nodos, recortándose de forma considerable el número de nodos a investigar. Esto sugiere realizar en cada iteración una exploración simultánea del retículo desde el ínfimo hacia arriba en busca de la 1–frontera y desde el supremo hacia abajo en busca de la 0–frontera.

De esta forma, si explorando el retículo hacia arriba encontramos un 1–nodo, podemos cortar el espacio de la búsqueda hacia abajo hasta ese nodo, ya que esos nodos intermedios por monotonía serán también 1–nodos. Y, si explorando hacia abajo, encontramos un 0–nodo, podemos cortar el espacio de la búsqueda hacia arriba hasta dicho nodo, ya que los nodos intermedios serán también 0–nodos.

Este recorrido simultáneo mejora el caso medio, aunque el coste en el caso peor sigue siendo exponencial, puesto que el problema es NP–completo.

Cálculo de $f_i^{\#}$

Supongamos que hemos calculado la frontera de una iteración $f_i^{\#}$ y aun no hemos llegado al punto fijo. A continuación hay que calcular la frontera de la función $f_{i+1}^{\#}$. Esto implica una nueva doble búsqueda y por tanto la evaluación de $f_{i+1}^{\#}$ en algunos nodos.

$f_{i+1}^{\#}$ está definido en función de $f_i^{\#}$, y ésta en función de $f_{i-2}^{\#}$, y así sucesivamente, tal como vimos en el ejemplo anterior f.

Sin embargo, puesto que una función está totalmente caracterizada por su 0–frontera, no es necesaria la expansión completa hasta $f_0^{\#}$ de la función para calcularla.

Para calcular $f_{i+1}^{\#}$ en un determinado nodo basta con calcular el valorde $f_i^{\#}$ en ese nodo y sustituirlo en el lado derecho de la definición de $f_{i+1}^{\#}$. Para calcular el valor de $f_i^{\#}$ basta con consultar su frontera, de la que ya disponíamos. Si el nodo está por debajo de la 0–frontera o pertenece a ella, entonces es un 0–nodo; si no, es un 1–nodo.

Una vez calculada la nueva frontera, es necesario comparar con la frontera anterior, para ver si se ha llegado al punto fijo. En el peor de los casos, esto tiene coste proporcional a la anchura del retículo de argumentos, es decir n, donde n es el número de argumentos. Puesto que el cálculo de la 0–frontera es también lineal, esto quiere decir, que en cada iteración el coste es lineal en el número de argumentos, en el peor de los casos.

El algoritmo de fronteras pretende solamente una mejora en el caso medio, a la vez que una representación cómoda de las funciones, pero sigue siendo exponencial en el caso peor.

Refinamiento de los algoritmos

Este método solamente se aplica a funciones de primer orden monomórficas sobre un dominio de dos puntos, como **2**.

Sin embargo, el método fue extendido a orden superior y a otros dominios finitos de mayor tamaño que **2**, aunque no a infinitos, ver Sección 5.3.3.

Una fuente de ineficiencia en el método es el hecho de que calcula las 0–fronteras completas para todas las aproximaciones del punto fijo. Esto es innecesario, ya que una vez se detectara una diferencia entre la 0–frontera que se está calculando y la siguiente iteración, se podría pasar directamente a la siguiente aproximación.

Es decir, en la i–ésima iteración, al añadir un elemento x a su 0–frontera, chequeamos si $f_{i+1}^{\#}\ x = 0$. Si no se cumple, entonces $f_i^{\#} \neq f_{i+1}^{\#}$, y podemos pasar directamente a la siguiente iteración. De esta forma, sin embargo perdemos la ventaja para el cálculo de $f_i^{\#}$ visto anteriormente ya que no disponemos de la frontera completa de la iteración anterior. No parece, por tanto, una optimización claramente ventajosa.

Otra pérdida de eficiencia se produce por la repetición del cálculo de $f_i^{\#}$ en algunos nodos. Esto se debe a cierta información que se ignora durante las búsquedas hacia arriba y hacia abajo. En la búsqueda hacia abajo se van buscando 0–nodos. Durante dicha búsqueda, sin embargo se van encontrando algunos 1–nodos, información que no queda registrada de ninguna forma. Por otro lado, la búsqueda hacia arriba va buscando 1–nodos, pero alguno de ellos puede ser un 1–nodo que ya había sido evaluado en la búsqueda hacia abajo. Al no haberse registrado este hecho, el nodo se volverá a evaluar.

Para resolver esto se podría mantener un conjunto de nodos que sabemos que son 1–nodos, y consultar dicho conjunto cuando se vaya a calcular $f_i^{\#}$ en un nuevo nodo x. Si x está por encima de algún elemento del conjunto entonces su valor es 1. Por ello, cada vez que se añada un elemento al conjunto, se pueden eliminar de él todos los que están por encima.

De forma dual se tratan los 0–nodos encontrados en la búsqueda hacia arriba. Esta posible solución, sin embargo, no parece que sea una fuente grande de optimización, puesto que implicaría el mantenimiento de conjuntos de nodos con operaciones de eliminación, lo cual podría resultar contraproducente. Una solución mejor a este problema se presenta en la Sección 5.3.3.

Otra forma de representación

Otra forma de representar las funciones booleanas son los **grafos o diagramas de decisión binarios**, introducidos por Bryant [Bry90]. Esta forma de representación maneja también funciones no monótonas (expresiones booleanas con negación). Sin embargo, los algoritmos de Bryant dependen de la elección de un buen orden entre los argumentos, de forma que la elección de un mal orden produce malos resultados incluso para funciones que se comportan bien.

5.3 Funciones de orden superior

5.3.1 Introducción

Cuando se desea introducir el orden superior en los análisis debemos tener en cuenta:

- A qué funciones están ligados los parámetros funcionales.
  Un ejemplo en análisis de estrictez es el siguiente, propuesto en [BHA86]:

$$\begin{array}{lcl} g\ w & = & 2 * w \\ h\ x\ y\ f & = & x + f\ (g\ y) \end{array}$$

  En este ejemplo, vemos que g es estricta en su argumento, por serlo el producto, y que h es estricta en x y f. Sin embargo no podemos saber nada de y, pues la estrictez de h en y depende de la estrictez de f. En una aplicación como $h\ e1\ e2\ g$ deberíamos ser capaces de detectar que h es estricta en y también.

- A qué función se evalúa realmente una expresión de tipo funcional.
  En el ejemplo que presentamos en el Capítulo 1 (y que se analiza en mayor profundidad en el Ejemplo 73 de la Sección 7.2.2):

$$
\begin{array}{ll}
\textbf{letrec} & \\
 & f\ x\ y = (x+y) \\
 & twice\ g\ z = g\ (g\ z) \\
\textbf{in} & \\
 & twice\ (f\ 7)\ 9
\end{array}
$$

  necesitamos saber, a qué función se liga el parámetro g, y como consecuencia a qué función se evalúa realmente la expresión $f\ 7$.

5.3.2 Aproximación de funciones mediante funciones abstractas

Los primeros en presentar un análisis de estrictez para un lenguaje funcional de orden superior, fueron Burn, Hankin y Abramsky [BHA86]. Posteriormente, Burn [Bur91] desarrolló un marco general de interpretación abstracta basada en estas ideas.

El lenguaje

El lenguaje con el que trabajan es un lenguaje de orden superior con tipos monomórficos planos. Las expresiones de tipo vienen dadas por:

$$\sigma ::= A \mid \sigma \to \sigma$$

donde A es un tipo base.

Y los términos por la gramática:

$$
\begin{array}{rcl}
Exp & ::= & c^{\sigma} \\
 & & \mid x^{\sigma} \\
 & & \mid \lambda x^{\sigma}.Exp \\
 & & \mid Exp_1\ Exp_2 \\
 & & \mid \mathbf{fix}_{(\sigma\to\sigma)\to\sigma}\ Exp
\end{array}
$$

donde $\mathbf{fix}_{(\sigma\to\sigma)\to\sigma}$ representa el operador de punto fijo. Este operador sirve para expresar la recursión[1].

[1] En el λ–cálculo sin tipos la recursión se puede expresar mediante el término

$$Y = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

En el λ–cálculo con tipos monomórficos, este término no es tipable, por lo que es necesario introducir el operador de punto fijo $\mathbf{fix}_{(\sigma\to\sigma)\to\sigma}$ para expresar la recursión. En los ejemplos nos hemos tomado la libertad de definir las funciones mediante ecuaciones recursivas para que la lectura resulte más intuitiva, aunque según la sintaxis dada debería hacerse usando el operador **fix** que corresponda en cada caso.

Semántica estándar

El dominio sobre el que se define la semántica (denotacional) estándar del lenguaje es:

$$D = + \{D_\sigma \mid \sigma \text{ es una expresión de tipo}\}$$

donde + representa la suma separada o unión disjunta, D_A es un dominio base plano que contiene, entre otras cosas, a los enteros y los booleanos, y

$$D_{\alpha\to\beta} = [D_\alpha \to D_\beta]$$

Para dar la semántica necesitamos dar una interpretación de constantes $K : constantes \to D$, donde $K(c^\sigma) \in D_\sigma$ y un cierto entorno $\rho^s : Env$ (denotaremos los entornos con un superíndice que nos indique si es un entorno en semántica estándar s o abstracta t), donde las variables están ligadas a elementos de D:

$$\begin{array}{lcl}
sem[\![]\!] : Exp \to Env \to D & & \\
sem[\![c^\sigma]\!]\rho^s & = & K(c^\sigma) \\
sem[\![x^\sigma]\!]\rho^s & = & \rho^s(x^\sigma) \\
sem[\![\lambda x^\sigma.e]\!]\rho^s & = & \lambda y^{D_\sigma}.(sem[\![e]\!]\rho^s[y^{D_\sigma}/x^\sigma]) \\
sem[\![e_1\ e_2]\!]\rho^s & = & (sem[\![e_1]\!]\rho^s)\ (sem[\![e_2]\!]\rho^s) \\
sem[\![\mathbf{fix}_{(\sigma\to\sigma)\to\sigma}\ e]\!]\rho^s & = & \bigsqcup_{n\geq 0}(sem[\![e]\!]\rho^s)^n(\perp_{D_\sigma})
\end{array}$$

donde $\rho^s[y^{D_\sigma}/x^\sigma]$ es el mismo entorno que ρ^s excepto en x^σ, donde su valor es y^{D_σ}.

Es decir, una función sintáctica de tipo $\alpha \to \beta$ se interpreta como una función continua en D, de D_α en D_β. La aplicación funcional sintáctica se interpreta como la aplicación de una función a su argumento.

Ejemplo 18 En este ejemplo, omitiremos los superíndices de las constantes, excepto en el caso del **if**, para simplificar notación, y en su lugar escribiremos una única vez $c : \sigma$. Si tenemos los tipos **int** y **bool** como tipos base, podríamos tener por ejemplo como constantes del lenguaje:

$$\begin{array}{rcl}
0 & : & \mathbf{int} \\
1 & : & \mathbf{int} \\
-1 & : & \mathbf{int} \\
\vdots & & \\
tt, f\!f & : & \mathbf{bool} \\
succ & : & \mathbf{int} \to \mathbf{int} \\
zero & : & \mathbf{int} \to \mathbf{bool} \\
\mathbf{if}^\sigma & : & \mathbf{bool} \to \sigma \to \sigma \to \sigma
\end{array}$$

y otras como los operadores aritméticos $+$, $-$, $*$, *div*, *mod*, los operadores lógicos entre booleanos *and*, *or* etc. La interpretación estándar sería la siguiente:

$$D_{\mathbf{int}} = \mathbb{Z}_\perp$$

$$
\begin{array}{rcl}
D_{\mathbf{bool}} & = & Bool_{\perp} \\
\\
K(0) & = & 0 \\
K(1) & = & 1 \\
K(-1) & = & -1 \\
\vdots & & \\
K(succ)\ \perp_{\mathbb{Z}_{\perp}} & = & \perp_{\mathbb{Z}_{\perp}} \\
K(succ)\ x & = & x+1 \text{ si } x \neq \perp_{\mathbb{Z}_{\perp}} \\
\\
K(tt) & = & true \\
K(ff) & = & false \\
\\
K(zero)\ \perp_{\mathbb{Z}_{\perp}} & = & \perp_{Bool_{\perp}} \\
K(zero)\ 0 & = & true \\
K(zero)\ n & = & false \text{ si } n \neq 0 \wedge n \neq \perp_{\mathbb{Z}_{\perp}}
\end{array}
$$

$$
K(\mathbf{if}^{\sigma})\ x\ y\ z = \begin{cases} \perp_{D_{\sigma}} & \text{si } x = \perp_{Bool_{\perp}} \\ y & \text{si } x = true \\ z & \text{si } x = false \end{cases}
$$

Los operadores aritméticos y booleanos se interpretarían de la forma usual.

□

Semántica abstracta

De la misma forma se define la semántica abstracta para el análisis. El dominio abstracto es:

$$B = +\ \{B_{\sigma} \mid \sigma \text{ es una expresión de tipo}\}$$

donde de nuevo $+$ representa la unión disjunta, B_A es un retículo finito correspondiente al tipo base.

Ejemplo 19 (análisis de estrictez)
Utilizaremos como ejemplo el análisis de estrictez de [BHA86] donde las constantes del lenguaje son las propuestas en el Ejemplo 18.

En este caso B_A es el formado por dos elementos $\mathbf{2}$, es decir, $B_{\mathbf{int}} = B_{\mathbf{bool}} = \mathbf{2}$ y $B_{\alpha\to\beta} = [B_{\alpha} \to B_{\beta}]$. En la Figura 5.4 podemos ver el dominio abstracto que resulta en este caso concreto. Puesto que B_A es un retículo, también lo es cada B_{σ}, y por ser finitos son completos.

La semántica abstracta *tabs* está definida de forma análoga a la estándar, con la diferencia de la interpretación de las constantes K'. En el caso del análisis de estrictez éstas estan definidas de la siguiente forma:

$$
\begin{array}{rcl}
K'(0) & = & 1 \\
K'(1) & = & 1 \\
K'(-1) & = & 1 \\
\vdots & & \\
K'(succ)\ x & = & x \\
& & \\
K'(tt) & = & 1 \\
K'(ff) & = & 1 \\
& & \\
K'(zero)\ x & = & x \\
& & \\
K'(\mathbf{if}^\sigma)\ 0\ x\ y & = & \bot_{B_\sigma} \\
K'(\mathbf{if}^\sigma)\ 1\ x\ y & = & x \sqcup y
\end{array}
$$

donde $x \sqcup y$ es el supremo calculado punto a punto.

Los operadores aritméticos y booleanos se interpretarían mediante el ínfimo de sus argumentos, por ser estrictos en ambos.

□

Relación entre las semánticas

A continuación debemos relacionar la interpretación abstracta y la estándar para garantizar la corrección del análisis.

En lugar de definir la función de abstracción $abs : D \to B$ como una única función se define una para cada componente del dominio:

$$
\begin{array}{l}
abs_A : D_A \to B_A \\
abs_{\alpha\to\beta} : D_{\alpha\to\beta} \to B_{\alpha\to\beta}
\end{array}
$$

Ejemplo 20 (análisis de estrictez)

En el caso del análisis de estrictez:

$$
abs_A(d) = \begin{cases} 0 & \text{si } d = \bot_A \\ 1 & \text{e.o.c.} \end{cases}
$$

para cada dominio básico A.

□

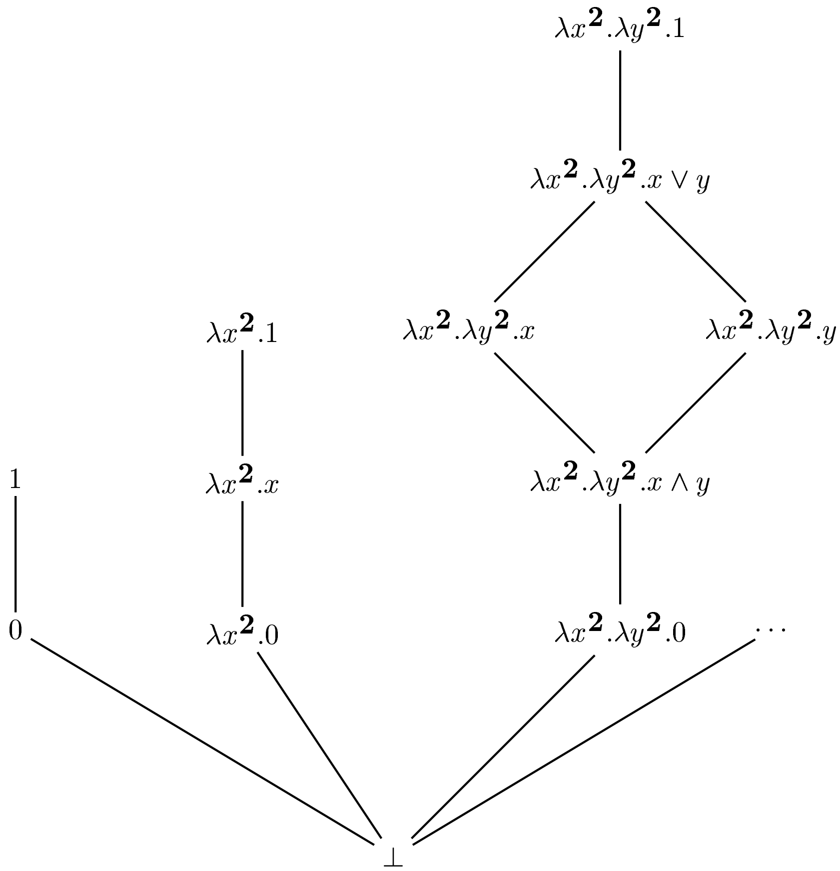


Figura 5.4: El dominio abstracto para el análisis de estrictez

La abstracción de las funciones es más compleja. Utilizando conceptos de dominios potencia de Hoare se define:

$$abs_{\alpha\to\beta}(f) = \bigsqcup(\mathcal{P}(abs_\beta) \circ \mathcal{P}(f) \circ \gamma_\alpha) = Abs_\beta \circ \mathcal{P}(f) \circ \gamma_\alpha$$

$$\begin{tikzcd}
B_\alpha \arrow[r, "abs_{\alpha\to\beta}(f)"] \arrow[d, "\gamma_\alpha"'] & B_\beta \\
\mathcal{P}(D_\alpha) \arrow[r, "\mathcal{P}(f)"'] & \mathcal{P}(D_\beta) \arrow[u, "Abs_\beta"']
\end{tikzcd}$$

donde:

- $\mathcal{P}$ es el funtor que hace corresponder a un dominio D su correspondiente dominio potencia de Hoare $\mathcal{P}(D)$ y a una función f le hace corresponder la función $\mathcal{P}(f)$, que toma un subconjunto cerrado X de D y devuelve el cierre inferior de las imágenes de los elementos de X por f, $LC(f(X))$.

- $\gamma_\sigma : B_\sigma \to \mathcal{P}(D_\sigma)$ es la función de concreción, que se define como:

$$\gamma_\sigma\ s = \bigcup\{S \mid Abs_\sigma(S) \sqsubseteq s\} = \{s' \mid abs_\sigma(s') \sqsubseteq s\}$$

- $Abs_\sigma = \bigsqcup \circ \mathcal{P}(abs_\sigma)$

La idea subyacente en esta definición, es que si b es una cierta información que se conoce sobre un argumento de f, entonces, para encontrar información sobre el resultado de f se calculan los resultados de aplicar f a los argumentos por debajo de b respecto al orden de aproximación de los elementos, y con todos ellos se aproxima el resultado para el argumento inicial (tomando mínima cota superior para mantener la seguridad).

Abramsky [AJ91] dio una definición equivalente sin usar explícitamente dominios potencia:

$$abs_{\alpha\to\beta}(f)b = \sqcup\{abs_\beta(f(d)) \mid abs_\alpha(d) \sqsubseteq b\}$$

Estas funciones de abstracción y concreción deben cumplir una serie de propiedades para garantizar la corrección del análisis.

Propiedades de las funciones de abstracción y concreción

Algunas proposiciones de esta sección hacen uso de las propiedades del dominio potencia de Hoare vistas en la Sección 2.3.3. Todas las demostraciones se pueden encontrar en [Bur91].

Proposición 21 *Se cumple que si abs_A es continua y estricta, entonces:*

- *abs_σ y Abs_σ son continuas para cualquier σ.*

- *abs_σ y Abs_σ son estrictas para cualquier σ.*

- *γ_σ está bien definida y es continua para cualquier σ.*

La estrictez de las funciones de abstracción es necesaria para que γ esté bien definida.

El hecho de exigir que los retículos abstractos sean completos garantiza la existencia de la cota superior que define las funciones de abstracción. La finitud de los mismos asegura que las funciones de concreción sean continuas. Abramsky [Abr90] demostró que si tenemos retículos completos arbitrarios (incluyendo infinitos), entonces las funciones de abstracción inducidas por las de concreción, pueden no ser continuas. Si las funciones de abstracción llevan elementos finitos a elementos finitos, entonces sí lo son. Y esto se cumple en el caso particular en que los retículos son finitos.

Supongamos a partir de ahora que disponemos de abs_A continua y estricta y que $abs_{\alpha\to\beta}$ está definida como hemos dicho anteriormente, es decir, que se cumplen todas las propiedades de la Proposición 21.

Proposición 22 *Se cumple que para todo σ, $abs_\sigma(s) \sqsubseteq s^\#$ si y solo si $s \in (\gamma_\sigma(s^\#))$.*

Proposición 23 *Para todo σ se cumple que $\gamma_\sigma(s^\#) = \{s \mid abs_\sigma(s) \sqsubseteq s^\#\}$*

La demostración se obtiene haciendo uso de la Proposición 22.

Proposición 24 *Tal como están definidas se cumple que γ_σ y Abs_σ son adjuntas, es decir:*

- $\gamma_\sigma \circ Abs_\sigma \sqsupseteq id_{\mathcal{P}(D_\sigma)}$
- $Abs_\sigma \circ \gamma_\sigma \sqsubseteq id_{B_\sigma}$

La demostración se obtiene usando la Proposición 22 y las propiedades del dominio de Hoare de la Sección 2.3.3.

Según la definición alternativa de conexión de Galois de la Sección 4.3.1, la Proposición 24 nos dice que γ_σ y Abs_σ forman una conexión de Galois.

Proposición 25 *abs_σ tiene propiedades semihomomórficas en la aplicación de funciones:*

$$abs_{\sigma\to\beta}(f) \circ abs_\sigma \sqsupseteq abs_\beta \circ f$$

La demostración se obtiene utilizando la Proposición 24 y las propiedades del dominio potencia de Hoare de la Sección 2.3.3. Como consecuencia de esto, se cumple:

Proposición 26

$$abs_\sigma(lfp\ f) \sqsubseteq lfp\ (abs_{\sigma\to\sigma}(f))$$

La demostración se obtiene usando las proposiciones 21 y 25. Estos resultados nos permiten demostrar la corrección del análisis planteado.

Se cumple también que si abs_A refleja el ínfimo $\perp_A$, entonces también lo hacen abs_σ y Abs_σ para cualquier σ.

El siguiente lema será útil en el ejemplo concreto que estamos desarrollando, el análisis de estrictez (ver Ejemplo 28), ya que la función de abstracción abs_A es estricta y refleja el ínfimo ($abs_A(\perp) = 0$):

Lema 27 *si abs_σ es estricta y refleja el ínfimo para todo σ entonces γ_σ también es estricta para todo σ:*

$$\gamma(\perp_{B_\sigma}) = \perp_{\mathcal{P}(D_\sigma)} = \{\perp_{D_\sigma}\}$$

La corrección del análisis

La propiedad de corrección del análisis dice que si:

$$(tabs[\![f]\!]\rho^t)\ s^{\#} \sqsubseteq t^{\#}$$

entonces se debe cumplir que:

$$\forall s \in \gamma(s^{\#}).(sem[\![f]\!]\rho^s)\ s \in \gamma(t^{\#})$$

donde para toda x^τ, $\rho^t(x^\tau) \sqsupseteq abs_\tau(\rho^s(x^\tau))$.

Ejemplo 28 (análisis de estrictez)
Para el análisis de estrictez es:

$$(tabs[\![f]\!]\rho^t)(\perp_{B_\alpha}) = \perp_{B_\beta} \Rightarrow (sem[\![f]\!]\rho^s)(\perp_{D_\alpha}) = \perp_{B_\beta}$$

ya que por el lema 27 se cumple $\gamma(\perp_{B_\sigma}) = \{\perp_{D_\sigma}\}$ y $\gamma(\perp_{B_\beta}) = \{\perp_{D_\beta}\}$.

□

La corrección depende de la **condición de seguridad** para las constantes:

$$abs_\sigma(K(c^\sigma)) \sqsubseteq K'(c^\sigma)$$

que se cumple en el caso del análisis de estrictez que hemos visto.

Ejemplo 29 (análisis de estrictez)
Así se obtiene por ejemplo, en el análisis de estrictez, que:

$$abs_{\mathbf{bool}\to\sigma\to\sigma\to\sigma}(\mathbf{if}^\sigma) \sqsubseteq \lambda x^{\mathbf{2}}.\lambda y^{B_\sigma}.\lambda z^{B_\sigma}.x\ \&_{B_\sigma}\ (\sqcup\{y,z\})$$

donde:

$$\begin{array}{rcl} 0\ \&_{B_\sigma}\ e & = & \perp_\sigma \\ 1\ \&_{B_\sigma}\ e & = & e \end{array}$$

y $\sqcup\ \{y,z\}$ se calcula punto a punto.
Veamos que en efecto esto se cumple. Por definición de abs:

$$\begin{array}{l} (abs_{\mathbf{bool}\to\sigma\to\sigma\to\sigma}\ \mathbf{if}^\sigma)\ x^{\#}\ y^{\#}\ z^{\#} = \\ \quad \bigsqcup\{abs_\sigma(\mathbf{if}^\sigma\ x\ y\ z) \mid (abs_{\mathbf{bool}}\ x) \sqsubseteq x^{\#}, (abs_\sigma\ y) \sqsubseteq y^{\#}, (abs_\sigma\ z) \sqsubseteq z^{\#}\} \end{array}$$

Luego:

$$\begin{array}{lll} \multicolumn{3}{l}{(abs_{\mathbf{bool}\to\sigma\to\sigma\to\sigma}\ \mathbf{if}^\sigma)\ 0\ y^{\#}\ z^{\#} =} \\ = & \bigsqcup\{abs_\sigma(\mathbf{if}^\sigma\ \perp_{Bool_\perp}\ y\ z) \mid (abs_\sigma\ y) \sqsubseteq y^{\#}, (abs_\sigma\ z) \sqsubseteq z^{\#}\} & \\ & \quad \text{porque } abs_{\mathbf{bool}}\text{refleja el ínfimo} & \\ = & \bigsqcup\{(abs_\sigma \perp_{D_\sigma})\} & \text{por la semántica estándar de } \mathbf{if} \\ = & \bigsqcup\{\perp_{B_\sigma}\} & \text{por ser } abs_\sigma \text{ estricta} \\ = & \perp_{B_\sigma} & \end{array}$$

y

$$\begin{array}{rl}
 & (abs_{\mathbf{bool}\to\sigma\to\sigma\to\sigma}\ \mathbf{if}^\sigma)\ 1\ y^\#\ z^\# = \\
= & \bigsqcup\{abs_\sigma(\mathbf{if}^\sigma\ x\ y\ z) \mid x \in Bool_\perp, (abs_\sigma\ y) \sqsubseteq y^\#, (abs_\sigma\ z) \sqsubseteq z^\#\} \\
= & \bigsqcup\{(abs_\sigma \perp_{D_\sigma}), (abs_\sigma\ y), (abs_\sigma\ z) \mid (abs_\sigma\ y) \sqsubseteq y^\#, (abs_\sigma\ z) \sqsubseteq z^\#\} \\
 & \text{por la semántica estándar de } \mathbf{if} \\
\sqsubseteq & \bigsqcup\{y^\#, z^\#\} \\
 & \text{por monotonía de } abs_\sigma \text{ y porque puede no ser sobreyectiva}
\end{array}$$

□

Proposición 30 *Si todas las constantes satisfacen la condición de seguridad entonces para todo* ρ^s, ρ^t *tales que para todo* x^τ, $abs_\tau(\rho^s x^\tau) \sqsubseteq (\rho^t x^\tau)$ *entonces, para toda expresión e se tiene:*

$$abs_\sigma(sem[\![e]\!]\rho^s) \sqsubseteq tabs[\![e]\!]\rho^t$$

Demostración:

Por inducción sobre los términos del lenguaje:

1. Si el término es una variable:

$$\begin{array}{rll}
tabs[\![x^\sigma]\!]\ \rho^t & = \rho^t(x^\sigma) & \\
 & \sqsupseteq abs_\sigma(\rho^s x^\sigma) & \text{por la condición del teorema} \\
 & = abs_\sigma(sem[\![x^\sigma]\!]\ \rho^s) &
\end{array}$$

2. Si el término es una constante:

$$\begin{array}{rll}
tabs[\![c^\sigma]\!]\ \rho^t & = K'(c^\sigma) & \\
 & \sqsupseteq abs_\sigma(K(c^\sigma)) & \text{por la condición del teorema} \\
 & = abs_\sigma(sem[\![c^\sigma]\!]\ \rho^s) &
\end{array}$$

3. Sea $s^\# \in B_\sigma$. Entonces, si el término es una λ–abstracción:

$$\begin{array}{rl}
(tabs[\![\lambda x^\sigma.e]\!]\ \rho^t)\ s^\# & = (\lambda y^{B_\sigma}.tabs[\![e]\!]\rho^t[y^{B_\sigma}/x^\sigma])\ s^\# \\
 & = tabs[\![e]\!]\ \rho^t[s^\#/x^\sigma]
\end{array}$$

y

$$\begin{array}{rl}
(abs_{\sigma\to\beta}(sem[\![\lambda x^\sigma.e]\!]\ \rho^s))\ s^\# & = (\lambda s^{B_\sigma}.\bigsqcup\{abs_\beta((sem[\![\lambda x^\sigma.e]\!]\ \rho^s)\ s) | (abs_\sigma s) \sqsubseteq s^{B_\sigma}\})\ s^\# \\
 & = \bigsqcup\{abs_\beta(\lambda y^{D_\sigma}.(sem[\![e]\!]\ \rho^s[y^{D_\sigma}/x^\sigma])\ s) | (abs_\sigma s) \sqsubseteq s^\#\} \\
 & = \bigsqcup\{abs_\beta(sem[\![e]\!]\ \rho^s[s/x^\sigma] | (abs_\sigma s \sqsubseteq s^\#\}^*
\end{array}$$

Los entornos $\rho^t[s^\#/x^\sigma]$ y $\rho^s[s/x^\sigma]$ cumplen las condiciones sobre los entornos pues $abs_\sigma s \sqsubseteq s^\#$, y por hipótesis de inducción, cada elemento del conjunto:

$$\{abs_\beta(sem[\![e]\!]\ \rho^s[s/x^\sigma]) \mid abs_\sigma s \sqsubseteq s^\#\}$$

aproxima a $tabs[\![e]\!]\ \rho^t[s^\#/x^\sigma]$ y por tanto se obtiene el resultado deseado.

4. Si el término es una aplicación:

$$\begin{array}{lll} tabs[\![e_1e_2]\!]\ \rho^t & = (tabs[\![e_1]\!]\ \rho^t)(tabs[\![e_2]\!]\ \rho^t) & \\ & \sqsupseteq (abs_{\sigma\to\beta}(sem[\![e_1]\!]\ \rho^s))(abs_\sigma(sem[\![e_2]\!]\ \rho^s)) & \text{por hip. de inducción} \\ & \sqsupseteq abs_\beta((sem[\![e_1]\!]\ \rho^s)(sem[\![e_2]\!]\ \rho^s)) & \text{por Proposición 25} \\ & = abs_\beta(sem[\![e_1e_2]\!]\ \rho^t) & \end{array}$$

5. Si el término es el operador **fix** aplicado a una función:

$$\begin{array}{lll} tabs[\![\mathbf{fix}_{(\sigma\to\sigma)\to\sigma}e]\!]\rho^t & = \bigsqcup_{n\geq 0}(tabs[\![e]\!]\ \rho^t)^n(\bot_{B_\sigma}) & \\ & \sqsupseteq \bigsqcup_{n\geq 0}(abs_{\sigma\to\sigma}(sem[\![e]\!]\ \rho^s))^n(\bot_{B_\sigma} & \text{por hipótesis de inducción} \\ & \sqsupseteq abs_\sigma(\bigsqcup_{n\geq 0}(sem[\![e]\!]\ \rho^s)^n\bot_{D_\sigma}) & \text{por Proposición 26} \\ & = abs_\sigma(sem[\![\mathbf{fix}_{(\sigma\to\sigma)\to\sigma}e]\!]\ \rho^t) & \end{array}$$

Como consecuencia de esto se demuestra la corrección:

$$\begin{array}{lll} t^\# & \sqsupseteq (tabs[\![f]\!]\ \rho^t)\ s^\# & \\ & \sqsupseteq (abs_{\sigma\to\beta}(sem[\![f]\!]\ \rho^s))\ s^\# & \text{por Proposición 30} \\ & \sqsupseteq (abs_{\sigma\to\beta}(sem[\![f]\!]\ \rho^s))(abs_\sigma s)\ \forall s\in(\gamma_\sigma s^\#) & \text{por Proposición 23} \\ & \sqsupseteq abs_\beta((sem[\![f]\!]\ \rho^s)\ s) & \text{por Proposición 25} \\ & \Rightarrow ((sem[\![f]\!]\ \rho^s)\ s)\in(\gamma_\sigma t^\#) & \text{por Proposición 22} \end{array}$$

Por tanto para demostrar la corrección de un análisis cualquiera basta con demostrar que:

- Las constantes cumplen la condición de seguridad.
- Las abs_A son continuas y estrictas.

Tales condiciones se cumplen en el análisis de estrictez visto.

Ejemplo 31 (análisis de estrictez)

En el análisis de estrictez, dada una función f de tipo $\sigma_1 \to \sigma_2 \to \ldots \sigma_n \to A$, entonces:

$$f^\# 1_{\sigma_1}\ldots 0_{\sigma_k}\ldots 1_{\sigma_n} = 0 \Rightarrow (\forall x_i \in D_{\sigma_i}.f x_1\ldots\bot_{D_k}\ldots x_n = \bot)$$

donde 0_{σ_i} y 1_{σ_i} representan el ínfimo y el supremo respectivamente del retículo B_{σ_i} (ver Figura 5.4).

□

Veamos un ejemplo de aplicación del análisis de estrictez a una función de orden superior.

Ejemplo 32 Sea la función de orden superior *twice* : (**int** → **int**) → **int** → **int**:

$$twice\ f\ x = f\ (f\ x)$$

La interpretación abstracta de *twice* viene dada por la siguiente tabla:

$twice^\#$

$x^\#/f^\#$	$\lambda x.0$	$\lambda x.x$	$\lambda x.1$
0	0	0	1
1	0	1	1

Puesto que $twice^\#$ $(\lambda x.0)$ $1 = 0$ y $twice^\#$ $(\lambda x.1)$ $0 = 1$, *twice* es estricta en su argumento f y no se puede decir nada acerca de x, puesto que depende de la estrictez de f respecto a su parámetro.

□

5.3.3 Cálculo eficiente de puntos fijos

El algoritmo propuesto por Chris Clack y Simon L. Peyton Jones [CP85] [JC87] mostrado en la Sección 5.2.5 fue optimizado y extendido a orden superior y retículos arbitrarios por Martin y Hankin [MH87].

El algoritmo anterior, como ya comentamos, puede calcular el valor de algunos nodos intermedios más de una vez. Para evitar esto Martin y Hankin modificaron el algoritmo de la siguiente manera: en lugar de construir la 0–frontera durante la búsqueda hacia abajo, y la 1–frontera en la búsqueda hacia arriba, se hace exactamente lo contrario.

El algoritmo de cálculo de la frontera

Nos centramos en este apartado en el algoritmo para calcular la frontera de una iteración a partir de la anterior. Cada búsqueda está representada mediante dos conjuntos. Para la búsqueda hacia arriba, que construye la 0–frontera tenemos $NewFr_0$ que contiene la máxima frontera calculada hasta el momento y $TrialFr_0$ que es una mínima 1–frontera que contiene todos los puntos que aun deben ser evaluados durante la búsqueda hacia arriba y que aún podrían evaluarse a 0. Para la búsqueda hacia abajo tenemos $NewFr_1$ y $TrialFr_1$.

En cada búsqueda hacia abajo, se toma un punto aun por evaluar, es decir, uno de $TrialFr_1$.

Si es un 1–nodo, se elimina de $TrialFr_1$ y se añade a $NewFr_1$ de forma que siga siendo una mínima 1–frontera. Es decir, para añadir un nodo, todos los que estén por encima de él deben ser eliminados, ya que se vuelven redundantes (procedimiento `Añadir-min`). Al quitarlo de $TrialFr_1$, deben eliminarse de él también todos los puntos que estén por encima de él, puesto que ya sabemos que son 1–nodos y no hace falta probarlos. Además añadimos a $TrialFr_1$ los que están inmediatamente por debajo (predecesores) de los recién eliminados y que no están por encima del punto añadido a $NewFr_1$ (procedimiento `Eliminar-max` en el algoritmo), pues se convierten en potenciales 1–nodos.

Si es un 0–nodo, entonces, puesto que todos los nodos por debajo de él serán 0–nodos, éstos deben eliminarse de $TrialFr_1$ (procedimiento `Quitar-inferiores`).

De forma simétrica cada vez que se realiza una búsqueda hacia arriba, se toma un punto aun por evaluar, es decir, uno de $TrialFr_0$.

Si es un 0–nodo, se quita de $TrialFr_0$ y se añade a $NewFr_0$ de forma que siga siendo una máxima 0–frontera. Es decir, para añadir un nodo, todos los que estén por debajo de él deben ser eliminados, ya que se vuelven redundantes (procedimiento `Añadir-max` en el algoritmo). Al quitarlo de $TrialFr_0$, deben eliminarse de él también todos los puntos que estén por debajo de él, puesto que ya sabemos que son 0–nodos y no hace falta probarlos. Además añadimos a $TrialFr_0$ los que están inmediatamente por encima (sucesores) de los recién eliminados y que no están por debajo del punto añadido a $NewFr_0$ (procedimiento `Eliminar-min` en el algoritmo).

Si es un 1–nodo, entonces, puesto que todos los nodos por encima de él serán 1–nodos, éstos deben eliminarse de $TrialFr_0$ (procedimiento `Quitar-superiores` en el algoritmo).

Por monotonía, en cada iteración del algoritmo de cálculo de punto fijo $TrialFr_1$ se inicializa de nuevo con $NewFr_0$ y $NewFr_1$ con $NewFr_1$ de la iteración anterior.

Así el algoritmo para el cálculo de la frontera en cada iteración es el mostrado en la Figura 5.5.

$TrialFr_0$ = {<0,0,...,0>}
$NewFr_0$ = { }
$TrialFr_1 = NewFr_0$ de la iteración anterior –inicialmente {<1,1,...,1>}
$NewFr_1 = NewFr_1$ de la iteración anterior –inicialmente {}
mientras ($TrialFr_0 \cup TrialFr_1 \neq \{\}$) **hacer**
  si $TrialFr_0 \neq \{\}$ **entonces**
    tomar un elemento x de $TrialFr_0$
    si $\exists z \in NewFr_1 . x \geq z$ **entonces**
      $resultado = 1$
    si no
      $resultado = f\ x$
    fsi
    `Buscar-hacia-arriba` (x, $resultado$)
    `Buscar-hacia-abajo` (x, $resultado$)
  fsi
  si $TrialFr_1 \neq \{\}$ **entonces**
    tomar un x elemento de $TrialFr_1$
    si $\exists z \in NewFr_0 . x \leq z$ **entonces**
      $resultado = 0$
    si no
      $resultado = f\ x$
    fsi
    `Buscar-hacia-arriba` (x, resultado)
    `Buscar-hacia-abajo` (x, resultado)
  fsi
fmientras

`Buscar-hacia-abajo`($punto$, $valor$):
  si ($valor = 1$) **entonces**
    $NewFr_1$:= `Añadir-min` ($NewFr_1$, $punto$)
    $TrialFr_1$:= `Eliminar-max` ($TrialFr_1$, $punto$)
  si no
    $TrialFr_1$:= `Quitar-inferiores` ($TrialFr_1$, $punto$)
  fsi

`Buscar-hacia-arriba` ($punto$, $valor$):
  si ($valor = 0$) **entonces**
    $NewFr_0$:= `Añadir-max` ($NewFr_0$, $punto$)
    $TrialFr_0$:= `Eliminar-min` ($TrialFr_0$, $punto$)
  si no
    $TrialFr_0$:= `Quitar-superiores` ($TrialFr_0$, $punto$)
  fsi

Figura 5.5: Algoritmo para el cálculo de la frontera

Las únicas operaciones que se necesitan implementar son por tanto las siguientes:

- Operadores de comparación sobre el dominio de argumentos posibles, al que llamaremos E.
- Operaciones para el cálculo de los predecesores ($PREDS$) y sucesores ($SUCCS$) de un nodo, necesarios, tal como hemos visto para los procedimientos $Eliminar-max$ y $Eliminar-min$.

Chris Martin demostró que este algoritmo es correcto, es decir que calcula las fronteras buscadas y que termina en todos los casos, y que en el peor de los casos cada uno de los nodos se evalua a lo sumo una vez.

El orden superior

El el caso de primer orden, el dominio de posibles argumentos E coincide con el dominio base, que en el caso de análisis de estrictez es $\mathbf{2}$. Si queremos extenderlo a orden superior debemos poder admitir que los argumentos sean funciones sobre el dominio base. Es decir, en el caso de análisis de estrictez los nodos del retículo serán tuplas cuyas componentes pueden ser funciones sobre $\mathbf{2}$.

Por tanto, para aplicar el algoritmo necesitamos un operador de comparación entre funciones y operadores que proporcionen las funciones sucesoras y predecesoras en el retículo de funciones correspondiente.

Para representar las funciones argumento se puede utilizar a su vez la 0–frontera, y por tanto la operación de comparación se establece entre 0–fronteras y los operadores de sucesor y predecesor deben proporcionar una frontera a partir de otra frontera.

Diremos que $F_1 \leq F_2$ si:

$$\forall x \in F_2 . \exists z \in F_1 . x \leq z$$

Ejemplo 33 Si el parámetro funcional tiene tipo $\mathbf{2} \to \mathbf{2} \to \mathbf{2}$, el retículo correspondiente y su representación mediante fronteras se muestran en la Figura 5.6.

Según esto, sería de esperar que

$$SUCCS(\lambda x.\lambda y.x \wedge y) = \{\lambda x.\lambda y.x, \lambda x.\lambda y.y\}$$

es decir

$$SUCCS(\{< 1, 0 >, < 0, 1 >\}) = \{\{< 1, 0 >\}, \{< 0, 1 >\}\}$$

□

Si utilizamos la 0–frontera para representar los argumentos funcionales, el cálculo de los sucesores es sencillo. Basta con sustituir cada vez un nodo de la 0–frontera original por sus nodos predecesores, manteniendo siempre el hecho de que no sea redundante y sea máxima.

$$
\begin{tikzcd}
& \{\} \arrow[d, dash] & \\
& \{(0,0)\} \arrow[dl, dash] \arrow[dr, dash] & \\
\{(0,1)\} \arrow[dr, dash] & & \{(1,0)\} \arrow[dl, dash] \\
& \{(1,0),(0,1)\} \arrow[d, dash] & \\
& \{(1,1)\} &
\end{tikzcd}
$$

Figura 5.6: El retículo de parámetros para una función cuyo único parámetro es otra función del tipo $\mathbf{2} \to \mathbf{2} \to \mathbf{2}$ representada mediante la 0–frontera

$$
\begin{tikzcd}
& {[\{\},\{(1,1)\}]} \arrow[d, dash] & \\
& {[\{(0,0)\},\{(1,0),(0,1)\}]} \arrow[dl, dash] \arrow[dr, dash] & \\
{[\{(0,1)\},\{(1,0)\}]} \arrow[dr, dash] & & {[\{(1,0)\},\{(0,1)\}]} \arrow[dl, dash] \\
& {[\{(1,0),(0,1)\},\{(1,1)\}]} \arrow[d, dash] & \\
& {[\{(1,1),\{\}\}]} &
\end{tikzcd}
$$

Figura 5.7: El retículo de parámetros para una función cuyo único parámetro es otra función del tipo $\mathbf{2} \to \mathbf{2} \to \mathbf{2}$ representada mediante ambas fronteras

Ejemplo 34 Continuando con el ejemplo anterior y según esto:

$$
\begin{array}{l}
SUCCS(\{< 1,0 >, < 0,1 >\}) = \\
\{\texttt{Anadir}-\texttt{max}\ (\{< 0,1 >\}, \texttt{Preds}\ (< 1,0 >)), \texttt{Anadir}-\texttt{max}\ (\{< 1,0 >\}, \texttt{Preds}\ (< 0,1 >))\} \\
= \{\texttt{Anadir}-\texttt{max}\ (\{< 0,1 >\}, \{< 0,0 >\}), \texttt{Anadir}-\texttt{max}\ (\{< 1,0 >\}, \{< 0,0 >\})\} = \\
\{\{< 1,0 >\}, \{< 0,1 >\}\}
\end{array}
$$

□

Sin embargo, el cálculo de los predecesores es difícil en el caso de representar mediante 0–fronteras y sencillo si representamos mediante 1–fronteras. Por ello, la mejor opción consiste en representar las funciones argumento mediante ambas: $[F0, F1]$. De esta forma el retículo para el caso $\mathbf{2} \to \mathbf{2} \to \mathbf{2}$ es el mostrado en la Figura 5.7.

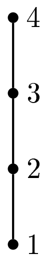


Figura 5.8: El retículo **4**

Así $SUCCS\ [F0, F1] = \{[NF0, NF1]\}$ donde cada $NF0$ se calcula de la forma especificada anteriormente y la $NF1$ correspondiente, añadiendo el elemento eliminado de $F0$ para construir $NF0$ a $F1$ (manteniendo las propiedades).

Y $PREDS\ [F0, F1] = \{[NF0, NF1]\}$ donde cada $NF1$ se calcula sustituyendo un nodo de $F1$ por sus sucesores (manteniendo minimidad) y su $NF0$ correspondiente, añadiendo el nodo eliminado o sustituido a $F0$ (manteniendo las propiedades).

Como vemos, los retículos de argumentos crecen al introducir orden superior y la complejidad se convierte en doblemente exponencial en el peor de los casos, lo cual es prácticamente inmanejable.

Retículos arbitrarios

Una función sobre un retículo base de n elementos $1, \ldots, n$ se puede representar mediante $n-1$ fronteras máximas. Cada m–frontera $(1 \leq m \leq n)$ está por encima de la máxima i–frontera para todo $i \leq m$, pero por debajo de la j–frontera para todo $m \leq j$.

Ejemplo 35 Supongamos que tenemos un retículo $\mathbf{4} = \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}\}$, ordenado como muestra la Figura 5.8 y una función $f : \mathbf{4} \rightarrow \mathbf{4} \rightarrow \mathbf{4}$ definida por $f = \lambda x_1.\lambda x_2.x_1$.

El retículo de argumentos es el mostrado en la Figura 5.9 y las fronteras son:

$$\begin{aligned} \mathbf{1} - frontera &= \{< 1, 4 >\} \\ \mathbf{2} - frontera &= \{< 2, 4 >\} \\ \mathbf{3} - frontera &= \{< 3, 4 >\} \end{aligned}$$

La $\mathbf{4} - frontera$ no hace falta calcularla pues se obtiene a partir de las anteriores: $\mathbf{4} - frontera = \{< 4, 4 >\}$.

□

Por tanto en el caso en que tengamos funciones como argumento, éstas se representarán en el retículo de argumentos mediante tuplas de $(n-1)$ componentes correspondientes a las fronteras.

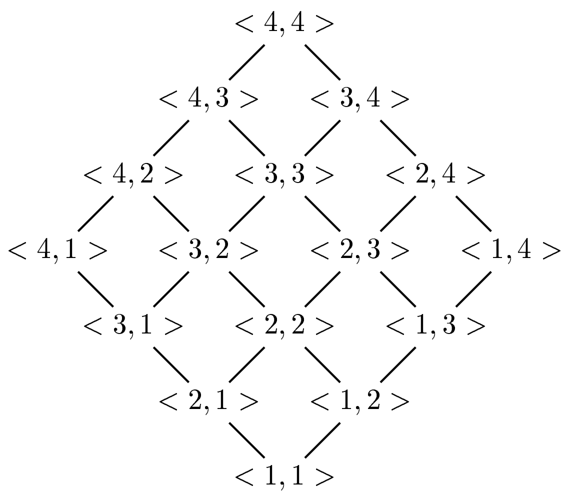


Figura 5.9: Retículo de argumentos para una función $f : \mathbf{4} \to \mathbf{4} \to \mathbf{4}$

En este caso la comparación entre funciones argumento consiste en comparar cada una de las fronteras. Si algunas son mayores y otras menores entonces son incomparables.

Los predecesores de una función se obtienen incrementando cada una de las fronteras que se pueden incrementar, teniendo en cuenta que la i–frontera no debe ser nunca mayor que la j–frontera donde $i \leq j$, o la función no sería monótona.

En el algoritmo llevamos una variable conjunto $NewFr_m$ para cada frontera que vamos a calcular. Y una variable conjunto $TrialFr_m$ que contiene todos los nodos posibles que podrían evaluarse todavía a m o a algún valor menor.

Tenemos un par de procesos de búsqueda para cada frontera calculada, uno hacia arriba que busca la máxima frontera y otro hacia abajo que busca la mínima frontera. Las variables $TrialFr$ se desdoblan en dos: $TrialFrUp$ y $TrialFrDown$ para las busquedas hacia arriba y hacia abajo respectivamente. En el ejemplo, tendríamos entonces 9 variables: $NewFr_i$, $TrialFrUp_i$ y $TrialFrDown_i$ para $i = 1, 2, 3$.

Inicialmente todos los $NewFr$ y $TrialFr$ están vacíos, excepto $TrialFrUp_1$ y $TrialDown_n$, los cuales contienen el ínfimo y el supremo del retículo de argumentos respectivamente.

Así el algoritmo es el mostrado en la Figura 5.10. Cuando se evalua un punto en el retículo de los argumentos se pasa dicha información a todos los procesos de búsqueda.

Una vez recibido un par (p, val) hay tres posibilidades en cada búsqueda. Veamos el caso en que es una búsqueda hacia arriba de la m-frontera (el caso hacia abajo es simétrico):

- Si $val \leq m$ añadimos el punto a $NewFr_m$ (manteniendo los máximos), se eliminan de $TrialFr_m$ los puntos por debajo de p y se añaden los sucesores de p.

- Si $val > m$ se eliminan de $TrialFr_m$ todos los puntos por encima de p, incluido p (pues ninguno de ellos va a poder ser un m-nodo), y se añaden los sucesores de p al $TrialFr_{val}$ (pues éstos no podrán evaluarse a ningún valor entre m y val).

Todos los $NewFrs$ y $TrialFrs$:={}
$TrialFrUp_1$:= {0,0, ...,0} –el ínfimo del retículo en general
$TrialFrDown_n$:= {1,1, ...,1} –el supremo del retículo en general
mientras ($\cup TrialFrs \neq \{\}$)
Tomar un punto del $TrialFrUp$ distinto del vacío más bajo
Obtener el valor de f en ese punto
Llamar a cada búsqueda hacia arriba y hacia abajo con (punto, valor)
Similar para los $TrialFrDown$ ...
fmientras

Figura 5.10: Algoritmo para el cálculo de las fronteras en retículos arbitrarios

- Si val y m son incomparables, entonces si $p \in TrialFr_m$ se sustituye por sus sucesores y en caso contrario no se hace nada, pues no se puede deducir nada en este caso.

Es importante darse cuenta de que cada vez que se encuentra un i–nodo, se añade a la $NewFr$ de todas las búsquedas desde 1 hasta i. Esto quiere decir que aunque no se encuentre ningún m–nodo en el retículo, existirá de todas formas una m–frontera, que será la misma que para $m-1$. En la búsqueda hacia arriba, inicialmente sólo un $TrialFr$ es distinto del vacío. A medida que va avanzando la búsqueda hacia arriba, se toman valores de esa frontera y se añaden a las fronteras superiores.

Cuando el número de elementos no comparables crece, la eficiencia de este algoritmo decrece rápidamente. Esto es así porque en tal caso, el número de veces que nos vamos por la rama "val y m no son comparables" es mayor. Como hemos visto, en dicha rama no se puede aprovechar la información de la búsqueda actual para las demás búsquedas.

En el caso de tener la suma coalescente de n retículos, una vez evaluados los nodos supremo e ínfimo de cada subretículo, los $TrialFr$ se separarán en n conjuntos disjuntos. Esto quiere decir que cuanto mayor sea n, habrá un menor aprovechamiento de la información producida por una búsqueda por parte de las demás. Solamente aquellas que pertenezcan al mismo subconjunto de $TrialFrs$ podrán aprovechar dicha información.

Comentarios

Los algoritmos basados en fronteras proporcionan una representación útil de las funciones, pero son altamente insatisfactorios en cuestiones de eficiencia, puesto que solamente funcionan relativamente bien para primer orden, funciones con pocos argumentos y dominios abstractos pequeños. Esta es la razón por la que se tiende a implementar la interpretación abstracta de una forma muy simplificada en los lenguajes reales. La atención se centra actualmente en el cálculo aproximado de los puntos fijos. Se prefiere renunciar a la exactitud frente a la ineficiencia. Para más detalles ver Sección 5.5.4.

5.4 Estructuras de datos: dominios no planos

En esta sección nos preocupamos por los tipos de datos construidos. En primer lugar estudiamos cómo se combinan las propiedades de las componentes de un par [Bur91] y posteriormente nos centramos en tipos de datos recursivos, en concreto las listas [Wad87].

5.4.1 Combinación de propiedades de pares: el producto tensor

Una de las técnicas en interpretación abstracta consiste en definir dominios abstractos para los tipos base que capturen las propiedades de interés, de forma que la interpretación abstracta de los constructores de tipo determina las propiedades que pueden definirse sobre los tipos algebraicos. Estudiemos el caso en que representamos las propiedades mediante conjuntos Scott–cerrados.

Dado un conjunto finito de propiedades, existen tres operaciones que podemos querer llevar a cabo sobre ellas: intersección, unión y complemento.

Las uniones e intersecciones finitas de conjuntos Scott–cerrados son conjuntos Scott–cerrados (propiedad de cierre), lo cual indica que las conjunciones y disyunciones de propiedades también son propiedades. No ocurre lo mismo con la negación (el complemento de un conjunto Scott–cerrado no es necesariamente Scott–cerrado).

A la hora de definir la interpretación abstracta se debe tener en cuenta que los operadores de supremo e ínfimo de los retículos finitos abstractos representen de forma adecuada la unión e intersección sobre los conjuntos que representan las propiedades.

Esto se refleja mediante la **Propiedad de Representación**. Los dominios abstractos cumplen la propiedad de representación si:

- $\gamma_\sigma(p \sqcap q) = (\gamma_\sigma(p)) \cap (\gamma_\sigma(q))$, es decir, el punto $p \sqcap q$ del dominio abstracto representa la intersección (conjunción) de las propiedades representadas por p y q.
- $\gamma_\sigma(p \sqcup q) = (\gamma_\sigma(p)) \cup (\gamma_\sigma(q))$, es decir, el punto $p \sqcup q$ del dominio abstracto representa la unión (disyunción) de las propiedades representadas por p y q.

Para cada nuevo tipo algebraico, se pretende encontrar una construcción sobre dominios abstractos con la propiedad de representación, tal que el dominio resultante tenga la propiedad de representación para el nuevo tipo.

El problema se resuelve para los pares mediante el producto tensorial pero aun es un problema abierto en el espacio de funciones.

Propiedades de los pares

Supongamos que disponemos de conjuntos finitos de propiedades $P = \{P_i \mid i \in I\}$ y $Q = \{Q_j \mid j \in J\}$ sobre dos dominios D y E respectivamente. Queremos expresar propiedades de pares de valores de $D \times E$.

Para expresar la propiedad de que el primer elemento del par tiene la propiedad P_i podemos usar el par de propiedades (P_i, E). Análogamente (D, Q_j) representa la propiedad de que la segunda componente del par tiene la propiedad Q_j.

Se cumple que

$$(P_{i1}, Q_{i1}) \bigcap (P_{i2}, Q_{i2}) = (P_{i1} \bigcap P_{i2}, Q_{i1} \bigcap Q_{i2})$$

y por ser P y Q cerrados bajo intersección, se cumple que la conjunción de propiedades de pares es un par de propiedades.

Esto significa que para representar conjunciones de propiedades basta con interpretar el producto como producto cartesiano. Es decir, todas las conjunciones de propiedades se pueden representar mediante pares de valores abstractos. Un análisis con una interpretación del producto de esta forma recibe el nombre de **método de atributos independientes**, pues no establece ninguna relación entre las componentes del par.

Consideremos, por ejemplo el análisis de estrictez visto anteriormente en el que disponemos también de un tipo producto $\sigma ::= \sigma \times \sigma$, interpretado de alguna forma aun por determinar. Y supongamos que se definen nuevas constantes:

$$\begin{array}{l} plus : \mathbf{int} \times \mathbf{int} \rightarrow \mathbf{int} \\ fst^{\sigma_1 \times \sigma_2} : \sigma_1 \times \sigma_2 \rightarrow \sigma_1 \\ snd^{\sigma_1 \times \sigma_2} : \sigma_1 \times \sigma_2 \rightarrow \sigma_2 \end{array}$$

y la expresión de formación de tuplas:

$$Exp ::= \mathbf{pair}_{(\sigma_1, \sigma_2)}(Exp_1, Exp_2)$$

La semántica estándar del producto es el producto cartesiano, es decir, $D_{\sigma_1 \times \sigma_2} = D_{\sigma_1} \times D_{\sigma_2}$ y por tanto

$$sem[\![\mathbf{pair}(e_1, e_2)]\!]\ \rho^s = (sem[\![e_1]\!]\rho^s, sem[\![e_2]\!]\rho^s)$$

y la de las constantes es, respectivamente, la suma de las componentes del par, y la proyección sobre la primera y segunda componente.

En los ejemplos, representaremos las tuplas sin el constructor **pair** por simplicidad.

Supongamos que la interpretación abstracta del producto es también el producto cartesiano $B_{\sigma_1 \times \sigma_2} = B_{\sigma_1} \times B_{\sigma_2}$, y por tanto

$$tabs[\![\mathbf{pair}(e_1, e_2)]\!]\ \rho^{\#} = (tabs[\![e_1]\!]\ \rho^{\#}, sem[\![e_2]\!]\ \rho^{\#})$$

La parte del dominio abstracto correspondiente a pares de enteros son por tanto pares de elementos de **2**. Intuitivamente, el par $(0,1)$ representa al conjunto $\{(\perp_{\mathbb{Z}_\perp}, b) \mid b \in \mathbb{Z}_\perp\}$ (la propiedad de que la primera componente del par está indefinida) y $(1,0)$ representa a $\{(a, \perp_{\mathbb{Z}_\perp}) \mid a \in \mathbb{Z}_\perp\}$ (la propiedad de que la segunda componente del par está indefinida).

El elemento $(0,0)$ representa la conjunción de ambas propiedades, es decir, al conjunto $\{(\perp_{\mathbb{Z}_\perp}, \perp_{\mathbb{Z}_\perp})\}$, que resulta de la intersección de los conjuntos anteriores.

Sin embargo, no se cumple la misma propiedad para las uniones de pares de propiedades. Continuando con el ejemplo anterior, sea la función:

$$f\ c\ x = \lambda c.\lambda x.plus\ (\mathbf{if}\ c\ (x, 12)\ (5, x))$$

Para determinar la estrictez de esta función en su segundo argumento, es necesario poder representar la propiedad de que la primera o la segunda componente del par está indefinida, es decir, el conjunto $\{(\perp_{\mathbb{Z}_\perp}, b) \mid b \in \mathbb{Z}_\perp\} \cup \{(a, \perp_{\mathbb{Z}_\perp}) \mid a \in \mathbb{Z}_\perp\}$. El punto en el dominio abstracto que mejor aproxima esta propiedad es $(0,1) \sqcup (1,0) = (1,1)$, pero $(1,1)$ representa a $\mathbb{Z}_\perp \times \mathbb{Z}_\perp$, lo cual no proporciona ninguna información.

Por otro lado, todas las propiedades se pueden expresar como unión de intersección de propiedades ya que se cumple que la intersección se distribuye sobre la unión (y viceversa). Como la intersección de propiedades es otra propiedad, todo se reduce a representar las uniones.

Una posibilidad consiste en representar la unión de propiedades

$$\bigcup\{(P_i, Q_j) | i \in I' \subseteq I \wedge j \in J' \subseteq J\}$$

mediante

$$\{(p_i, q_j) | i \in I' \subseteq I \wedge j \in J' \subseteq J\}$$

Esta interpretación, sin embargo, falla al intentar capturar dos propiedades importantes de la unión. La primera de ellas es:

$$P_{i_1} \subseteq P_{i_2} \wedge Q_{j_1} \subseteq Q_{j_2} \Rightarrow (P_{i_1}, Q_{j_1}) \cup (P_{i_2}, Q_{j_2}) = (P_{i_2}, Q_{j_2})$$

es decir, los conjuntos $\{(p_{i_1}, q_{j_1}), (p_{i_2}, q_{j_2})\}$ y $\{(p_{i_2}, q_{j_2})\}$ deberían identificarse.

Esto se consigue mediante la clausura por la izquierda, es decir, la unión de propiedades se representaría mediante:

$$\mathbf{LC}(\{(p_i, q_j) \mid i \in I' \subseteq I \wedge j \in J' \subseteq J\})$$

La segunda:

$$(P_{i_1}, Q_j) \cup (P_{i_2}, Q_j) = (P_{i_1} \cup P_{i_2}, Q_j)$$

es decir, el conjunto que contenga a (p_{i_1}, q_j) y a (p_{i_2}, q_j) también debe contener a $(p_{i_1} \sqcup p_{i_2}, q_j)$. Lo mismo sucede para la segunda componente del par.

Esto motiva la siguiente interpretación para el dominio $D \times E$:

$$\{Y \subseteq D^\# \times E^\# \mid Y = \mathbf{LC}(Y) = \mathbf{CC}_1(Y) = \mathbf{CC}_2(Y)\}$$

donde $D^\#$ y $E^\#$ son dominios abstractos correspondientes a D y E respectivamente, y:

$$\begin{array}{rcl}\mathbf{CC}_1(Y) & = & \{(d_1 \sqcup d_2, e) \mid (d_1, e), (d_2, e) \in Y\} \\ \mathbf{CC}_2(Y) & = & \{(d, e_1 \sqcup e_2) \mid (d, e_1), (d, e_2) \in Y\}\end{array}$$

Se trata de un producto tensorial de dos retículos finitos $D^\#$ y $E^\#$ con respecto a funciones lineales (o aditivas) (ver Sección 2.2), donde $cross(d, e) = \mathbf{LC}\{(d, e)\}$ y $f^\otimes(Y) = \sqcup\{f(l, l') \mid (l, l') \in Y\}$

Los análisis que interpretan el producto de esta manera reciben el nombre de **análisis relacionales**.

De esta forma, en el análisis de estrictez mencionado anteriormente, interpretamos el producto mediante el producto tensorial $B_{\sigma_1 \times \sigma_2} = B_{\sigma_1} \otimes B_{\sigma_2}$, y por tanto

$$tabs[\![\mathbf{pair}(e_1, e_2)]\!]\ \rho^t = cross(tabs[\![e_1]\!]\ \rho^t, tabs[\![e_2]\!]\ \rho^t)$$

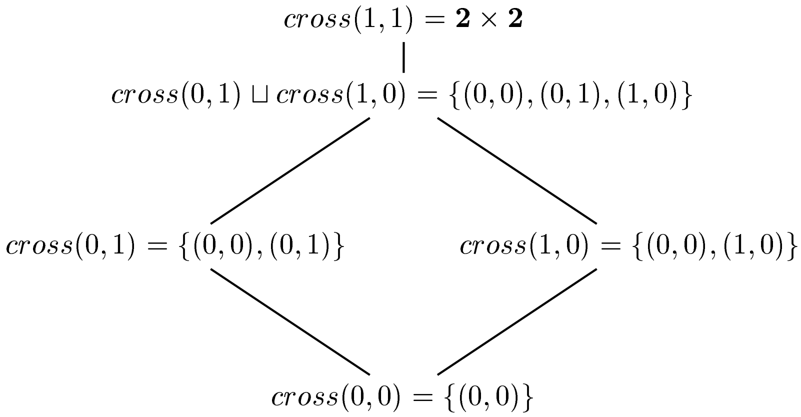


Figura 5.11: El retículo abstracto de los pares de propiedades

Existe entonces un punto adicional $cross(0,1) \sqcup cross(1,0) = \{(0,0),(0,1),(1,0)\}$ diferente del $cross(1,1) = \{(0,0),(0,1),(1,0),(1,1)\}$ (ver Figura 5.11).

Las funciones de abstracción y concreción son:

$$abs_{\sigma_1 \times \sigma_2}(d_1, d_2) = cross(\sigma_1(d_1), \sigma_2(d_2))$$

y

$$\gamma_{\sigma_1 \times \sigma_2}(Y) = \{(d_1, d_2) \mid \exists (y_1, y_2) \in Y : d_1 \in \gamma_{\sigma_1}(y_1) \wedge d_2 \in \gamma_{\sigma_2}(y_2)\}$$

La interpretación de las constantes es ahora la siguiente:

$$\begin{array}{l} K'(plus) = \lambda v^{\mathbf{2}\otimes\mathbf{2}}. \begin{cases} 1 & \text{si } v = \mathbf{LC}\{(1,1)\} \\ 0 & \text{e.o.c} \end{cases} \\ K'(fst^{\sigma_1 \times \sigma_2}) = \lambda Y^{B_{\sigma_1} \otimes B_{\sigma_2}}. \sqcup \{s_1 \mid (s_1, s_2) \in Y\} \\ K'(snd^{\sigma_1 \times \sigma_2}) = \lambda Y^{B_{\sigma_1} \otimes B_{\sigma_2}}. \sqcup \{s_2 \mid (s_1, s_2) \in Y\} \end{array}$$

las cuales cumplen la condición de seguridad de las constantes.

Así, la función f definida anteriormente, cuya interpretación abstracta es:

$$f^{\#} = \lambda c^{\mathbf{2}}.\lambda x^{\mathbf{2}}.c \wedge K'(plus)(cross(x,1) \sqcup \; cross(1,x))$$

cumple que

$$f^{\#}\ 0\ 1 = 0$$

y

$$f^{\#}\ 1\ 0 = K'(plus)(\{(0,0),(0,1),(1,0)\}) = 0$$

lo cual quiere decir que f es estricta en ambos argumentos.

Propiedades de las funciones

En el caso de las funciones, solamente se pueden representar conjunciones de propiedades si se interpretan las funciones como funciones abstractas. Hasta el momento no se conoce ninguna construcción que proporcione un dominio abstracto que verifique la primera condición del teorema de representación.

Ejemplo 36 La función:

$$f = \lambda c^{\mathbf{bool}}.\mathbf{if}\ c\ (\lambda x^{\mathbf{bool}}.\lambda y^{\mathbf{bool}}.x)\ (\lambda x^{\mathbf{bool}}.\lambda y^{\mathbf{bool}}.y)$$

se encuentra en la concreción de la función $\lambda c^{\mathbf{2}}.\lambda x^{\mathbf{2}}.\lambda y^{\mathbf{2}}.x \sqcup \lambda c^{\mathbf{2}}.\lambda x^{\mathbf{2}}.\lambda y^{\mathbf{2}}.y$ pero no se encuentra en la concreción de ninguna de las dos, y por tanto tampoco en la unión.

□

El análisis llevado a cabo por Burn, Hankin y Abramsky [BHA86] de la Sección 5.3.2 es un método independiente de los atributos, puesto que aunque en dicho análisis no se utilizaban explícitamente productos, las funciones están currificadas y se cumple que:

$$A_\sigma \to A_\tau \to A_\psi \simeq A_\sigma \times A_\tau \to A_\psi$$

pues los dominios con funciones continuas forman una categoría cartesianamente cerrada. Esto no contradice la intuición, pues no se puede establecer ninguna relación entre los argumentos cuando éstos se proporcionan de uno en uno.

Ejemplo 37 En el análisis de estrictez anterior, consideramos la versión currificada de $plus$, $plus' : \mathbf{int} \to \mathbf{int} \to \mathbf{int}$ con interpretación abstracta $K'(plus'_{\mathbf{int}\to\mathbf{int}\to\mathbf{int}}) = \lambda x^{\mathbf{2}}.\lambda y^{\mathbf{2}}.x \sqcap y$.

Sean las funciones

$$\begin{aligned} f &= \lambda x^{\mathbf{int}\times\mathbf{int}}.plus_{\mathbf{int}\times\mathbf{int}\to\mathbf{int}}\ x \\ f' &= \lambda x^{\mathbf{int}\times\mathbf{int}}.plus'_{\mathbf{int}\to\mathbf{int}\to\mathbf{int}}\ (fst_{\mathbf{int}\times\mathbf{int}}\ x)\ (snd_{\mathbf{int}\times\mathbf{int}}\ x) \end{aligned}$$

Su interpretación estándar es la misma. Sin embargo si aplicamos la interpretación abstracta de cada una de ellas a $cross(0,1) \sqcup cross(1,0)$, vemos que f produce 0, mientras que f' produce 1. Mediante la currificación se ha perdido información.

□

5.4.2 Tipos de datos recursivos: el análisis de listas

Introducción

Si el argumento de una función es una lista, a diferencia de los valores atómicos hay varias formas en que una función puede usar dicho argumento. Para ilustrar el problema podemos ver algunos ejemplos propuestos por Wadler [Wad87].

Ejemplo 38 Sean las funciones *isempty*, *length* y *sum*, definidas por:

$$\begin{array}{lcl} isempty\ nil & = & True \\ isempty\ (cons\ x\ xs) & = & False \\ & & \\ length\ nil & = & 0 \\ length\ (cons\ x\ xs) & = & 1 + length\ xs \\ & & \\ sum\ nil & = & 0 \\ sum\ (cons\ x\ xs) & = & x + sum\ xs \end{array}$$

□

Es fácil ver que cada una de ellas exige un grado distinto de evaluación del argumento lista. La primera función, *isempty* solamente necesita saber el constructor en cabeza de la lista, es decir, si se trata de la lista vacía o tiene al menos un elemento.

La segunda, *length*, sin embargo, necesita evaluar toda la lista, aunque no le importa los valores contenidos en ella.

La tercera, *sum*, no solo necesita evaluar toda la lista sino que además necesita conocer los valores numéricos para sumarlos.

Por tanto, cuando se introducen tipos estructurados, la información proporcionada en el análisis debería ser más compleja, para reflejar los distintos grados requeridos de evaluación de la estructura.

Wadler [Wad87] presentó un enfoque del análisis de estrictez sobre dominios no planos, (en concreto las listas, aunque se puede extender a cualquier otro tipo de datos) usando interpretación abstracta con dominios finitos. Esto le permitía usar las técnicas habituales para encontrar los puntos fijos, tratar el polimorfismo y el orden superior.

Listas de enteros

Los enteros se representan mediante un dominio abstracto de dos elementos $\{\perp, \top\}$. Supongamos que existe una función que a cada elemento x concreto le hace corresponder uno abstracto $x^{\#}$.

Wadler utilizó como dominio abstracto de las lista de enteros $\{\perp, \infty, \perp_\epsilon, \top_\epsilon\}$. El orden se muestra en la Figura 5.12.

Cada uno de los puntos del dominio abstracto representa una propiedad, es decir, un conjunto Scott–cerrado. Por tanto, el conjunto de listas representadas por cada punto del dominio contiene a todas las representadas por los puntos inferiores y algunas más. En la Figura 5.12 se presenta junto a cada punto del dominio un elemento perteneciente al conjunto representado por dicho punto, que no pertenece a los representados por los puntos inferiores.

El elemento $\perp$ representa a la lista totalmente indefinida $\perp$. El elemento ∞ representa a la lista $\perp$ y además a todas las listas infinitas y sus aproximaciones (ej. $1 : \perp$ es aproximación de $[1, 1, \ldots]$). $\perp_\epsilon$ representa a las anteriores y además a todas las listas finitas, alguno de cuyos elementos es $\perp$. $\top_\epsilon$ representa a las anteriores y además a las listas finitas y totalmente definidas (ningún elemento es $\perp$), es decir, representa a todas las listas. Este dominio abstracto pretende

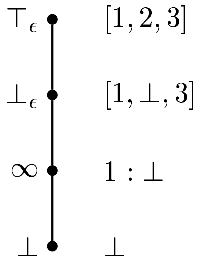


Figura 5.12: Dominio abstracto para las listas de enteros

capturar los distintos modos de evaluación: evaluar una lista a forma normal débil de cabeza, o evaluar la estructura de una lista, o evaluar la estructura de la lista y cada elemento de ella a forma normal débil de cabeza.

A continuación se definen abstracciones de los constructores: $nil^{\#} = \top_\epsilon$ y $cons^{\#}$, definido por la siguiente tabla:

$x^{\#}/xs^{\#}$	$\top_\epsilon$	$\perp_\epsilon$	∞	$\perp$
$\top$	$\top_\epsilon$	$\perp_\epsilon$	∞	∞
$\perp$	$\perp_\epsilon$	$\perp_\epsilon$	∞	∞

Esta función es monótona y cumple $abs\ (cons\ x\ xs) = cons^{\#}\ x^{\#}\ xs^{\#}$.

En el análisis que hace Wadler es esencial que las funciones estén definidas por casos pues si no se pierde información.

Ejemplo 39 Si definiéramos por ejemplo la función *sum* de la forma:

$$sum\ xs = \mathbf{if}\ (null\ xs)\ 0\ ((head\ xs) + sum\ (tail\ xs))$$

entonces el resultado del análisis de estrictez sería el siguiente:

$xs^{\#}$	$sum^{\#}\ xs^{\#}$
$\top_\epsilon$	$\top$
$\perp_\epsilon$	$\top$
∞	$\top$
$\perp$	$\perp$

Solamente nos dice que es estricta en la lista, pero no que además necesita evaluar todos y cada uno de los elemento de la lista; lo cual sí se obtiene al hacer un análisis por casos,como se verá más adelante.

□

Si una función h se define por casos de la siguiente manera:

$$\begin{array}{rcl} h\ nil & = & a \\ h\ (cons\ x\ xs) & = & f\ x\ xs \end{array}$$

entonces podemos definir su abstracción usando la tabla de definición de $cons^{\#}$ hacia atrás. Si una lista us es de la forma $cons\ x\ xs$ y $us^{\#} = \bot_\epsilon$, entonces o bien $x^{\#} = \bot$ y $xs^{\#} = \top_\epsilon$, o bien $x^{\#} = \top$ y $xs^{\#} = \bot_\epsilon$, o bien $x^{\#} = \bot$ y $xs^{\#} = \top_\epsilon$. Es decir

$$\begin{array}{rcl} h^{\#}\ \bot_\epsilon & = & (f^{\#}\ \bot\ \top_\epsilon) \sqcup (f^{\#}\ \top\ \bot_\epsilon) \sqcup (f^{\#}\ \bot\ \bot_\epsilon) \\ & = & (f^{\#}\ \bot\ \top_\epsilon) \sqcup (f^{\#}\ \top\ \bot_\epsilon) \end{array}$$

La última igualdad se cumple por monotonía: $f^{\#}\ \bot\ \bot_\epsilon \sqsubseteq f^{\#}\ \top\ \bot_\epsilon$. Razonando de la misma manera en el resto de los casos la abstracción es:

$$\begin{array}{rcl} h^{\#}\ \top_\epsilon & = & a^{\#} \sqcup (f^{\#}\ \top\ \top_\epsilon) \\ h^{\#}\ \bot_\epsilon & = & (f^{\#}\ \bot\ \top_\epsilon) \sqcup (f^{\#}\ \top\ \bot_\epsilon) \\ h^{\#}\ \infty & = & f^{\#}\ \top\ \infty \\ h^{\#}\ \bot & = & \bot \end{array}$$

Esta definición verifica que $abs\ (h\ xs) \sqsubseteq h^{\#}\ xs^{\#}$, es decir, cumple la propiedad de seguridad.

Ejemplo 40 La definición de *sum* del Ejemplo 38 en la Sección 5.4.2 es recursiva y por tanto la obtención de su interpretación abstracta implica el cálculo de un punto fijo:

$$\begin{array}{rcl} sum^{\#}\ \top_\epsilon & = & \top \sqcup (\top \sqcap (sum^{\#}\ \top_\epsilon)) \\ & = & \top \\ sum^{\#}\ \bot_\epsilon & = & (\bot \sqcap (sum^{\#}\ \top_\epsilon)) \sqcup (\top \sqcap (sum^{\#}\ \bot_\epsilon)) \\ & = & sum^{\#}\ \bot_\epsilon \\ sum^{\#}\ \infty & = & \top \sqcap (sum^{\#}\ \infty) \\ & = & sum^{\#}\ \infty \\ sum^{\#}\ \bot & = & \bot \end{array}$$

Obtenemos el siguiente resultado:

$xs^{\#}$	$sum^{\#} xs^{\#}$
$\top_\epsilon$	$\top$
$\bot_\epsilon$	$\bot$
∞	$\bot$
$\bot$	$\bot$

□

Los resultados de la interpretación abstracta se pueden entender de la siguiente manera:

- Si $h^{\#}\ \bot = \bot$ entonces h es estricta en la lista argumento y éste se puede evaluar a forma normal débil para exponer el constructor en cabeza.
- Si $h^{\#}\ \infty = \bot$ entonces es seguro evaluar el argumento y todas sus colas, es decir, se puede construir la lista con una versión de *cons* estricta en el segundo parámetro.
- Si $h^{\#}\ \bot_\epsilon = \bot$ entonces es seguro evaluar el argumento, así como todas sus cabezas y colas, es decir, se puede construir la lista con una versión de *cons* estricta en ambos argumentos (hiperestrictez).

Listas de cualquier tipo

Dado un dominio D, llamemos D^* al dominio de las lista de elementos de D. Si disponemos de una abstracción $D^{\#}$ del dominio D, entonces el dominio abstracto correspondiente a D^* es $D^{*\#}$ formado de la siguiente forma:

- Se añaden dos elementos, ∞ y $\bot$. $\bot$ representa la lista totalmente indefinida $\bot$ y ∞ representa a $\bot$ y a las listas infinitas y sus aproximaciones.
- A cada elemento de $D^{\#}$ le añadimos ϵ, creando así un dominio $D_\epsilon{}^{\#}$. Estos elementos representan a las listas representadas por los elementos inferiores y además todas las posibilidades de listas finitas, algunos de cuyos elementos poseen las propiedades representadas por $D^{\#}$.
- Los elementos se ordenan: $\bot \sqsubseteq \infty \sqsubseteq \bot_\epsilon \sqsubseteq \ldots \sqsubseteq \top_\epsilon$ (donde $\bot_\epsilon$ y $\top_\epsilon$ representan la extensión con ϵ del $\bot$ y $\top$ de $D^{\#}$).

Otra forma de expresar estas ideas es mediante la elevación de dominios, es decir:

$$D^{*\#} = ((D^{\#})_\bot)_\bot$$

Por ejemplo, en el caso de listas de listas de enteros el dominio es el presentado en la Figura 5.13

El valor abstracto que le corresponde a una lista concreta $[x_1, \ldots, x_n]$ se calcula como el ínfimo de las abstracciones de los elementos de la lista:

$$xs^{\#} = x_1^{\#} \sqcap \ldots \sqcap x_n^{\#}$$

Por ello, la abstracción de *cons* se define de la siguiente manera:

$$\begin{array}{lcl} cons^{\#}\ x^{\#}\ (y_\epsilon^{\#}) & = & (x^{\#} \sqcap y^{\#})_\epsilon \\ cons^{\#}\ x^{\#}\ \infty & = & \infty \\ cons^{\#}\ x^{\#}\ \bot & = & \infty \end{array}$$

Se puede ver fácilmente que en el caso de las listas de enteros la definición dada de $cons^{\#}$ es un caso particular del general.

El resto del análisis funciona exactamente igual que en la parte anterior.

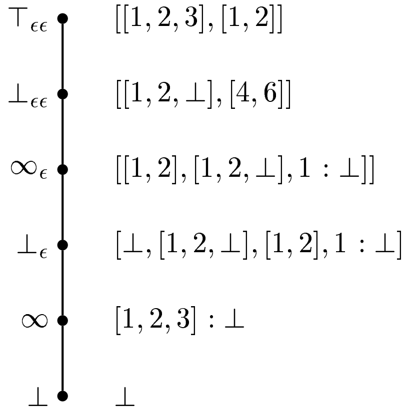


Figura 5.13: Dominio abstracto para las listas de listas de enteros

Problemas de este enfoque

En algunos casos, hacer una lista estricta en la cola puede provocar una **complejidad mayor en espacio** del programa. Por ejemplo, sea el término:

$$sum\ (upto\ 1\ n)$$

donde *upto* 1 n devuelve la lista de numeros de 1 a n. Usando una version estricta en la cola de *cons* la complejidad en espacio pasa a ser lineal en lugar de constante. Para evitar esto, se podría hacer estricta una componente solamente en el caso es que el espacio que ocupase un elemento evaluado estuviera acotado por una constante.

Este método se caracteriza por que trata todos los elementos de la lista de la misma manera. Esto contrasta con otros métodos como el de Hughes [Hug86] que puede producir resultados del tipo: una lista tendrá al menos dos elementos y es seguro evaluar la cola del primero y la cabeza del segundo. Aunque en algunos ejemplos los resultados de Hughes no son tan buenos. Por ejemplo en:

$$last\ xs = head\ (rev\ xs)$$

(donde *head* es la función que toma la cabeza de una lista y *rev* calcula la lista invertida) el resultado esperado, que el análisis de Wadler sí detecta, es que los *cons* que construyen xs se pueden hacer estrictos en la cola, pero el análisis de Hughes no encuentra dicho resultado.

El análisis de Wadler no es capaz de detectar la estrictez de cabeza, que se da por ejemplo en funciones como *head*. Sin embargo sí se puede expresar mediante análisis de proyecciones. Kamin [Kam92] demostró que la propiedad de estrictez de cabeza no se puede representar mediante conjuntos Scott–cerrados, por lo que en una interpretación abstracta del estilo que está siendo desarrollada no puede representarse. Sin embargo, el análisis de proyecciones se corresponde con la representación de propiedades mediante relaciones de equivalencia total, que sí son capaces de representar esta propiedad (ver Sección 6.2.6). La interpretación abstracta usando relaciones de equivalencia parcial también es capaz de detectarla (ver Capítulo 7.1).

5.4.3 El enfoque de los Nielson

Si A es el dominio abstracto correspondiente al tipo τ, los Nielson [NN92] propusieron el uso de ciertos subconjuntos de A como elementos del dominio abstracto para el tipo de las listas de elementos de tipo τ.

El dominio abstracto

Sea $\mathcal{O}(A)$ el conjunto de todos los conjuntos abiertos no vacíos en la topología de Scott inducida por A, ver Sección 2.3.2:

$$\mathcal{O}(A) = \{X^{\uparrow} \mid X \subseteq A \text{ y } X \neq \emptyset\}$$

$\mathcal{O}(A)$ con $\supseteq$ (los conjuntos abiertos están cerrados superiormente) forma un retículo con ínfimo A, supremo $\{\top_A\}$, y operadores de supremo e ínfimo los operadores de intersección e unión de conjuntos respectivamente.

El dominio abstracto es:

$$L(A) = \{\perp, \infty\} \sqcup \{X_\epsilon \mid X \in \mathcal{O}(A)\} = ((\mathcal{O}(A))_\perp)_\perp$$

donde el orden, ver Figura 5.14 viene dado por:

$$\begin{array}{lcll} \perp & \sqsubseteq & Z & \forall Z \in L(A) \\ \infty & \sqsubseteq & Z & \text{si } Z \in L(A) \text{ y } Z \neq \perp \\ X_\epsilon & \sqsubseteq & Y_\epsilon & \text{si } X \subseteq Y \text{ en } \mathcal{O}(A) \end{array}$$

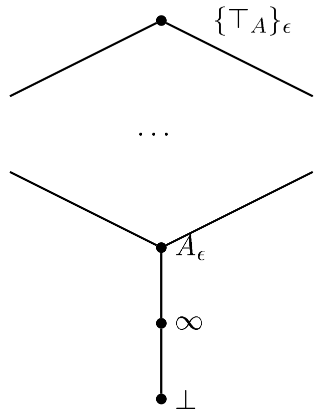


Figura 5.14: El retículo L(A)

Los elementos de $L(A)$ se interpretan de la siguiente manera:

- $\perp$ representa a la lista $\perp$.
- ∞ representa a todas las listas infinitas, las listas parciales y $\perp$.

- Si $X = \{a_1, \ldots, a_k\}$ es un conjunto abierto, entonces X_ϵ representa a todas las listas descritas por ∞ y todas las listas finitas no vacías $[x_1, \ldots, x_n]$ en las que cada a_i representa a algún x_j y todos los x_j están representados.

- $\{\top\}_\epsilon$ describe todas las listas (la lista vacía sólo se representa mediante este punto del dominio).

En el caso en que trabajamos con retículos A totalmente ordenados, como por ejemplo **2**, el dominio propuesto es el mismo que el presentado por Wadler. La principal diferencia se observa en listas de pares de elementos, es decir, cuando aparecen productos.

Por ejemplo, en el caso de listas de pares de enteros, el dominio de Wadler tiene el aspecto de la Figura 5.15. Está formado por 6 puntos. En el análisis de Wadler, la lista $[(4, \perp), (\perp, 5)]$ está representada mediante el punto $(0, 0)_\epsilon$ del dominio.

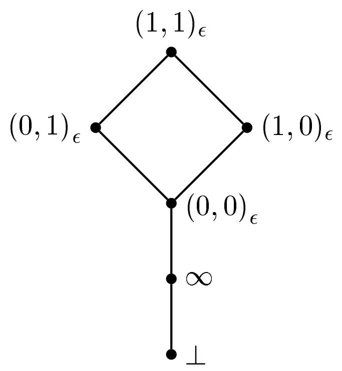


Figura 5.15: El dominio abstracto de Wadler para listas de pares de enteros

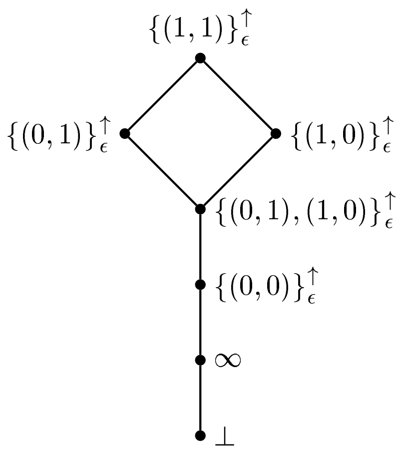


Figura 5.16: El retículo abstracto L (**2**$\times$ **2**)

Sin embargo, el dominio propuesto por los Nielson, formado por 7 puntos, tiene el aspecto de la Figura 5.16 y la lista $[(4,\bot),(\bot,5)]$ está representada mediante el valor abstracto $\{(0,1),(1,0)\}^{\uparrow} = \{(0,1),(1,0),(1,1)\}$.

Mediante este dominio, por tanto, se puede detectar el hecho de que los pares que forman parte de una lista tienen una de sus dos componentes indefinidas.

Interpretación de las funciones

Si A es el dominio de los elementos, la interpretación de nil_A es $nil_A^{\#} = \{\top_A\}_\epsilon$. La interpretación de $cons$ es ahora la función continua $cons_A^{\#} : A \to L(A) \to L(A)$:

$$\begin{array}{lcl} cons_A^{\#}\ x\ \bot & = & \infty \\ cons_A^{\#}\ x\ \infty & = & \infty \\ cons_A^{\#}\ x\ X_\epsilon & = & (\{x\}^{\uparrow} \sqcap X)_\epsilon,\ \ X_\epsilon \in L(A) \end{array}$$

Consideramos las funciones definidas en función de la función $case$:

$$g\ ys = case(a, f, ys)$$

donde

$$\begin{array}{lcl} case(a, f, nil) & = & a \\ case(a, f, cons\ x\ xs) & = & f\ x\ xs \end{array}$$

cuya interpretación estándar es:

$$\begin{array}{lcl} case(a, f, \bot) & = & \bot \\ case(a, f, nil) & = & a \\ case(a, f, cons\ x\ xs) & = & f\ x\ xs \end{array}$$

El uso del $case$ es equivalente a la definición por casos de Wadler en la Sección 5.4.2.

La interpretación abstracta del $case$ es la función continua $case^{\#}$:

$$\begin{array}{lcl} case^{\#}(b, f, \bot) & = & \bot \\ case^{\#}(b, f, \infty) & = & f\ \top\ \infty \\ case^{\#}(b, f, \{\top\}_\epsilon) & = & b \sqcup (f\ \top\ \{\top\}_\epsilon) \\ case^{\#}(b, f, Y_\epsilon) & = & \bigsqcup_{Y' \in U_Y} (f\ (\sqcap Y')\ (Y \ominus Y')_\epsilon) \quad \text{si } Y \neq \{\top\} \end{array}$$

donde $U_Y = \{Y' \mid Y' \subseteq Y, Y' \text{ abierto y no vacío}\}$, y para cada X:

$$Y \ominus X = (Y - X^{\uparrow})^{\uparrow} \cup \{\top\}$$

La última ecuación tiene la siguiente motivación: si Y_ϵ representa a la lista xs, entonces podemos suponer que la lista es de la forma $cons\ x\ xs'$. Si el conjunto abierto $Y' \subseteq Y$ está formado por los puntos que describen a x, entonces $\sqcap Y'$ también describe a x (ya que si $abs(x) \sqsubseteq y$ para cada $y \in Y'$ entonces $abs(x) \sqsubseteq \sqcap Y'$). Si $y \in Y - Y'$ entonces, por definición de Y', y no representa a x. Pero Y_ϵ representa a $cons\ x\ xs'$, luego y debe representar a algún miembro de xs'. Por tanto $(Y \ominus Y')_\epsilon$ debe representar a xs'.

Seguridad y corrección

La función de abstracción sobre las listas $abs : D^* \rightarrow L(A)$ viene dada por:

$$abs(xs) = \begin{cases} \bot & \text{si } xs = \bot \\ \{\top\}_\epsilon & \text{si } xs = [] \\ \infty & \text{si xs es infinita o parcial} \\ \{abs(x_i) \mid 1 \leq i \leq n\}_\epsilon^\uparrow & \text{si } xs = [x_1, \ldots, x_n] \end{cases}$$

Es continua, estricta, refleja el $\bot$ y además $abs(case) \sqsubseteq case^\#$.

Veamos un ejemplo en el que el resultado de este análisis es más preciso que el de Wadler.

Ejemplo 41 Sea la siguiente función:

$$\begin{array}{lcl} g\ nil & = & 0 \\ g\ (cons\ x\ xs) & = & \mathbf{if}\ c\ ((fst\ x) + (g\ xs))\ ((snd\ x) + (g\ xs)) \end{array}$$

Supongamos que $c^\# = 1$

- El análisis de Wadler de g produce la función:

$$g^\# z = \begin{cases} 0 & \text{si } z = \bot \text{ ó } z = \infty \\ 1 & \text{e.o.c} \end{cases}$$

  y por tanto $g^\#(0,0)_\epsilon = 1$. El análisis no es capaz de detectar que si algún par de la lista tiene sus dos componentes indefinidas, entonces no termina.

- Sin embargo, el análisis de los Nielson produce la siguiente función:

$$g^\# z = \begin{cases} 0 & \text{si } z = \bot \text{ ó } z = \infty \text{ ó } z = \{(0,0)\}_\epsilon^\uparrow \\ 1 & \text{e.o.c} \end{cases}$$

  y por tanto $g^\#\{(0,0)\}_\epsilon^\uparrow = 0$.

□

Una forma alternativa de presentar este análisis es mediante el uso del producto tensorial en la interpretación del producto junto con una versión inversa de *cons* llamada *split*. De esta forma, se puede evitar que las definiciones de las funciones tengan que ser por casos y basta utilizar el condicional.

La definición de la función *split* es la siguiente:

$$split\ xs = \begin{cases} cross(\bot, \bot) & \text{si } xs = \bot \\ cross(\top, \infty) & \text{si } xs = \infty \\ \bigsqcup\{cross(\sqcap Y', (Y \ominus Y')_\epsilon) \mid Y' \subseteq Y\} & \text{si } xs = Y_\epsilon \neq \top_\epsilon \\ cross(\top, \top_\epsilon) & \text{si } xs = \top_\epsilon \end{cases}$$

Como esta función ya lleva a cabo el análisis por casos, basta con usar el operador de punto fijo y el condicional para definir las funciones. Por ejemplo, la función *sum*, antes definida por casos, se podría definir usando *split* de la siguiente manera:

$$sum = fix(\lambda f.cond(isnil, zero, add \circ \mathbf{pair}(id, f) \circ split))$$

donde $cond\ f\ g\ h\ xs = \mathbf{if}\ (f\ xs)\ (g\ xs)\ (h\ xs)$, *isnil* es una función que dice si una lista es vacía o no, *zero* devuelve el número 0 al aplicarla a una lista, $\mathbf{pair}(f, g)$ aplicado a un par devuelve el par que resulta de aplicar f a la primera componente y g a la segunda, y *add* suma las componentes de un par.

5.5 Polimorfismo

5.5.1 Introducción

La mayoría de los lenguajes funcionales tienen sistemas de tipos polimórficos. Los ejemplares monomórficos de una función polimórfica se pueden analizar usando los métodos definidos en los capítulos anteriores. En esta sección vamos a estudiar cómo tratar el polimorfismo.

Los cuerpos de todos los ejemplares de una función polimórfica son esencialmente los mismos. La semejanza a nivel sintáctico debería implicar alguna relación semántica entre los distintos ejemplares.

Los métodos de análisis que pretenden tratar el polimorfismo intentan establecer conexiones entre las propiedades e interpretación abstracta de los distintos ejemplares.

Una forma simple de ver la semántica de una función polimórfica es considerarla como una colección de funciones continuas, donde cada una de ellas corresponde a un ejemplar monomórfico. Esta visión, sin embargo, no establece ninguna relación entre los ejemplares.

La relación entre los ejemplares puede establecerse dando un modelo categórico al lenguaje funcional, en el que los objetos son dominios, correspondientes a los tipos monomórficos, los constructores de tipos son funtores, y las funciones polimórficas transformaciones naturales o alguna variante de ellas (ver Sección 5.5.3). La relación entre las distintas componentes de la transformación natural define la relación semántica entre los distintos ejemplares de la función.

Los morfismos de la categoría varían de unos modelos a otros dependiendo del lenguaje y del análisis con el que se esté tratando. Pueden ser funciones estrictas si el lenguaje solamente es de primer orden, pares inmersión–proyección si se trata del λ–cálculo polimórfico, o pares inmersión–clausura, que resultan más convenientes que los anteriores para llevar a cabo una interpretación abstracta, como veremos más adelante.

5.5.2 Antecedentes

Invarianza polimórfica

Abramsky [Abr86] demostró que el análisis de estrictez de un lenguaje con sistema de tipos Hindley–Milner es **polimórficamente invariante**.

Esto significa que el análisis puede detectar la estrictez de un ejemplar de una función polimórfica si y solamente si puede detectar la estrictez de todos los ejemplares. Por tanto, basta con trabajar con el ejemplar más pequeño de la función.

Para cada análisis que se desee realizar es necesario demostrar que es polimórficamente invariante. Por esto, y porque la invarianza polimórfica se deduce de resultados posteriores más potentes, no se presenta con detalle este enfoque.

Funciones de primer orden

Hughes [Hug89] investigó una interpretación abstracta de funciones polimórficas de primer orden, obteniendo una aproximación a la interpretación abstracta de cualquier ejemplar de una función polimórfica a partir del más pequeño.

Para ello proporcionó al lenguaje funcional un modelo categórico, en el que los objetos son dominios que interpretan a los tipos monomórficos, y los morfismos son funciones continuas estrictas. En este modelo, los constructores de tipo se pueden interpretar como funtores sobre esta categoría y **las funciones polimórficas como transformaciones naturales**. Las componentes de dichas transformaciones naturales son la semántica de los ejemplares monomórficos de la función polimórfica. De esta forma, los ejemplares están relacionados de cierta manera.

Por ejemplo, sean dos tipos con variable de tipo t, $F(t)$ y $G(t)$, y sea f la transformación natural de F a G que es semántica de una función polimórfica $\mathbf{f} : F(t) \to G(t)$. Entonces para cualesquiera dominios A y B, si $\alpha : A \to B$ es una función estricta que relaciona dos dominios monomórficos se cumple:

$$G(\alpha) \circ f_A = f_B \circ F(\alpha)$$

es decir, el siguiente diagrama conmuta:

$$\begin{tikzcd} F(A) \arrow[r, "f_A"] \arrow[d, "F(\alpha)"'] & G(A) \arrow[d, "G(\alpha)"] \\ F(B) \arrow[r, "f_B"] & G(B) \end{tikzcd}$$

Es decir, los ejemplares $f_A : F(A) \to G(A)$ y $f_B : F(B) \to G(B)$ están relacionados en base a una función α que relacione A y B.

Además, Hughes demostró que para cualquier tipo monomórfico τ, existe un término $E(f_\tau)$ tal que:

$$f_{Abs\ \tau} = E(f_{\mathbf{2}})$$

donde $Abs\ \tau$ es el tipo obtenido de τ sustituyendo las apariciones de los tipos básicos (enteros y booleanos) por $\mathbf{2}$ y $f_{\mathbf{2}}$ es aquel ejemplar obtenido sustituyendo las variables de tipo por el dominio $\mathbf{2}$. A partir de lo cual se demostró que

$$f_\tau^{\#} \sqsubseteq E'(f_{\mathbf{2}}^{\#})$$

donde E' se obtiene a partir de E sustituyendo los términos por sus valores abstractos.

Es decir, $E'(f_{\mathbf{2}}^{\#})$ es una aproximación a la interpretación abstracta del ejemplar f_τ.

De esta forma, se puede calcular una aproximación a la interpretación abstracta de cualquiera de los ejemplares de una función polimórfica a partir de la interpretación abstracta de su ejemplar más pequeño.

Aparecen problemas con el orden superior debidos a la contravarianza en su primer argumento del funtor correspondiente al constructor de funciones.

Puesto que a continuación vamos a desarrollar con más detalle el caso de orden superior, no se proporcionan más detalles sobre este trabajo.

Invarianza polimórfica semántica

Existe un resultado más potente que la invarianza polimórfica, demostrado por Abramsky y Jensen [AJ91], que es la **invarianza polimórfica semántica** de la estrictez de un lenguaje de orden superior.

Ésta consiste en demostrar que un ejemplar de una función polimórfica *es* estricta si y solo si todos los ejemplares lo son. De esta forma, si una técnica de análisis detecta que un ejemplar es estricto, entonces todos los ejemplares deben serlo. Pero esto ocurre incluso para aquellos ejemplares cuya estrictez no puede ser detectada por dicho análisis. Recordemos que la invarianza polimórfica vista anteriormente habla solamente de detección. La invarianza polimórfica semántica se deriva del análisis de Baraki desarrollado en la siguiente sección, por lo que no proporcionamos más detalles.

5.5.3 El polimorfismo de Baraki

Introducción

Baraki [Bar93] se dió cuenta de que cuando se llama a distintos ejemplares de una misma función polimórfica en distintos puntos del programa, la invarianza polimórfica puede no ser muy útil, ya que para obtener el máximo de información posible, es necesario calcular las funciones abstractas de todos esos ejemplares. Por ejemplo, sea la función:

$$f\ x\ y = \mathbf{if}\ (x = 0)\ y\ (f\ (x - 1)\ y)$$

El tipo Hindley–Milner de f es $\forall a.\mathbf{int} \to a \to a$, luego la interpretación abstracta del ejemplar más pequeño de f es una función en $\mathbf{2} \to \mathbf{2} \to \mathbf{2}$, dada posiblemente mediante una tabla.

Sea ahora la función:

$$g\ x = \mathbf{fst}\ (f\ 0\ x)$$

donde **fst** es la función que devuelve la primera componente de un par. Para calcular la interpretación abstracta de g necesitamos la de algún ejemplar de f, que no es precisamente el más pequeño. Por tanto, la tabla de que disponemos no sirve de mucho, sino que debemos construir otra más para el ejemplar adecuado. Esto resulta bastante ineficiente.

Es deseable, por tanto, usar la información obtenida de un ejemplar para estudiar propiedades de otros.

Funciones de orden superior

Para trabajar con orden superior, Baraki [Bar93] trabajó con la categoría de los dominios y pares inmersión–clausura C^{ec}, y en concreto en la categoría de los retículos finitos y los pares inmersión–clausura A^{ec}. Todas las demostraciones de los teoremas y proposiciones de esta sección se pueden encontrar en [Bar93].

Coquand et al [CGW89] proporcionaron una semántica a las funciones polimórficas utilizando la categoría de los dominios y los pares inmersión–proyección, pero en este caso se obtienen aproximaciones desde abajo, y no desde arriba como deseamos. Por ello se prefieren los pares inmersión–clausura.

Semántica de un lenguaje polimórfico

El polimorfismo más habitual en los lenguajes funcionales tales como ML y Miranda es el polimorfismo de Hindley–Milner. Normalmente las expresiones no necesitan tiparse explícitamente, y sus tipos se infieren en tiempo de compilación. Sin embargo, para utilizar la información de tipos en los análisis, a continuación se presenta un lenguaje en el que ciertas informaciones de tipos son explícitas, en particular las variables y las funciones polimórficas **if**, **fix**, **fst** y **snd**. Esto no representa ninguna restricción, pues los tipos se pueden inferir en tiempo de compilación y etiquetar con ellos las expresiones posteriormente.

La sintaxis del lenguaje

Las **expresiones de tipo monomórficas** se definen mediante la gramática:

$$\tau ::= t \mid b \mid \tau \times \tau \mid \tau \to \tau$$

donde t denota una variable de tipo, b un tipo básico (como **int** o **bool**). Los tipos polimórficos reciben el nombre de **esquemas de tipo**:

$$\sigma ::= \tau \mid \forall t.\sigma$$

Son de la forma general $\forall t_1.\ldots.\forall t_n.\tau$ (no son cerrados bajo $\times$ ni $\to$).

Los términos del lenguaje son:

$$\begin{array}{rl} E ::= & x \mid c \mid \lambda x:\tau.E \mid (E\ E) \mid \mathbf{if}_\tau\ E\ E\ E \mid \mathbf{fix}_\tau\ E \\ & \mid\ <E,E>\ \mid \mathbf{fst}_{\tau\times\tau}\ E \mid \mathbf{snd}_{\tau\times\tau}\ E \\ & \mid \mathbf{let}\ x:\sigma = E\ \mathbf{in}\ E \mid \Lambda t.E \mid E[\tau] \end{array}$$

donde $\Lambda t.E$ representa una abstracción de tipo monomórfico, y $E[\tau]$ la aplicación de una expresión a un tipo monomórfico. Estas dos últimas expresiones nos permiten definir funciones polimórficas y ejemplares de las mismas. Solamente se permiten términos bien tipados, según las reglas de chequeo de tipos habituales. Como es usual, $\Gamma \vdash E : \sigma$ denota que usando dichas reglas se demuestra que la expresión E tiene tipo σ bajo la asignación Γ de tipos a variables de tipo.

Semántica

Semántica de los tipos

La semántica de los tipos monomórficos se especifica normalmente asociando a cada uno un dominio D_τ. Se asocia a cada tipo base b un dominio D_b, por ejemplo $\mathbb{Z}_\perp$ a **int** y $Bool_\perp = \{\perp, True, False\}$ a **bool**.

El dominio asociado a $\tau_1 \times \tau_2$ suele ser $D_{\tau_1 \times \tau_2} = D_{\tau_1} \times D_{\tau_2}$ (en ciertos análisis a veces conviene darle otra semántica al producto, como por ejemplo el producto tensor, ver Sección 5.4.1), y el asociado a $\tau_1 \to \tau_2$ es $D_{\tau_1 \to \tau_2} = [D_{\tau_1} \to D_{\tau_2}]$.

La semántica de un tipo τ con variables de tipo depende de los dominios asociados a dichas variables. Por ello necesitamos una asignación de dominios.

La función ζ proporciona la semántica de los tipos monomórficos, donde S es una asignación de dominios:

$$\begin{array}{lcl}
\zeta[\![b]\!]\ S & = & D_b \\
\zeta[\![t]\!]\ S & = & S(t) \\
\zeta[\![\tau_1 \times \tau_2]\!]\ S & = & (\zeta[\![\tau_1]\!]\ S) \times (\zeta[\![\tau_2]\!]\ S) \\
\zeta[\![\tau_1 \to \tau_2]\!]\ S & = & [(\zeta[\![\tau_1]\!]\ S) \to (\zeta[\![\tau_2]\!]\ S)]
\end{array}$$

Los constructores de tipo se interpretan por tanto como funtores sobre la categoría de los dominios y pares inmersión–clausura.

Semántica de los términos

El significado de un término depende de los valores, del dominio correspondiente a su tipo, que toman las variables libres de ese término. Este dominio depende a su vez de la asignación de dominios a las variables de tipo.

Por ello para dar semántica a un término hay que proporcionar una asignación de dominios S y un entorno de valores $\eta : Var \to +D_\tau$, que debe ser consistente con la asignación de tipos a las variables, representado por Γ.

Para cada asignación de tipos Γ y tipo τ, llamamos $A_{\Gamma,\tau}$ al conjunto de términos E tales que $\Gamma \vdash E : \tau$.

Sobre este conjunto se define la función $\mu_{\Gamma,\tau}$, que da semántica a un término. $\mu_{\Gamma,\tau}[\![E]\!]\ S\ \eta$ nos da un valor en $\zeta[\![\tau]\!]\ S$. Suponemos que para cada variable x, $\eta(x) \in \zeta[\![\Gamma(x)]\!]\ S$

Suponemos también que para cada constante de tipo base, la función α_b nos da la correspondiente interpretación: $\alpha_b : K_b \to D_b$ (donde K_b es el conjunto de todas las constantes de tipo b).

La función semántica se muestra en la Figura 5.17. La semántica de $f = \Lambda t_1 \ldots \Lambda t_n.E$ con tipo $\forall t_1 \ldots \forall t_n.\tau$ es la colección $\{f_{D_1 \ldots D_n}\}$ donde $f_{D_1 \ldots D_n} = \mu_{\Gamma,\tau}[\![E]\!]\ S\ \eta$ para Γ, η y S tal que $S(t_i) = D_i$.

Semántica relacional

Hemos visto que la semántica de una función polimórfica suele considerarse como una colección de funciones continuas, cada una de las cuales representa la semántica de un ejemplar

- Si $k \in K_b$ entonces $\mu_{\Gamma,\tau}[\![k]\!]\ S\ \eta = \alpha_b(k)$
- Si $x \in dom(\Gamma)$ entonces $\mu_{\Gamma,\tau}[\![x]\!]\ S\ \eta = \eta(x)$
- Si $E_1 \in A_{\Gamma,\tau\to\tau'}$ y $E_2 \in A_{\Gamma,\tau}$ entonces

$$\mu_{\Gamma,\tau}[\![(E_1\ E_2)]\!]\ S\ \eta = (\mu_{\Gamma,\tau}[\![E_1]\!]\ S\ \eta)(\mu_{\Gamma,\tau}[\![E_2]\!]\ S\ \eta)$$

- Si $E \in A_{(\Gamma,x:\tau),\tau'}$ entonces

$$\mu_{\Gamma,\tau}[\![\lambda x:\tau.E]\!]\ S\ \eta = \lambda d \in (\zeta[\![\tau]\!]\ S).\mu_{(\Gamma,x:\tau),\tau'}[\![E]\!]\ S\ (\eta[d/x])$$

- Si $E_1 \in A_{\Gamma,\tau}$ y $E_2 \in A_{\Gamma,\tau'}$ entonces

$$\mu_{\Gamma,\tau}[\![< E_1, E_2 >]\!]\ S\ \eta = (\mu_{\Gamma,\tau}[\![E_1]\!]\ S\ \eta, \mu_{\Gamma,\tau'}[\![E_2]\!]\ S\ \eta)$$

- Si $E \in A_{\Gamma,\tau\times\tau'}$ entonces

$$\mu_{\Gamma,\tau}[\![\mathbf{fst}_{\tau\times\tau'}\ E]\!]\ S\ \eta = \mathbf{fst}\ (\mu_{\Gamma,\tau}[\![E]\!]\ S\ \eta)$$

$$\mu_{\Gamma,\tau}[\![\mathbf{snd}_{\tau\times\tau'}\ E]\!]\ S\ \eta = \mathbf{snd}\ (\mu_{\Gamma,\tau}[\![E]\!]\ S\ \eta)$$

- Sea $c = \mu_{\Gamma,\mathbf{bool}}[\![E_0]\!]\ S\ \eta$ entonces

$$\mu_{\Gamma,\tau}[\![\mathbf{if}_\tau\ E_0\ E_1\ E_2]\!]\ S\ \eta = \begin{cases} \mu_{\Gamma,\tau}[\![E_1]\!]\ S\ \eta & \text{si } c = true \\ \mu_{\Gamma,\tau}[\![E_2]\!]\ S\ \eta & \text{si } c = false \\ \bot & \text{si } c = \bot \end{cases}$$

- $\mu_{\Gamma,\tau}[\![\mathbf{fix}_\tau\ E]\!]\ S\ \eta = \sqcup_{n=0}^{\infty}(\mu_{\Gamma,\tau}[\![E]\!]\ S\ \eta)^n(\bot)$
- $\mu_{\Gamma,\tau}[\![\mathbf{let}\ x:\sigma = E_1\ \mathbf{in}\ E_2]\!]\ S\ \eta = \mu_{\Gamma,\tau}[\![E_2[E_1/x]]\!]\ S\ \eta$
- Si $\Gamma \vdash \Lambda t_1 \ldots \Lambda t_n.E : \forall t_1 \ldots \forall t_n.\tau$ entonces para $\tau_1, \ldots, \tau_n$:

$$\mu_{\Gamma,\tau}[\![(\Lambda t_1 \ldots \Lambda t_n.E)[\tau_1]\ldots[\tau_n]]\!]\ S\ \eta = \mu_{\Gamma,\tau}[\![E]\!]\ S[D_1/t_1]\ldots[D_n/t_n]\ \eta$$

  donde $D_i = \zeta[\![\tau_i]\!]\ S$.

Figura 5.17: Semántica de los términos

monomórfico. Si se desea establecer alguna relación entre los ejemplares, se debe proporcionar una semántica relacional.

Llamamos **representación** entre dos dominios D y D' a un par (Φ, Ψ) de funciones continuas $\Phi : D \to D'$, $\Psi : D' \to D$ que satisfacen:

$$\begin{array}{c} \Psi \circ \Phi \sqsupseteq id \\ \Phi \circ \Psi \sqsubseteq id \end{array}$$

Un elemento $d \in D$ representa a otro elemento $d' \in D'$ si $d \sqsubseteq \Psi(d')$ (o equivalentemente si $\Phi(d) \sqsubseteq d'$). En nuestro caso vamos a trabajar con pares inmersión–clausura (h^e, h^c), ver Sección 2.4, que son un caso particular de representación entre dominios.

Por tanto, diremos que dos elementos $d \in D$ y $d' \in D'$ están relacionados a través de $h : D \to^{ec} D'$ si $h^e(d) \sqsupseteq d'$ (o equivalentemente si $d \sqsupseteq h^c(d')$).

La semántica de un término depende de los dominios asignados a sus variables de tipo libres, y de los valores asignados a sus variables libres.

Queremos establecer una relación entre $\mu_{\Gamma,\tau}[\![E]\!]\ S_1\ \eta_1$ y $\mu_{\Gamma,\tau}[\![E]\!]\ S_2\ \eta_2$. Estos valores pertenecen a los espacios $\zeta[\![\tau]\!]\ S_1$ y $\zeta[\![\tau]\!]\ S_2$ respectivamente. Para relacionarlos, supongamos que tenemos pares inmersión–clausura entre $S_1(t)$ y $S_2(t)$ para toda variable de tipo t. Entonces decimos que los entornos η_1 y η_2 están relacionados si para cada x, $\eta_1(x)$ y $\eta_2(x)$ están relacionados a través de los pares inmersión–clausura entre los espacios asociados con el tipo de x, es decir si para todo $x \in dom(\Gamma)$:

$$h^e_{\Gamma(x)}(\eta_1(x)) \sqsupseteq \eta_2(x)$$

donde recordemos que $\eta_1(x) \in \zeta[\![\Gamma(x)]\!]\ S_1$, $\eta_2(x) \in \zeta[\![\Gamma(x)]\!]\ S_2$ y $h^e_{\Gamma(x)} : \zeta[\![\Gamma(x)]\!]\ S_1 \to \zeta[\![\Gamma(x)]\!]\ S_2$.

Supongamos que ρ es una asignación de pares inmersión–clausura a variables de tipo, en lugar de dominios, de forma que para cada t, $\rho(t)$ es un par inmersión–clausura de $S_1(t)$ a $S_2(t)$. Para cada tipo τ podemos extender ρ a un par inmersión–clausura de $\zeta[\![\tau]\!]\ S_1$ a $\zeta[\![\tau]\!]\ S_2$, denotada por $\zeta[\![\tau]\!]\rho$:

$$\begin{array}{lcl} \zeta[\![b]\!]\ \rho & = & id_{D_b} \\ \zeta[\![t]\!]\ \rho & = & \rho(t) \\ \zeta[\![\tau_1 \times \tau_2]\!]\ \rho & = & (\zeta[\![\tau_1]\!]\ \rho) \times (\zeta[\![\tau_2]\!]\ \rho) \\ \zeta[\![\tau_1 \to \tau_2]\!]\ \rho & = & (\lambda k.(\zeta[\![\tau_1]\!]\ \rho)^e \circ k \circ (\zeta[\![\tau_1]\!]\ \rho)^c, \lambda k.(\zeta[\![\tau_1]\!]\ \rho)^c \circ k \circ (\zeta[\![\tau_1]\!]\ \rho)^e) \end{array}$$

El resultado principal es el **teorema de representación**:

Teorema 42 *(teorema de representación) Sean S_1 y S_2 dos asignaciones de dominio, y Γ una asignación de tipos. Sean η_1 y η_2 entornos. Supongamos que para cada variable de tipo t disponemos de un par inmersión–clausura $\rho(t) : S_1(t) \to^{ec} S_2(t)$. Y llamemos h_τ a $\zeta[\![\tau]\!]\ \rho$.*

Si η_1 y η_2 están relacionados, entonces para cada fórmula $\Gamma \vdash E : \tau$ se tiene:

$$h^e_\tau(\mu_{\Gamma,\tau}[\![E]\!]\ S_1\ \eta_1) \sqsupseteq \mu_{\Gamma,\tau}[\![E]\!]\ S_2\ \eta_2$$

Consecuencias del teorema de representación

La primera consecuencia del teorema de representación es la relación entre los ejemplares de una función polimórfica.

Nos interesa el caso particular en que la expresión es de la forma $\Gamma \vdash \Lambda t.E : \forall t.\tau$ (donde t no está libre en Γ), ya que deseamos relacionar la semántica de $(\Lambda t.E)[\tau_1]$ y $(\Lambda t.E)[\tau_2]$, dados ciertos S y η.

La relación entre estos ejemplares depende de la relación entre τ_1 y τ_2, establecida mediante un par inmersión–clausura $h : \zeta[\![\tau_1]\!]\ S \to \zeta[\![\tau_2]\!]\ S$.

Sean:

$$S_i(s) = \begin{cases} \zeta[\![\tau_i]\!]\ S & \text{si } s = t \\ S(s) & e.o.c \end{cases}$$

para $i = 1, 2$, y

$$\rho(s) = \begin{cases} h & \text{si } s = t \\ id & e.o.c \end{cases}$$

Puesto que t no está libre en Γ, $h_{\Gamma(x)} = id$ para cada x (pues $h_{\Gamma(x)} : \zeta[\![\Gamma(x)]\!]\ S_1 \to \zeta[\![\Gamma(x)]\!]\ S_1$ y $\zeta[\![\Gamma(x)]\!]\ S_i = S_i(\Gamma(x)) = S(\Gamma(x))$ para $i = 1, 2$), luego η está relacionado consigo mismo y por tanto, por el teorema de representación:

$$h^e_\tau(\mu_{\Gamma,\tau}[\![E]\!]\ S_1\ \eta) \sqsupseteq \mu_{\Gamma,\tau}[\![E]\!]\ S_2\ \eta$$

Y por tanto, como $\mu_{\Gamma,\tau}[\![E]\!]\ S_i\ \eta = \mu_{\Gamma,\tau[\tau_i/t]}[\![(\Lambda t.E)[\tau_i]]\!]\ S\ \eta$, entonces:

$$h^e_\tau(\mu_{\Gamma,\tau[\tau_1/t]}[\![(\Lambda t.E)[\tau_1]]\!]\ S\ \eta) \sqsupseteq \mu_{\Gamma,\tau[\tau_2/t]}[\![(\Lambda t.E)[\tau_2]]\!]\ S\ \eta$$

Los constructores de tipo como funtores

Hemos visto que la semántica de un tipo monomórfico τ es $\zeta[\![\tau]\!]S$. Veamos cual es la semántica de un tipo polimórfico cerrado $\forall t_1 \ldots t_n.\tau$. Los tipos polimórficos cerrados se interpretan como funtores sobre el producto de la categoría de dominios y pares inmersión–clausura.

Sean S_1 y S_2 dos asignaciones de dominio y supongamos que para cada variable de tipo t, $\rho(t)$ es un par inmersión–clausura de $S_1(t)$ a $S_2(t)$.

Sea el funtor de $(C^{ec})^n$ a C^{ec} que:

- Toma una tupla de dominios $(D_1, \ldots, D_n)$ y devuelve $\zeta[\![\tau]\!]\ S[D_1/t_1, \ldots, D_n/t_n]$
- Dada una tupla $(h_1, \ldots, h_n)$ de pares inmersión–clausura donde $h_i : D_i \to^{ec} D'_i$ para cada i, devuelve $\zeta[\![\tau]\!]\ \rho'$ donde:

$$\rho'(t_i) = \begin{cases} h_i & \text{para cada } i : 1 \leq i \leq n \\ \rho(t) & e.o.c. \end{cases}$$

Más claramente y definido por inducción, el funtor es el siguiente:

$$\begin{aligned}
F[\![\forall t_1 \ldots t_n.b]\!] &= Id \\
F[\![\forall t_1 \ldots t_n.t_i]\!] &= Sel_i \\
F[\![\forall t_1 \ldots t_n.\tau_1 \times \tau_2]\!] &= (F[\![\forall t_1 \ldots t_n.\tau_1]\!]) \times (F[\![\forall t_1 \ldots t_n.\tau_2]\!]) \\
F[\![\forall t_1 \ldots t_n.\tau_1 \to \tau_2]\!] &= (F[\![\forall t_1 \ldots t_n.\tau_1]\!]) \to (F[\![\forall t_1 \ldots t_n.\tau_2]\!])
\end{aligned}$$

donde Sel_i es el funtor que selecciona el dominio y el morfismo i–ésimo de una n–tupla.

Funciones polimórficas como transformaciones naturales laxas

El hecho de que las funciones polimórficas sean transformaciones naturales laxas es otra consecuencia del teorema de representación. Supongamos que

$$\Gamma \vdash \Lambda\vec{t}.E : \forall\vec{t}.F(\vec{t}) \to G(\vec{t})$$

donde $\Lambda\vec{t}.E$ es un término cerrado.

Dada una tupla de dominios $(D_1, \ldots, D_n)$ la semántica del ejemplar correspondiente a esa tupla viene dada por:

$$f_{D_1 \ldots D_n} = \mu_{\Gamma, F(\vec{t}) \to G(\vec{t})} [\![E]\!]\ S[D_i/t_i]\eta$$

Si para cada i, D_i es la semántica de τ_i entonces:

$$f_{D_1 \ldots D_n} = \mu_{\Gamma, F(\tau_1, \ldots, \tau_n) \to G(\tau_1, \ldots, \tau_n)} [\![(\Lambda\vec{t}.E)[\tau_1] \ldots [\tau_n]]\!]\ S[D_i/t_i]\ \eta$$

Si para cada i tenemos un par inmersión–clausura $h_i : D_i \to^{ec} D'_i$, el teorema de representación dice que:

$$(F \to G)^e(h_1, \ldots, h_n)(f_{D_1, \ldots, D_n}) \sqsupseteq f_{D'_1, \ldots, D'_n}$$

y por tanto:

$$\begin{array}{lll} & (F \to G)^e(h_1, \ldots, h_n)(f_{D_1, \ldots, D_n}) & \sqsupseteq f_{D'_1, \ldots, D'_n} \\ \Rightarrow & (\lambda k. G^e(\vec{h}) \circ k \circ F^c(\vec{h}))(f_{D_1, \ldots, D_n}) & \sqsupseteq f_{D'_1, \ldots, D'_n} \\ \Rightarrow & G^e(\vec{h}) \circ f_{D_1, \ldots, D_n} \circ F^c(\vec{h}) & \sqsupseteq f_{D'_1, \ldots, D'_n} \quad (*) \\ \Rightarrow & G^e(\vec{h}) \circ f_{D_1, \ldots, D_n} & \sqsupseteq f_{D'_1, \ldots, D'_n} \circ F^e(\vec{h}) \end{array}$$

es decir, $\{f_{D_1, \ldots, D_n}\}$ es una transformación natural laxa de F a G.

La desigualdad (*) nos proporciona una aproximación a la semántica de un ejemplar de una función polimórfica a partir de la semántica de otro ejemplar en algún sentido más pequeño (pues las $h_i : D_i \to D'_i$).

En el caso de primer orden, las funciones polimórficas resultan ser transformaciones naturales, usando solamente las clausuras como morfismos.

Y tal como vimos en la Sección 2.4.4, si trabajamos con retículos finitos, tenemos:

$$f_{A_1 \ldots A_n} \sqsubseteq \sqcap G^e(h_1, \ldots, h_n) \circ f_{\mathbf{2} \ldots \mathbf{2}} \circ F^c(h_1, \ldots, h_n)$$

donde $\sqcap$ se toma sobre todas las tuplas posibles de n morfismos $(h_1, \ldots, h_n)$ con $h_i : \mathbf{2} \to^{ec} A_i$ para cada i. En primer orden además se da la igualdad.

Como consecuencia de esto, a partir de la semántica definida se puede derivar la invarianza polimórfica semántica. Si A es un retículo con $\top$, entonces existe un cierto morfismo $h : \mathbf{2} \to^{ec} A$ según vimos en la Sección 2.4.4. Podemos elegirla de forma que h^e sea estricta. Si h^e es estricta, entonces $G^e(h)$ es estricta. Luego, si una función f es estricta en el ejemplar $\mathbf{2}$ entonces $f_{\mathbf{2}}(\bot) = \bot$. Y por tanto, puesto que $F^c(h)$ siempre es estricta, se cumple que $f_A(\bot) = \bot$. Es decir, si una función es estricta en el ejemplar $f_{\mathbf{2}}$, también lo es en cualquier otro ejemplar. Se puede demostrar la implicación contraria utilizando pares inmersión–proyección.

Un ejemplo de aplicación: análisis de estrictez

De forma semejante a la semántica estándar proporcionada para un término del lenguaje $\mu_{\Gamma,\tau}[\![E]\!]\ S\ \eta$, se define una semántica abstracta $\nu_{\Gamma',Abs\ \tau}[\![E']\!]\ S'\ \eta'$ donde:

- Abs hace corresponder a un tipo ordinario, un tipo cuyo único tipo base es $\mathbf{2}$:

$$\begin{array}{lcl} Abs\ \mathbf{int} & = & \mathbf{2} \\ Abs\ \mathbf{bool} & = & \mathbf{2} \\ Abs\ t & = & t \\ Abs\ (\tau_1 \times \tau_2) & = & (Abs\ \tau_1) \times (Abs\ \tau_2) \\ Abs\ (\tau_1 \to \tau_2) & = & (Abs\ \tau_1) \to (Abs\ \tau_2) \\ Abs\ (\forall t.\sigma) & = & \forall t.(Abs\ \sigma) \end{array}$$

  Es decir, cada aparición de **int** y **bool** se sustituye por $\mathbf{2}$.

- Si $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ entonces $\Gamma' = \{x_1 : Abs\ \sigma_1, \ldots, x_n : Abs\ \sigma_n\}$.
- $S'(t) = Abs\ S(t)$ y $\eta'(x) = abs(\eta(x))$, donde abs es la función de abstracción entre los dominios semánticos concretos y los dominios semánticos abstractos proporcionada en apartados anteriores.
- E' se obtiene a partir de E mediante sustitución de cada subtérmino por su interpretación abstracta según visto en apartados anteriores.

Nos centramos en las funciones polimórficas. Sea un término cerrado $\Lambda\vec{t}.E$ con tipo $\forall\vec{t}.F(\vec{t}) \to G(\vec{t})$. Este término se puede traducir a un término $\Lambda\vec{t}.E'$ con tipo $\forall\vec{t}.F'(\vec{t}) \to G'(\vec{t})$ donde $F'(t)$ y $G'(t)$ se obtienen a partir de $F(t)$ y $G(t)$ sustituyendo **int** y **bool** por $\mathbf{2}$.

Entonces F' y G' se intrepretan como funtores de la categoría $(A^{ec})^n$ en A^{ec}, y la semántica de $\Lambda\vec{t}.E'$ es una transformación natural laxa de F' a G'.

Por tanto, supongamos que $\tau_1, \ldots, \tau_n, \tau'_1, \ldots, \tau'_n$ son tipos monomórficos, y sean B_i y B'_i los retículos finitos que son semántica de $Abs\ \tau_i$ y $Abs\ \tau'_i$ respectivamente. Y supongamos que para cada i, $h_i : B_i \to^{ec} B'_i$ es un par inmersión–clausura. Entonces, puesto que $\Lambda\vec{t}.E'$ es cerrado, el valor $\nu[\![\Lambda\vec{t}.E']\!]\ S\ \eta$ es independiente de S y η, y tenemos:

$$\nu[\![(\Lambda\vec{t}.E')[Abs\ \tau'_1]\ldots[Abs\ \tau'_n]]\!] \sqsubseteq G^e(\vec{h}) \circ \nu[\![(\Lambda\vec{t}.E')[Abs\ \tau_1]\ldots[Abs\ \tau_n]]\!] \circ F^c(\vec{h})$$

Puesto que la semántica de $\Lambda\vec{t}.E'$ se puede expresar como la colección $\{g_{B_1\ldots B_n}\}$ donde $g_{B_1\ldots B_n} = \nu[\![E']\!]\ S\ \eta$ y $S(t_i) = B_i$ para cada i, entonces si h_i es un morfismo de $\mathbf{2}$ a B_i:

$$g_{B_1\ldots B_n} \sqsubseteq G^e(\vec{h}) \circ g_{\mathbf{2}\ldots\mathbf{2}} \circ F^c(\vec{h})$$

Esto quiere decir que se puede usar $G^e(\vec{h}) \circ g_{\mathbf{2}\ldots\mathbf{2}} \circ F^c(\vec{h})$ en lugar de $g_{B_1\ldots B_n}$ en el análisis.

Supongamos que $n = 1$. Si elegimos $h : \mathbf{2} \to^{ec} B$ tal que h^e es estricta, entonces si $g_{\mathbf{2}}$ es estricta, también lo es $G^e(h) \circ g_{\mathbf{2}} \circ F^c(h)$, y por tanto g_B.

Un resultado aún mejor para aproximar g_B consiste en tomar la máxima cota inferior sobre todas las aproximaciones posibles:

$$g_B \sqsubseteq \sqcap G^e(h) \circ g_{\mathbf{2}} \circ F^c(h)$$

En el caso de primer orden se cumple la igualdad.

Ejemplo 43 Sea la función:

$$f = \Lambda\alpha.\mathbf{fix}_\tau\ (\lambda g : \tau.\lambda x : \mathbf{bool}.\lambda y : \mathbf{int}.\lambda z : \alpha.\mathbf{if}_\alpha\ x\ z\ (g\ x\ (y-1)\ z))$$

donde $\tau = \mathbf{bool} \to \mathbf{int} \to \alpha \to \alpha$.

Su interpretación abstracta sobre el ejemplar más pequeño es la función $f^\#_{\mathbf{2}}$ dada por la siguiente tabla:

x	y	z	$f^\#_{\mathbf{2}}\ x\ y\ z$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

la cual se obtiene de forma iterativa.

La interpretación abstracta de la función sobre el tipo $\mathbf{2}\times\mathbf{2}$, se puede aproximar:

$$f^\#_{\mathbf{2}\times\mathbf{2}} \sqsubseteq \sqcap_a(\lambda u.(\lambda l.h^e_a \circ l \circ h^c_a) \circ u \circ id^c) \circ f^\#_{\mathbf{2}} \circ id^c$$

donde $a \in \mathbf{2}\times\mathbf{2}$, y las h_a son las especificadas en la Sección 2.4.4.

Llamando

$$k_a = (\lambda u.(\lambda l.h^e_a \circ l \circ h^c_a) \circ u \circ id^c) \circ f^\#_{\mathbf{2}} \circ id^c$$

obtenemos que

$$k_a\ x\ y\ z = h^e_a\ (f^\#_{\mathbf{2}}\ x\ y\ (h^c_a\ z))$$

y por tanto podemos construir la siguiente tabla:

z	$k_{(0,0)}\ x\ y\ z$	$k_{(0,1)}\ x\ y\ z$	$k_{(1,0)}\ x\ y\ z$
$(0,0)$	$(0,0)$	$(0,1)$	$(1,0)$
$(0,1)$	$h^e_{(0,0)}(x)$	$(0,1)$	$h^e_{(1,0)}(x)$
$(1,0)$	$h^e_{(0,0)}(x)$	$h^e_{(0,1)}(x)$	$(1,0)$
$(1,1)$	$h^e_{(0,0)}(x)$	$h^e_{(0,1)}(x)$	$h^e_{(1,0)}(x)$

Se ve fácilmente que:

$$k_{(0,1)} \sqcap k_{(1,0)} \sqsubseteq k_{(0,0)}$$

Luego al tomar el ínfimo no hace falta tener en cuenta a $k_{(0,0)}$. Y obtenemos:

x	y	z	$f^{\#}_{\mathbf{2}\times\mathbf{2}}\ x\ y\ z$
0	0	$(0,0)$	$(0,0)$
0	0	$(0,1)$	$(0,0)$
0	0	$(1,0)$	$(0,0)$
0	0	$(1,1)$	$(0,0)$
0	1	$(0,0)$	$(0,0)$
0	1	$(0,1)$	$(0,0)$
0	1	$(1,0)$	$(0,0)$
0	1	$(1,1)$	$(0,0)$
1	0	$(0,0)$	$(0,0)$
1	0	$(0,1)$	$(0,1)$
1	0	$(1,0)$	$(1,0)$
1	0	$(1,1)$	$(1,1)$
1	1	$(0,0)$	$(0,0)$
1	1	$(0,1)$	$(0,1)$
1	1	$(1,0)$	$(1,0)$
1	1	$(1,1)$	$(1,1)$

$\square$

5.5.4 Aproximación de puntos fijos usando ensanchamiento y estrechamiento

En el apartado anterior hemos visto cómo calcular aproximaciones de la interpretación abstracta de un ejemplar de una función polimórfica a partir de la del ejemplar más pequeño.

Sin embargo, hay funciones monomórficas que no son ejemplares de ninguna función polimórfica. El cálculo de la interpretación abstracta de dichas funciones es costoso e implica el cálculo de puntos fijos.

Para tratar con estos casos Hankin y Hunt [HH92] introdujeron un método para el **cálculo de puntos fijos aproximados**, y demostraron su correspondencia con unos operadores llamados de ensanchamiento y estrechamiento presentado por los Cousot [Cou78][Cou81].

Ensanchamiento y estrechamiento

Los operadores de ensanchamiento y estrechamiento[2] se utilizan para asegurar la terminación de los análisis cuando los dominios abstractos poseen cadenas ascendentes infinitas, es decir para asegurar la convergencia, o bien para acelerar la misma en dominios grandes. El caso que nos interesa es el segundo.

El precio a pagar es la pérdida de la exactitud. Es decir, estas técnicas calculan solamente una aproximación al punto fijo.

Definición 44 *(operador de ensanchamiento) Dado un retículo completo L, un operador $\bigtriangledown \in \mathbb{N} \to ((L \times L) \to L)$ es un* ***operador de ensanchamiento*** *si satisface las siguientes condiciones:*

- $\forall j > 0, x, y \in L. x \sqcup y \sqsubseteq x \bigtriangledown(j)\ y$

[2]En inglés *widening* y *narrowing*

- *Para toda cadena* $x_0 \sqsubseteq x_1 \ldots \sqsubseteq x_n \sqsubseteq \ldots$ *en* L*, la cadena* $y_0 \sqsubseteq y_1 \sqsubseteq \ldots \sqsubseteq y_n \sqsubseteq \ldots$ *definida por* $y_0 = x_0, y_1 = y_0 \triangledown(1)\ x_1, \ldots y_n = y_{n-1} \triangledown(n)\ x_n$ *es eventualmente estable, es decir, existe un* $k \geq 0$ *tal que para todo* $i \geq k : y_i = y_k$.

Proposición 45 *Sea* f *una función monótona sobre* L *y* $\triangledown$ *un operador de ensanchamiento. El límite* u *de la siguiente secuencia:*

$$\begin{array}{lcll} x_0 & = & \bot & \\ x_{n+1} & = & x_n & si\ f(x_n) \sqsubseteq x_n \\ x_{n+1} & = & x_n \triangledown(n+1)\ f(x_n) & en\ otro\ caso \end{array}$$

se puede calcular en un número finito de pasos y $fix(f) \sqsubseteq u$ *y* $f(u) \sqsubseteq u$.

Es decir, si utilizamos un operador de ensanchamiento obtenemos una aproximación segura del punto fijo.

El proceso de iteración y su relación con la cadena de Kleene se ilustra en la Figura 5.18.

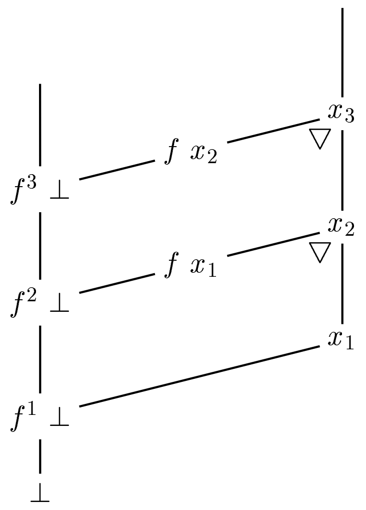


Figura 5.18: Relación entre la cadena de Kleene y la obtenida mediante el operador de ensanchamiento

Definición 46 *(operador de estrechamiento)*

Un operador $\triangle : \mathbb{N} \to ((L \times L) \to L)$ *es un* ***operador de estrechamiento*** *si y solo si satisface las siguientes condiciones:*

- $\forall j > 0. \forall x, y \in L. y \sqsubseteq x \Rightarrow y \sqsubseteq x \triangle(j)\ y \sqsubseteq x$

- *Para toda cadena* $x_0 \sqsupseteq x_1 \ldots \sqsupseteq x_n \sqsupseteq \ldots$ *en* L*, la cadena* $y_0 = x_0, y_1 = y_0 \triangle(1)\ x_1, \ldots y_n = y_{n-1} \triangle(n)\ x_n$ *es eventualmente estable, es decir, existe un* $k \geq 0$ *tal que para todo* $i \geq k. y_i = y_k$.

Proposición 47 *Sea f una función monótona sobre L y $\triangle$ un operador de estrechamiento. Sea $u \in L$ tal que $fix(f) \sqsubseteq u$ y $f(u) \sqsubseteq u$. La cadena decreciente:*

$$\begin{aligned} x_0 &= u \\ x_{n+1} &= x_n \ \triangle(n+1) \ f(x_n) \end{aligned}$$

es eventualmente estable, y cumple $\forall k \geq 0 : fix(f) \sqsubseteq x_k$.

Es decir, si partimos de la aproximación del punto fijo calculada mediante ensanchamiento, el operador de estrechamiento obtiene una aproximación mejor, que aún se encuentra por encima del punto fijo, es decir, que aún es segura.

Aproximación del punto fijo

El método propuesto por Hunt y Hankin [HH92] consiste en calcular el punto fijo de una función f sobre un retículo A, trasladando esta función a un retículo más pequeño A', calculando en él el punto fijo de la función trasladada y trasladando de vuelta el resultado al retículo A.

Trabajamos con una familia de retículos finitos L:

$$\begin{array}{ll} \mathbf{2} \in L & \\ D_\perp \in L & \text{si } D \in L \\ D_\top \in L & \text{si } D \in L \\ D_1 \times D_2 \in L & \text{si } D_i \in L, i = 1,2 \\ [D \to D'] \in L & \text{si } D, D' \in L \end{array}$$

entre los que se establece una relación de orden:

$$\begin{array}{ll} \mathbf{2} \sqsubseteq D & \forall D \in L \\ D_\perp \sqsubseteq D'_\perp & \text{si } D \sqsubseteq D' \\ D_\top \sqsubseteq D'_\top & \text{si } D \sqsubseteq D' \\ D_1 \times D_2 \sqsubseteq D'_1 \times D'_2 & \text{si } D_1 \sqsubseteq D'_1 \text{ y } D_2 \sqsubseteq D'_2 \\ [D_1 \to D_2] \sqsubseteq [D'_1 \to D'_2] & \text{si } D_1 \sqsubseteq D'_1 \text{ y } D_2 \sqsubseteq D'_2 \end{array}$$

Las funciones de traslado entre retículos son pares inmersión–clausura:

$$S_{A,B} : A \to^{ec} B$$

y pares inmersión–proyección:

$$L_{A,B} : A \to^{ep} B$$

Llamemos $UpS_{A,B}$ a $S^e_{A,B}$, $DownS_{A,B}$ a $S^c_{A,B}$, $UpL_{A,B}$ a $L^e_{A,B}$ y $DownL_{A,B}$ a $L^p_{A,B}$.

Definimos pares inmersión–clausura de la siguiente manera:

$$\begin{array}{lcl} UpS_{\mathbf{2},\mathbf{B}}\ a & = & h^e_\perp \\ DownS_{B,\mathbf{2}}\ b & = & h^c_\perp \\ S_{A_\perp,B_\perp} & = & (S_{A,B})_\perp \\ S_{A_\top,B_\top} & = & (S_{A,B})_\top \\ S_{A_1\times B_1,A_2\times B_2} & = & S_{A_1,A_2} \times S_{B_1,B_2} \\ S_{[A_1\to B_1],[A_2\to B_2]} & = & S_{A_1,A_2} \to S_{B_1,B_2} \end{array}$$

donde $h_\perp$ es un caso particular de los pares inmersión–clausura h_a definidos en la Sección 2.4.4 donde $a = \perp$.

Los pares inmersión–proyección se definen de forma semejante donde:

$$\begin{array}{lcl} UpL_{\mathbf{2},\mathbf{B}}\ a & = & k_\top^e \\ DownL_{B,\mathbf{2}}\ b & = & k_\top^p \end{array}$$

y $k_\top$ es el caso particular de los pares inmersión–proyección k_a definidos en la Sección 2.4.7 donde $a = \top$.

Hankin y Hunt [HH92] demostraron que cuando un punto fijo de la aproximación de una función g en un retículo más pequeño se sumerge en un retículo mayor cualquiera, se convierte en un punto posfijo de la aproximación de g en dicho retículo.

Teorema 48 *Para cualesquiera $[A \to A]$, $[A' \to A']$, $[A'' \to A'']$, donde $A'' \sqsubseteq A' \sqsubseteq A$, entonces para cada $g \in [A \to A]$ y cualquier punto fijo x de $(DownS_{[A\to A],[A'\to A']}\ g)$, tenemos:*

$$(DownS_{[A\to A],[A'\to A']}\ g)(UpS_{A'',A'}\ x) \sqsubseteq (UpS_{A'',A'}\ x)$$

El resultado dual para los pares inmersión–proyección es que cualquier punto fijo se convierte en un punto prefijo en el retículo mayor. Además

Teorema 49 *Para cualesquiera $[A \to A]$, $[A' \to A']$, $[A'' \to A'']$, donde $A'' \sqsubseteq A' \sqsubseteq A$ y $f \in [A \to A]$ se cumple:*

- $UpS_{A'',A'}(fix_{A''}(DownS_{[A\to A],[A''\to A'']}\ f) \sqsupseteq fix'_A(DownS_{[A\to A],[A'\to A']}\ f)$
- $UpL_{A'',A'}(fix_{A''}(DownL_{[A\to A],[A''\to A'']}\ f) \sqsubseteq fix'_A(DownS_{[A\to A],[A'\to A']}\ f)$

Es decir, el punto posfijo obtenido es además una aproximación superior al punto fijo de la aproximación de g en A'. Y el punto prefijo es una aproximación inferior a punto fijo de la aproximación de g en A'.

A partir de estos resultados y las propiedades de los puntos prefijos y posfijos de la Sección 2.1.3 se obtiene el siguiente algoritmo para el cálculo de aproximaciones puntos fijos:

1. Seleccionar algún retículo A'' más pequeño en el que el problema del cálculo del punto fijo sea tratable. Calcular el mínimo punto fijo de las aproximaciones de f: $S_{A,A''}(f)$ y $L_{A,A''}(f)$ en dicho retículo iterando desde $\perp$ como es usual.

2. El paso anterior proporciona cotas superior s e inferior i del verdadero punto fijo. Si coinciden, o bien si basta con una aproximación, tomar la cota superior, si no pasar al siguiente paso.

3. Aplicar $UpS_{A'',A'}$ a s y $UpL_{A'',A'}$ a i, donde A' es un retículo intermedio mayor que A'', e iterar desde el punto posfijo resultante hacia abajo y desde el punto prefijo resultante hacia arriba. Volver al punto 2.

$$
\begin{tikzcd}
A & f \arrow[dd] & & UpS_{A',A}\ x' \\
A' & & y' = UpS_{A'',A'}\ x'' \arrow[r] & x' = (DownS_{[A\to A],[A'\to A']}\ f)^n\ y' \arrow[u] \\
A'' & f'' = DownS_{[A\to A],[A''\to A'']}\ f \arrow[r] & x'' = fix_{A''} f'' \arrow[u] &
\end{tikzcd}
$$

Figura 5.19: Algoritmo de aproximación del punto fijo

En la Figura 5.19 se muestra el proceso del algoritmo para el caso particular en que solamante intervienen dos retículos adicionales A' y A''.

Si el punto posfijo (prefijo) de la imagen mediante el par inmersión–clausura (proyección) en un retículo intermedio ya es un punto fijo, entonces no se puede mejorar la cota superior. Pero si el punto prefijo es un punto fijo, entonces es el mínimo punto fijo.

Hankin y Hunt [HH92] demostraron que el primer paso en el algoritmo es equivalente al uso del operador de ensanchamiento:

$$\lambda j.\lambda(x,y).x \sqcup UpS_{D,D'}(Down_{D,D'}\ y)$$

y que el refinamiento de la cota superior iterando desde ella hacia abajo en el tercer paso del algoritmo corresponde al uso del operador de estrechamiento:

$$\lambda j.\lambda(x,y).x \sqcap UpS_{A_j,A}(DownS_{A,A_j}\ y)$$

donde $A_1, \ldots A_n$ es una secuencia de retículos tales que $A_i \sqsubseteq A_{i+1}$ y $A_n = A$

Comentarios

La renuncia a la exactitud en el cálculo del punto fijo parece una buena forma de evitar su alto coste en casos de retículos grandes. En dichos casos la técnica de fronteras no parece ser muy satisfactoria. Lamentablemente no existe ninguna medida de la ganancia en tiempo de cálculo que se puede obtener aplicando esta técnica de aproximación. Tampoco existe una cuantificación de la pérdida de exactitud producida, aunque ésta depende de los operadores seleccionados. En una versión experimental de Haskell se implementó un análisis de estrictez de orden superior que utilizaba un operador de ensanchamiento para la aproximación de puntos fijos. El operador de ensanchamiento utilizado no era demasiado sutil y además no se combinaba con uno de estrechamiento, puesto que se deseaba una aproximación lo más rápida posible. Aun así se obtenían mejoras en la ejecución de los programas de hasta un 30%. Sería interesante estudiar si en retículos más complejos que el utilizado para el análisis de estrictez, se mantienen los mismos resultados, o si son necesarios operadores algo más "finos".

Capítulo 6

Análisis hacia atrás

6.1 Análisis hacia delante y hacia atrás

El análisis hacia delante o interpretación abstracta consiste esencialmente en ejecutar un programa con información parcial acerca de sus entradas para derivar información parcial acerca de su resultado. El análisis hacia atrás se usa sin embargo, para derivar información contextual.

Ambos tipos de análisis de programas propagan información a través del árbol sintáctico del programa analizado. Un analizador hacia delante parte de información acerca de las variables que aparecen en la expresión y la propaga hacia arriba (hacia la raiz) en el árbol sintáctico para derivar información sobre el valor de la expresión completa.

Un analizador hacia atrás parte de información acerca del contexto en el que se evalúa la expresión completa y la propaga hacia abajo (hacia las hojas) en el árbol sintáctico para derivar información acerca de los contextos en los que se evalúan las subexpresiones.

Por ejemplo, consideremos el análisis de estrictez de la siguiente función:

$$f\ x\ y = \textbf{if}\ (x = 0)\ 0\ (x * y)$$

El árbol sintáctico del cuerpo de la función se muestra en la Figura 6.1.

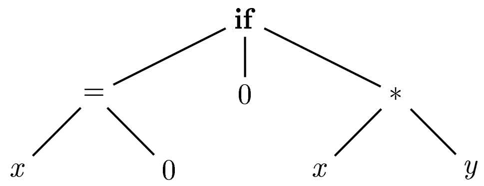


Figura 6.1: Arbol sintáctico de la función $f\ x\ y = \textbf{if}\ (x = 0)\ 0\ (x * y)$

En las Figuras 6.2 y 6.3 se muestra la propagación de información hacia las variables x e y respectivamente en un análisis de estrictez hacia delante. La información propagada desde las hojas hacia la raiz son los elementos del dominio abstracto **2** habitual. Este análisis detecta que

f es estricta en x (ver Figura 6.2) y no detecta nada sobre el segundo parámetro (ver Figura 6.3).

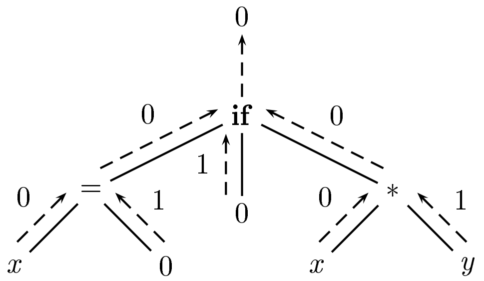


Figura 6.2: Propagación de la información de estrictez hacia x en la función $f\ x\ y = \textbf{if}\ (x = 0)\ 0\ (x * y)$

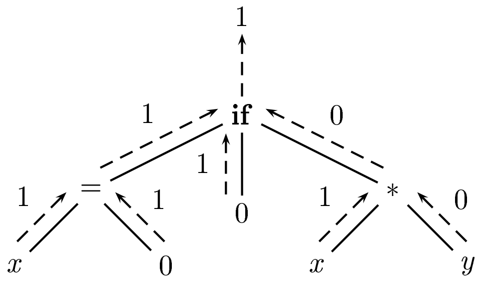


Figura 6.3: Propagación de la información de estrictez hacia y en la función $f\ x\ y = \textbf{if}\ (x = 0)\ 0\ (x * y)$

La Figura 6.4 muestra la propagación de información en un análisis hacia atrás de la misma función. Se trata de una versión simple de la propagación de la información en el **if**. Más adelante veremos una forma mejor de analizarlo. En este caso, la información propagada de la raíz hacia las hojas son los valores S y L que representan la evaluación estricta y (posiblemente) perezosa respectivamente de las subexpresiones (usando proyecciones son respectivamente las proyecciones STR e ID de la sección 6.2). Como es una expresión **if**, se propaga un contexto estricto hacia la condición, mientras que se propaga un contexto perezoso hacia las ramas. Posteriormente, $=$ y $*$ son estrictas en un contexto estricto, y perezosas en un contexto perezoso. Una vez que se ha propagado la información hacia las hojas es necesario combinarla de forma adecuada. Para ello se usa un operador & (que más adelante generalizaremos) que no es más que la máxima cota inferior. En el caso de la x, obtenemos que recibe un contexto S en la condición y uno L en cada rama. Esto quiere decir que la función f es estricta en x en un contexto estricto. Por otro lado y recibe un contexto L, luego no sabemos nada acerca de este parámetro. El resultado es pues el mismo que en el análisis hacia delante.

Esta versión de la propagación en el **if** tiene un defecto. Con ella, no se puede detectar que $g\ x\ y = \textbf{if}\ (x = 0)\ y\ y$ es estricta en y en un contexto estricto, pues se propaga directamente L a las ramas y su combinación a través de & nos produce de nuevo L. Por ello más adelante combinaremos el uso de & con el de la mínima cota superior, de forma semejante a la interpretación hacia delante del **if**.

Hemos visto que para analizar hacia delante una llamada a una función f, utilizamos una función abstracta $f^{\#}$ que hace corresponder a los n valores abstractos de los argumentos el valor abstracto de la llamada. En el análisis hacia atrás, se utilizan n funciones de propagación de contextos $f^1 \ldots f^n$, donde f^i propaga el contexto de la llamada de la función hacia el argumento i–ésimo.

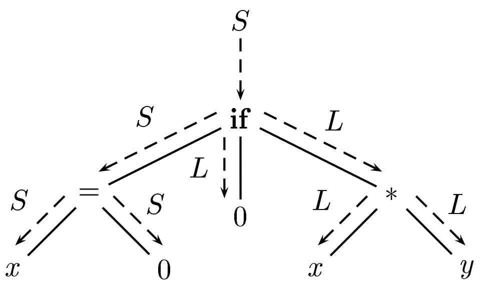


Figura 6.4: Propagación de la información de estrictez hacia atrás en la función $f\ x\ y = \textbf{if}\ (x = 0)\ 0\ (x * y)$

Es de notar la semejanza con la evaluación de atributos: el análisis hacia delante usa atributos sintetizados mientras que el análisis hacia atrás usa atributos heredados.

Una comparación entre estos ejemplos permite observar una de las ventajas del análisis hacia atrás frente al análisis hacia delante: es más eficiente, puesto que, al menos en este ejemplo, una única propagación es suficiente para mostrar que f es estricta en x y que no sabemos nada de y. En la interpretación abstracta, sin embargo, es necesaria una fase de cálculo de las funciones abstractas y una posterior de aplicación a los valores abstractos, lo que implica una propagación para cada parámetro.

Además, en ciertos casos, como en la estrictez de cabeza, el análisis hacia atrás resulta ser más potente que el análisis hacia delante basado en propiedades representadas con conjuntos Scott–cerrados.

Sin embargo, su extensión a orden superior es más compleja que en el análisis hacia delante. Para ver por qué, consideremos una expresión como:

$$apply\ f\ x$$

donde *apply* simplemente llama a su primer argumento con su segundo argumento como parámetro. Supongamos que el contexto de esta expresión es tan restrictivo que sabemos que el valor de la expresión es 0. ¿Qué sabemos entonces de x, independientemente del valor de f? Lo único que sabemos de x es que hay alguna función f que le hace corresponder el valor 0, pero esto esta es una propiedad que se cumple para todos los valores, luego, en realidad no se propaga ninguna información. La información propagada a f no es mucho mejor. Solamente sabemos que debe ser una función que al menos en un valor devuelva 0, pero esta puede ser la función constantemente 0, o podría ser la función head y x ser $cons$ 0 nil, o podría ser la función incremento y x ser -1.

El análisis hacia atrás se podría entender como una interpretación abstracta en la que los valores abstractos son funciones sobre contextos (funciones de propagación de contextos).

Los contextos nos proporcionan información sobre las propiedades exigidas al resultado de una determinada aplicación de función. Por ejemplo, en el caso de análisis de estrictez, un contexto nos proporciona información sobre el grado de definición que se le exige al resultado de una aplicación de función, a partir de la cual podemos intentar inferir el grado de definición exigido a sus argumentos.

Estos contextos se pueden representar de diversas maneras: mediante continuaciones o mediante proyecciones. El uso de continuaciones [Hug87] fue previo en el tiempo y posteriormente prevaleció el uso de las proyecciones [WH87]. A continuación estudiaremos un análisis hacia atrás en el que la información propagada a través del árbol sintáctico son contextos representados mediante proyecciones.

6.2 Análisis hacia atrás mediante proyecciones

6.2.1 Las proyecciones como contextos

Una **proyección** α sobre un dominio D es una función continua tal que para cada $u \in D$ se cumple:

$$\begin{array}{l} \alpha\ u \sqsubseteq u \\ \alpha\ (\alpha\ u) = \alpha\ u \end{array}$$

La primera condición establece que las proyecciones pueden eliminar información de los objetos a los que se aplican, y la segunda que la eliminan toda de una vez, por lo que aplicar la proyección por segunda vez no tiene ningún efecto.

Las proyecciones sobre un dominio D forman un retículo completo con respecto al orden $\sqsubseteq$. El supremo es la proyección ID definida por $ID\ \ u = u$ para todo u. Y el ínfimo es la proyección BOT definida por $\forall u. BOT\ \ u = \bot$

El supremo $\sqcup$ de dos proyecciones es la menor función mayor o igual que ambas, es decir, se calcula punto a punto y es una proyección. Sin embargo, no ocurre lo mismo con el ínfimo. $\alpha \sqcap \beta$ se define como la mayor proyección menor o igual que α y β, puesto que la mayor función menor que α y β puede no ser una proyección.

Ejemplo 50 Sea por ejemplo el dominio $D = \{a, b, c\}$ con $a \sqsubset b \sqsubset c$. Sean $\alpha = \{a \mapsto a, b \mapsto a, c \mapsto c\}$ y $\beta = \{a \mapsto a, b \mapsto b, c \mapsto b\}$.

Entonces:

$$\alpha \sqcup \beta = ID$$

que es claramente una proyección.

Sin embargo la mayor función menor o igual que ambas es:

$$f = \{a \mapsto a, b \mapsto a, c \mapsto b\}$$

que no es una proyección, puesto que no es idempotente:

$$f(f\ c) = a \neq b = f\ c$$

En este ejemplo:

$$\alpha \sqcap \beta = \{a \mapsto a, b \mapsto a, c \mapsto a\}$$

□

Se puede interpretar que una proyección especifica el grado de definición requerido para un argumento, de forma que la parte cuya imagen es $\perp$ es la parte innecesaria mientras que la que se deja invariante es la que se necesita.

El contexto en el que se evalua una función puede permitir a su argumento estar menos definido que en el caso general.

Ejemplo 51 Sean las siguientes proyecciones sobre pares:

$$F(u, v) = (u, \perp)$$
$$S(u, v) = (\perp, v)$$

Sea una función f de pares en pares, y supongamos un contexto en el que solamente necesitamos la primera componente del par resultado que produce f. En tal caso f se puede sutituir en ese contexto de forma segura por $F \circ f$. Se dice entonces que f se evalúa en el contexto F.

Supongamos que

$$f(u, v) = (v, u)$$

Entonces para f evaluada en el contexto F, es seguro evaluar su argumento en el contexto S, es decir:

$$F \circ f = F \circ f \circ S$$

□

Decimos que una función f es α-estricta en su argumento si $f \circ \alpha = f$. Decimos que una función f es β-estricta en el contexto α, y se denota por $f : \alpha \Rightarrow \beta$, si

$$\alpha \circ f = \alpha \circ f \circ \beta$$

Se demuestra que:

$$f : \alpha \Rightarrow \beta \text{ si y solo si } \alpha \circ f \sqsubseteq f \circ \beta$$

Se puede pensar que β es una estimación de la cantidad de información que se puede descartar del argumento de f sin afectar al resultado suponiendo que cierta información se ha descartado del mismo debido al contexto α.

6.2.2 Proyecciones para análisis de estrictez, ausencia y divergencia

Hasta el momento, las proyecciones definidas especifican la información suficiente, pero no la necesaria. Para introducirla, es necesario extender el dominio con un nuevo elemento $\jmath$, que representa la necesidad de que un cierto valor debe tener un mínimo de definición. Así, $\alpha\ u = \jmath$ significa que α requiere un valor más definido de u.

Por tanto, la estrictez simple se define mediante la proyección STR que hace corresponder a $\perp$ el elemento $\jmath$, y deja invariables el resto de los elementos:

$$
\begin{aligned}
STR\ \ \jmath &= \jmath \\
STR\ \ \perp &= \jmath \\
STR\ \ u &= u \quad \text{si } \perp \sqsubset u
\end{aligned}
$$

Dado que STR es una proyección, se debe cumplir que $SRT\ \perp \sqsubseteq \perp$, y por tanto $\jmath \sqsubset \perp$.

Esto quiere decir que el nuevo elemento es en realidad un nuevo ínfimo, que debemos añadir a cada dominio D. El nuevo dominio es la elevación de D, $D_\jmath$. Cada función $f : D_1 \to D_2$ debe ser por tanto extendida a otra en $D_{1_\jmath} \to D_{2_\jmath}$ obligando a f a ser estricta en $\jmath$.

Podemos usar STR para definir la estrictez:

$$f : STR \Rightarrow STR \ \text{ si y solo si } f \text{ es estricta}$$

Una vez introducido el nuevo ínfimo BOT deja de ser el ínfimo en el retículo de las proyecciones y pasa a serlo $FAIL$ definida por:

$$\forall u. FAIL\ \ u = \jmath$$

Puesto que toda función es estricta en $\jmath$ se cumple que para toda función f, $f : FAIL \Rightarrow FAIL$.

Se dice que una función es **divergente** si $\forall u. f\ u = \perp$. Se cumple:

$$f : STR \Rightarrow FAIL \ \text{ si y solo si } f \text{ es divergente}$$

La anterior proyección BOT pasa a ser la proyección ABS definida por:

$$
\begin{aligned}
ABS\ \ \jmath &= \jmath \\
ABS\ \ u &= \perp \quad \forall u \neq \jmath
\end{aligned}
$$

Esta proyección detecta cuándo una función ignora su argumento. Se dice que una función **ignora su argumento** si $f\ u = f\ \perp\ \forall u$. Se cumple que para toda función f, $f : ABS \Rightarrow ABS$. Además se cumple que $f : STR \Rightarrow ABS$ si y solo si f ignora su argumento.

La Figura 6.5 presenta el orden entre las proyecciones discutidas hasta el momento. Forman un subdominio del dominio de todas las proyecciones sobre un dominio D.

El análisis de contextos pretende etiquetar cada subexpresión e con una proyección α de forma que $\alpha\ e$ y e produzcan el mismo resultado, en el contexto en el que se halla situada.

Una vez encontrada dicha proyección, es seguro etiquetar también e con cualquier β tal que $\alpha \sqsubseteq \beta$. Por ello, para un determinado análisis podemos elegir un subconjunto de proyecciones siempre que esté cerrado bajo $\sqcup$ y $\sqcap$ y contenga a ID.

Para hacer solamente un análisis de estrictez basta con utilizar el subdominio de proyecciones $\{STR, ID\}$, pero ABS y $FAIL$ proporcionan algunas optimizaciones adicionales, como el análisis de ausencia y de divergencia.

La información obtenida por el triple análisis es la siguiente:

$$\begin{tikzcd}[column sep=small] & ID \arrow[dl, dash] \arrow[dr, dash] & \\ ABS \arrow[dr, dash] & & STR \arrow[dl, dash] \\ & FAIL & \end{tikzcd}$$

Figura 6.5: Parte del retículo de proyecciones

- Si etiquetamos e con $FAIL$ (rechaza todos los valores) el compilador puede implementar e de forma segura mediante un código que aborte directamente la ejecución.
- Si etiquetamos e con ABS (el valor de e es ignorado) el compilador puede implementar e de forma segura mediante un código que devuelva un valor cualquiera.
- Si etiquetamos e con STR (el valor de e se requiere) el compilador puede implementar e de forma segura mediante un código que evalúa e inmediatamente.
- Si etiquetamos e con ID (no sabemos nada) el compilador puede implementar e de forma segura solamente mediante un código que construya un grafo (lo usual).

Dada una proyección α, decimos que es **estricta** si $\alpha = \alpha \sqcap STR$. Si no es estricta llamamos a $\alpha' = \alpha \sqcap STR$ **parte estricta** de α. El problema de analizar la estrictez en un contexto α se puede reducir al análisis de su parte estricta α', ya que:

$$\text{Si } \alpha \text{ no es estricta y } f : \alpha' \Rightarrow \beta \text{ entonces } f : \alpha \Rightarrow ABS \sqcup \beta.$$

6.2.3 Proyecciones para el dominio de los enteros

Sea $\mathbb{Z}_\perp$ el dominio plano de los enteros. Hay infinitas proyecciones sobre dicho dominio. Un ejemplo de ellas son las definidas por:

$$\begin{array}{lcll} EQUAL_m\ m & = & m & \\ EQUAL_m\ u & = & \jmath & \forall u \neq m \end{array}$$

Así, si $f : EQUAL_m \Rightarrow EQUAL_n$ se obtiene que $f\ n = m$.

6.2.4 Proyecciones para las listas

Veamos un dominio finito de proyecciones para analizar las listas. Definamos las proyecciones NIL y $CONS\ \ \alpha\ \beta$, donde α es una proyección sobre el dominio de elementos de la lista D, y β es una proyección sobre el dominio de listas de D, $List\ D$:

$$\begin{array}{lcllcl} NIL\ \ \jmath & = & \jmath & CONS\ \ \alpha\ \beta\ \jmath & = & \jmath \\ NIL\ \ \perp_{List\ D} & = & \jmath & CONS\ \ \alpha\ \beta\ \perp & = & \jmath \\ NIL\ \ [\,] & = & [\,] & CONS\ \ \alpha\ \beta\ [\,] & = & \jmath \\ NIL\ \ (u:v) & = & \jmath & CONS\ \ \alpha\ \beta\ (u:v) & = & (\alpha\ u):(\beta\ v) \end{array}$$

Cuando se estudia la estrictez podemos considerar las siguientes proyecciones, que determinan la estrictez de cola y una forma de estrictez de cabeza respectivamente:

$$T\ \ u = \begin{cases} \jmath & \text{si } u = \jmath \\ \perp_{List\ D} & \text{si } u \text{ es lista infinita o aproximación de una} \\ u & \text{e.o.c} \end{cases}$$

$$\begin{array}{lcl} H\ \ \jmath & = & \jmath \\ H\ \ \perp_{List\ D} & = & \perp_{List\ D} \\ H\ \ (\perp_D : v) & = & \perp_{List\ D} \\ H\ \ [\,] & = & [\,] \\ H\ \ (u : v) & = & u : (H\ \ v) \end{array}$$

Esta noción de estrictez de cabeza no es la noción pura puesto que se aplica recursivamente al resto de la lista. Por ello a las funciones f tales que $f = f \circ H$ las llamaremos H-estrictas. En la Sección 6.2.6 se muestra la estrictez de cabeza pura y se comparan ambas.

Ejemplo 52 Sean las siguientes funciones:

before xs = **case** xs **of**
    [ ] ⇒ [ ]
    (y:ys) ⇒ **if** (y = 0) [ ] (y:before ys)
length xs = **case** xs **of**
    [ ] ⇒ 0
    (y:ys) ⇒ 1 + length ys
doubles xs = **case** xs **of**
    [ ] ⇒ [ ]
    y:ys ⇒ (2 * y):(doubles ys)

Se cumple que $before$ es una función H -estricta, ya que debe examinar la cabeza de la lista para ver si es 0 o no. La función $length$ es T-estricta ya que exige que la lista sea finita.

La función $doubles$ no es H -estricta ni T-estricta, puesto que aunque en la lista resultado aparece $2 * y$, si no se demanda el cálculo de la cabeza de la lista resultado, no es necesario calcularlo.

La diferencia se ve más fácilmente si se llama a $doubles$ en un contexto H –estricto. Por ejemplo si llamáramos a $(before \circ doubles)\ xs$, entonces sí se necesitaría evaluar la cabeza de xs. Esto quiere decir que $doubles$ es H -estricta en el contexto H :

$$before \circ doubles = before \circ H\ \circ doubles = before \circ H\ \circ doubles \circ H$$

□

Las partes estrictas de H y T son $H' = H \sqcap STR$ y $T' = T \sqcap STR$.

Algunas proyecciones de interés especial son las que tratan todos los elementos de una lista de la misma forma. Por ejemplo, si α es una proyección sobre D, sean las proyecciones:

$$\begin{array}{lcl} FIN\ \ \alpha & = & NIL \sqcup CONS\ \ \alpha\ (FIN\ \ \alpha) \\ INF\ \ \alpha & = & NIL \sqcup CONS\ \ \alpha\ (ABS \sqcup INF\ \ \alpha) \end{array}$$

Es decir, $FIN\ \alpha$ acepta solamente listas finitas, cada uno de cuyos elementos es aceptado por α, e $INF\ \alpha$ acepta listas finitas o infinitas cada uno de cuyos elementos es aceptado por α.

En relación a las proyecciones estudiadas anteriormente, se cumplen las siguientes ecuaciones:

$$\begin{array}{llllll} STR & = & INF\ ID & ID & = & ABS \sqcup INF\ ID \\ H' & = & INF\ STR & H & = & ABS \sqcup INF\ STR \\ T' & = & FIN\ ID & T & = & ABS \sqcup FIN\ ID \\ H' \sqcap T' & = & FIN\ STR & H \sqcap T & = & ABS \sqcup FIN\ STR \end{array}$$

Sea D_C un dominio finito de proyecciones sobre D. Por ejemplo, si $D = \mathbb{Z}_\perp$, D_C podría ser $\{STR, ID\}$. El dominio finito de proyecciones sobre listas $List_C D_C$ sobre $List\ D$ consiste en las proyecciones:

$$\begin{array}{lll} INF\ \alpha & ABS \sqcup INF\ \alpha & FAIL \\ FIN\ \alpha & ABS \sqcup FIN\ \alpha & ABS \end{array}$$

para cada $\alpha \in D_C$.

Por ejemplo, sin $D_C = \{STR, ID\}$ entonces $List_C\ D_C$ es el dominio de diez puntos mostrado en la Figura 6.6.

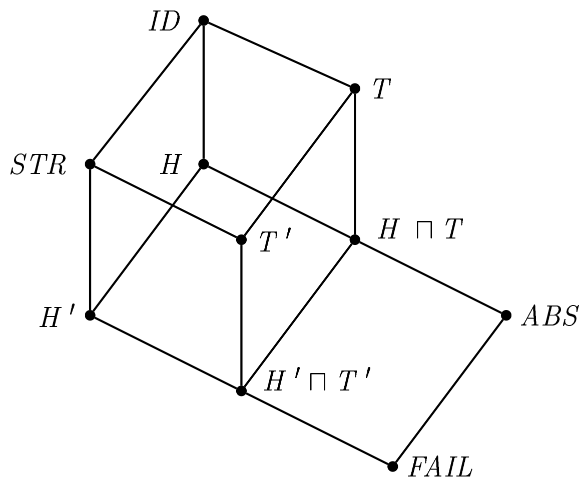


Figura 6.6: Retículo $List_C\ D_C$ donde $D_C = \{STR, ID\}$

6.2.5 El análisis de proyecciones

El problema a resolver en el análisis de proyecciones es el siguiente: dada una función f y una proyección α, se busca una proyección β tal que $f : \alpha \Rightarrow \beta$ (ID siempre vale, pero se intenta encontrar una más ajustada). Es decir, se trata de encontrar un contexto en el que evaluar el argumento de la función, conocido el contexto en el que se evalua la función completa.

Para ello se definen los llamados **transformadores de proyecciones**. Son funciones de proyecciones en proyecciones, encargadas de propagar los contextos de una expresión a sus subexpresiones. Es decir, son los que propagan la información de la raíz a las hojas del árbol.

Si tenemos un lenguaje como el siguiente:

$$
\begin{array}{rlll}
e ::= & x & & \text{variables} \\
| & k & & \text{constantes} \\
| & f\ e_1 \ldots e_n & & \text{aplicación de funciones} \\
| & \mathbf{if}\ e_0\ e_1\ e_2 & & \text{condicional} \\
| & \mathbf{case}\ e_0\ \mathbf{of}\ [\,] \Rightarrow e_1 \mid y : ys \Rightarrow e_2 & & \text{análisis por casos}
\end{array}
$$

donde las funciones se definen de la forma $f\ x_1 \ldots x_n = e$. Para cada una de las expresiones e del lenguaje, y para cada variable del programa, hay que definir un transformador e^x que tome una proyección aplicada a e y devuelva una proyección que se puede aplicar de forma segura a cada aparición de x en e.

La condición de seguridad exigida es la siguiente:

$$\alpha(sem[\![e]\!]\ \rho) \sqsubseteq sem[\![e]\!]\ (\rho[\beta(\rho(x))/x])$$

donde $\beta = e^x\alpha$.

Como hemos dicho anteriormente el análisis hacia atrás se puede considerar una interpretación abstracta en la que los valores abstractos son transformadores de proyecciones. La semántica estándar es la vista en secciones anteriores. La semántica abstracta es la que hace corresponder a cada expresión e y a cada variable x un transformador de proyecciones e^x que cumple la propiedad de seguridad exigida. En el análisis aqui presentado, el transformador de proyecciones se calcula para cada variable por separado, mientras que en la Sección 6.4 veremos un enfoque más general.

Por los resultados anteriores de la Sección 6.2.1 sabemos que es seguro que $e^x\ FAIL\ = FAIL$ y $e^x\ ABS\ = ABS$, y que si α no es estricta podemos definir $e^x\ \alpha = ABS\ \sqcup\ e^x\ \alpha'$. Luego e^x se define de la siguiente manera:

$$e^x\ \alpha = \alpha \rhd e^x\ \alpha'$$

donde

$$
\begin{array}{lcll}
FAIL\ \rhd \beta & = & FAIL & \\
ABS\ \rhd \beta & = & ABS & \\
\alpha \rhd \beta & = & \beta & \text{si } \alpha \text{ es estricta y } \alpha \neq FAIL \\
(ABS\ \sqcup \alpha) \rhd \beta & = & ABS\ \sqcup\ \beta & \text{si } \alpha \text{ es estricta y } \alpha \neq FAIL
\end{array}
$$

y si α es estricta y distinta de $FAIL$:

$$
\begin{array}{lcl}
x^x\ \alpha & = & \alpha \\
y^x\ \alpha & = & ABS \quad \text{si } x \neq y \\
k^x\ \alpha & = & ABS \quad \text{si k es una constante} \\
(f\ e_1 \ldots e_n)^x\ \alpha & = & e_1^x\ (f^1\ \alpha)\ \&\ \ldots\ \&\ e_n^x\ (f^n\ \alpha) \\
(\mathbf{if}\ e_0\ e_1\ e_2)^x\ \alpha & = & e_0^x\ (\mathbf{if}^\#\ \alpha)\ \&\ (e_1^x\ \alpha \sqcup e_2^x\ \alpha) \\
(\mathbf{case}\ e_0\ \mathbf{of}\ [\,] \Rightarrow e_1 & & \\
\qquad \mid y : ys \Rightarrow e_2)^x\ \alpha & = & (e_0^x\ NIL\ \ \&\ e_1^x\ \alpha) \sqcup (e_0^x\ (CONS\ \ (e_2^y\ \alpha)\ (e_2^y s\ \alpha))\ \&\ e_2^x\ \alpha)
\end{array}
$$

donde:

- $\mathbf{if}^{\#}$ es un transformador de estados cuya definición depende del análisis realizado. En el caso de análisis de estrictez $\mathbf{if}^{\#}\ \alpha = STR$.
- La definición de & depende del análisis que se está realizando y debe cumplir algunas propiedades. En el caso concreto del triple análisis de la Sección 6.2.2, & es una variante de $\sqcup$ definida por:

$$(\gamma_1\ \&\ \gamma_2)\ u = \begin{cases} \jmath & \text{si } \gamma_1\ u = \jmath \text{ ó } \gamma_2\ u = \jmath \\ \gamma_1\ u\ \sqcup\ \gamma_2\ u & \text{e.o.c} \end{cases}$$

  Cuando se trabaja en el análisis de estrictez, & coincide con el operador de ínfimo $\sqcap$.
- Dada f, f^i es un trasformador que toma una proyección aplicada a f y devuelve una proyección que se puede aplicar de forma segura al argumento i–ésimo:

$$f^i\ \alpha = e^{x_i}\ \alpha$$

  En el caso de análisis de estrictez, si f es una función primitiva estricta en todos los argumentos (por ejemplo $+$) entonces $\forall i. f^i\alpha = \alpha \rhd STR$. En el caso de las listas, llamamos $HEAD$ a $(:)^1$ y $TAIL$ a $(:)^2$, definidas de la siguiente manera:

$$\begin{array}{lcl} (HEAD\ \ \alpha)\ u & = & \bigsqcup_{v \in List\ D} head\ (\alpha\ (u : v)) \\ (TAIL\ \ \alpha)\ v & = & \bigsqcup_{u \in D} tail\ (\alpha\ (u : v)) \end{array}$$

  donde *head* y *tail* son las funciones que toman la cabeza y la cola de una lista respectivamente.

Por tanto, para un análisis concreto, es necesario proporcionar el operador & y un transformador de proyecciones para cada primitiva del lenguaje (como $\mathbf{if}^{\#}$). Wadler y Hughes [WH87] demostraron la propiedad de seguridad para todos los casos en el triple análisis de la Sección 6.2.2.

Las definiciones de funciones recursivas producen definiciones recursivas de transformadores de proyecciones, que se pueden resolver usando la cadena ascendente de Kleene.

Ejemplo 53 Supongamos que solamente estamos haciendo un análisis de estrictez, es decir, trabajamos con $\{STR\ , ID\ \}$, y sea la función:

$$f\ x\ y = \mathbf{if}\ (x = 0)\ y\ (f\ (x-1)\ y)$$

Obtenemos:

$$\begin{array}{lcl} f^1\ STR & = & STR\ \ \&\ (ID\ \sqcup f^1\ STR\ ) \\ f^2\ STR & = & ID\ \ \&\ (STR\ \sqcup f^2\ STR\ ) \end{array}$$

lo que nos lleva a:

$$\begin{array}{lcl} f^1\ STR & = & STR \\ f^2\ STR & = & STR \end{array}$$

y por tanto f es estricta en ambos argumentos.

□

Ejemplo 54 La aplicación del método a las funciones del Ejemplo 52 da los siguientes resultados suponiendo α estricta y distinta de $FAIL$. Si no, se usa la regla $f^i\alpha = \alpha \rhd f^i\alpha'$:

$$\begin{array}{lcl} length^1\ \alpha & = & NIL \sqcup CONS\ \ ABS\ \ (length^1\ \alpha) \\ before^1\ \alpha & = & NIL \sqcup CONS\ \ (STR\ \&HEAD\ \alpha)(ABS\ \ \sqcup (before^1\ (TAIL\ \alpha)) \\ doubles^1\alpha & = & NIL \sqcup CONS\ \ (HEAD\ \alpha)(doubles^1\ (TAIL\ \alpha)) \end{array}$$

Se trata de definiciones recursivas. La cadena ascendente de Kleene en el caso de *length* es la siguiente:

$$\begin{array}{lcl} length^{1\,(0)}\ \alpha & = & FAIL \\ length^{1\,(i+1)}\ \alpha & = & NIL \sqcup CONS\ \ ABS\ \ (length^{1\,(i)}\ \alpha) \end{array}$$

luego

$$\begin{array}{lcl} length^{1\,(0)}\ STR & = & FAIL \\ length^{1\,(1)}\ STR & = & NIL \sqcup CONS\ \ ABS\ \ FAIL \\ & = & FIN\ \ FAIL \\ length^{1\,(2)}\ STR & = & NIL \sqcup CONS\ \ ABS\ \ (FIN\ FAIL\ ) \\ & = & FIN\ \ ABS \\ length^{1\,(3)}\ STR & = & NIL \sqcup CONS\ \ ABS\ \ (FIN\ ABS\ ) \\ & = & FIN\ \ ABS \end{array}$$

El punto fijo es $length^1\ STR\ = FIN\ \ ABS$, que era exactamente lo que esperábamos: *length* espera una lista finita pero ignora sus elementos.

□

Kei Davis y Wadler [DW90] mejoraron posteriormente este análisis haciéndolo más exacto. La debilidad del análisis presentado aquí estriba en que los parámetros de las funciones se tratan por separado, mientras que en el análisis mejorado se lleva un entorno de variables ligadas a transformadores de proyecciones ϕ, no solamente una variable cada vez. Además, no se centra solemente en las listas, sino que además presenta una construcción **case** general para cualquier tipo de datos construido, imponiendo ciertas restricciones a las proyecciones definidas sobre ellos. Este análisis se presenta en la Sección 6.4.

6.2.6 La estrictez de cabeza

La proyección H de la Sección 6.2.4 no representa la estrictez de cabeza pura, puesto que se aplica recursivamente al resto de la lista. Esta noción de estrictez significa que si se necesita evaluar la lista, entonces se necesita evaluar también la cabeza y el resto de la lista se evaluará en un contexto H.

La estrictez de cabeza pura viene representada por H_B:

$$
\begin{array}{lcl}
H_B\ \jmath & = & \jmath \\
H_B\ \perp_{List\ D} & = & \perp_{List\ D} \\
H_B\ (\perp_D : v) & = & \perp_{List\ D} \\
H_B\ [\,] & = & [\,] \\
H_B\ (u : v) & = & u : v
\end{array}
$$

Ser H_B estricta significa que si se necesita evaluar la lista entonces también se necesita evaluar la cabeza. Se cumple que $H \sqsubset H_B$.

H y H_B capturan nociones de estrictez de cabeza que se superponen pero cada una proporciona información que la otra no proporciona.

Si una función es H_B-estricta entonces no es necesariamente H-estricta. Por ejemplo, la siguiente función es H_B-estricta pero no H-estricta:

$$f\ xs = (hd\ xs) + (length\ xs)$$

El hecho de que H proporciona información que H_B no proporciona se puede ver en el siguiente ejemplo:

$$
\begin{array}{lcl}
f\ [\,] & = & [\,] \\
f\ (x : xs) & = & \textbf{if}\ (x = 0)\ [\,]\ (\textbf{if}\ (null\ xs)\ [\,]\ (f\ xs))
\end{array}
$$

Esta función es tanto H-estricta como H_B-estricta. Sin embargo, el hecho de ser H-estricta nos lleva a evaluar el argumento creando una constructora de lista *cons* con una cabeza evaluada y cuya cola se halla en un contexto H, mientras que H_B no hace nada con la cola. En este ejemplo, H provocará la evaluación de todas las celdas *cons* con cabezas evaluadas, mientras que H_B solamente hará esto para el primer *cons*.

6.2.7 Análisis de compartición

El análisis de compartición pretende estimar el número de veces que se usa una variable. Se distingue entre cero (0), uno (1) y muchos usos (M). Usaremos como contextos proyecciones sobre el dominio de los subconjuntos de $\{0, 1, M\}$ ordenados por inclusión de conjuntos.

Estos subconjuntos representan las posibles veces que se usa una expresión. Por ejemplo, $\{0, 1\}$ representa el hecho de que se puede usar o no, pero que en el caso de que se use será como mucho una vez. En este caso el ínfimo es $\emptyset$ y el supremo $\{0, 1, M\}$.

Las proyecciones que se utilizan son de la siguiente manera:

$$\alpha_c\ u = \begin{cases} \emptyset & \text{si } u \sqsubseteq c \\ u & \text{e.o.c} \end{cases}$$

donde c es un subconjunto concreto de $\{0, 1, M\}$.

Es decir, aplastan los subconjuntos hacia el conjunto vacio hasta un determinado elemento del dominio y dejan el resto invariables.

Podemos identificar, por tanto estas proyecciones con el subconjunto c correspondiente. Así:

- $\emptyset$ representa a la proyección BOT
- $\{0\}$ representa la ausencia
- $\{1, M\}$ representa la estrictez
- $\{0, 1, M\}$ representa a la proyección ID

El operador de supremo entre proyecciones se traduce a la unión de conjuntos, y el operador & se define de la siguiente manera:

$$\alpha \ \& \ \beta = \{a + b \mid a \in \alpha, b \in \beta\}$$

donde:

$$\begin{array}{lcllclcl} 0+0 & = & 0 & 1+0 & = & 0+1 & = & 1 \\ 1+1 & = & M & 0+M & = & M+0 & = & M \\ M+M & = & M & 1+M & = & M+1 & = & M \end{array}$$

En cuanto a las primitivas:

- $(:)^i \ \alpha = \{b \mid \exists a \in \alpha, b \sqsubseteq a\}$.
- $\mathbf{if}^\#$ y las demás primitivas son la identidad respecto a α.

Ejemplo 55 Sea la misma función del Ejemplo 53. Obtenemos:

$$\begin{array}{lcl} f^1 \ \alpha & = & \alpha \ \& \ (\{0\} \sqcup f^1 \ \alpha) \\ f^2 \ \alpha & = & \alpha \sqcup f^2 \ \alpha \end{array}$$

Supongamos que nos hallamos en un contexto representado por $\{0, 1\}$. Entonces, calculando el punto fijo:

$$\begin{array}{lcl} f^1 \ \{0, 1\} & = & \{0, 1, M\} \\ f^2 \ \{0, 1\} & = & \{0, 1\} \end{array}$$

Es decir, el segundo parámetro se puede pasar por nombre en un contexto en el que la función f se use a lo sumo una vez.

□

6.3 Relación entre el análisis de proyecciones y la interpretación abstracta

Burn [Bur90] estudió la relación entre una cierta clase de proyecciones, y una cierta clase de interpretaciones abstractas.

Una proyección $\alpha : \sigma \to \sigma$ es una proyección **aplastante** si para cada $u \in \sigma$, o bien $\alpha\ u = \perp_{D_\sigma}$, o bien $\alpha\ u = u$. Es decir, o bien son la identidad sobre el elemento, o bien lo aplastan al ínfimo.

Sea una interpretación abstracta del estilo de la Sección 5.3.2, donde las funciones de abstracción de los tipos base *abs* reflejan el $\perp$. Entonces la función de abstracción para cualquier tipo refleja el $\perp$.

Decimos que un elemento del dominio abstracto $\overline{u} \in B_\sigma$ y una proyección $\alpha : \sigma \to \sigma$ están **relacionados** si:

- $\forall u \in \gamma_\sigma(\overline{u}) : \alpha\ u = \perp_{D_\sigma}$
- $\forall u \notin \gamma_\sigma(\overline{u}) : \alpha\ u = u$

Es decir, están relacionados si α solamente lleva a $\perp$ a las concreciones de $\overline{u}$.

Ejemplo 56 Como ejemplo usamos el dominio abstracto de la Figura 6.7 para la interpretación abstracta de listas. Con este dominio se distinguen listas con cabeza indefinida de las que no la tienen, para capturar la noción de que una función es estricta de cabeza si devuelve $\perp$ cuando se aplica a una lista con cabeza indefinida.

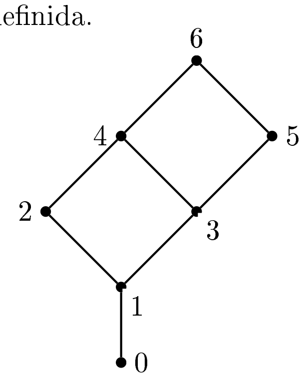


Figura 6.7: Dominio abstracto para las listas

Si se unen los puntos 1 y 2, y los puntos 3, 4 y 5, entonces obtenemos el dominio abstracto de la Sección 5.4.2.

Dado $u \in D_{List\ D}$, entonces $abs_{List\ D}\ u$ tiene el valor:

- 0 si $u = \perp_{List\ D}$.
- 1 si u es una lista infinita, o una aproximación de una lista infinita con la cabeza indefinida.
- 2 si u es una lista infinita, o una aproximación de una lista infinita cuya cabeza no está indefinida.

- 3 si u es una lista finita cuya cabeza está indefinida, y que tiene algún elemento posterior indefinido.
- 4 si u es una lista finita cuya cabeza no está indefinida, pero que tiene algún elemento posterior indefinido.
- 5 si u es una lista finita cuya cabeza está indefinida, pero ningún otro elemento de la lista lo está.
- 6 si u es una lista finita ninguno de cuyos elementos está indefinido.

Esta función de abstracción es estricta, continua y refleja el $\perp$. Se cumple que ID está relacionado con 0, T está relacionado con 2, H_B está relacionado con 5 y $T \sqcap H$ está relacionada con 4. Si queremos trabajar con las partes estrictas de las proyecciones, entonces hay que elevar el dominio abstracto, y entonces se cumple que $ID\,' = STR$ está relacionada con $< 0,0 >$, T' está relacionada con $< 0,2 >$, $H_B{}'$ está relacionada con $< 0,5 >$ y $(T \sqcap H\,)'$ está relacionada con $< 0,4 >$. Recordemos que una proyección α es estricta si $\alpha \perp = \jmath$

$\square$

El principal teorema que demuestra Burn es el siguiente, que relaciona las proyecciones aplastantes, (por ejemplo las usadas en la Sección 6.2, excepto H) y las interpretaciones abstractas que usan funciones de abstracción que reflejan el $\perp$ (por ejemplo el análisis de estrictez presentado como ejemplo en la Sección 5.3.2).

Teorema 57 *Sea* $f : \sigma \to \tau$ *con* $\overline{f} = abs_{\sigma \to \tau}\ f$. *Supongamos que* $\overline{u}$ *está relacionado con* $\alpha : \sigma \to \sigma$ *y* $\overline{v}$ *está relacionado con* $\beta : \tau \to \tau$. *Entonces*

- $f : \beta \Rightarrow \alpha$ *y* f *estricta implica* $\overline{f}\ \overline{u} \sqsubseteq \overline{v}$.
- $\overline{f}\ \overline{u} \sqsubseteq \overline{v}$ *implica que* $f : \beta \Rightarrow \alpha$.

En este teorema existe una asimetría. Si $f : \alpha \Rightarrow \beta$ y f no es estricta, entonces no es cierto necesariamente que $\overline{f}\ \overline{u} \sqsubseteq \overline{v}$. Por ejemplo, si $\alpha = ID$ y $\beta = T$, entonces la función f definida por

$$f\ xs = [\,]$$

que no es estricta, cumple que $f : T \Rightarrow ID$ pero $\overline{f}\ 0 \not\sqsubseteq 2$, ya que $\overline{f}\ \overline{u} = 6$ para todo $\overline{u}$.

Tampoco es cierto que si $\overline{f}\ \overline{u} \sqsubseteq \overline{v}$ esto implique que f es estricta. Por ejemplo, sea la función

$$f\ xs = append\ [\,]\ xs$$

Esta función no es estricta, pero $\forall \overline{u}. \overline{f}\ \overline{u} = \overline{u}$.

Solamente podemos concluir que es estricta si encontramos un $\overline{u}$ que cumpla que $\overline{f}\ \overline{u} \sqsubseteq \perp$.

Sin embargo, aunque no sea estricta obtenemos que $f : \alpha \Rightarrow \beta$, lo cual nos proporciona información útil. Por ejemplo, sea la función:

$$f\ xs\ y = \textbf{if}\ (y = 0)\ 0\ (\textbf{if}\ (null\ xs)\ [\,]\ (1 + (f\ (tl\ xs)\ y)))$$

Aunque esta función no es estricta en el primer argumento, es sin embargo T-estricta (en el contexto ID), lo cual resulta útil, puesto que en lugar de crear una clausura para cada cola de la lista, se puede crear desde el principio una única clausura para evaluar toda la estructura de la lista.

Esto quiere decir que las proyecciones proporcionan más información puesto que el resultado de la interpretación abstracta siempre implica el resultado obtenido a través del análisis de proyecciones, mientras que la implicación inversa solamente se obtiene cuando la función en cuestión es estricta.

6.4 Un análisis más general

A continuación se presenta una generalización del análisis presentado anteriormente desarrollada por Kei Davis y Wadler [DW90]. En el leguaje se admiten además de listas, otros valores de tipos construidos:

$$
\begin{array}{rll}
e ::= & x & \text{variables} \\
| & c\ e_1 \ldots e_n & \text{constantes construidas} \\
| & f\ e_1 \ldots e_n & \text{aplicación de funciones} \\
| & p\ e_1 \ldots e_n & \text{funciones estrictas primitivas} \\
| & \mathbf{if}\ e_0\ e_1\ e_2 & \text{condicional} \\
| & \mathbf{case}\ e_0\ \mathbf{of} & \text{análisis por casos} \\
 & \quad c_1\ x_{1,1} \ldots x_{1,a_1} \Rightarrow e_1 & \\
 & \quad \vdots & \\
 & \quad c_m\ x_{m,1} \ldots x_{m,a_m} \Rightarrow e_m & \\
p ::= & \{f_i\ x_1 \ldots x_{n_i} = e_i \mid 1 \leq i \leq n\} &
\end{array}
$$

un programa p consta de varias definiciones de función y c, c_i son constructores.

Como ya hemos visto anteriormente el análisis hacia atrás se puede ver como una interpretación abstracta en la que los valores abstractos son los transformadores de proyecciones.

La semántica estándar es la habitual vista anteriormente. Se llevan dos entornos, uno para variables ρ y otro para funciones definidas σ, una función C que a cada constructor c le hace corresponder una función, y otra P que a cada p le hace corresponder una función:

$$
\begin{array}{lcl}
sem[\![x]\!]\ \rho\ \sigma & = & \rho(x) \\
sem[\![c\ e_1 \ldots e_n]\!]\ \rho\ \sigma & = & C(c)\ (sem[\![e_1]\!]\ \rho\ \sigma) \ldots (sem[\![e_n]\!]\ \rho\ \sigma) \\
sem[\![f\ e_1 \ldots e_n]\!]\ \rho\ \sigma & = & \sigma(f)\ (sem[\![e_1]\!]\ \rho\ \sigma) \ldots (sem[\![e_n]\!]\ \rho\ \sigma) \\
sem[\![p\ e_1 \ldots e_n]\!]\ \rho\ \sigma & = & P(p)\ (sem[\![e_1]\!]\ \rho\ \sigma) \ldots (sem[\![e_n]\!]\ \rho\ \sigma)
\end{array}
$$

$$
sem \left[\!\!\left[\begin{array}{l} \mathbf{case}\ e_0\ \mathbf{of} \\ \quad c_1\ x_{1,1} \ldots x_{1,a_1} \Rightarrow e_1 \\ \quad \vdots \\ \quad c_m\ x_{m,1} \ldots x_{m,a_m} \Rightarrow e_m \end{array} \right]\!\!\right] \rho\ \sigma
$$

$$
\begin{array}{l}
= sem[\![e_i]\!]\ \rho[(sel_{i,1}\ v)/x_{i,1}, \ldots, (sel_{i,a_i}\ v)/x_{i,a_i}]\ \sigma \\
\qquad\qquad\qquad\qquad \text{si } isc_i\ (sem[\![e_0]\!]\ \rho\ \sigma)\ (1 \leq i \leq n) \\
= \perp \text{ e.o.c}
\end{array}
$$

$$sem[\![\{f_i\ x_1 \ldots x_{n_i} = e_i \mid 1 \leq i \leq n\}]\!] = fix(\lambda\sigma.[(\lambda y_1 \ldots y_{n_i}.sem[\![e_i]\!][\overline{y}/\overline{x}]\ \sigma)/f_i \mid 1 \leq i \leq n)$$

donde isc_i es una función que devuelve cierto si su parámetro es un valor construido con constructor c_i y falso en caso contrario, y $sel_{i,j}$ es una función que dado un valor construido con constructor c_i devuelve su componente j.

Las proyecciones permitidas para los valores construidos son aquellas de la forma $C_i\ \alpha_1 \ldots \alpha_{a_i}$ donde:

$$C_i\ \alpha_1 \ldots \alpha_{a_i}\ (c_j\ v_1 \ldots v_{a_j}) = \begin{cases} c_j\ (\alpha_1\ v_1) \ldots (\alpha_{a_j}\ v_{a_j}) & \text{si } i = j \\ \jmath & \text{e.o.c.} \end{cases}$$

y sus menores cotas superiores: $\sqcup_{i=1}^{m} C_i\ \alpha_{i,1} \ldots \alpha_{i,a_i}$.

Esta restricción excluye a aquellas proyecciones que representan alguna interdependencia entre sus componentes.

La función de semántica abstracta es muy similar a la vista anteriormente con la excepción de que llevamos un entorno de variables ligadas a transformadores de proyecciones ϕ en lugar de solamente una variable cada vez. Además llevamos un entorno de funciones ligadas a funciones entre transformadores de proyecciones ψ.

La semántica abstracta es ahora:

$$B[\![e]\!]\ \phi\ \psi\ \alpha \quad = \quad \alpha \rhd B[\![e]\!]\ \phi\ \psi\ (\alpha \sqcap STR\ )$$

y si α es estricta y distinta de $FAIL$:

$$\begin{array}{lcl}
B[\![x]\!]\ \phi\ \psi & = & \phi(x) \\
B[\![c\ e_1 \ldots e_n]\!]\ \phi\ \psi & = & C_B(c)\ (B[\![e_1]\!]\ \phi\ \psi) \ldots (B[\![e_n]\!]\ \phi\ \psi) \\
B[\![f\ e_1 \ldots e_n]\!]\ \phi\ \psi & = & \psi(f)\ (B[\![e_1]\!]\ \phi\ \psi) \ldots (B[\![e_n]\!]\ \phi\ \psi) \\
B[\![p\ e_1 \ldots e_n]\!]\ \phi\ \psi & = & P_B(p)\ (B[\![e_1]\!]\ \phi\ \psi) \ldots (B[\![e_n]\!]\ \phi\ \psi)
\end{array}$$

$$B\left[\!\!\left[\begin{array}{l} \textbf{case } e_0 \textbf{ of} \\ \quad c_1\ x_{1,1} \ldots x_{1,a_1} \Rightarrow e_1 \\ \quad \vdots \\ \quad c_m\ x_{m,1} \ldots x_{m,a_m} \Rightarrow e_m \end{array}\right]\!\!\right]\ \phi\ \psi$$

$$\begin{array}{l} = \lambda\alpha. \sqcup_{i=1}^{m} (B[\![e_i]\!]\ \phi[(\lambda\alpha.ABS)/x_{i,1} \mid 1 \leq i \leq a_i]\ \psi\ \& \\ B[\![e_0]\!]\ \phi\ \psi\ (C_i\ (B[\![e_i]\!]\ \phi_{x_{i,1}}\ \psi\ \alpha) \ldots (B[\![e_i]\!]\ \phi_{x_{i,a_i}}\ \psi\ \alpha))) \end{array}$$

$$B[\![\{f_i\ x_1 \ldots x_{n_i} = e_i \mid 1 \leq i \leq n\}]\!] \quad = \quad fix(\lambda\psi.[(\lambda y_1 \ldots y_{n_i}.B[\![e_i]\!][\overline{y}/\overline{x}]\ \psi)/f_i \mid 1 \leq i \leq n])$$

donde ϕ_x denota el entorno que hace corresponder $\lambda\alpha.\alpha$ a la variable x y $\lambda\alpha.ABS$ al resto de variables. Además:

$$C_B(c)\ \tau_1 \ldots \tau_n\ (\sqcup_{j=1}^{m} C_j\ \alpha_{j,1} \ldots \alpha_{j,a_j}) = \&_{j=1}^{a_i} B[\![e_j]\!]\ \phi\ \psi\ \alpha_{i,j}$$

y si $f\ x_1 \ldots x_n = e$ entonces:

$$\phi(f) = f^B = \lambda\tau_1 \ldots \tau_n.B[\![e]\!]\ [\tau_1/x_1 \ldots \tau_n/x_n]\ \psi$$

y si p es una función primitiva estricta de aridad n:

$$P_B(p)\ \tau_1 \ldots \tau_n = \lambda\alpha.\&_{j=1}^{a_i} B[\![e_j]\!]\ \phi\ \psi\ STR$$

Ejemplo 58 Sea la función:

$$head\ xs = \textbf{case}\ xs\ \textbf{of}\ y : ys \Rightarrow y$$

Asumiendo que en cada abstracción lambda está implícita $\rhd$, tenemos:

$$\begin{array}{rcl}
head^B\ \tau & = & B[\![\textbf{case}\ xs\ \textbf{of}\ y : ys \Rightarrow y]\!]\ [\tau/xs][\ ] \\
 & = & \lambda\alpha.B[\![xs]\!]\ [\tau/xs]\ [\ ]\ (CONS\ \ (B[\![y]\!]\ \phi_y\ [\ ]\ \alpha)\ (B[\![y]\!]\ \phi_{ys}\ [\ ]\ \alpha)) \\
 & & \&\ B[\![y]\!]\ [\tau/xs, (\lambda\alpha.ABS\ )/y, (\lambda\alpha.ABS\ )/ys]\ [\ ]\ \alpha \\
 & = & \lambda\alpha.\tau\ (CONS\ \ (\alpha \rhd \alpha)\ (\alpha \rhd ABS\ )) \\
 & = & \lambda\alpha.\tau\ (CONS\ \ \alpha\ ABS\ )
\end{array}$$

Para determinar un transformador de proyecciones seguro para xs con respecto a $head\ xs$:

$$head^B\ (\lambda\alpha.\alpha)\ \alpha = \alpha \rhd CONS\ \ \alpha\ ABS$$

Por ejemplo, en el contexto STR:

$$head^B\ (\lambda\alpha.\alpha)\ STR\ = CONS\ \ STR\ \ ABS$$

es decir, requiere que xs tenga un valor $x : xs$ donde $x \neq \bot$. De la misma forma se obtiene que:

$$tail^B\ \tau = \lambda\alpha.\tau\ (CONS\ \ ABS\ \ \alpha)$$

y por tanto la estrictez de la expresión $head(tail\ xs)$ respecto a xs viene dad por:

$$\begin{array}{l}
B[\![head\ (tail\ xs)]\!]\ [(\lambda\alpha.\alpha)/xs]\ [head^B/head, tail^B/tail] \\
\quad = \quad head^B(tail^B(\lambda\alpha.\alpha)) \\
\quad = \quad \lambda\alpha.(CONS\ \ ABS\ \ (CONS\ \ \alpha\ ABS\ ))
\end{array}$$

□

La condición de seguridad es ahora la siguiente: τ es un transformador de proyecciones seguro para x con respecto a e en σ si para todas las proyecciones α y entornos de variables ρ:

$$\alpha(sem[\![e]\!]\ \rho\ \sigma) \sqsubseteq sem[\![e]\!](\rho[\tau\ \alpha\ (\rho(x))/x])\ \sigma \tag{6.1}$$

Es decir, si en la evaluación de e usamos $\tau\ \alpha\ (\rho(x))$ en lugar de $\rho(x)$ el resultado es al menos tan definido como el resultado de aplicar α al resultado de evaluar e. Si la expresión es una función f con un único parámetro x esto se traduce en: $\forall\alpha.f : \alpha \Rightarrow \tau\ \alpha$.

Davis y Wadler demostraron que B cumple la condición de seguridad, es decir, tomando $\tau = B[\![e]\!]\ [\overline{\tau}/\overline{z}]\ \psi$ se cumple 6.1. Para ello se demuestra algo más general: sean z_i las variables libres de e, e_i expresiones y τ_i transformadores seguros para x con respecto a e_i para cada i. Entonces:

$$\alpha(sem[\![e[\overline{e}/\overline{z}]]\!]\ \rho\ \sigma) \sqsubseteq sem[\![e[\overline{e}/\overline{z}]]\!](\rho[\tau\ \alpha\ (\rho(x))/x])\ \sigma$$

de lo cual se deduce lo que queremos.

Un ejemplo que muestra el mayor potencial de este análisis respecto al anterior es el siguiente:

Ejemplo 59

$$
\begin{aligned}
f\ x\ y\ z = \textbf{case}\ x\ \textbf{of} & \\
& true \Rightarrow y \\
& false \Rightarrow z
\end{aligned}
$$

con la llamada $f\ x\ y\ y$. En este caso el primer análisis no es capaz de deducir que es estricta en y mientras que B si lo detecta:

$$f^B\ (\lambda\alpha ABS\ )\ (\lambda\alpha.\alpha)\ (\lambda\alpha.\alpha) = STR$$

□

La restriccion del primer análisis se encuentra en que en el entorno de variables solamente una variable cada vez puede tener un valor distinto de $\lambda\alpha.ABS$. Llamemos A a la función de semántica abstracta del análisis anterior (a la que no dimos nombre anteriormente). A se puede obtener a partir de B sustituyendo en los lados izquierdos de la definición B por A, y en los lados derechos B por A' añadiendo la regla:

$$A'[\![e]\!]\ [\overline{\tau}/\overline{z}]\ \psi = \&_{i=1}^{n} A[\![e]\!]\ \phi_{ABS}\ [\tau_i/z_i]\ \psi$$

donde ϕ_{ABS} lleva a $\lambda\alpha.ABS$ a todas las variables.

Comentarios

Hemos visto en la primera parte de esta sección, cómo el análisis hacia atrás de una función con n parámetros, implica el cálculo de n funciones de propagación de contextos, cuyos resultados después se combinan de la forma adecuada. El cálculo de estas funciones implica la resolución de ecuaciones recursivas a través de iteración, como hemos visto en algunos ejemplos. Se podría pensar por tanto, que la inicial esperanza de mayor eficiencia en este tipo de análisis no se cumple. Sin embargo, el coste de estas iteraciones es menor en el caso del análisis hacia atrás. Una función n–aria analizada hacia delante, da lugar a una función abstracta en el dominio $A^n \to A$. Tal función requeriría una tabla de $|A|^n$ entradas para representarla (en el caso de usar tablas), y el cálculo del punto fijo tendría coste exponencial en el número de argumentos n. En contraste, el análisis hacia atrás de la misma función da lugar a n funciones de propagación de contextos en el dominio $C \to C$ (donde C es el dominio de contextos). Cada función se representaría mediante una tabla de $|C|$ entradas, y el cálculo del punto fijo no llevaría más de $n \times |C|^2$ pasos. Considerando el coste de cada iteración proporcional al tamaño del programa, obtenemos que en el peor de los casos el análisis hacia atrás requiere un tiempo cuadrático en el tamaño del programa en lugar de exponencial.

El punto débil del análisis hacia atrás, es como ya explicamos al principio de la sección, su difícil extension a orden superior. Sin embargo, una posible solución a este problema se encontraría, como indicó Hughes [Hug88], en la combinación del análisis hacia delante y hacia atrás. La idea es proporcionar a las funciones de orden superior de una interpretación abstracta que es una función de orden superior, no sobre valores abstractos, sino sobre funciones de propagación de contextos para cada argumento de la función original. No se desarrollan aquí con más detalle estas ideas, pero son motivo de un estudio futuro en mayor profundidad.

Capítulo 7

Otros enfoques relacionados

7.1 Interpretación abstracta usando relaciones de equivalencia parcial

7.1.1 Introducción

En los análisis vistos en el Capítulo 5, las propiedades representadas son conjuntos Scott–cerrados. Sin embargo, ya comentamos en la Sección 5.1.1 que algunas propiedades como ser una función estricta de cabeza o ser una función constante, no se pueden representar mediante conjuntos Scott–cerrados. Sí se puede hacer mediante relaciones de equivalencia. Una **relación de equivalencia total** es una relación binaria que cumple las propiedades reflexiva, simétrica y transitiva. Una **relación de equivalencia parcial** es una relación binaria que cumple las propiedades simétrica y transitiva.

Por ejemplo, la propiedad de ser constante se puede representar usando relaciones de equivalencia de la siguiente manera. Sean las relaciones de equivalencia total Id_D y All_D sobre un dominio D definidas por:

$$\forall x, x' \in D.x \ All_D \ x'$$
$$\forall x, x' \in D.x \ Id_D \ x' \Leftrightarrow x = x'$$

Sea una función $f : D \to E$ y dos relaciones S y T sobre D y E respectivamente. Escribimos $f : S \Rightarrow T$ si

$$\forall x, x' \in D.x \ S \ x' \Rightarrow (f \ x) \ T \ (f \ x')$$

El hecho de que una función es constante se representa mediante la fórmula $\forall x, x'.(f \ x) = (f \ x')$ y por tanto se puede escribir igualmente $f : All_D \Rightarrow Id_D$.

En el momento en que queremos introducir orden superior surge la necesidad de trabajar con relaciones de equivalencia parcial. Dadas dos relaciones de equivalencia parcial, S y T sobre dos dominios D y E respectivamente, y dos funciones $f, g : D \to E$, sea la relación de equivalencia parcial $(S \Rightarrow T)$ sobre el dominio $[D \to E]$ de las funciones continuas de D en E:

$$f \ (S \Rightarrow T) \ g \Leftrightarrow (\forall x, x' \in D.x \ S \ x' \Rightarrow (f \ x) \ T \ (g \ x'))$$

Según esta definición, f es constante si $f \ (All_D \Rightarrow Id_D) \ f$.

El hecho que obliga a hacer uso de las relaciones de equivalencia parcial es que aunque S y T sean relaciones de equivalencia total, $S \Rightarrow T$ no lo es necesariamente. Basta con ver que $All_D \Rightarrow Id_D$ no es total, porque solamente las funciones constantes cumplen la propiedad reflexiva.

Hunt [HS91] [Hun91] [Hun91] desarrolló una interpretación abstracta en la que las propiedades representadas por los puntos del dominio abstracto son relaciones de equivalencia parcial en lugar de conjuntos Scott–cerrados.

7.1.2 Relaciones de equivalencia parcial

Ya hemos dicho que una relación de equivalencia parcial es una relación binaria que cumple las propiedades simétrica y transitiva. A partir de este momento, por abreviar las llamaremos "pers".

Dada una relación de equivalencia parcial P sea $|P|$ el conjunto $\{x \in D \mid (x,x) \in P\}$. La restricción de P a $|P|$ es una relación de equivalencia total. Escribimos $x : P$ si $x \in |P|$.

Puesto que trabajamos con dominios de Scott, nos fijamos en aquellas pers que respetan su estructura. Estas son las pers **completas**. Son necesarias para tratar la recursión. Una per es completa si es estricta e inductiva, donde una relación $P \subseteq D \times E$ es **estricta** si $\perp_D P \perp_E$, y es **inductiva** si para toda cadena $\{x_i\}_{i \in w} \subseteq D$ y $\{y_i\}_{i \in w} \subseteq E$, se cumple

$$\bigsqcup_{i \in w} x_i \; P \; \bigsqcup_{i \in w} y_i$$

Denotamos por $CPER\ (D)$ a la clase de las pers completas definidas sobre un dominio D. $CPER\ (D)$ es un $\sqcap$–semiretículo completo ordenado por inclusión. El supremo de $CPER\ (D)$ es All_D. Llamemos cpers a las pers completas.

Dadas dos cpers $P \in CPER\ (D)$ y $Q \in CPER\ (E)$, podemos construir una nueva cper $(D \Rightarrow E) \in CPER\ ([D \to E])$ definida por:

$$f\ (P \Rightarrow Q)\ g \Leftrightarrow \forall x, x' \in D. x\ P\ x' \Rightarrow (f\ x)\ Q\ (g\ x')$$

A partir de las definiciones se cumple trivialmente que:

$$f : P \Rightarrow Q \Leftrightarrow f\ (P \Rightarrow Q)\ f$$

El siguiente resultado, que va a ser utilizado más adelante, nos dice cómo tratar funciones con varios argumentos.

Lema 60 *Sea* $f : D_1 \to \ldots \to D_k \to E$ *y pers* $P_1 \subseteq D_1^2 \ldots P_k \subseteq D_k^2, Q \subseteq Q^2$, *entonces* $f : P_1 \Rightarrow (\ldots (P_k \Rightarrow Q))$ *si y solo si:*

$$(x_1\ P_1\ x_1') \wedge \ldots \wedge (x_k\ P_1\ x_k') \Rightarrow (f\ x_1 \ldots x_k)\ Q\ (f\ x_1' \ldots x_k')$$

Usando la misma convención que en los tipos escribiremos $f : P_1 \Rightarrow \ldots \Rightarrow P_k \Rightarrow Q$.

Un ejemplo de aplicación de las cpers para representar propiedades en un análisis, es el análisis del momento de vinculación.

Ejemplo 61 Dada una descripción de los parámetros de un programa que se conocerán en momento de especialización, un análisis del momento de vinculación debe determinar qué partes de un programa dependen solamente de estas partes conocidas (y por tanto son también conocidas en momento de evaluación parcial).

En la literatura el hecho de que x quedará fijado en momento de evaluación parcial se describe diciendo que x es *estático*, mientras que el hecho contrario se describe diciendo que x es *dinámico*. En el marco de las pers identificaremos estático con Id y dinámico con All. Sea por ejemplo la función $K(x,y) = x$, que cumple que $K : Id \times All \Rightarrow Id$. Podemos leer esta propiedad diciendo que dado un primer argumento estático y un segundo argumento dinámico, K devuelve un resultado estático. Luego, si tenemos $sem[\![e]\!] : D_1 \to \ldots \to D_k \to E$ y $P_1, \ldots, P_k, Q$ son cpers, mediante el análisis queremos comprobar condiciones de la forma:

$$sem[\![e]\!] : P_1 \Rightarrow \ldots \Rightarrow P_k \Rightarrow Q$$

□

7.1.3 La interpretación abstracta

El lenguaje con el que trabajamos en esta sección es el λ–cálculo con tipos de la Sección 5.3.2, al que se puede añadir los tipos producto y lista. La semántica estándar es la habitual, ya presentada en la Sección 5.3.2. Las constantes del lenguaje son el operador de punto fijo **fix**, el condicional **if**, las constructoras y destructoras de pares (*pair*, *fst* y *snd*), y algunas funciones sobre listas (*nil*, *cons*, *null*, *hd* y *tl*).

En esta interpretación abstracta, cada B_σ es, igual que en capítulos anteriores, un retículo finito. La diferencia estriba en que a cada elemento del dominio abstracto $a \in B_\sigma$, la función de concreción γ_σ le hace corresponder una cper $\gamma_\sigma(a) \in CPER\ (D_\sigma)$.

La interpretación de las constantes debe cumplir ahora una nueva condición para asegurar la corrección del análisis. Deseamos que para cada término cerrado $e : \sigma_1 \to \ldots \to \sigma_k \to \tau$ y para cada $a_1 \in B_{\sigma_1}, \ldots, a_k \in B_{\sigma_k}, b \in B_\tau$ se cumpla:

$$tabs[\![e]\!]\ a_1 \ldots a_k = b \Rightarrow sem[\![e]\!] : \gamma_1(a_1) \Rightarrow \ldots \Rightarrow \gamma_k(a_k) \Rightarrow \gamma_\tau(b)$$

Así, en el caso en que $k = 1$, si $\gamma_{\sigma_1}(a) = P$ y $\gamma_\tau(b) = Q$, entonces para comprobar si $sem[\![e]\!] : P \Rightarrow Q$ basta con comprobar si $tabs[\![e]\!]\ a = b$.

El dominio abstracto

El dominio abstracto depende del análisis realizado, al igual que antes. Para cada análisis concreto será necesario proporcionar un retículo para los tipos básicos B_A y para cada tipo construido $B_{\sigma\to\tau}$, $B_{\sigma\times\tau}$ y $B_{list(\tau)}$. Al igual que en los análisis del Capítulo 5, $B_{\sigma\to\tau} = [B_\sigma \to B_\tau]$ en general para todos los análisis. Veamos un ejemplo concreto.

Ejemplo 62 (análisis del momento de vinculación)
En el caso del análisis del momento de vinculación la definición del dominio abstracto viene dado en la Figura 7.1. Para los tipos base tenemos el dominio formado por dos puntos S y D que significan intuitivamente estático y dinámico respectivamente. La construcción para las listas

añade un elemento supremo. Los demás puntos, $SPINE(a)$, representan listas de estructura estática y elementos cuya propiedad es descrita por a.

$\square$

$$
\begin{array}{lcl}
B_A & = & \{D, S\} \text{ donde } S \sqsubset D \\
B_{\sigma \to \tau} & = & [B_\sigma \to B_\tau] \\
B_{\sigma \times \tau} & = & B_\sigma \times B_\tau \\
B_{list(\tau)} & = & \{SPINE(a) \mid a \in B_\tau\} \cup \{D\} \\
 & & \text{donde } SPINE(a) \sqsubseteq SPINE(b) \Leftrightarrow a \sqsubseteq b \text{ y } SPINE(a) \sqsubset D
\end{array}
$$

Figura 7.1: Dominio abstracto

Las funciones de concreción

Para cada tipo τ debemos definir una función de concreción γ_τ que asocie a cada punto del dominio abstracto $a \in B_\tau$ una cper $\gamma_\tau(a) \in CPER\ (D_\tau)$. Para un análisis concreto se debe definir una para el dominio base, y otra para cada tipo construido. La definición para las funciones, en la Figura 7.2 es general a todos los análisis y surge a partir de la prueba de corrección como veremos más adelante (Lema 67).

Se cumplen algunas propiedades que serán útiles más adelante:

- $P \subseteq Q \Rightarrow P \cap Q = P$
- Para cualquier $P \subseteq D^2$, $P \Rightarrow All_E = All_{[D \to E]}$
- $P \supseteq P' \wedge Q \subseteq Q' \Rightarrow (P \Rightarrow Q) \subseteq (P' \Rightarrow Q')$, es decir $\Rightarrow$ es antimonótona en su primer argumento y monótona en el segundo.

Ejemplo 63 En el caso del análisis del momento de vinculación las funciones de concreción se definen en la Figura 7.2. Se demuestra que para todo σ y a, $\gamma_\sigma(a)$ es una cper.

La concreción de los elementos del dominio base son las cpers All e Id. Como ejemplo que ilustra la definición dada para las funciones, sea una función de tipo $int \to int$. El dominio abstracto $D^{\#}_{int \to int}$ es:

$$
\begin{array}{c}
\lambda x.D \\
| \\
\lambda x.x \\
| \\
\lambda x.S
\end{array}
$$

y, usando las propiedades vistas anteriormente, la función de concreción es la siguiente (para complicar lo menos posible la notación omitimos aquí los corchetes de $[int \to int]$):

$$\begin{array}{l}\gamma_{int\to int}(\lambda x.D)\\ \quad = \ (\gamma_{int}(S) \Rightarrow \gamma_{int}(D)) \cap (\gamma_{int}(D) \Rightarrow \gamma_{int}(D))\\ \quad = \ (Id_{int} \Rightarrow All_{int}) \cap (All_{int} \Rightarrow All_{int})\\ \quad = \ All_{int\to int} \cap All_{int\to int}\\ \quad = \ All_{int\to int}\end{array}$$

$$\begin{array}{l}\gamma_{int\to int}(\lambda x.x)\\ \quad = \ (\gamma_{int}(S) \Rightarrow \gamma_{int}(S)) \cap (\gamma_{int}(D) \Rightarrow \gamma_{int}(D))\\ \quad = \ (Id_{int} \Rightarrow Id_{int}) \cap (All_{int} \Rightarrow All_{int})\\ \quad = \ (Id_{int} \Rightarrow Id_{int}) \cap All_{int\to int}\\ \quad = \ Id_{int} \Rightarrow Id_{int}\end{array}$$

$$\begin{array}{l}\gamma_{int\to int}(\lambda x.S)\\ \quad = \ (\gamma_{int}(S) \Rightarrow \gamma_{int}(D)) \cap (\gamma_{int}(S) \Rightarrow \gamma_{int}(D))\\ \quad = \ (Id_{int} \Rightarrow Id_{int}) \cap (All_{int} \Rightarrow Id_{int})\\ \quad = \ All_{int} \Rightarrow Id_{int}\end{array}$$

Esto quiere decir intuitivamente que representan respectivamente funciones dinámicas, estáticas y constantes.

Como ejemplo para las listas, el dominio para las listas de booleanos $D^{\#}_{list(bool)}$ es:

$$\begin{array}{c} D \\ | \\ SPINE(D) \\ | \\ SPINE(S) \end{array}$$

y la función de concreción hace corresponder a D con $All_{list(bool)}$, a $SPINE(S)$ con el menor punto fijo de F_S que es $Id_{list(bool)}$ y a $SPINE(D)$ con el punto fijo de F_D, que es la cper que relaciona todas las listas finitas de la misma longitud, todas las listas parciales de la misma longitud y todas las lista infinitas.

$\square$

Es importante en relación a la corrección del análisis que las γ_σ sean monótonas.

$$\gamma_\sigma : B_\sigma \to CPER\ (D_\sigma)$$

$$\begin{array}{lcl}
\gamma_A(a) & = & \left\{ \begin{array}{ll} All_A & \text{si } a = D \\ Id_A & \text{si } a = S \end{array} \right. \\[2ex]
\gamma_{\sigma\to\tau}(f) & = & \bigcap_{a\in B_\sigma} (\gamma_\sigma(a) \Rightarrow \gamma_\tau(f\ a)) \\[2ex]
\gamma_{\sigma\times\tau}(a,b) & = & \gamma_\sigma(a) \times \gamma_\tau(b) \\[2ex]
\gamma_{list(\tau)}(a) & = & \left\{ \begin{array}{ll} All_{list(\tau)} & \text{si } a = D \\ lfp\ F_b & \text{si } a = SPINE(b) \end{array} \right. \\[2ex]
 & & \text{donde } F_b : CPER\ (D_{list(\tau)}) \to CPER\ (D_{list(\tau)}) \\
 & & F_b(P) = \{(\bot,\bot)\} \cup \{([],[])\} \cup \{(x : l, x' : l') \mid (x,x') \in \gamma_\tau(b), (l,l') \in P\}
\end{array}$$

Figura 7.2: Función de concreción

Interpretación de las constantes

La interpretación abstracta de las constantes depende del análisis que estemos realizando, aunque hay que tener en cuenta algunas condiciones semejantes a la condición de seguridad de las constantes de los análisis del Capítulo 5 para que el análisis en cuestión sea correcto (ver Sección 7.1.5).

Ejemplo 64 Continuamos con el ejemplo del momento de vinculación. Para simplificar notación omitiremos los subíndices de tipo de las constantes, pues el tipo está suficientemente claro en cada caso.

La interpretación abstracta de algunas de las constantes, como **fix**, *pair*, *fst* y *snd* es la misma que la estándar:

$$\begin{array}{lcl}
\mathbf{fix}^\#(f) & = & \bigsqcup_{n=0}^{\infty} f^n(\bot_{B_\tau}) \\[2ex]
pair^\#\ x\ y & = & (x,y) \\
fst^\#\ (x,y) & = & x \\
snd^\#\ (x,y) & = & y
\end{array}$$

donde $f : \tau \to \tau$. Las demás se interpretan de la siguiente forma:

- En el condicional, si la condición es dinámica, el resultado es claramente dinámico. Si la condición es estática, la monotonía de γ_τ se puede usar para mostrar que es correcto tomar el supremo de las alternativas:

  $$\mathbf{if}^\#\ a\ b\ c = \left\{ \begin{array}{ll} \top_{B_\tau} & \text{si } a = D \\ b \sqcup c & \text{si } a = S \end{array} \right.$$

  donde $b, c : \tau$.

- Para las listas:

$$\begin{array}{rcl} nil^{\#} & = & SPINE(\bot_{B_\tau}) \\ cons^{\#}\ a\ l & = & \left\{ \begin{array}{ll} SPINE(a \sqcup a') & \text{si } l = SPINE(a') \\ D & \text{si } l = D \end{array} \right. \\ null^{\#}\ l & = & \left\{ \begin{array}{ll} S & \text{si } l = SPINE(a) \\ D & \text{si } l = D \end{array} \right. \\ hd^{\#}\ l & = & \left\{ \begin{array}{ll} a & \text{si } l = SPINE(a) \\ \top_{B_\tau} & \text{si } l = D \end{array} \right. \\ tl^{\#}\ l & = & l \end{array}$$

  donde $l : list(\tau)$, $a : \tau$. La definición de $cons^{\#}$ viene motivado por lo siguiente: añadir un nuevo elemento a una lista con estructura dinámica produce una lista que sigue teniendo estructura dinámica; mientras que añadir un elemento a una lista con estructura estática produce una lista con estructura estática.

- Para el resto de constantes:

$$c^{\#}_{A_1 \to \ldots \to A_n \to B}\ a_1 \ldots a_n = \bigsqcup_{i=1}^{n} a_i$$

□

7.1.4 Relaciones lógicas

Ya comentamos en la Sección 5.1.2 que una forma alternativa a la usada en el Capítulo 5 para dar la corrección de los análisis basados en interpretación abstracta es el uso de las relaciones lógicas.

Supongamos que tenemos dos interpretaciones distintas de un lenguaje, $I = (\{D_A\}, \{c_\sigma\})$ y $K = (\{B_\sigma\}, \{c^{\#}_\sigma\})$, donde la primera componente del par denota la interpretación de los tipos y la segunda denota la interpretación de las constantes.

Una **relación lógica binaria** $R : I, K$ es una familia $\{R_\sigma\}$ donde $R_\sigma \subseteq D_\sigma \times B_\sigma$ tales que para todo σ, τ y para toda $f \in D_{\sigma \to \tau}, h \in B_{\sigma \to \tau}$:

$$\begin{array}{c} f\ R_{\sigma \to \tau}\ h \\ \Leftrightarrow \\ \forall x \in D_\sigma, a \in B_\sigma . x\ R_\sigma\ a \Rightarrow (f\ x)\ R_\tau\ (h\ a) \end{array}$$

Las relaciones lógicas binarias son suficientes para establecer la corrección de los análisis del estilo del Capítulo 5. Sin embargo, no son suficientes para los análisis que utilizan pers. En este caso se necesitan relaciones ternarias $R_\sigma \subseteq D_\sigma \times D_\sigma \times B_\sigma$ de forma que cada punto $a \in B_\sigma$ está asociado con una per dada por $\{(x, y) \in D_\sigma \times D_\sigma \mid R_\sigma(x, y, a)\}$.

Generalizando la definición de relación lógica al caso n–ario, $R : I_1 \dots I_n$, donde $I_j = (\{D^j_A\}, \{c^j_\sigma\})$, es una relación lógica $R = \{R_\sigma\}$ si para toda $f_1 \in D^1_{\sigma\to\tau}, \dots f_n \in D^n_{\sigma\to\tau}$:

$$(f_1, \dots, f_n) \in R_{\sigma\to\tau}$$
$$\Leftrightarrow$$
$$\forall a_1 \in D^1_\sigma, \dots, a_n \in D^n_\sigma.(a_1, \dots, a_n) \in R_\sigma \Rightarrow (f_1\ a_1, \dots, f_n\ a_n) \in R_\tau$$

Se cumple la siguiente propiedad, demostrada por Abramsky [Abr90].

Proposición 65 *Sea $R : I_1 \dots I_k$ una relación lógica. Supongamos que para cada constante c_τ, $(c^1_\tau, \dots, c^k_\tau) \in R_\tau$. Entonces para toda $e : \sigma$, y para toda $\rho_i \in Env^i$:*

$$(\rho_1, \dots, \rho_k) \in R \Rightarrow ([\![e]\!]^{I_1}\ \rho_1, \dots, [\![e]\!]^{I_k}\ \rho_k) \in R_\sigma$$

donde $[\![e]\!]^{I_j}$ denota la semántica de e bajo la interpretación I_j.

Esta propiedad nos permitirá más adelante reducir la corrección del análisis a la de las constantes del lenguaje.

7.1.5 Corrección

Recordemos que para un término cerrado $e : \sigma_1 \to \dots \to \sigma_k \to \tau$, nuestra exigencia a la interpretación es que para todo $a_1 \in B_{\sigma_1}, \dots, a_k \in B_{\sigma_k}$ se cumpla:

$$sem[\![e]\!] : \gamma_{\sigma_1}(a_1) \Rightarrow \dots \Rightarrow \gamma_{\sigma_k}(a_k) \Rightarrow \gamma_\tau(tabs[\![e]\!]\ a_1 \dots a_k)$$

Lema 66 *Para cada par de funciones continuas $f : D_{\sigma_1} \to \dots \to D_{\sigma_k} \to D_\tau$ y $h : B_{\sigma_1} \to \dots \to B_{\sigma_k} \to B_\tau$, se cumple que:*

$$\forall a_1 \in B_{\sigma_1}, \dots, a_k \in B_{\sigma_k}.f : \gamma_{\sigma_1}(a_1) \Rightarrow \dots \Rightarrow \gamma_{\sigma_k}(a_k) \Rightarrow \gamma_\tau(h\ a_1 \dots a_k)$$
$$\Leftrightarrow$$
$$f \in |\gamma_{\sigma_1\to\dots\to\sigma_k\to\tau}(h)|$$

Luego el hecho de que la interpretación abstracta es correcta para una expresión cerrada $e : \sigma$ se puede expresar diciendo que $sem[\![e]\!] \in |\gamma_\sigma(tabs[\![e]\!])|$.

Es necesario generalizar la idea de corrección a los términos con variables. Diremos que dados dos entornos $\rho \in Env$ y $\rho' \in Env^\#$, ρ' es correcto respecto a ρ si $\forall x^\tau.\rho\ x^\tau \in |\gamma_\tau(\rho'\ x^\#)|$.

Luego la condición general de corrección es: la interpretación abastracta es correcta si para toda expresión $e : \sigma$, y para todo par de entornos $\rho \in Env$ y $\rho' \in Env^\#$ tales que ρ' es correcto respecto a ρ, entonces:

$$(sem[\![e]\!]\ \rho) \in |\gamma_\sigma(tabs[\![e]\!]\ \rho')|$$

Intentemos ahora expresar la condición de corrección en función de una familia de relaciones lógicas. Sea una familia $R = \{R_\sigma\}$ donde $R_\sigma \subseteq D_\sigma \times D_\sigma \times B_\sigma$ está definida por:

$$R_\sigma(x, x', a) \Leftrightarrow (x, x') \in \gamma_\sigma(a)$$

Escribiremos $R(\rho_1, \rho_2, \rho')$ cuando queramos decir:

$$\forall x^\tau . R_\tau(\rho_1\ x^\tau, \rho_2\ x^\tau, \rho'\ x^\tau)$$

Se cumple que la R así definida es una relación lógica ternaria tal como la definimos anteriormente. La demostración de este hecho motiva la definición de la función de concreción para las funciones dada en la Figura 7.2.

Lema 67 *$R : I, I, K$ es lógica si y solo si para cada σ, τ y para toda $f \in [B_\sigma \to B_\tau]$:*

$$\gamma^R_{\sigma\to\tau}(f) = \bigcap_{a\in B_\sigma} (\gamma^R_\sigma(a) \Rightarrow \gamma^R_\tau(f\ a))$$

donde $\gamma^R_\sigma(a) = \{(x, y) \in D_\sigma \times D_\sigma \mid (x, y, a) \in R_\sigma\}$.

Usando estas relaciones lógicas podemos reformular la corrección diciendo que la interpretación abstracta es correcta si para toda expresión $e : \sigma$, para todo par de entornos $\rho \in Env$ y $\rho' \in Env^\#$ tales que $R(\rho, \rho, \rho')$ se cumple:

$$R_\sigma(sem[\![e]\!]\ \rho, sem[\![e]\!]\ \rho, tabs[\![e]\!]\ \rho')$$

Por ser R relación lógica se cumple la proposición 65. Esto quiere decir que en un análisis particular, para probar que es correcto bastará con probar que para cada constante c_τ, se cumple $c_\tau \in |\gamma_\tau(c^\#_\tau)|$.

Para demostrar esta propiedad de seguridad de las constantes en el caso de recursión, por ejemplo para **fix**, es necesario que las R_σ sean estrictas e inductivas, lo cual se puede demostrar fácilmente si cada $\gamma_\sigma(a)$ es estricta e inductiva. Por ello hemos introducido las cpers.

Veamos un par de ejemplos con el análisis del momento de vinculación. El primero de ellos muestra que **if**cumple la condición de seguridad para las constantes y en el segundo se muestra un ejemplo de aplicación del análisis.

Ejemplo 68 Sea la constante $\mathbf{if}_{bool\to\tau\to\tau\to\tau}$. Veamos que cumple la condición deseada para las constantes. Tenemos que comprobar que $\mathbf{if}_{bool\to\tau\to\tau\to\tau} \in |\gamma_{bool\to\tau\to\tau\to\tau}(\mathbf{if}^\#_{bool\to\tau\to\tau\to\tau})|$. Lo cual, por el Lema 66, es equivalente a demostrar (omitimos los subíndices de **if** por legibilidad):

$$\forall a \in B_{bool}, b \in B_\tau, c \in B_\tau . \mathbf{if} : \gamma_{bool}(a) \Rightarrow \gamma_\tau(b) \Rightarrow \gamma_\tau(c) \Rightarrow \gamma_\tau(\mathbf{if}^\#\ a\ b\ c)$$

A su vez, por el Lema 60 es equivalente a demostrar:

$$\begin{gathered}\forall a \in B_{bool}, b \in B_\tau, c \in B_\tau . x_1\ \gamma_{bool}(a)\ x'_1 \wedge x_2\ \gamma_\tau(b)\ x'_2 \wedge x_3\ \gamma_\tau(c)\ x'_3 \\ \Rightarrow \\ (\mathbf{if}\ x_1\ x_2\ x_3)\ \gamma_\tau(\mathbf{if}^\#\ a\ b\ c)\ (\mathbf{if}\ x'_1\ x'_2\ x'_3)\end{gathered}$$

Se puede dividir en dos casos: $a = S$ y $a = D$. El caso interesante se da cuando $a = S$, pues si $a = D$, $\mathbf{if}^\#\ a\ b\ c = \top_\tau$ y $\gamma_\tau(\top_\tau) = All_\tau$ con lo que se cumple trivialmente.

Si $a = S$, entonces $\gamma_{bool}(a) = Id_{bool}$, $x_1\ \gamma_{bool}(a)\ x'_1 \Leftrightarrow x_1 = x'_1$ y $\mathbf{if}^\#\ a\ b\ c = b \sqcup c$. Luego, en este caso se reduce a demostrar:

$$x_2\ \gamma_\tau(b)\ x'_2 \wedge x_3\ \gamma_\tau(c)\ x'_3 \Rightarrow (\mathbf{if}\ x_1\ x_2\ x_3)\ \gamma_\tau(b \sqcup c)\ (\mathbf{if}\ x_1\ x'_2\ x'_3)$$

Por monotonía de γ_τ tenemos que $\gamma_\tau(b \sqcup c) \supseteq \gamma_\tau(b)$ y $\gamma_\tau(b \sqcup c) \supseteq \gamma_\tau(c)$. Por tanto para demostrar lo que queremos bastaría con demostrar que $\forall P, P', Q \in CPER(D_\tau)$ tales que $P, P' \subseteq Q$ se cumple:

$$x_2 \ P \ x_2' \wedge x_3 \ P' \ x_3' \Rightarrow (\textbf{if} \ x_1 \ x_2 \ x_3) \ Q \ (\textbf{if} \ x_1 \ x_2' \ x_3')$$

lo cual se cumple trivialmente por la definición de **if**.

□

Ejemplo 69 Sea el análisis visto del momento de vinculación, y sea el programa

$$map \ snd \ l$$

donde $l : list(int \times bool)$ y map está definida como sigue:

$$map \ f \ l = \textbf{if} \ (null \ l) \ nil \ (cons \ (f \ (hd \ l)) \ (map \ f \ (tl \ l)))$$

El cálculo del punto fijo para map es el siguiente:

$$
\begin{array}{rcl}
map_0^{\#} \ h \ c & = & SPINE(S) \\[2ex]
map_1^{\#} \ h \ c & = & \left\{ \begin{array}{ll} D & \text{si } c = D \\ cons^{\#} \ (h(a,b)) \ (map_0^{\#} \ h \ SPINE((a,b))) & \text{si } c = SPINE((a,b)) \\ = SPINE(h(a,b) \sqcup h(a,b)) & \\ = SPINE(h(a,b)) & \end{array} \right. \\
map_2^{\#} \ h \ c & = & \left\{ \begin{array}{ll} D & \text{si } c = D \\ cons^{\#} \ (h(a,b)) \ (map_1^{\#} \ h \ SPINE((a,b))) & \text{si } c = SPINE((a,b)) \\ = SPINE(h(a,b) \sqcup h(a,b)) & \\ = SPINE(h(a,b)) & \end{array} \right. \\
& \vdots & \\[2ex]
map^{\#} \ h \ c & = & \left\{ \begin{array}{ll} D & \text{si } c = D \\ SPINE(h(a,b)) & \text{si } c = SPINE((a,b)) \end{array} \right.
\end{array}
$$

Luego:

$$map^{\#} \ snd^{\#} \ D = D$$

y

$$map^{\#} \ snd^{\#} \ SPINE((D,S)) = SPINE(snd^{\#}(D,S)) = SPINE(S)$$

Esto quiere decir que si la estructura de la lista es dinámica entonces el resultado es también dinámico. Mientras que si la estructura de la lista es estática y todas las segundas componentes son estáticas entonces el resultado es completamente estático.

□

7.1.6 Funciones de abstracción e interpretaciones mejores

Ya hemos dicho anteriormente que cuando exigimos que la función de abstracción y la de concreción formen una conexión de Galois (o lo que es lo mismo, que la función de abstracción sea adjunta por la izquierda de la función de concreción) entonces a través de la función de abstracción $abs(f)$ obtenemos la mejor interpretación posible de f.

En este caso sucede lo mismo. Cada γ_σ tiene adjunto por la izquierda Abs_σ definido por:

$$Abs_\sigma\ X = \sqcap\{a \in B_\sigma \mid X \subseteq \gamma_\sigma(a)\}$$

Se cumple que si $F \subseteq [D_\sigma \to D_\tau]^2$ y $a \in B_\sigma$:

$$Abs_{\sigma\to\tau}\ F\ a = Abs_\tau\{(f\ x, f'\ x') \mid (f, f') \in F, (x, x') \in \gamma_\sigma(a)\}$$

Entonces la mejor interpretación correcta (aquella tal que $\gamma_\sigma(c^{\#}_\sigma)$ es el conjunto más pequeño) que se puede tomar para las constantes es $Abs_\sigma\{(c_\sigma, c_\sigma)\}$.

Ejemplo 70 Para el caso base del análisis del momento de vinculación tenemos por ejemplo:

$$abs_A(d, d') = \begin{cases} S & \text{si } d = d' \\ D & \text{e.o.c.} \end{cases}$$

Para las funciones:

$$abs_{\sigma\to\tau}\ (f, f')\ a = \bigsqcup\{abs_\tau(f\ d, f'\ d') \mid abs_\sigma(d, d') \sqsubseteq a\}$$

y para los productos:

$$abs_{\sigma\times\tau}((d_1, d_2), (d_1', d_2')) = (abs_\sigma(d_1, d_1'), abs_\tau(d_2, d_2'))$$

Abs es la extensión de estas funciones a conjuntos.

□

7.1.7 Relación con los análisis anteriores

Recordemos que una proyección es una función continua $\alpha : D \to D$ tal que $\alpha \sqsubseteq id_D$ y $\alpha \circ \alpha = \alpha$.

Dada una función $\alpha : D \to E$ se puede definir una relación de equivalencia $E_\alpha \subseteq D^2$ de la siguiente manera:

$$x\ E_\alpha\ x' \Leftrightarrow \alpha\ x = \alpha\ x'$$

Por ejemplo $E_{\lambda x.x} = Id$ y $E_{\lambda x.\perp} = All$. Trivialmente las E_α son relaciones de equivalencia total.

La conexión entre el análisis con pers y el análisis usando proyecciones viene dada por la siguiente proposición [Hun91].

Proposición 71 *Para toda proyección $\alpha : D \to D$, $\beta : E \to E$, y función $f : D \to E$:*

$$\beta \circ f = \beta \circ f \circ \alpha \Leftrightarrow f : E_\alpha \Rightarrow E_\beta$$

Esto quiere decir que cualquier pregunta sobre una función que se pueda hacer usando proyecciones, de la forma $\beta \circ f = \beta \circ f \circ \alpha$, se puede hacer también usando pers, de la forma $f : P \Rightarrow Q$. En concreto $E_\alpha \Rightarrow E_\beta$ es una relación de equivalencia total. Esto quiere decir que el análisis de proyecciones es equivalente al uso de relaciones de equivalencia total.

Puesto que hay pers que no se corresponden con ninguna proyección, hay preguntas que se pueden hacer usando pers, que no se pueden hacer usando proyecciones. En conclusión, la interpretación abstracta usando pers *es más potente* que al análisis de proyecciones. Además incorpora de forma sencilla el orden superior. Y por supuesto *es más potente* que la interpretación abstracta usando conjuntos Scott–cerrados, puesto que por ejemplo permite expresar el hecho de que una función es constante. Además cualquier análisis que use conjuntos Scott–cerrados se puede expresar usando pers, ya que un conjunto se puede representar como una relación en la que cada elemento solamente se relaciona consigo mismo. El precio a pagar por esta potencia es la complejidad del marco teórico desarrollado.

7.2 Interpretación abstracta basada en semántica operacional

7.2.1 Introducción

Las técnicas de interpretación abstracta vistas hasta ahora están basadas en la semántica denotacional proporcionada al λ–cálculo, la cual sirve de referencia para demostrar la corrección del análisis desarrollado.

Una de las principales aplicaciones de la interpretación abstracta es la optimización de los compiladores. En la semántica denotacional se pierden muchas propiedades que dependen de la implementación concreta de los lenguajes. En algunos análisis la información que se pretende obtener es de tipo operacional.

Una forma de obtener información operacional sobre la ejecución de los programas consiste en **instrumentar** la semántica denotacional, es decir, en extender la semántica denotacional con la información deseada. La instrumentación podría, por ejemplo, recoger una traza o historia del cómputo, secuencias de referencias a variables, dependencias entre valores, propiedades de los entornos etc.

Sestoft [Ses88] [Ses89] [SA89] [Ses91] y otros autores (ver Capítulo 3) se decantan por esta vertiente operacional de la interpretación abstracta y proporcionan numerosos análisis útiles en la optimización de los compiladores. Se presenta aquí con más detalle el enfoque de Sestoft por ser el más nuevo y potente.

En el trabajo llevado a cabo por Sestoft, cada uno de los análisis desarrollados se aplican a una transformación posterior a nivel de lenguaje objeto. Aunque en esta sección nos centramos solamente en el estilo de análisis realizado, Sestoft presenta además:

- Una forma de implementar las optimizaciones que se pretenden llevar a cabo a partir de la información obtenida por los distintos análisis. La forma de implementar las optimizaciones consiste en especializar el código generado para una máquina abstracta, según se ha visto en el Capítulo 1. En concreto, las máquina abstractas en las que se basa Sestoft son:
    - Una variante de la máquina SECD de Peter Landin [Lan63] para un lenguaje estricto al que se aplica globalización de variables.

– Una extensión de la máquina de Krivine [Cré90] [Cur90], llamada máquina K, extendida con valores básicos, pereza y estructuras de datos (codificados como funciones) partiendo de ideas de la TIM de Fairbairn y Wray [FW86] [WF89], para el resto de análisis aplicados a un lenguaje perezoso.

- Una demostración de corrección respecto a la máquina abstracta correspondiente.

7.2.2 Los análisis

Los lenguajes funcionales a los que se aplican los análisis de esta sección no son necesariamente con tipos. En los dos primeros análisis de esta sección se trabaja con un lenguaje funcional perezoso de orden superior que incluye definiciones locales:

$$
\begin{array}{lll}
e ::= & \lambda x.e & \lambda - \text{abstracción} \\
 & \mid x & \text{Variable} \\
 & \mid (e\ e) & \text{Aplicación} \\
 & \mid (\mathbf{A}\ e_1 \ldots e_n) & \text{Función primitiva totalmente aplicada} \\
 & \mid \mathbf{if}\ e_1\ e_2\ e_3 & \text{Condicional} \\
 & \mid \mathbf{letrec}\ f_1 = e_1 \ldots f_n = e_n\ \mathbf{in}\ e_0 & \text{Definiciones locales}
\end{array}
$$

Se considerará que los tipos de datos

$$t ::= C_1\ t_{11} \ldots t_{1c(1)} \mid \ldots \mid C_n\ t_{n1} \ldots t_{nc(n)}$$

están codificados mediante funciones:

$$C_i = \lambda x_1 . \ldots \lambda x_{c(i)} . \lambda c_1 . \ldots \lambda c_n . c_i\ x_1 \ldots x_{c(i)}$$

y el destructor

$$
\begin{array}{l}
\mathbf{case}\ e\ \mathbf{of} \\
\quad C_1\ x_{11} \ldots x_{1c(1)} \Rightarrow e_1 \\
\quad \ldots \\
\quad C_n\ x_{n1} \ldots x_{nc(n)} \Rightarrow e_n
\end{array}
$$

está codificado mediante la siguiente aplicación:

$$e\ (\lambda x_{11} . \ldots \lambda x_{1c(1)} . e_1) \ldots (\lambda x_{n1} . \ldots \lambda x_{nc(n)} . e_n)$$

Así, si el discriminante del **case** fuera de la forma $(C_i\ e'_1 \ldots e'_{c(i)})$, entonces se seleccionará la rama $(\lambda x_{i1} . \ldots \lambda x_{ic(i)} . e_i)$ y se aplicará a $e'_1, \ldots, e'_{c(i)}$. Luego el resultado del **case** será el lado derecho e_i evaluado con x_{ij} ligada a la componente e'_j de la estructura de datos.

El análisis de globalización se lleva a cabo sobre un lenguaje estricto, donde un programa tiene la forma:

$$
\begin{array}{ll}
\mathbf{letrec} & f^1\ x^1_1 \ldots x^1_{i_1} = e^1 \\
 & \ldots \\
 & f^n\ x^n_1 \ldots x^n_{i_n} = e^n \\
\mathbf{in}\ e_0 &
\end{array}
$$

donde

$$
\begin{array}{rll}
e ::= & x & \text{Variable} \\
 & \mid f^i & \text{Función definida} \\
 & \mid (e\ e) & \text{Aplicación} \\
 & \mid (\mathbf{A}\ e_1 \ldots e_n) & \text{Función primitiva totalmente aplicada} \\
 & \mid \mathbf{if}\ e_1\ e_2\ e_3 & \text{Condicional}
\end{array}
$$

Los dominios abstractos con los que se trabaja son en casi todos los casos retículos formados por conjuntos ordenados por inclusión. Una vez definido el dominio particular sobre el que trabaja, por ejemplo conjuntos de etiquetas, define un conjunto de ecuaciones mutuamente recursivas (que surgen del uso de funciones auxiliares a la que interesa) cuyo punto fijo debe calcularse. Se asegura la existencia de solución por ser funciones monótonas definidas sobre retículos completos.

En los dos primeros análisis consideramos que cada una de las λ–abstracciones está etiquetada de forma única $\lambda^l x.e$ y cada una de las apariciones de x ligada por dicha λ–abstracción con la misma etiqueta x^l. Supongamos también que cada ligadura en un **letrec** se etiqueta de la forma $f =^l e$. Estas etiquetas son disjuntas de las etiquetas correspondientes a las λ–abstracciones.

Análisis de clausuras

Recordemos que este análisis pretende encontrar el conjunto de clausuras a las que se puede evaluar una expresión, y el conjunto de funciones argumento a los que una determinada función de orden superior se puede aplicar, ver Sección 1. Los resultados buscados son por tanto conjuntos de etiquetas $l \in Label$, cada una de las cuales representa a la λ–abstracción que etiqueta.

El dominio abstracto con el que se trabaja es, por tanto, el retículo completo formado por conjuntos de etiquetas ordenados por inclusión. El ínfimo es el conjunto vacío; el supremo es el conjunto total de etiquetas del programa, y los operadores de supremo e ínfimo vienen dados por la unión e intersección de conjuntos.

Se definen las siguientes funciones:

- $\phi\ l$, llamada **función de descripción de clausuras resultado**, calcula el conjunto de λ–abstracciones a las que se evalúa el cuerpo e_1 de $\lambda^l x.e_1$[1].

- $\rho\ l$, llamada **función de descripción de clausuras argumento**, calcula el conjunto de λ–abstracciones a las que $\lambda^l x.e_1$ se puede aplicar, es decir, el conjunto de λ–abstracciones a las que se puede ligar la variable x^l.

- $P_e[\![e]\!]\ \phi\ \rho$, **llamada función de análisis de clausuras**, calcula el conjunto de clausuras a las que se puede evaluar la expresión e.

- $P_v[\![e]\!]\ \phi\ \rho\ l'$, llamada **función de propagación de clausuras**, calcula el conjunto de λ–asbtracciones a las que la λ–abstracción etiquetada con l' se puede aplicar en la expresión e.

[1]Nótese que estamos en un lenguaje sin tipos y, en principio, e_1 podría ser de tipo funcional. Si no lo es, se devuelve el conjunto vacío.

En el caso de variables ligadas en un **letrec** de la forma $f =^l e$, $\phi\ l$ y $\rho\ l$ tienen el siguiente significado: el conjunto de λ–abstracciones a las que se puede evaluar e y el conjunto de las λ–abstracciones a las que se puede ligar f respectivamente. Los dos conjuntos coinciden.

$P_e[\![e]\!]\ \phi\ \rho$ (Figura 7.3) y $P_v[\![e]\!]\ \phi\ \rho\ l'$ (Figura 7.4) son mutuamente recursivas.

$$
\begin{array}{lcl}
P_e[\![\lambda^l x.e]\!]\ \phi\ \rho & = & \{l\} \\
P_e[\![x^l]\!]\ \phi\ \rho & = & \rho\ l \\
P_e[\![(e_1\ e_2)]\!]\ \phi\ \rho & = & \bigcup \{\phi\ l \mid l \in P_e[\![e_1]\!]\ \phi\ \rho\} \\
P_e[\![(\mathbf{A}\ e_1\ e_2 \ldots e_n)]\!]\ \phi\ \rho & = & \{\} \\
P_e[\![\mathbf{if}\ e_1\ e_2\ e_3]\!]\ \phi\ \rho & = & P_e[\![e_2]\!]\ \phi\ \rho\ \cup\ P_e[\![e_3]\!]\ \phi\ \rho \\
\multicolumn{3}{l}{P_e[\![\mathbf{letrec}\ f_1 =^{l_1} e_1 \ldots f_m =^{l_m} e_m\ in\ e_0]\!]\ \phi\ \rho\ =\ P_e[\![e_0]\!]\ \phi\ \rho}
\end{array}
$$

Figura 7.3: Función de análisis de clausuras

$$
\begin{array}{lcl}
P_v[\![\lambda^l x.e]\!]\ \phi\ \rho\ l' & = & P_v[\![e]\!]\ \phi\ \rho\ l' \\
P_v[\![x^l]\!]\ \phi\ \rho\ l' & = & \{\} \\
P_v[\![(e_1\ e_2)]\!]\ \phi\ \rho\ l' & = & P_v[\![e_1]\!]\ \phi\ \rho\ l' \cup P_v[\![e_2]\!]\ \phi\ \rho\ l' \cup \\
 & & \qquad \bigcup \{P_e[\![e_2]\!]\ \phi\ \rho \mid l' \in P_e[\![e_1]\!]\ \phi\ \rho\} \\
P_v[\![(\mathbf{A}\ e_1\ e_2 \ldots e_n)]\!]\ \phi\ \rho\ l' & = & \bigcup_{j=1}^{n}\ P_v[\![e_j]\!]\ \phi\ \rho\ l' \\
P_v[\![\mathbf{if}\ e_1\ e_2\ e_3]\!]\ \phi\ \rho\ l' & = & \bigcup_{j=1}^{3}\ P_v[\![e_j]\!]\ \phi\ \rho\ l' \\
\multicolumn{3}{l}{P_v[\![\mathbf{letrec}\ f_1 =^{l_1} e_1 \ldots f_m =^{l_m} e_m\ in\ e_0]\!]\ \phi\ \rho\ l' = \bigcup_{i=0}^{m}\ P_v[\![e_i]\!]\ \phi\ \rho\ l'}
\end{array}
$$

Figura 7.4: Función de propagación de clausuras

Se establece una serie de ecuaciones mutuamente recursivas para cada programa e_p, cuyo mínimo punto fijo (ϕ, ρ) hay que calcular:

$$
\begin{array}{lcl}
\phi\ l & = & P_e[\![e_1]\!]\ \phi\ \rho \text{ para cada } \lambda^l x.e_1 \text{ en } e_p \\
\rho\ l & = & P_v[\![e_p]\!]\ \phi\ \rho\ l \\
\phi\ l_i\ =\ \rho\ l_i & = & P_e[\![e_i]\!]\ \phi\ \rho\ \text{ para cada }\ letrec\ f_1 =^{l_1} e_1 \ldots f_m =^{l_m} e_m\ in\ e_0
\end{array}
$$

El punto fijo que queremos calcular pertenece al retículo completo $CDescription^2$ de los pares de funciones de etiquetas en conjuntos de etiquetas ($CDescription = Label \to P(Label)$). La existencia de punto fijo (ϕ, ρ) está asegurada por ser las funciones P_e y P_v monótonas (Sestoft no lo demuestra formalmente) y el dominio al que pertenece el punto fijo un retículo completo.

Análisis de la frecuencia de uso

Este análisis encuentra una aproximación del número de veces que se usa una variable en una evaluación de una expresión, así como del número de veces que se usa una λ–abstracción en un programa.

El análisis calcula una cota superior y otra inferior del número de veces que se evalúa una subexpresión cuando la expresión que la contiene se evalúa una vez. Incluye al análisis de estrictez y al de compartición. Los resultados de este análisis pretenden la optimización de suspensiones:

- La **cota inferior** proporciona información de estrictez. Si la cota inferior es mayor o igual que uno, se lleva a cabo la transformación de paso por necesidad al paso por valor del parámetro de la función. Si sabemos que se va a evaluar un parámetro al menos una vez, entonces podemos evaluar el argumento antes de la aplicación de la función, de forma que no es necesaria una actualización pues ya estará en forma normal débil de cabeza. (en la máquina presentada por Sestoft no se aprovecha bien el análisis de estrictez ya que no evita la creación de una clausura, lo que sí se hace en otras máquinas)

- La **cota superior** contiene información de compartición. Si la cota superior es menor o igual a uno, se lleva a cabo la transformación del paso por necesidad al paso por nombre del parámetro de la función. Cuando sabemos que un parámetro va a ser usado a lo sumo una vez, nos ahorramos la actualización llevada a cabo tras la obtención de su forma normal débil de cabeza (y en general todo el tratamiento que supone la compartición de los lenguajes perezosos).

El **contador de uso** es un "número" dentro del conjunto $U = \{Zero,\ One,\ Many\}$, donde $Zero \leq One \leq Many$. Se pueden definir operaciones de suma y multiplicación entre contadores de uso, así como máximo y mínimo: $One + One = Many$, $\forall u \in U.Many + u = Many$, $\forall u \in U.Zero \times u = Zero$ etc.

Los **intervalos de uso** son conjuntos de contadores de uso de la forma $[u_{min}, u_{max}] = \{u \in U \mid u_{min} \leq u \leq u_{max}\}$. Llamemos UI al conjunto de los intervalos de uso. El dominio abstracto UI es un retículo completo formado por algunos subconjuntos de $\{Zero, One, Many\}$ ordenados por inclusión. El ínfimo es el conjunto vacío; el supremo es el conjunto $\{Zero, One, Many\}$, y la operación de supremo $\sqcup$ de define como el menor intervalo que los contiene a ambos, es decir, en términos de conjuntos se trata de la unión. También se extienden las operaciones de $+$ y $\times$:

$$\begin{array}{rcl}[u_{11}, u_{12}] + [u_{21}, u_{22}] & = & [u_{11} + u_{21}, u_{12} + u_{22}] \\ [u_{11}, u_{12}] \times [u_{21}, u_{22}] & = & [u_{11} \times u_{21}, u_{12} \times u_{22}]\end{array}$$

Se definen las siguientes funciones:

- $\psi\ l$, llamada **función de descripción de uso**, calcula el intervalo de uso de la variable x^l.

- $\chi\ l$, llamada **función de descripción de la compartición**, calcula el intervalo de uso de $\lambda^l x.e$

- $U_e[\![e]\!]\ \psi\ \chi\ l'$, llamada **función de análisis del uso**, calcula el intervalo de uso de la variable $x^{l'}$ en la expresión e cuando ésta se evalúa una vez.
- $U_v[\![e]\!]\ \psi\ l'$, llamada **función de propagación del uso**, calcula el intervalo de uso de cada $\lambda^{l'}x.e_1$ en la expresión e.

$U_e[\![e]\!]\ \psi\ \chi\ l'$ (Figura 7.5) y $U_v[\![e]\!]\ \psi\ l'$ (Figura 7.6) son mutuamente recursivas.

$$
\begin{array}{lcl}
U_e[\![\lambda^l x.e]\!]\ \psi\ \chi\ l' & = & \left\{\begin{array}{ll} [Zero, Zero] & \text{si } l = l' \\ \chi\ l\ \times\ U_e[\![e]\!]\ \psi\ \chi\ l' & \text{e.o.c.} \end{array}\right. \\
U_e[\![x^l]\!]\ \psi\ \chi\ l' & = & \left\{\begin{array}{ll} [One, One] & \text{si } l = l' \\ [Zero, Zero] & \text{e.o.c.} \end{array}\right. \\
U_e[\![(e_1\ e_2)]\!]\ \psi\ \chi\ l' & = & U_e[\![e_1]\!]\ \psi\ \chi\ l'\ + \\
 & & \quad U_e[\![e_2]\!]\ \psi\ \chi\ l'\ \times\ \bigsqcup\{\chi\ l \mid l \in P_e[\![e_1]\!]\ \phi\ \rho\} \\
U_e[\![(\mathbf{A}\ e_1\ e_2 \ldots e_n)]\!]\ \psi\ \chi\ l' & = & \sum_{j=1}^{n} U_e[\![e_j]\!] \psi\ \chi\ l' \\
U_e[\![\mathbf{if}\ e_1\ e_2\ e_3]\!]\ \psi\ \chi\ l' & = & U_e[\![e_1]\!]\ \psi\ \chi\ l'\ + \\
 & & \quad U_e[\![e_2]\!]\ \psi\ \chi\ l'\ \sqcup\ U_e[\![e_3]\!]\ \psi\ \chi\ l' \\
U_e[\![\mathbf{letrec}\ f_1 =^{l_1} e_1 \ldots f_m =^{l_m} e_m\ in\ e_0]\!]\ \psi\ \chi\ l' = & & \left\{\begin{array}{l} [Zero, Zero]\ \text{si}\ \ l' \in \{l_1, \ldots, l_m\} \\ U_e[\![e_0]\!]\ \psi\ \chi\ l'\ + \\ \quad \sum_{j=1}^{m} \chi\ l_j \times U_e[\![e_j]\!]\ \psi\ \chi\ l' \quad \text{e.o.c.} \end{array}\right.
\end{array}
$$

Figura 7.5: Función de análisis de uso

Se establecen por tanto una serie de ecuaciones recursivas para el programa e_p, cuyo mínimo punto fijo (ψ, χ) hay que calcular:

$$
\begin{array}{lcl}
\psi\ l & = & U_e[\![e_1]\!]\ \psi\ \chi\ l \quad \text{para cada } \lambda^l x.e_1 \text{ en } e_p \\
\chi\ l & = & U_v[\![e_p]\!]\ \psi\ l \\
\psi\ l_i & = & U_e[\![e_0]\!]\ \psi\ \chi\ l_i\ +\ \sum_{j=1}^{m} \psi\ l_j\ \times\ U_e[\![e_j]\!]\ \psi\ \chi\ l_i \\
 & & \text{para cada } \mathbf{letrec}\ f_1 =^{l_1} e_1 \ldots f_m =^{l_m} e_m\ in\ e_0
\end{array}
$$

El punto fijo que queremos calcular pertenece al retículo completo $UDescription^2$ de los pares de funciones de etiquetas en intervalos de uso ($UDescription = Label \rightarrow UI$) .

La existencia de punto fijo (ψ, χ) está asegurada por ser las funciones U_e y U_v monótonas (Sestoft no lo demuestra formalmente) y el dominio al que pertenece el punto fijo un retículo completo.

Veamos un par de ejemplos.

$$
\begin{array}{lcl}
U_v[\![\lambda^l x.e]\!]\ \psi\ l' & = & \bigsqcup\{\psi\ l_1 \mid l'\ \in\ \rho\ l_1\}\ \sqcup\ U_v[\![e]\!]\ \psi\ l' \\
U_v[\![x^l]\!]\ \psi\ l' & = & [Many, Zero] \\
U_v[\![(e_1\ e_2)]\!]\ \psi\ l' & = & U_v[\![e_1]\!]\ \psi\ l' \sqcup U_v[\![e_2]\!]\ \psi\ l' \sqcup \\
 & & \quad \bigsqcup\ \{[One, One] \mid l' \in P_e[\![e_1]\!]\ \phi\ \rho\} \\
U_v[\![(\mathbf{A}\ e_1\ e_2 \ldots e_n)]\!]\ \psi\ l' & = & \bigsqcup_{j=1}^{n}\ U_v[\![e_j]\!]\ \psi\ l' \\
U_v[\![\mathbf{if}\ e_1\ e_2\ e_3]\!]\ \psi\ l' & = & \bigsqcup_{j=1}^{3}\ U_v[\![e_j]\!]\ \psi\ l' \\
\end{array}
$$

$$
\begin{array}{lcl}
U_v[\![\mathbf{letrec}\ f_1 =^{l_1} e_1 \ldots f_m =^{l_m} e_m\ in\ e_0]\!]\ \psi\ l' & = & \bigsqcup_{i=0}^{m}\ P_v[\![e_i]\!]\ \psi\ l' \sqcup \\
 & & \bigsqcup_{j=1}^{m}(\bigsqcup\{\psi\ l_j \mid l'\ \in\ \rho\ l_j\}
\end{array}
$$

Figura 7.6: Función de propagación de uso

Ejemplo 72 Sea el programa:

$$
\begin{array}{lll}
\mathbf{letrec} & & \\
 & d\ w = & w + w \\
 & g\ x\ y\ z\ v = & \mathbf{if}\ (z = 0) \\
 & & \quad ((d\ x) + y) \\
 & & \quad (y + (z + v)) \\
in & g\ e_1\ e_2\ e_3\ e_4 &
\end{array}
$$

Las variables tienen los siguientes intervalos de uso: w tiene como intervalo de uso $[Many, Many]$, x tiene $[Zero, Many]$, y tiene $[One, One]$, z tiene $[One, Many]$ y v tiene $[Zero, One]$.

□

En el ejemplo motivador presentado en la Sección 1 se ve la importancia de usar el análisis de clausuras para poder aplicar el análisis a funciones de orden superior:

Ejemplo 73 En la expresión:

$$
\begin{array}{ll}
\mathbf{letrec} & \\
 & f\ x\ y = (x + y) \\
 & twice\ g\ z = g\ (g\ z) \\
in & \\
 & twice\ (f\ 7)\ 9
\end{array}
$$

se obtiene el intervalo de uso $[One, One]$ para la variable y. El análisis de clausuras nos dice que g se puede ligar a f aplicada a un argumento v, así que $g\ (g\ z)$ es en realidad $f\ v\ (f\ v\ z)$ para algún valor v. Por tanto, como sabemos el intervalo de uso para y, obtenemos que el intervalo de uso para z es $[One, One]$, el de g es $[Many, Many]$ y el de x es $[One, Many]$.

□

Algunas medidas comparativas de tiempo de cpu (excluyendo recolección de basura), número de comprobaciones de marcas de actualización, y número de marcas de actualización encontradas en tales comprobaciones son:

- Con la optimización de paso por necesidad a paso por nombre se reduce el tiempo de ejecución de la mayoría de los programas ejemplo de un 3 a un 10%. Además produce una reducción considerable en comprobaciones y actualizaciones, excepto para casos de prueba que trabajan exclusivamente con valores básicos, como la función que calcula el factorial.
- Con ambas optimizaciones (paso por necesidad a paso por nombre y paso por necesidad a paso por valor) las ventajas obtenidas en tiempo de ejecución por la optimización anterior aislada se pierden, debido al poco partido que se saca del análisis de estrictez en la máquina presentada por Sestoft. Además se incrementa el número de comprobaciones a realizar, aunque el número de actualizaciones se reduce, siendo nulo en los casos de valores básicos.

Análisis de globalización

Recordemos que este análisis pretende detectar cuándo un parámetro de una función se puede sustituir por una variable global y encontrar grupos de variables que se pueden asociar con la misma variable global en lenguajes estrictos. En ellos el análisis de clausuras es esencialmente el mismo que el descrito anteriormente para un lenguaje perezoso. Supongamos que disponemos de un análisis de clausuras cuyos resultados vienen dados de la forma $(f^i, m) \in AClo$ (donde $AClo$ es el conjunto de clausuras), el cual representa una clausura de la forma $f^i[v_1, \ldots, v_m]$, donde los v_i son valores. Sea $ca[\![]\!] : Expr \to P(AClo)$ la función que dada la expresión e devuelve un superconjunto del conjunto de clausuras a las que se puede evaluar dicha expresión.

Para llevar a cabo este análisis se necesita el concepto de viveza. Existen distintos tipos de viveza:

- **Viveza local:** x está localmente viva en un determinado punto del programa si hay un posible uso, temporalmente posterior de la misma en el ámbito actual. Por ejemplo, en el cuerpo de $f\ x\ y\ z = x + y + z$: x, y y z están están localmente vivas en el punto anterior a la evaluación de x; las variables y y z están localmente vivas tras la evaluación de x pero antes de la de y; z está localmente viva despues de la evaluación de x e y y antes de la evaluación de z; y ninguna lo está tras la evaluación de z. Calcularemos un conjunto aproximado, puesto que si tenemos la expresión **if** e *then* x *else* 7, diremos que la variable x está localmente viva tras la evaluación de e porque hay una posibilidad de que x se use posteriormente, aunque no sea seguro que esto vaya a ocurrir.
- **Viveza indirecta (no local):** x está indirectamente viva si puede ser capturada en el valor de alguna variable localmente viva. Por ello es necesario un previo análisis de clausuras.

En el ejemplo 74 más adelante, el parámetro z de h está indirectamente vivo. En dicho ejemplo, se crea una clausura h 2 y se guarda en la variable localmente viva t_1. Luego, durante la evaluación de t_4, aunque el primer parámetro z de h no aparece en el cuerpo de k, y por tanto no está localmente viva, existe un ejemplar de z en una clausura , y por tanto está viva de forma indirecta.

- **Viveza global:** incluye a las anteriores. x está globalmente viva si puede usarse posteriormente en el cómputo dinámico circundante, no necesariamente en el entorno léxico. Así, una variable puede estar viva en una función distinta de la que la tiene como parámetro. Por ejemplo, en:

$$\begin{array}{l} f1\ z = (f2\ 3) + z \\ f2\ x = ... \end{array}$$

  z está localmente viva en el cuerpo de $f1$ tras la llamada a $f2$. Por tanto, cuando se llama a $f2$, hay un uso posterior de la variable no local z. Luego z está globalmente viva en $f2$.

La globalización requiere un análisis previo de interferencia de variables. Una variable x interfiere con otra z, si se crea un nuevo ejemplar de x cuando z esta globalmente viva.

El análisis busca los pares de interferencia. A partir de ellos, un algoritmo calcula un conjunto de agrupaciones de variables que no interfieren mutuamente entre sí (no es único). Las variables que pertenecen al mismo grupo podrán asociarse con la misma variable global.

El dominio abstracto en este caso son conjuntos de pares de variables, llamados **pares de interferencia**, ordenados por inclusión, con ínfimo el conjunto vacío, supremo el conjunto de todos los pares posibles de variables y los operadores de supremo e ínfimo la unión e intersección de conjuntos.

Para llevar a cabo este análisis primero se secuencializan las expresiones (se introducen variables temporales, a las cuales se ligan las subexpresiones de la expresión dentro de un let).

Ejemplo 74

$$\begin{array}{l} k\ y = (cons\ (h\ 2)\ (g\ 3)) \\ h\ z\ w = \ldots \\ g\ x = \ldots \end{array}$$

se secuencializa como:

$$\begin{array}{lcllcl}
k\ y & = & \mathbf{let}\ t1 & & = & \mathbf{let}\ t2 = h \\
 & & & & & in \\
 & & & & & \quad \mathbf{let}\ t3 = 2 \\
 & & & & & \quad \mathbf{in}\ (t2\ t3) \\
 & & in & & & \\
 & & \quad \mathbf{let}\ t4 & & = & \mathbf{let}\ t5 = g \\
 & & & & & in \\
 & & & & & \quad \mathbf{let}\ t6 = 3 \\
 & & & & & \quad \mathbf{in}\ (t5\ t6) \\
 & & \quad in & & & \\
 & & & & & (cons\ t1\ t4)
\end{array}$$

□

Esto nos evita tener que hablar de la viveza de resultados temporales y pasar a hablar de viveza de variables (temporales), lo cual hace el análisis más sencillo de expresar.

Además esto significa que el lenguaje sobre el que se realiza el análisis se modifica ligeramente: convertimos el **letrec** recursivo en varios *let* en secuencia.

Se definen las siguientes funciones:

- Una función $acap : P(AClo) \rightarrow P(Param)$ que calcula una aproximación al conjunto de parámetros capturados en un conjunto de clausuras.Una clausura $f^i[v_1, \ldots, v_m]$ captura los primeros m parámetros $x_1^i, \ldots, x_m^i$ de f^i y aquellos parámetros capturados en los valores $v_1, \ldots, v_m$.

$$
\begin{array}{lcl}
acap(\{\}) & = & \{\} \\
acap(\{(f^i, m)\}) & = & \{x_1^i, \ldots, x_m^i\} \cup acap(ca[\![x_1^i]\!] \cup \ldots \cup ca[\![x_m^i 1]\!]) \\
acap(V_1 \cup V_2) & = & acap(V_1) \cup acap(V_2)
\end{array}
$$

- Una función $ccl : P(Var) \rightarrow P(Param)$ que calcula el conjunto de variables que son indirectamente vivas debido a una variable localmente viva. Una variable parámetro no temporal se captura a sí misma y a las variables capturadas en el conjunto de valores que puede tomar dicha variable. Sin embargo, si se trata de una variable temporal, no la incluimos porque no nos interesa conocer su viveza.

$$
\begin{array}{lcl}
ccl(\{x\}) & = & \{x\} \cup acap(ca[\![x]\!]) \\
ccl(\{t\}) & = & acap(ca[\![t]\!]) \\
ccl(V_1 \cup V_2) & = & ccl(V_1) \cup ccl(V_2)
\end{array}
$$

- Una función $L : Expr \rightarrow Varset \rightarrow (Varset \times GLEnv)$ (donde $GLEnv = Fun \rightarrow P(Param)$), que, conociendo el conjunto de variables vivas δ tras la evaluación de una expresión e, calcula un par cuya primera componente δ' es el conjunto de variables vivas antes de la evaluación de la expresión, y la segunda θ' representa la contribución de la expresión a los conjuntos de variables vivas de las funciones del programa (a cada función del programa f^i le hace corresponder el conjunto de variables que pueden estar vivas mientras el cuerpo e_i de la función se está evaluando), ver Figura 7.7.

  El resultado deseado es la menor solución $\theta \in GLEnv$ a la ecuación:

$$\theta = \bigsqcup snd(L[\![e^i]\!](\theta f^i)) \mid f^i \text{es una función}$$

- Una función $I : Expr \rightarrow Varset \rightarrow P(Inter)$ (donde $Inter$ son pares de parámetros), que, dado el conjunto de variables vivas, calcula el conjunto de pares de interferencia en un programa dado (ver Figura 7.8).

La regla más interesante es la de la aplicación. Se produce un par de interferencia (z, x) por cada variable x de la que se puede crear una nueva instancia en la aplicación y para cada variable z viva cuando se crea el nuevo ejemplar. El resultado final del análisis viene dado por:

$$\xi = \bigcup \{I[\![e^i]\!](\theta f^i) \mid i = 1, \ldots, n\}$$

$$
\begin{array}{lcl}
L[\![x_j^i]\!]\ \delta & = & (\{x_j^i\} \cup \delta, \bot_{GLEnv}) \text{ donde } \bot_{GLEnv} = [f^i \longmapsto \{\}] \\
L[\![f^i]\!]\ \delta & = & (\delta, \bot_{GLEnv}) \\
L[\![\mathbf{if}\ t\ e_1\ e_2]\!]\ \delta & = & (\{t\} \cup \delta_1 \cup\ \delta_2, \theta_1 \sqcup \theta_2) \\
 & & \qquad \text{donde } (\delta_1, \theta_1) = L[\![e_1]\!]\ \delta \\
 & & \qquad \qquad \quad (\delta_2, \theta_2) = L[\![e_2]\!]\ \delta \\
L[\![\mathbf{A}(t_1, \ldots, t_n)]\!]\ \delta & = & (\{t_1, \ldots, t_n\} \cup\ \delta, \bot_{GLEnv}) \\
L[\![\mathbf{let}\ t_1 = e_1\ \mathbf{in}\ e]\!]\ \delta & = & (\delta_1, \theta_0 \sqcup \theta_1) \\
 & & \qquad \text{donde } (\delta_0, \theta_0) = L[\![e]\!]\delta \\
 & & \qquad \qquad \quad (\delta_1, \theta_1) = L[\![e_1]\!](\delta_0 \backslash \{t_1\}) \\
L[\![t_1\ t_2]\!]\ \delta & = & (\delta \cup \{t_1, t_2\}, \bigsqcup\{[f^i \longmapsto ccl(\delta)] \mid (f^i, arity_i - 1)\ \in\ ca[\![t_1]\!]\})
\end{array}
$$

Figura 7.7: Función de análisis de la viveza

$$
\begin{array}{lcl}
I[\![x_j^i]\!]\ \delta & = & \{\} \\
I[\![f^i]\!]\ \delta & = & \{\} \\
I[\![\mathbf{if}\ t\ e_1\ e_2]\!]\ \delta & = & I[\![e_1]\!]\delta \cup I[\![e_2]\!]\delta \\
I[\![\mathbf{A}(t_1, \ldots, t_n)]\!]\ \delta & = & \{\} \\
I[\![\mathbf{let}\ t_1 = e_1\ \mathbf{in}\ e]\!]\ \delta & = & I[\![e_1]\!](\delta_1 \backslash \{t_1\}) \cup I[\![e]\!]\ \delta \\
 & & \qquad \text{donde } (\delta_1, \theta_1) = L[\![e]\!]\delta \\
I[\![t_1\ t_2]\!]\ \delta & = & \{(z, x) \mid z \in vivas, x \in upds\} \\
 & & \qquad \text{donde } upds = \{x_{j+1}^i \mid (f^i, j) \in ca[\![t_1]\!]\} \\
 & & \qquad \qquad \quad vivas = ccl(\delta \cup \{t_1, t_2\})
\end{array}
$$

Figura 7.8: Función de análisis de interferencias

7.2.3 La complejidad

Estudiemos la complejidad de estos análisis. Ésta se calcula en función del número de pasadas que deben darse para calcular el punto fijo, el número de operaciones elementales realizadas en cada pasada y el tiempo que tarda cada una de ellas. El número de pasadas se calcula en función de la profundidad del dominio al que pertenece el punto fijo que estamos calculando.

Por ejemplo, en el análisis de clausuras, pretendemos calcular un punto fijo (ϕ, ρ), donde cada una de las funciones es de tipo: $CDescription = Label \rightarrow P(Label)$. Por tanto el dominio al que pertenece el punto fijo es $CDescription^2$. La profundidad de este retículo, y por tanto el número de pasadas para calcular el punto fijo, es $2n^2$. El número de etiquetas de un programa n se estima proporcional al tamaño del programa p. Por tanto el coste del análisis de clausuras es del orden de $0(p^4)$ y el de frecuencia de uso es del orden de $0(p^3)$.

Capítulo 8

Análisis basados en tipos

8.1 Introducción

El primer uso explícito de los tipos en el análisis de programas lo llevaron a cabo Kuo y Mishra [KM89]. Presentaron un sistema de tipos para inferir la estrictez en programas funcionales. La mayor atracción de su enfoque, en comparación con la interpretación abstracta era la posibilidad de diseñar algoritmos eficientes para llevar a cabo el análisis, puesto que aunque el algoritmo de inferencia en el sistema de tipos de Hindley–Milner es exponencial en el caso peor, en la mayoría de los casos reales que se presentan tiene un comportamiento polinómico. Su desventaja era que el análisis era más débil, es decir, proporcionaba información menos útil que la interpretación abstracta. El sistema de tipos que usaron para expresar las propiedades de estrictez tenía dos constantes (o tipos básicos), ϕ y $\Box$:

$$\alpha ::= \phi \mid \Box \mid \alpha \to \alpha$$

Semánticamente ϕ representa el conjunto de todos los términos divergentes, y $\Box$ el conjunto de todos los términos. El tipo $\alpha \to \beta$ representa los términos F tal que $F\ E$ pertenece al conjunto representado por β cuando E está en el conjunto reperesentado por α. Por ejemplo, las constantes enteras tienen el tipo $\Box$, y $\phi \to \phi$ es el tipo de los términos estrictos. Los operadores primitivos, por ejemplo $+$, pueden tener más de un tipo. Por ejemplo $\phi \to \Box \to \phi$ y $\Box \to \phi \to \phi$ son tipos de $+$. Hay incluso términos con infinitos tipos. Por ejemplo, $\lambda x.x$ tiene tipos $\alpha \to \alpha$ para cada tipo α. Por ello surge la necesidad de incorporar conjunciones de tipos, de forma que el tipo de $+$ sería $\phi \to \Box \to \phi \wedge \Box \to \phi \to \phi$. Esta falta de tipos conjuntivos en el análisis de Kuo y Mishra es la causa de que sea más débil que la interpretación abstracta. Posteriormente Jensen [Jen91] [Jen92] incorporó la conjunción de tipos y demostró la equivalencia entre los análisis basados de tipos y la interpretación abstracta con respecto al análisis de estrictez. También Benton [Ben92] [Ben93] trabajó en la recuperación de la potencia de la interpretación abstracta. Ambos se centraron en el análisis de estrictez. En este proceso olvidaron la conexión con el aspecto algorítmico y la eficiencia. Burn [Bur92] generalizó su enfoque, pero tampoco se fijó en cuestiones algorítmicas. Más recientemente Hankin y Le Métayer [HL94b] [HL94a] [HM94] han introducido una nueva idea, los tipos perezosos, que permite mantener la potencia de la interpretación abstracta y a la vez desarrollar algoritmos eficientes. En este capítulo presentamos el marco general de análisis de programas basados en tipos presentado por ellos que reune todas las ideas de los autores previos. Dicho marco estará parametrizado por el conjunto de tipos básicos y las reglas de asignación de tipos para las constantes. Para construir un análisis concreto, se debe proporcionar lo siguiente:

- La lista de constantes del lenguaje.
- La lista de constantes de tipo.
- Una estructura de tipos que incluye una asociación de los tipos a conjuntos de valores en el dominio estándar (propiedades que representan los tipos).
- Las reglas de asignación de tipo a las constantes del lenguaje y la verificación de las condiciones de corrección asociadas.
- Reglas que definen el tratamiento de las constantes por parte de la máquina abstracta que implementa el algoritmo de deducción de tipos y la corrección de dichas reglas. En esta sección nos vamos a centrar en los puntos anteriores y no entraremos en detalles respecto a la máquina abstracta que implementa el algoritmo.

El lenguaje básico con el que trabajaremos será el λ–cálculo simple con tipos visto ya en secciones anteriores.

8.2 Tipos y su interpretación

Suponemos que hay ciertos conjuntos de constantes de tipo con un cierto orden $\leq$. Ejemplos de conjuntos de constantes podrían ser:

$$\begin{array}{c} B_{ss} = \{\mathbf{f}, \mathbf{t}\} \text{ donde } \mathbf{f} \leq \mathbf{t} \\ B_{sl} = \{\mathbf{f}, \infty, \mathbf{f}_\epsilon, \mathbf{t}\} \text{ donde } \mathbf{f} \leq \infty \leq \mathbf{f}_\epsilon \leq \mathbf{t} \\ B_{bt} = \{\mathit{static}, \mathit{dynamic}\} \text{ donde } \mathit{static} \leq \mathit{dynamic} \end{array}$$

usadas respectivamente en un análisis simple de estrictez, en uno de estrictez de listas y en un análisis del momento de vinculación. Se pueden construir tipos más complejos usando $\wedge$ (conjunción) y $\rightarrow$ (funciones):

$$\alpha ::= b \mid \alpha \wedge \alpha \mid \alpha \rightarrow \alpha$$

donde b es un tipo base. El orden entre los tipos se puede extender a partir del básico como se muestra en la Figura 8.1. Obsérvese la contravarianza de la regla correspondinte al tipo funcional. Se define además $=$ como la equivalencia inducida por el orden entre tipos:

$$\sigma = \tau \Leftrightarrow \sigma \leq \tau \wedge \tau \leq \sigma$$

Las fórmulas de tipo representan propiedades de la misma forma que los puntos de un dominio abstracto representan propiedades o conjuntos de valores en la interpretación abstracta.

Para dar una interpretación de los tipos, se proporciona una estructura de tipo.

Definición 75 *Una* ***estructura de tipo*** $\mathcal{M}$ *es una tupla* $(X, \sqsubseteq, \sqcap, \Rightarrow, norm)$ *donde:*

$$\phi \leq \phi$$

$$\frac{\phi \leq \psi, \psi \leq \chi}{\phi \leq \chi} \qquad \frac{\phi \leq \psi_1,\, \psi \leq \psi_2}{\phi \leq \psi_1 \wedge \psi_2} \qquad \phi \wedge \psi \leq \phi \qquad \phi \wedge \psi \leq \psi$$

$$\phi \to \psi_1 \wedge \phi \to \psi_2 \leq \phi \to (\psi_1 \wedge \psi_2) \qquad \frac{\phi' \leq \phi, \psi \leq \psi'}{\phi \to \psi \leq \phi' \to \psi'}$$

Figura 8.1: Orden entre los tipos

- $(X, \sqsubseteq)$ *es un dominio de propiedades que incluye la interpretación de las constantes.*
- $\sqcap : X \times X \to X$ *interpreta a* $\wedge$.
- $\Rightarrow : X \times X \to X$ *interpreta a* $\to$.
- $norm : X \to P(D)$ *hace corresponder a cada propiedad su correspondiente conjunto de elementos del dominio.*

Además se satisfacen las siguientes propiedades:

$$f \in norm(x \Rightarrow y) \textit{ si y solo si } \forall a.a \in norm(x) \Rightarrow (f\ \ a) \in norm(y)$$

$$x \sqsubseteq y \Rightarrow norm(x) \subseteq norm(y)$$

Dada una estructura particular $\mathcal{M}$ y una interpretación de las constantes de tipo $I : B \to X$ denotamos la interpretación de σ por $[\![\sigma]\!]^{\mathcal{M},I}$ o simplemente $[\![\sigma]\!]$

Definición 76 *Decimos que la estructura es un* ***modelo*** *si para toda* ϕ *y* ψ*:*

$$\phi \leq \psi \Rightarrow [\![\phi]\!] \sqsubseteq [\![\psi]\!]$$

Veamos un par de ejemplos.

Ejemplo 77 Sea $CPER(D)$ el conjunto de todas las relaciones de equivalencia parcial sobre un dominio D, ya visto en la Sección 7.1.2. Sea la siguiente estructura:

$$\mathcal{M}_{cper} = (CPER(D), \sqsubseteq_{cper}, \sqcap_{cper}, \Rightarrow_{cper}, norm_{cper})$$

donde:

- $\sqsubseteq_{cper} = \subseteq$ (inclusión de conjuntos)
- $\sqcap_{cper} = \cap$ (intersección de conjuntos)
- $f\ (Q \Rightarrow_{cper} R)\ g \equiv [\forall q, q'.q\ Q\ q' \Rightarrow (f\ \ q)\ R\ (g\ q')]$

- $norm_{cper}(P) = \{x \in D \mid x\ P\ x\}$

Se cumplen las propiedades exigidas. La estructura no está ligada a una interpretación particular de las constantes. Más adelante se usará esta estructura en un análisis del momento de vinculación (Ejemplo 94).

□

Ejemplo 78 Llamemos $SCS(D)$ al conjunto de todos los conjuntos Scott–cerrados sobre D. Sea la siguiente estructura:

$$\mathcal{M}_{scs} = (SCS(D), \sqsubseteq_{scs}, \sqcap_{scs}, \Rightarrow_{scs}, norm_{scs})$$

donde:

- $\sqsubseteq_{scs} = \subseteq$ (inclusión de conjuntos)
- $\sqcap_{scs} = \cap$ (intersección de conjuntos)
- $S_1 \Rightarrow_{scs} S_2 = \{f \mid \forall s_1 \in S_1. f\ s_1 \in S_2\}$
- $norm_{scs}(S) = S$

También se cumplen las propiedades exigidas. Más adelante se usará esta estructura en un análisis de estrictez.

□

8.3 El análisis basado en tipos

Una vez tenemos los tipos y el orden definido entre ellos, se define una lógica con reglas para la deducción de tipos sobre un lenguaje tipo λ–cálculo sin *let*. Estas reglas se muestran en la Figura 8.2, donde Γ, como es habitual, representa un contexto de asignaciones de tipos a variables y $\Gamma[x \mapsto \phi]$ representa el contexto Γ aumentado con la asignación del tipo ϕ a la variable x. Escribimos $\Gamma \leq \Delta$ si para toda variable x en el dominio de Γ se cumple que $\Gamma(x) \leq \Delta(x)$.

Además de estas reglas, para un análisis particular es necesario que se proporcionen las reglas para las constantes y estas reglas deben ser correctas en algún sentido. Se establece la noción de corrección respecto a la semántica estándar.Utilizaremos la siguiente notación:

Definición 79

- $d \models_{el} \phi \equiv d \in norm([\![\phi]\!])$
- $\rho \models_{el} \Gamma \equiv \forall x. \rho\ x \models_{el} \Gamma\ x$
- $\Gamma \models e : \phi \equiv \forall \rho \models_{el} \Gamma. sem[\![e]\!]\ \rho \models_{el} \phi$ *donde* $sem[\![e]\!]\ \rho$ *denota, como hasta ahora, la semántica estándar de* e *en el entorno* ρ.

Así, definimos la corrección de las constantes de la siguiente manera:

$$\textbf{Conj}\ \frac{\Gamma \vdash_T e : \psi_1 \qquad \Gamma \vdash e : \psi_2}{\Gamma \vdash_T e : \psi_1 \wedge \psi_2} \qquad \textbf{Weak}\ \frac{\Gamma \leq \Delta \qquad \Delta \vdash_T e : \phi \qquad \phi \leq \psi}{\Gamma \vdash_T e : \psi}$$

$$\textbf{Var}\ \Gamma[x \mapsto \phi] \vdash_T x : \phi \qquad \textbf{Abs}\ \frac{\Gamma[x \mapsto \phi] \vdash_T e : \psi}{\Gamma \vdash_T \lambda x.e : (\phi \to \psi)}$$

$$\textbf{App}\ \frac{\Gamma \vdash_T e_1 : (\phi \to \psi) \qquad \Gamma \vdash_T e_2 : \phi}{\Gamma \vdash_T e_1\ e_2 : \psi} \qquad \textbf{Fix}\ \frac{\Gamma \vdash_T (\lambda g.e) : \phi \to \phi}{\Gamma \vdash_T \textbf{fix}(\lambda g.e) : \phi}$$

Figura 8.2: Sistema de tipos

Definición 80 *Una regla para una constante n–aria:*

$$\textbf{Const}\ \frac{\Gamma \vdash_T e_1 : \phi_1 \ldots \Gamma \vdash_T e_n : \phi_n}{\Gamma \vdash_T c\ e_1 \ldots e_n : \phi}$$

es correcta si bajo la suposición de que:

$$\Gamma \vdash_T e_i : \phi_i \Rightarrow \Gamma \models e_i : \phi_i$$

entonces $\Gamma \models c\ e_1 \ldots e_n : \phi$.

El siguiente teorema nos dice que basta que las reglas para las constantes sean correctas para que todo el sistema sea correcto.

Teorema 81 *Supongamos que las reglas para cada constante n–aria son correctas. Sea* e *un término con tipo* σ *y sea* Γ *un entorno que asocia fórmulas a las variables libres. Entonces:*

$$\Gamma \vdash_T e : \phi \Rightarrow \Gamma \models e : \phi$$

donde ϕ *es una fórmula consistente con* σ.

El teorema anterior se demuestra por inducción sobre la forma del árbol de derivación [HL94b].

Ejemplo 82 (Análisis de estrictez)
Veamos un análisis de estrictez propuesto por Jensen [Jen91], donde la estructura de tipo es la presentada en el Ejemplo 78.

Se dispone de dos constantes de tipo básicas $\mathbf{f}$ y $\mathbf{t}$, tales que $\mathbf{f} \leq \mathbf{t}$. El tipo base $\mathbf{f}$ representa la propiedad de ser estricto y $\mathbf{t}$ representa a todos los valores (semejantes a los valores abstractos 0 y 1 de la interpretación abstracta). A partir de ellas se definen los tipos compuestos, usando $\wedge$ y $\to$. Para cada tipo real σ denotamos por $\mathbf{f}_\sigma$ al menor elemento del retículo de tipos de estrictez que tienen la misma estructura de flechas que σ, y por $\mathbf{t}_\sigma$ al mayor elemento del retículo de tipos de estrictez que tienen la misma estructura de flechas que σ. Por ejemplo, si $\sigma = b \to b$, donde b es un tipo básico, entonces $\mathbf{t}_\sigma$ denota $\mathbf{f} \to \mathbf{t}$, y $\mathbf{f}$ denota $\mathbf{t} \to \mathbf{f}$.

El orden entre los tipos viene dado en la Figura 8.3, donde ϕ es un tipo consistente (con la misma estructura de flechas) con σ. La desigualdad $\mathbf{t}_{\sigma\to\tau} \leq \mathbf{t}_\sigma \to \mathbf{t}_\tau$, establece una equivalencia entre fórmulas.

$$\boxed{\mathbf{f}_\sigma \leq \phi \qquad \phi \leq \mathbf{t}_\sigma \qquad \mathbf{t}_{\sigma\to\tau} \leq \mathbf{t}_\sigma \to \mathbf{t}_\tau}$$

Figura 8.3: Orden entre los tipos para el análisis de estrictez

Por ejemplo, en el caso de que $\sigma = b \to b$, esta desigualdad establece que $\mathbf{f} \to \mathbf{t} \leq \mathbf{t} \to \mathbf{t}$, y por tanto que son equivalentes, ya que $\mathbf{f} \to \mathbf{t}$ es el supremo del dominio. Luego ahora $\mathbf{t}_\sigma$ denota a la clase de equivalencia con elementos $\mathbf{f} \to \mathbf{t}$ y $\mathbf{t} \to \mathbf{t}$. En la Figura 8.4 se muestra el retículo de los tipos consistentes con $\sigma = b \to b$.

$$\begin{array}{c} \mathbf{t}_{b\to b} = \{\mathbf{t} \to \mathbf{t}, \mathbf{f} \to \mathbf{t}\} \\ | \\ \mathbf{f} \to \mathbf{f} \\ | \\ \mathbf{f}_{b\to b} = \{\mathbf{t} \to \mathbf{f}\} \end{array}$$

Figura 8.4: Retículo de tipos estrictos para funciones de un argumento

Esta equivalencia es necesaria porque tal como hemos definido la interpretación de $\to$, $\mathbf{f} \to \mathbf{t}$ y $\mathbf{t} \to \mathbf{t}$ denotan el mismo conjunto de funciones, que son todas las de un argumento.

En la Figura 8.5 se muestra el retículo de los tipos consistentes con $\sigma = b \to (b \to b)$. En este caso, se identifican $\mathbf{f} \to \mathbf{t}_{b\to b}$ y $\mathbf{t} \to \mathbf{t}_{b\to b}$ y por tanto $\mathbf{t}_\sigma$ denota a la clase formada por estos dos tipos.

La estructura de tipo es la presentada en el Ejemplo 78 como ya hemos mencionado anteriormente. En este caso concreto la interpretación de los tipos base es la siguiente:

$$\begin{array}{r} I_{scs}(\mathbf{f}_\sigma) = \{\perp_{D_\sigma}\} \\ I_{scs}(\mathbf{t}_\sigma) = D_\sigma \end{array}$$

Se cumple que $I_{scs}(\mathbf{f}_\sigma) \subseteq I_{scs}(\mathbf{t}_\sigma)$, por lo que $\phi \leq \psi \Rightarrow [\![\phi]\!] \sqsubseteq [\![\psi]\!]$, con lo que la estructura es un modelo.

Si estamos en el caso, por ejemplo, de una función con un argumento, el tipo $\mathbf{t} \to \mathbf{f}$ se interpreta como el conjunto que contiene como único elemento la función indefinida; el tipo

$$\begin{tikzcd}[column sep=small]
 & \mathbf{t}_{b\to(b\to b)} = \{\mathbf{t} \to \mathbf{t}_{b\to b}, \mathbf{f} \to \mathbf{t}_{b\to b}\} \arrow[d, dash] & \\
 & \mathbf{f} \to (\mathbf{f} \to \mathbf{f}) \arrow[dl, dash] \arrow[dr, dash] & \\
\mathbf{f} \to \mathbf{f}_{b\to b} \arrow[dr, dash] & & \mathbf{t} \to (\mathbf{f} \to \mathbf{f}) \arrow[dl, dash] \\
 & \mathbf{f} \to \mathbf{f}_{b\to b} \wedge \mathbf{t} \to (\mathbf{f} \to \mathbf{f}) \arrow[d, dash] & \\
 & \mathbf{f}_{b\to(b\to b)} = \{\mathbf{t} \to \mathbf{f}_{b\to b}\} &
\end{tikzcd}$$

Figura 8.5: Retículo de tipos estrictos para funciones de dos argumentos

$\mathbf{f} \to \mathbf{f}$ se interpreta como el conjunto de todas las funciones estrictas; y el tipo $\mathbf{t}_{b\to b}$, o cualquiera de sus representantes, se interpreta como el conjunto de todas las funciones de un argumento. De la misma forma, $\mathbf{f} \to \mathbf{f}_{b\to b}$ y $\mathbf{t} \to (\mathbf{f} \to \mathbf{f})$ se interpretan como el conjunto de funciones estrictas en el primer y segundo argumentos respectivamente. El tipo $\mathbf{f} \to \mathbf{f}_{b\to b} \wedge \mathbf{t} \to (\mathbf{f} \to \mathbf{f})$ se interpreta como el conjunto de funciones estrictas en los dos argumentos. Para poder representar esta información es imprescindible la conjunción de tipos, como ya hemos dicho anteriormente.

Supongamos que tenemos en el lenguaje constantes c de tipo base y el condicional. Las reglas para estas constantes se muestran en la Figura 8.6. Las reglas son correctas. bsérvese que aunque aparentemente se podrían hacer dos inferencias distintas a partir de $e_1 : \mathbf{f}$, $e_2 : \phi$, $e_3 : \phi$, de hecho se deduce $\mathbf{f}$ para el condicional debido a las reglas de $\wedge$ y a que $\forall \phi . \mathbf{f} \leq \phi$.

$$\textbf{Taut}\ \Gamma \vdash_T c : \mathbf{t}$$

$$Cond_1\ \frac{\Gamma \vdash_T e_1 : \mathbf{f}}{\Gamma \vdash_T \textbf{if}\ e_1\ e_2\ e_3 : \mathbf{f}_\sigma} \qquad Cond_2\ \frac{\Gamma \vdash_T e_2 : \phi \qquad \Gamma \vdash_T e_3 : \phi}{\Gamma \vdash_T \textbf{if}\ e_1\ e_2\ e_3 : \phi}$$

Figura 8.6: Reglas para las constantes en un análisis de estrictez

Veamos un ejemplo. Sea

$$f\ x\ y\ z = \textbf{if}\ (z = 0)\ (x + y)\ (f\ y\ x\ (z - 1))$$

Queremos ver que esta función es estricta en sus dos primeros argumentos. Para ello tendremos que demostrar que tiene tipo:

$$\psi = \mathbf{f} \to \mathbf{t} \to \mathbf{t} \to \mathbf{f} \wedge \mathbf{t} \to \mathbf{f} \to \mathbf{t} \to \mathbf{f}$$

Llamemos ψ_1 al primer elemento de la conjunción anterior y ψ_2 al segundo. Y sea $\Gamma = [f : \psi, x : \mathbf{t}, y : \mathbf{f}, z : \mathbf{t}]$. Entonces el árbol de demostración es el siguiente:

$$
\dfrac{
  \dfrac{\vdots}{[f:\psi] \vdash E : \psi_1}
  \qquad
  \dfrac{
    \dfrac{
      (+)\;\dfrac{\Gamma \vdash x : \mathbf{t} \qquad \Gamma \vdash y : \mathbf{f}}{\Gamma \vdash x + y : \mathbf{f}}
      \qquad
      \dfrac{
        \dfrac{\Gamma \vdash f : \psi \quad \Gamma \vdash \psi \leq \psi_1}{\Gamma \vdash f : \psi_1}\;\mathbf{Weak}
        \quad \Gamma \vdash y : \mathbf{f} \quad \Gamma \vdash x : \mathbf{t} \quad \Gamma \vdash (z-1) : \mathbf{t}
      }{\Gamma \vdash f\ y\ x\ (z-1) : \mathbf{f}}\;\mathbf{App}
    }{\Gamma \vdash if\ (z=0)\ (f\ y\ x\ (z-1)) : \mathbf{f}}\;\mathbf{Cond_2}
  }{[f:\psi] \vdash E : \psi_2}\;\mathbf{Abs}
}{
  \dfrac{[f:\psi] \vdash \lambda x.\lambda y.\lambda z.if\ (z=0)\ (x+y)\ (f\ y\ x\ (z-1)) : \psi}{
    \dfrac{\vdash \lambda f.\lambda x.\lambda y.\lambda z.if\ (z=0)\ (x+y)\ (f\ y\ x\ (z-1)) : \psi \to \psi}{\vdash fix(\lambda f.\lambda x.\lambda y.\lambda z.if\ (z=0)\ (x+y)\ (f\ y\ x\ (z-1)) : \psi}\;\mathbf{Fix}
  }\;\mathbf{Abs}
}\;\mathbf{Conj}
$$

donde con (+) se pretenden resumir varios pasos, partiendo del hecho de que se cumple $\vdash + : \mathbf{f} \to \mathbf{t} \to \mathbf{f} \wedge \mathbf{t} \to \mathbf{f} \to \mathbf{f}$.

□

8.4 Tipo más general

Las reglas de la Figura 8.2 no sugieren de forma inmediata un algoritmo para llevar a cabo el análisis. El problema se encuentra en la regla de debilitamiento, la cual introduce no determinismo. Para llegar a un algoritmo se introduce primero una restricción sobre los tipos que se pueden construir: no se permiten conjunciones en el lado derecho de una flecha. No se pierde poder expresivo puesto que $a \to (b \wedge c) \equiv (a \to b) \wedge (a \to c)$. Llamemos a este nuevo lenguaje de tipos T_I. La formación de los tipos se muestra en la Figura 8.7, donde b representa un tipo básico.

$$
b \in T_S \qquad \dfrac{\sigma \in T_I \quad \psi \in T_S}{\sigma \to \psi \in T_S} \qquad \dfrac{\phi_1 \in T_S \ldots \phi_n \in T_S}{\phi_1 \wedge \ldots \wedge \phi_n \in T_I}
$$

Figura 8.7: El nuevo lenguaje de tipos T_I

A continuación se introduce la noción de **tipo completo**. La restricción a tipos completos nos permite evitar la regla de debilitamiento porque un tipo completo es la conjunción de todos los elementos mayores o iguales que él.

Definición 83

$$\begin{array}{rcl} CT(\tau) & = & \wedge\{ct(\sigma) \mid \sigma \in Sup(\tau)\} \\ Sup(\sigma) & = & \{\sigma' \in T_S \mid \sigma' \geq \sigma\} \\ \\ ct(b) & = & b \textit{ si } b \textit{ es un tipo base} \\ ct(\sigma \wedge \tau) & = & ct(\sigma) \wedge ct(\tau) \\ ct(\sigma \to \tau) & = & CT(\sigma) \to ct(\tau) \end{array}$$

Definición 84 *El* ***tipo más general*** *de una expresión e es la conjunción de todos los tipos de la expresión en el entorno dado:*

$$MGT(\Gamma, e) = CT(\wedge\{\sigma_i \in T_S \mid \Gamma \vdash_T e : \sigma_i\}$$

La lógica que calcula el tipo más general se muestra en la Figura 8.8.

$$\textbf{Var } \Gamma[x \mapsto \phi] \vdash_S x : \phi$$

$$\textbf{Abs } \frac{\Gamma[x \mapsto \phi_1] \vdash_S e : \psi_1^1 \wedge \ldots \wedge \psi_{n_1}^1 \ldots \Gamma[x \mapsto \phi_k] \vdash_S e : \psi_1^k \wedge \ldots \wedge \psi_{n_k}^k}{\Gamma \vdash_S \lambda x.e : ((\phi_1 \to \psi_1^1) \wedge \ldots \wedge (\phi_1 \to \psi_{n_1}^1) \wedge \ldots \wedge (\phi_k \to \psi_{n_k}^k))}$$

$$\textbf{App } \frac{\Gamma \vdash_S e_1 : ((\phi_1^1 \wedge \ldots \wedge \phi_{n_1}^1) \to \psi_1) \wedge \ldots \wedge ((\phi_1^m \wedge \ldots \wedge \phi_{n_m}^m) \to \psi_m) \quad \Gamma \vdash_S e_2 : \theta_1 \wedge \ldots \wedge \theta_k}{\Gamma \vdash_S e_1 \ e_2 : \wedge_i \psi_i \quad i \textbf{ tal que } \{\phi_1^i, \ldots \phi_{n_i}^i\} \equiv \{\theta_1, \ldots \theta_k\}}$$

$$\textbf{Fix } \frac{\Gamma \vdash_S (\lambda g.e) : \sigma}{\Gamma \vdash_S \textbf{fix}(\lambda g.e) : \wedge_{i=1}^n \theta_i \qquad \textbf{donde } \forall i.(\theta_1 \wedge \ldots \wedge \theta_n \to \theta_i) \in \sigma}$$

Figura 8.8: Sistema del tipo más general

Se cumple que el tipo derivado para una expresión e mediante $\vdash_S$ es equivalente a la conjunción de todos los tipos derivados mediante $\vdash_T$:

Teorema 85 *(Corrección)*

$$\Gamma \vdash_S e : \sigma \Rightarrow \Gamma \vdash_T e : \sigma$$

Teorema 86 *(Completitud)*

$$CT(\Gamma) \vdash_S e : MGT(\Gamma, e)$$

En [HM94] se demuestra que el tipo más general de una expresión es precisamente la información producida por un análisis basado en interpretación abstracta. Esto implica que los algoritmos derivados son de la misma complejidad que la interpretación abstracta. Para evitar este coste se introduce la noción de **tipo perezoso**: permitimos que los tipos contengan clausuras introducidas por la regla de aplicación, que se evaluan de forma perezosa, es decir, cuando se requieren.

Ejemplo 87 (Análisis de estrictez)
Continuando con el análisis de estrictez anterior las reglas para las constantes en el sistema de tipos más general se muestran en la Figura 8.9, donde en la regla $\mathbf{Cond_1}$, C_σ denota los tipos con la misma estructura que σ.

□

$$\mathbf{Taut}\ \Gamma \vdash_S c : \mathbf{t} \qquad\qquad \mathbf{Cond_1}\ \frac{\Gamma \vdash_S e_1 : \mathbf{f}}{\Gamma \vdash_S \mathbf{if}\ e_1\ e_2\ e_3 : \wedge CT(C_\sigma)}$$

$$\mathbf{Cond_2}\ \frac{\Gamma \vdash_S e_1 : \mathbf{t} \qquad \Gamma \vdash_S e_2 : \phi_1^2 \wedge \ldots \phi_{n_2}^2 \qquad \Gamma \vdash_S e_3 : \phi_1^3 \wedge \ldots \phi_{n_3}^3}{\Gamma \vdash_S \mathbf{if}\ e_1\ e_2\ e_3 : \wedge\{\phi_i \mid \exists k, l. \phi_i = \phi_k^2 = \phi_l^3\}}$$

Figura 8.9: Sistema de tipos más general para análisis de estrictez

8.5 Tipos perezosos

8.5.1 Introducción

La ineficiencia de la interpretación abstracta se debe a que calcula más información de la que realmente se necesita. Si g es una función de n argumentos, la versión abstracta de g considera todas las posibles combinaciones de valores abstractos de esos argumentos. Por ejemplo $g^{\#}\ 1\ 0\ 1\ 0\ 0 = 0$ significa que una llamada a g está indefinida si lo están el segundo, cuarto y quinto argumentos. En algunos casos esta información puede ser útil pero en la mayoría no lo será. La idea detrás de los tipos perezosos es calcular solamente la información necesaria para responder a una determinada pregunta. Un tipo perezoso es equivalente al tipo más general de una expresión, pero en una forma no evaluada.

Un ejemplo que puede servir para proporcionar una intuición acerca de los tipos perezosos es el siguiente.

Ejemplo 88

$$\begin{array}{l} foldr\ g\ b\ nil = b \\ foldr\ g\ b\ (cons(x, xs)) = g\ x\ (foldr\ b\ g\ xs) \\ cat\ l = foldr\ append\ nil\ l \end{array}$$

Utilizando **fix** y *case*, la función *foldr* se escribe:

$$foldr = \mathbf{fix}(\lambda f.\lambda g.\lambda b.\lambda l.case(l, b, \lambda x.\lambda xs.g\ x\ (f\ b\ g\ xs))$$

donde recordemos que $case(l, b, f)$ actúa de la siguiente forma: si $l = nil$ entonces devuelve b, y si es de la forma $cons(x, xs)$ entonces devuelve $f\ x\ xs$. El análisis de esta función usando interpretación abstracta con el dominio de cuatro puntos de Wadler es intratable. Si queremos saber si *cat* es estricta , necesitamos conocer la versión abstracta de *foldr*, la cual es una función en el dominio $(\mathbf{2} \to \mathbf{2} \to \mathbf{2}) \to \mathbf{2} \to \mathbf{2} \to \mathbf{2}$.

Veremos más adelante con más detalle el análisis de estrictez incluyendo listas, pero supongamos de momento que $\mathbf{f}$ es un tipo base correspondiente a la propiedad de ser estricto y $\mathbf{t}$ es otro tipo base correspondiente a todos los valores (semejante a los valores abstractos 0 y 1 de la interpretación abstracta). En términos de tipos, la propiedad que queremos buscar es $cat : \mathbf{f} \to \mathbf{f}$. Utilizando los tipos perezosos esto requerirá probar que $foldr : append \to \mathbf{f} \to \mathbf{t} \to \mathbf{f}$ donde la primera componente del tipo es una clausura no evaluada que se corresponde con la conjunción de todos los tipos de *append*. Esto devuelve el valor *true* directamente, porque si l tiene tipo $\mathbf{f}$ entonces $l = nil$ también y por tanto también el cuerpo de *foldr*. Este caso es extremo porque no necesitamos ninguna información acerca de *append* para obtener la respuesta.

□

8.5.2 Los tipos perezosos

La formación de los tipos perezosos T_G se muestra en la Figura 8.10. Los tipos perezosos son de la forma (Γ, e), y se introducen en los lados izquierdos de las flechas. (Γ, e) pretende representar $MGT(\Gamma, e)$, aunque como ya hemos mencionado, de forma no evaluada. La definición de la función *Expand* establece una correspondencia entre los tipos perezosos y los ordinarios, que se extiende de forma directa a los entornos.

Definición 89

$$\begin{array}{rcl} Expand & : & T_G \to T_I \\ Expand(b) & = & b \qquad si\ b\ es\ un\ tipo\ base \\ Expand(\sigma_1 \wedge \sigma_2) & = & Expand(\sigma_1) \wedge Expand(\sigma_2) \\ Expand(\sigma_1 \to \sigma_2) & = & Expand(\sigma_1) \to Expand(\sigma_2) \\ Expand((\Gamma, e)) & = & MGT(Expand(\Gamma), e) \end{array}$$

$$b \in T'_S \qquad\qquad nil \in Env$$

$$\frac{\Gamma \in Env \qquad \sigma \in T_G}{\Gamma[x \mapsto \sigma] \in Env} \qquad\qquad \frac{\sigma \in T_G \quad \psi \in T'_S}{\sigma \to \psi \in T'_S}$$

$$\frac{\Gamma \in Env \qquad e \in Exp}{(\Gamma, e) \in T_G} \qquad\qquad \frac{\phi_1 \in T'_S \ldots \phi_n \in T'_S}{\phi_1 \wedge \ldots \wedge \phi_n \in T_G}$$

Figura 8.10: El lenguaje de tipos perezosos T_G

$$\frac{\forall j \in [1,m], \exists i \in [1,n]\ \phi_i \leq_G \psi_j}{\phi_1 \wedge \ldots \wedge \phi_n \leq_G \psi_1 \wedge \ldots \wedge \psi_n} \qquad\qquad \frac{\forall \phi.(\Gamma \vdash_G e : \phi) \Rightarrow \psi \leq_G \phi}{\psi \leq_G (\Gamma, e)}$$

$$\frac{\Gamma \vdash_G e : \phi}{(\Gamma, e) \leq_G \phi}\ \phi \neq (\Gamma', e') \qquad\qquad \frac{\phi' \leq_G \phi,\ \psi \leq_G \psi'}{\phi \to \psi \leq_G \phi' \to \psi'}$$

$$\textbf{Conj}\ \frac{\Gamma \vdash_G e : \psi_1 \qquad \Gamma \vdash_G e : \psi_2}{\Gamma \vdash_G e : \psi_1 \wedge \psi_2} \qquad\qquad \textbf{Var}\ \frac{\psi_1 \leq_G \psi_2}{\Gamma[x \mapsto \psi_1] \vdash_G x : \psi_2}$$

$$\textbf{Abs}\ \frac{\Gamma[x \mapsto \phi] \vdash_G e : \psi}{\Gamma \vdash_G \lambda x.e : (\phi \to \psi)} \qquad\qquad \textbf{App}\ \frac{\Gamma \vdash_G e_1 : ((\Gamma, e_2) \to \psi)}{\Gamma \vdash_G e_1\ e_2 : \psi}$$

$$\textbf{Fix}\ \frac{\Gamma \vdash_G (\lambda g.e) : (\wedge_{i=1}^n \phi_i \to \phi_1) \wedge \ldots \wedge (\wedge_{i=1}^n \phi_i \to \phi_n)}{\Gamma \vdash_G \textbf{fix}(\lambda g.e) : \phi_k \qquad (k \in [1,n])}$$

Figura 8.11: Sistema de tipos perezosos

Las nuevas reglas de orden y de deducción se muestran en la Figura 8.11.

Se puede establecer la corrección y la completitud de este sistema respecto al original, [HM94] y [HL94b].

Teorema 90 *(Corrección)*

$$\Gamma \vdash_G e : \phi \Rightarrow \mathit{Expand}(\Gamma) \vdash_T e : \mathit{Expand}(\phi)$$

Teorema 91 *(Completitud)*

$$\mathit{Expand}(\Gamma) \vdash_T e : \mathit{Expand}(\phi) \Rightarrow \Gamma \vdash_G e : \phi$$

Luego

Teorema 92 *(Equivalencia)*

$$\Gamma \vdash_T e : \phi \Leftrightarrow \Gamma \vdash_G e : \phi$$

si $\Gamma : Var \to T_I$ *y* $e : \phi \in T_I$.

Ejemplo 93 (Análisis de estrictez)
Las reglas específicas para el análisis de estrictez se muestran en la Figura 8.12.

□

$$\mathbf{f}_\sigma \leq_G \phi \qquad \phi \leq_G \mathbf{t}_\sigma \qquad \phi_1 \to \ldots \to \phi_n \to \phi \leq_G \psi_1 \to \ldots \to \psi_n \to \mathbf{t}$$

$$\mathbf{Taut}\ \Gamma \vdash_G c : \mathbf{t} \qquad \mathbf{Cond_1}\ \frac{\Gamma \vdash_G e_1 : \mathbf{f}}{\Gamma \vdash_G \mathbf{if}\ e_1\ e_2\ e_3 : \phi} \qquad \mathbf{Cond_2}\ \frac{\Gamma \vdash_G e_2 : \phi \qquad \Gamma \vdash_G e_3 : \phi}{\Gamma \vdash_G \mathbf{if}\ e_1\ e_2\ e_3 : \phi}$$

Figura 8.12: Sistema de tipos perezosos para el análisis de estrictez

Ejemplo 94 (Análisis del momento de vinculación usando cpers)
Veamos un análisis del momento de vinculación. Como constantes del lenguaje consideramos constantes de tipo básico c y dos constantes funcionales: $+$ y el condicional **if**. Las constantes de tipo son dos, como ya sugerimos al principio del capítulo: *static* y *dynamic*, donde $\mathit{static} \leq \mathit{dynamic}$.

Modelizamos los tipos usando cpers, como mostramos en el Ejemplo 77. La interpretación de las constantes es la siguiente:

$$I_{cper}(\mathit{static}) = \{(x, x) \mid x \in D\} (= Id)$$
$$I_{cper}(\mathit{dynamic}) = \{(x, y) \mid x, y \in D\} (= All)$$

Es inmediato verificar que $I_{cper}(static) \subseteq I_{cper}(dynamic)$ y por tanto que:

$$\phi \leq \psi \Rightarrow [\![\phi]\!] \sqsubseteq [\![\psi]\!]$$

luego la estructura es un modelo.

Con respecto a las reglas de asignación de tipos a las constantes, todas las constantes de tipo base son estáticas, luego tenemos el siguiente axioma:

$$\textbf{Const} \ \vdash_G c : static$$

el cual es correcto puesto que $norm([\![static]\!]) = D$. Las reglas para + y el condicional son respectivamente:

$$+ \ \frac{\Gamma \vdash_G e_1 : static \qquad \Gamma \vdash_G e_2 : static}{\Gamma \vdash_G (e_1 + e_2) : static}$$

$$+ \ \frac{\Gamma \vdash_G e_1 : \phi \qquad \Gamma \vdash_G e_2 : \psi}{\Gamma \vdash_G (e_1 + e_2) : dynamic}$$

$$\textbf{Cond} \ \frac{\Gamma \vdash_G b : static \qquad \Gamma \vdash_G e_1 : \phi \qquad \Gamma \vdash_G e_2 : \phi}{\Gamma \vdash_G \textbf{if } b \ e_1 \ e_2 : \phi}$$

$$\textbf{Cond} \ \Gamma \vdash_G \textbf{if } b \ e_1 \ e_2 : dynamic$$

Se ve fácilmente que las reglas del + son correctas. Veamos como ejemplo de demostración de corrección la primera regla para el condicional. Por hipótesis se cumple:

$$\begin{array}{lcl} Expand(\Gamma) & \models & b : static \\ Expand(\Gamma) & \models & e_1 : Expand(\phi) \\ Expand(\Gamma) & \models & e_2 : Expand(\phi) \end{array}$$

$sem[\![\textbf{if } b \ e_1 \ e_2]\!] \ \rho$ tiene cuatro casos distintos. Si $sem[\![b]\!] \ \rho = \bot$ entonces $sem[\![\textbf{if } b \ e_1 \ e_2]\!] \ \rho = \bot$ y por ser $Expand(\phi)$ una cper se cumple $(\bot, \bot) \in Expand(\phi)$, y por tanto

$$\bot \in norm([\![Expand(\phi)]\!])$$

En caso contrario, entonces $sem[\![\textbf{if } b \ e_1 \ e_2]\!] \ \rho = e_1$ o $sem[\![\textbf{if } b \ e_1 \ e_2]\!] \ \rho = e_2$, en cuyo caso por hipótesis $sem[\![e_i]\!] \ \rho \in norm([\![Expand(\phi)]\!])$. Luego:

$$Expand(\Gamma) \models \textbf{if } b \ e_1 \ e_2 : Expand(\phi)$$

□

Un ejemplo experimental

En la Figura 8.13 se presentan las reglas para un análisis de estrictez de listas. Las constantes de tipo son las ya presentadas en la Sección 8.2:

$$B_{sl} = \{\mathbf{f}, \infty, \mathbf{f}_\epsilon, \mathbf{t}\} \text{ donde } \mathbf{f} \leq \infty \leq \mathbf{f}_\epsilon \leq \mathbf{t}$$

cada una de las cuales representan al mismo tipo de listas que en el correspondiente dominio de cuatro puntos de Wadler. Para distinguir $\mathbf{t}$ y $\mathbf{f}$ del tipo base y de listas del tipo base, vamos a indicar los del tipo base de la forma $\mathbf{t}_b$ y $\mathbf{f}_b$. Observese la analogía entre las reglas dadas en la Sección 5.4.2 para el análisis por casos y las de la Figura 8.13.

Ya vimos anteriormente los motivos por los que la interpretación abstracta es muy costosa frente al uso de los tipos perezosos. El algoritmo de tipos perezosos ha sido implementado en CAML light [HL94b]. Para la función *cat* del Ejemplo 88 se obtuvieron los siguientes resultados:

$$\begin{array}{ll} cat : \mathbf{f}_\epsilon \rightarrow \mathbf{f} & \text{False } 0.002s \\ cat : \infty \rightarrow \mathbf{f} & \text{False } 0.003s \\ cat : \infty \rightarrow \infty & \text{True } 0.001s \end{array}$$

Esto quiere decir, que dado un tipo, tarda milésimas de segundo en comprobar si es o no tipo de la función *cat*. Utilizando interpretación abstracta mediante un algoritmo de fronteras se tarda alrededor de 30 minutos. En [HL94b] se presentan algunos resultados experimentales más.

$$\textbf{Taut}-\textbf{hd}\quad \Gamma \vdash_G hd(e) : \mathbf{t}_t \qquad \textbf{Hd}\ \frac{\Gamma \vdash_G e : \mathbf{f}}{\Gamma \vdash_G hd(e) : \phi}$$

$$\textbf{Taut}-\textbf{tl}\quad \Gamma \vdash_G tl(e) : \mathbf{t} \qquad \textbf{Tl}-\textbf{1}\ \frac{\Gamma \vdash_G e : \mathbf{f}}{\Gamma \vdash_G tl(e) : \mathbf{f}} \qquad \textbf{Tl}-\textbf{2}\ \frac{\Gamma \vdash_G e : \infty}{\Gamma \vdash_G tl(e) : \infty}$$

$$\textbf{Taut}-\textbf{cons}\quad \Gamma \vdash_G cons(e_1, e_2) : \mathbf{t} \qquad \textbf{Cons}_\mathbf{1}\ \frac{\Gamma \vdash_G e_2 : \infty}{\Gamma \vdash_G cons(e_1, e_2) : \infty}$$

$$\textbf{Cons}_\mathbf{2}\ \frac{\Gamma \vdash_G e_2 : \mathbf{f}_\epsilon}{\Gamma \vdash_G cons(e_1, e_2) : \mathbf{f}_\epsilon} \qquad \textbf{Cons}_\mathbf{3}\ \frac{\Gamma \vdash_G e_1 : \mathbf{f}_b}{\Gamma \vdash_G cons(e_1, e_2) : \mathbf{f}_\epsilon}$$

$$\textbf{Case}_\mathbf{1}\ \frac{\Gamma \vdash_G e_1 : \mathbf{f}}{\Gamma \vdash_G case(e_1, e_2, e_3) : \phi}$$

$$\textbf{Case}_\mathbf{2}\ \frac{\Gamma \vdash_G e_1 : \infty \quad \Gamma \vdash_G e_3 : \mathbf{t}_b \rightarrow \infty \rightarrow \phi}{\Gamma \vdash_G case(e_1, e_2, e_3) : \phi}$$

$$\textbf{Case}_\mathbf{3}\ \frac{\Gamma \vdash_G e_1 : \mathbf{f}_\epsilon \quad \Gamma \vdash_G e_3 : \mathbf{t}_b \rightarrow \mathbf{f}_\epsilon \rightarrow \phi \wedge \mathbf{f}_b \rightarrow \mathbf{t} \rightarrow \phi}{\Gamma \vdash_G case(e_1, e_2, e_3) : \phi}$$

$$\textbf{Case}_\mathbf{4}\ \frac{\Gamma \vdash_G e_2 : \phi \quad \Gamma \vdash_G e_3 : \mathbf{t}_b \rightarrow \mathbf{t} \rightarrow \phi}{\Gamma \vdash_G case(e_1, e_2, e_3) : \phi}$$

Figura 8.13: Análisis de estrictez para listas

Capítulo 9

Conclusiones y trabajo futuro

Los análisis de programas ofrecen técnicas para predecir de forma segura y computable aproximaciones del conjunto de valores o de los comportamientos de un programa.

En este trabajo hemos tratado de sistematizar las distintas técnicas utilizadas en análisis de lenguajes funcionales. Se trata de un campo muy amplio, por lo que se ha seguido un proceso de selección, explicando en cada momento los motivos por los que no se ha profundizado en una determinada técnica. En dichos casos, se proporciona bibliografía relacionada, por si el lector se halla interesado.

Se ha seguido un orden cronológico, exponiendo no solamente las técnicas más arraigadas sino también las que se encuentran en pleno proceso de desarrollo. Y dentro de una misma técnica se ha seguido un orden de complejidad creciente. Por ejemplo, en el Capítulo 5 partimos de un lenguaje de primer orden y después pasamos a orden superior, dominios no planos y finalmente polimorfismo.

Se ha proporcionado una comparación entre las distintas técnicas a distintos niveles. Por una parte se comparan a nivel de potencia. Por ejemplo, se ha visto que la interpretación abstracta usando conjuntos Scott–cerrados permite hacer menos análisis que la que utiliza relaciones de equivalencia parcial; y que la técnica basada en tipos exige la existencia de tipos conjuntivos para mantener la potencia de la intepretación abstracta. Por otra parte, se comparan a nivel de eficiencia, proporcionando en el caso de la interpretación abstracta métodos para calcular de una forma algo más eficiente puntos fijos o aproximaciones a ellos.

Además se han presentado ejemplos de análisis en todas las técnicas. Aunque el análisis sobre el que más se ha investigado es el análisis de estrictez, se han presentado también el análisis de compartición, globalización de variables, análisis de clausuras, análisis del momento de vinculación, de propagación de constantes, de ausencia y de divergencia.

9.1 Comparación de enfoques

Según lo estudiado en este trabajo de investigación, podemos clasificar las técnicas de análisis de programas funcionales en base a distintos criterios. En la Figura 9.1 se muestra un cuadro resumen de la clasificación de lás técnicas junto con los inconvenientes de cada una de ellas. La primera clasificación se puede hacer en dos grandes grupos: técnicas basadas en interpretación abstracta y técnicas basadas en tipos.

La mayor parte de la investigación en el área se ha llevado a cabo en interpretación abstracta. Se trata de una técnica muy general que ha avanzado de forma simultánea en distintos frentes y que se aplica no solamente a lenguajes funcionales sino también a lenguajes lógicos e imperativos.

La aplicación de la interpretación abstracta a los lenguajes imperativos está fundamentalmente orientada a la verificación y a la evaluación parcial aplicada a la generación de compiladores. En la programación lógica existe también una gran aplicación de esta técnica: análisis de independencia, de circularidad, de modo etc.

A su vez la interpretación abstracta se puede dividir en función de distintos criterios. Se puede establecer una primera clasificación en base a la forma de propagar la información a lo largo del árbol sintáctico. Según este criterio se dividen en análisis hacia delante y análisis hacia atrás.

Los **análisis hacia delante** propagan la información de las hojas a la raiz del árbol sintáctico, es decir, a partir de información sobre los argumentos de una función obtenemos información sobre el resultado de la función. La interpretación de una función f de n argumentos es una función n–aria $f^{\#}$ sobre valores abstractos. Si la función f es recursiva entonces $f^{\#}$ es solución de una ecuación recursiva. Para resolverla es necesario el cálculo de un punto fijo. Aquí es donde se encuentra la ineficiencia del análisis hacia delante, ya que el cálculo de dicho punto fijo es exponencial en el número de argumentos de f. Si además f es una función de orden superior, el coste es doblemente exponencial. Por otra parte surge la necesidad de representar de una forma lo más compacta posible las funciones abstractas. Lo más inmediato es representarlas mediante *tablas argumento–resultado*. El inconveniente de esta forma de representación es la gran cantidad de espacio que ocupa y que la comparación de dos funciones (necesaria para determinar si se ha alcanzado el punto fijo) es siempre lineal, por lo que el coste del cálculo del punto fijo es siempre exponencial. Hemos visto que una forma alternativa en el caso del análisis de estrictez es el uso de *expresiones booleanas* (formas normales conjuntivas o disyuntivas). Los inconvenientes de esta forma de representación son que sólo tiene sentido en análisis cuyo dominio abstracto es de aspecto booleano, que es prácticamente inmanejable su extensión a orden superior y que resulta difícil establecer cuándo dos expresiones booleanas con distinto aspecto son en realidad la misma. La última forma de representación expuesta es el uso de *fronteras* (nodos del retículo de parámetros donde la función cambia de valor). Se trata de una forma bastante compacta de representar las funciones. Además la frontera determina de forma única a cada función, por lo que para comparar dos funciones basta con comparar sus fronteras. En cada paso de iteración es necesario calcular la frontera de la nueva aproximación en función de la anterior. Para ello se utilizan búsquedas hacia arriba y hacia abajo en el retículo de argumentos, cada una de las cuales influye en las demás podando su espacio de búsqueda. Aunque el coste del cálculo del punto fijo en el caso peor sigue siendo exponencial, en el caso medio funciona mejor.

Una forma de paliar el problema de la ineficiencia que puede resultar más satisfactoria que el uso de fronteras consiste en renunciar a la exactitud en el cálculo de punto fijo en favor de la eficiencia. Así, en la Sección 5.5.4 vimos una forma de aproximar el punto fijo de una función mediante operadores de ensanchamiento y estrechamiento, y cómo ésto se traduce en el cálculo del punto fijo en un retículo más simple.

Los **análisis hacia atrás** propagan la información de la raiz a las hojas del árbol sintáctico, es decir, a partir de información sobre el contexto en que se llama a una función obtenemos información sobre el contexto en que se evalúan los argumentos. En realidad, el análisis hacia atrás se puede considerar un análisis hacia delante donde la información propagada son transformadores de contextos, es decir funciones de contextos en contextos. El concepto de contexto se puede formalizar de distintas formas, de las cuales hemos elegido las proyecciones. La interpretación de una función f de n parámetros es un conjunto de n transformadores de proyecciones f^i. Cada f^i propaga el contexto en que se evalua la función hacia el argumento i–ésimo. Si f es una función recursiva, cada f^i es solución de una ecuación recursiva, lo cual implica el cálculo de n puntos fijos. Pero puesto que cada función a calcular es ahora una función unaria, el coste es en el peor de los casos cuadrático en el tamaño del programa. Los análisis hacia atrás tienen una seria limitación: su extensión a orden superior no parece proporcionar información con la precisión que desearíamos.

La segunda clasificación de los análisis basados en interpretación abstracta se puede hacer en base a la forma de representar las propiedades de los valores. Se usan conjuntos Scott–cerrados, relaciones de equivalencia total o relaciones de equivalencia parcial.

Inicialmente se utilizaron los **conjuntos Scott–cerrados** (cerrados inferiormente). La semántica denotacional de los lenguajes funcionales suele utilizar los dominios de Scott como dominios semánticos donde las expresiones del lenguaje toman valores. Los elementos de un dominio de Scott se relacionan mediante una relación de orden $\sqsubseteq$, de forma que el elemento que se encuentra más abajo respecto a esta relación de orden es el elemento con menos información, y a medida que subimos en el dominio los elementos tienen cada vez más información acumulada. Esto quiere decir que a partir de un elemento podemos deducir todas las informaciones de los elementos menores que el. Esta es la idea que pretenden capturar los conjuntos Scott–cerrados. Si un elemento pertenece a un conjunto Scott–cerrado, todos los que están por debajo de él también pertenecen, puesto que todos ellos tienen también la propiedad que posee ese elemento.

Como hemos visto en el trabajo, existen algunas propiedades que no se pueden representar utilizando conjuntos Scott– cerrados. Los dos ejemplos que hemos visto son la estrictez de cabeza y la propiedad de que una función es constante. El uso de **relaciones de equivalencia total** resuelve parcialmente el problema. Ambas propiedades se pueden representar de esta forma, pero solamente para funciones de primer orden. La técnica de análisis hacia atrás usando proyecciones es equivalente al uso de relaciones de equivalencia total. El uso de **relaciones de equivalencia parcial** resuelve por completo el problema, puesto que permite representar ambas propiedades para funciones de orden superior. Es por tanto la técnica más potente hasta el momento.

Finalmente, las técnicas basadas en interpretación abstracta se pueden clasificar en base al tipo de semántica respecto a la cual se demuestra la corrección del análisis: denotacional u operacional.

Una gran parte de los análisis desarrollados están basados en **semántica denotacional**, debido a la naturaleza declarativa de los lenguajes funcionales, que nos permite abstraernos de la implementación concreta del lenguaje. Hemos visto dos formas de dar la corrección en este caso.

La primera consiste en usar **conexiones de Galois**. Una conexión de Galois, como ya hemos visto, es un par de funciones, una de abstracción y otra de concreción, con unas determinadas propiedades que aseguran que la pérdida de información producida en el proceso de abstracción es segura. Se define una función de abstracción particular para los tipos base, que se puede extender de forma general a los tipos funcioneales. Estas funciones de abstracción deben cumplir una serie de propiedades, como ser continuas, estrictas etc. La corrección del análisis depende de que las constantes particulares del lenguaje cumplan una condición de seguridad. Es la forma habitual de proporcionar la corrección en la interpretación abstracta basada en conjuntos Scott–cerrados.

La segunda consiste en usar **relaciones lógicas**, de forma que la semántica estándar y abstracta de las expresiones pertenezcan a dicha relación de alguna forma. La interpretación abstracta basada en relaciones de equivalencia parcial utiliza relaciones lógicas ternarias para demostrar la corrección.

Como ya hemos visto en la Sección 7.2 existen algunas propiedades intensionales que no pueden ser capturadas por una semántica denotacional. La solución se encuentra en la **semántica instrumentada**, que es normalmente una extensión de la semántica denotacional con la información deseada de tipo operacional.

Mediante el análisis de clausuras los análisis basados en semántica operacional se pueden extender bien a orden superior y en algunos casos la complejidad parece ser menor (polinomial).

La corrección de estos análisis se establece respecto a una semántica operacional, y esto implica tener en cuenta la estrategia de evaluación que se está usando en la implementación del lenguaje.

Y por último tenemos los **análisis basados en tipos**. Este tipo de análisis se introduce con la esperanza de incrementar la eficiencia. Hemos visto que para capturar toda la potencia de la interpretación abstracta es necesario la introducción de los llamados tipos conjuntivos (conjunciones de tipos). Además, para que la eficiencia mejore respecto a la interpretación abstracta, es necesaria la introducción de los tipos perezosos. Estos son tipos equivalentes al tipo más general de una expresión pero en una forma no evaluada, lo cual se representa mediante la introducción de clausuras en los lados izquierdos de los tipos funcionales.

9.2 Juicio crítico

A lo largo de este trabajo se ha puesto de manifiesto que el desarrollo teórico realizado en torno al análisis de programas en lenguajes funcionales ha sido bastante amplio.

La década de los 80 y principios de los 90 ha sido la época más productiva en este área. Este desarrollo teórico parece en cierto modo provocado por las necesidades de potencia que se van planteando a la hora de realizar ciertos análisis, así como por el deseo de mejorar la eficiencia. La interpretación abstracta basada en relaciones de equivalencia parcial es la técnica más potente hasta el momento, pero la mayoría de los análisis planteados no requieren tal potencia. Por el momento, parece que solamente dos análisis se benefician de ella: estrictez de cabeza y propagación de constantes. Por ello, parece lógico mantener el uso de los conjuntos Scott–cerrados mientras no sea necesario complicar el marco teórico. Se echa de menos un estudio en profundidad del tipo de propiedades que no pueden expresarse mediante conjuntos Scott–cerrados y sí mediante relaciones de equivalencia parcial.

Por otra parte, existe una deficiencia importante con respecto al análisis de lenguajes funcionales: la implementación práctica de los mismos.

En la mayoría de los análisis presentados no se proporcionan medidas de la mejora producida en el código generado como resultado del análisis, por lo que no se puede sopesar frente al coste que conlleva el análisis. Además, en aquellos casos en los que el análisis sí se ha implementado, como por ejemplo el análisis de estrictez, se ha optado por versiones bastante simplificadas.

Por ejemplo en Haskell se ha implementado un análisis de estrictez de primer orden del estilo de Mycroft [Myc81]. En una versión experimental posterior [PP93] se implementó un análisis de estrictez de orden superior sobre dominios no planos no recursivos. Para calcular los puntos fijos se usó un operador de *ensanchamiento* muy sencillo. Con el análisis de estrictez activado existían mejoras de hasta un 30% en el tiempo de ejecución del código generado. Sería necesario un mayor esfuerzo implementador para estudiar si compensa utilizar un operador de ensanchamiento más sutil y uno de estrechamiento combinado con él, como vimos en la Sección 5.5.4.

Aun así, en la actualidad sigue utilizándose el análisis de primer orden. Si para evitar el coste de la extensión a orden superior se mantiene un análisis simple de primer orden, sería mejor entonces utilizar un análisis hacia atrás, que es bastante más eficiente.

La técnica basada en tipos es bastante reciente y aun se encuentra en pleno desarrollo. Como ya dijimos en el Capítulo 8, surgió a partir del deseo de eficiencia en los análisis. Es conocido que el algoritmo de inferencia para el sistema de tipos Hindley–Milner es exponencial en el caso peor, aunque en los casos habituales que se presentan se comporta de forma polinómica. No existen aun las suficientes garantías de que esta técnica sea mucho más eficiente. Como ya hemos dicho anteriormente, la introducción de los tipos perezosos parece que mejora la eficiencia, pero aun son necesarias medidas rigurosas que apoyen esta intuición.

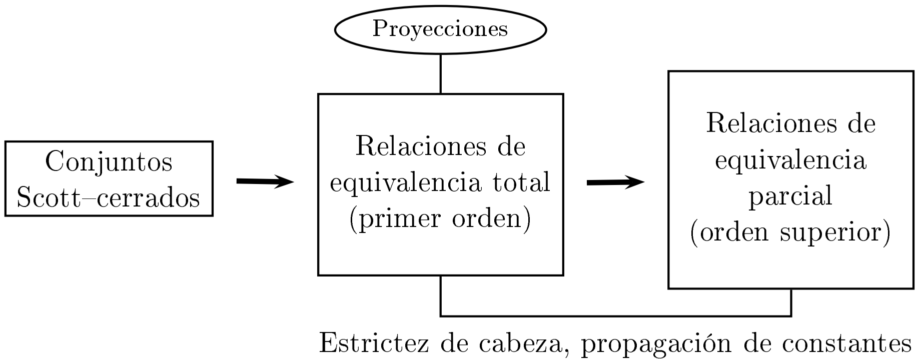
El análisis de programas sigue siendo un tema de plena actualidad y considero que es un área con futuro. Actualmente existe un gran esfuerzo investigador en torno a la evaluación parcial, a la que hay incluso dedicada una conferencia: el PEPM (Partial Evaluation and Semantic–Based Program Manipulation). Otra conferencia dedicada al campo del análisis estático de programas es el SAS (International Static Analysis Symposium). En ella se presentan trabajos sobre interpretación abstracta, análisis de complejidad, especialización de programas, flujo de datos, sistemas de verificación y de inferencia de tipos etc. Se tratan, no solamente la programación funcional, sino también programación lógica, imperativa, orientada a objetos, concurrente y con restricciones.

Existen varios grupos investigadores a destacar. En la Universidad de Copenhaguen (DIKU) existe un grupo de investigación (TOPPS) dedicado a la evaluación parcial. Trabajan en varios proyectos de investigación, entre ellos los proyectos DART (Design, Analysis and Reasoning about Tools) y PEKIS. El primero en un proyecto de colaboración que desarrolla técnicas de análisis y manipulación de programas relevantes en el proceso de producción de software. En el contexto de este proyecto se han desarrollado varios sistemas de evaluación parcial: Similix (para Scheme), C–Mix (para C) y ML–Mix (para Standard ML). El segundo aplica la evaluación parcial en un sistema industrial complejo. En la Universidad de Aarhus (DAIMI), Fleming y Hanne Riis Nielson desarrollan un marco teórico basado en lenguajes funcionales multinivel. La idea consiste en factorizar la semántica denotacional de un lenguaje funcional en dos partes: el núcleo de la semántica y una interpretación. La interpretación especifica los detalles relevantes a una interpretación concreta (estándar o abstracta) de los valores y operaciones de un programa. El núcleo especifica aquellas partes de la semántica que se van a usar de la misma forma en todas las interpretaciones. Es la parte de la semántica que consideramos fija (por ejemplo la interpretación de las λ-abstracciones siempre como funciones).

En el École Normale Supérieure (ENS) Patrick Cousot desarrolla una teoría general de interpretación abstracta y la aplica a la verificación de programas imperativos. En el análisis basado en tipos trabajan fundamentalmente en colaboración Chris Hankin (Imperial College) y Daniel Le Métayer (INRIA/IRISA).

INTERPRETACIÓN ABSTRACTA:

- **Propagación de información**
  - Hacia delante: costosa
  - Hacia atrás: difícil extender a orden superior
- **Representación de propiedades:**



- **Semántica “base”:**
  - Denotacional: no captura propiedades operacionales
  - Operacional: demasiado dependiente de la máquina concreta

ANÁLISIS BASADOS EN TIPOS $\Rightarrow$ Necesidad de tipos conjuntivos

Figura 9.1: Panorama de las técnicas vistas

9.3 Trabajo futuro

El campo del análisis de programas funcionales es muy amplio, y existen algunas vertientes que han quedado aun por estudiar.

Por una parte hemos visto que la técnica de análisis hacia atrás es difícil de extender a orden superior. Un camino para solucionar esto, que ya se sugirió en [Hug88], consiste en combinar interpretación abstracta hacia delante y hacia atrás para extender un análisis hacia atrás a orden superior. Queda pendiente como trabajo futuro estudiar algunos trabajos relacionados con esta técnica.

Por otra parte hemos visto por separado los análisis basados en semántica denotacional y operacional. Una forma de conectarlas que ha quedado pendiente por estudiar es el uso de los transformadores de evaluación de Burn [Bur87] [FB92]. Se establece un criterio por el cual cambiar la estrategia de evaluación de un programa no cambia su semántica denotacional. Se trata de un análisis hacia atrás, que toma un evaluador para el resultado y devuelve los evaluadores apropiados para los argumentos. Se han proporcionado esquemas de compilación que usan este modelo para generar código para la máquina–G sin espina y algunas variantes paralelas de la misma.

En cuanto a los análisis basados en tipos, queda pendiente el estudio del polimorfismo [Mos94].

En el contexto de cómputos paralelos existe aun mucho campo por investigar. Un problema crucial en la paralelización implícita de lenguajes funcionales es decidir cuándo una expresión debe ser evaluada localmente o por otro procesador. Éste no es solamente un problema de cúando es seguro que una expresión se evalúe remotamente (ya hemos dicho que de ello se puede encargar el análisis de estrictez) sino también de cuándo es eficiente, puesto que la creación de un proceso tiene un coste mayor que la llamada local a una función y por tanto sólo vale la pena crear un proceso cuando el “grano” de la expresión es lo suficientemente grueso. Esto exige realizar un análisis de coste, que permita anotar cada expresión de un programa con el número de pasos que requiere su evaluación. Este problema es bastante complejo y existe abundante literatura sobre el mismo.

Existen pocos lenguajes funcionales con paralelismo explícito. Entre ellos se encuentran Concurrent ML y Eden. Actualmente trabajo en el desarrollo de este último. En la siguiente sección se describe brevemente el lenguaje Edén y los problemas susceptibles de análisis que la autora se propone abordar en un futuro próximo.

El lenguaje Edén

El lenguaje Eden [KOMP98] [BLOMP96] extiende el lenguaje funcional Haskell mediante construcciones sintácticas que permiten definir procesos de forma explícita. Existen una nueva expresión `process x -> e` y un nuevo tipo `Process a b`, para definir una *abstracción de proceso* que tiene a la variable `x::a` como entrada y a la expresión `e::b` como salida. Las abstracciones de proceso se pueden comparar con las funciones. La principal diferencia entre ellas es que cuando se crean las concreciones de proceso, éstas se ejecutan en paralelo. Una *concreción de proceso* se obtiene usando el operador infijo predefinido `(#)::Process a b -> a -> b`. Las abstracciones de proceso son valores de primera clase, luego se pueden usar como parámetros o resultados de funciones. Esto facilita la definición de esqueletos genéricos [GPP96]. Por ejemplo, a continuación se muestra una tubería:

```
pipe :: [Process a a] -> Process a a
pipe [p] = p
pipe (p:ps) = process z -> pipe ps # (p # z)
y = pipe [p1,p2,p3] # x
```

Un lenguaje como Edén ofrece muchas oportunidades para utilizar las técnicas vistas en este trabajo.

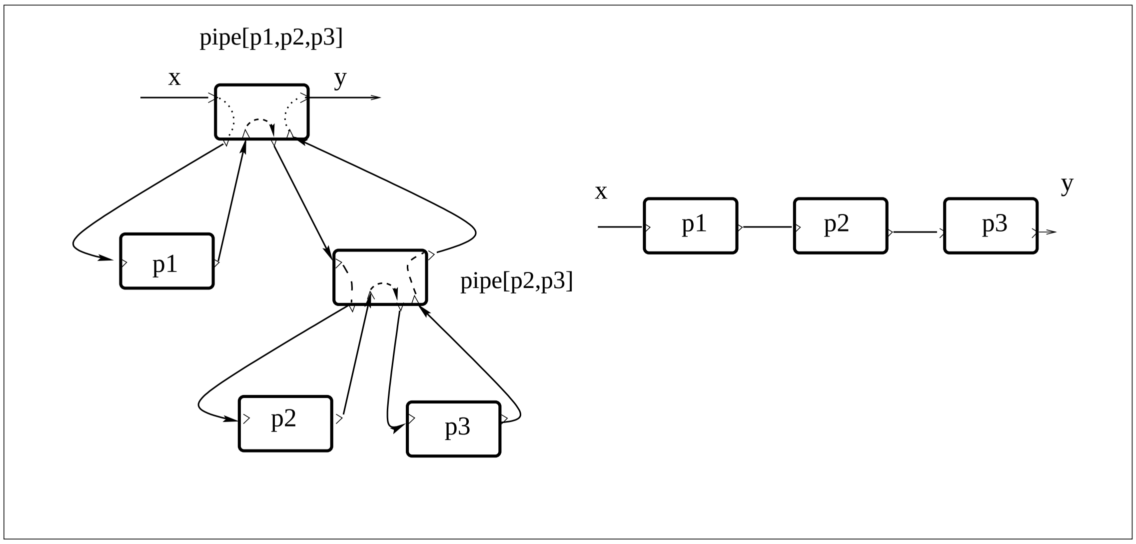


Figura 9.2: Una tubería

Conexión directa

Cuando un proceso Edén crea otros procesos-hijo, se comunica con ellos por medio de canales. En muchos casos el padre no ha de intervenir posteriormente en las comunicaciones más que como un mero intermediario. En esos casos es más eficiente conectar directamente a los hijos entre sí. Llamamos a esta optimización *conexión directa* o *automatic bypassing*. Retomemos el ejemplo de la tubería. En la implementación actual de Eden [KOMP98] se despliega la definición recursiva en tiempo de ejecución generándose la topología de la parte izquierda de la Figura 9.2. Esta estructura no es la deseable, la cual se muestra en la parte derecha de la Figura 9.2, pues no se crean solamente procesos para `p1`, `p2` y `p3`, sino también para `pipe [p1,p2,p3]` and `pipe [p2,p3]`, que únicamente transmiten datos de sus entradas a sus salidas. Además los canales se conectan de una forma jerárquica siguiendo el despliegue de la llamada recursiva. Es deseable detectar tales situaciones, para conectar de forma directa al productor y al consumidor, como en la parte derecha de la Figura 9.2. Esta detección se puede realizar en tiempo de compilación mediante un análisis semejante al de uso. De esta forma, aunque los procesos adicionales aun se crearían, estos terminarían inmediatamente tras crear a sus procesos hijos, al carecer de entradas y salidas.

Tratamiento impaciente de procesos

En Edén los procesos declarados a nivel superior se lanzan de forma impaciente, mientras que aquellos que se hallan encerrados en una clausura que es a su vez una expresión de tipo *let*, solamente se crean cuando se usa la variable a la que están ligados. Si, mediante un análisis semejante al análisis de estrictez, supiéramos que dicha variable se va a usar, podríamos adelantar el lanzamiento del proceso.

Detección de interbloqueos

Para algunas topologías y para algunos procesos con cierta estructura sintáctica —por ejemplo, los llamados procesos *guardados*— es posible asegurar en tiempo de compilación que no se van a producir interbloqueos en tiempo de ejecución. Se trata de aplicar las técnicas de interpretación abstracta a esta cuestión buscando ampliar la clase de programas "seguros". Para el resto, el compilador podría emitir un aviso indicando que es necesaria una prueba formal de ausencia de bloqueo.

No determinismo

La única fuente de no determinismo en Edén es el proceso `merge`, el cual, dada una lista de canales que son listas potencialmente infinitas, mezcla de forma justa las entradas, generando como salida una única lista. Su influencia se puede extender a una buena parte del programa, por lo que sería interesante detectar en tiempo de compilación las porciones de texto que siguen conservando una semántica funcional pura y las porciones que han perdido tal característica debido al uso directo o indirecto de procesos no deterministas. La idea es definir un sistema de tipos con anotaciones, siguiendo la línea de los *tipos únicos* de Clean [PvE93] o de los sistemas *lineales* de tipos utilizados en Haskell [TWM95] que permita realizar tal inferencia.

Bibliografía

[Abr86] S. Abramsky. **Strictness analysis and polymorphic invariance**. In H. Ganzinger and N. D. Jones, editors, *Proceedings of a Workshop on Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 1–23. Springer Verlag, October 1986.

[Abr90] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1(1):5–40, July 1990.

[AJ89] L. Augustsson and T. Johnsson. Parallel graph reduction with the <ν, G>-machine. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture '89, Imperial College, London*, pages 202–213, New York, NY, 1989. ACM.

[AJ91] S. Abramsky and T. P. Jensen. A relational approach to strictness analysis for higher-order polymorphic functions. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, pages 49–54, Orlando, FL, USA, January 1991. ACM Press.

[AJ94] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science Volume 3*, pages 1–168. Oxford University Press, 1994.

[Bar93] G. Baraki. ***Abstract Interpretation of Polymorphic Higher-Order Functions***. PhD thesis, University of Glasgow, February 1993.

[Ben92] P. N. Benton. Strictness logic and polymorphic invariance. *Lecture Notes in Computer Science*, 620, 1992.

[Ben93] P. N. Benton. Strictness properties of lazy algebraic datatypes. *Lecture Notes in Computer Science*, 724, 1993.

[BH86] A. Bloss and P. Hudak. Variations on Strictness Analysis. In *1986 ACM Symposium on Lisp and Functional Programming*, pages 132–142, Cambridge, Massachusetts, August 4–6, 1986. ACM Press, New York.

[BH88] A. Bloss and P. Hudak. Path Semantics. In M. Mislove, editor, *Mathematical Foundation of Programming Language Semantics'87, New Orleans, Louisiana*, volume 298 of *LNCS*, pages 476–489. Springer-Verlag, April 1988.

[BHA86] G. L. Burn, C. L. Hankin, and S. Abramsky. **The Theory of Strictness Analysis for Higher Order Functions**. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, volume 217 of *LNCS*, pages 42–62. Springer-Verlag, October 1986.

[BLOMP96] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language definition and operational semantics. Technical Report, Bericht 96-10, revised version, Philipps-Universität Marburg, Germany, 1996.

[Bon90] A. Bondorf. Self-Applicable Partial Evaluation. Ph.D. Thesis 90/17, DIKU, Univ. of Copenhagen, Denmark, 1990.

[Bry90] R. E. Bryant. Graph-based algorithms for boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.

[Bur87] G. L. Burn. Evaluation transformers - A model for the parallel evaluation of functional languages. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 446–470. Springer-Verlag, Berlin, DE, 1987. Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, OR.

[Bur90] G. L. Burn. A Relationship between Abstract Interpretation and Projection Analysis. In *17th POPL, San Fransisco, California*, pages 151–155. ACM Press, January 1990.

[Bur91] G. L. Burn. ***Lazy Functional Languages: Abstract Interpretation and Compilation***. Research Monographs in Parallel and Distributed Computing. Pitman in association with MIT Press, 1991. 238pp.

[Bur92] G. L. Burn. A logical framework for program analysis. In J. Launchbury and P. Sansom, editors, *Glasgow functional programming workshop*, Workshops in Computer Science, pages 30–42. Springer Verlag, July 1992.

[Bur91] G. L. Burn. The abstract interpretation of higher-order functional languages: From properties to abstract domains. In R. Heldal, C. Holst Kehler, and P. Wadler, editors, *Glasgow functional programming workshop*, pages 56–72, August 91.

[CC77] P. Cousot and R. Cousot. **Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixed Points**. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[CC92b] P. Cousot and R. Cousot. **Abstract Interpretation and applications to logic programs**. *J of Logic Programming*, 13(2-3):103–180, July 1992.

[CGW89] T. Coquand, C. Gunter, and G. Winskel. Domain theoretic models of polymorphism. *Information and Computation*, 81(2):123–167, May 1989.

[Cou78] P. Cousot. Méthodes itératives de construction et d'approximation de point fixes d'opérateurs monotone sur un treilis, analyse sémantique des programmes. Thése de Doctoratés Science Mathématiques, Univ. of Grenoble, France, 1978.

[Cou81] P. Cousot. Semantics Foundation of Program Analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

[CP85] C. Clack and S. L. Peyton Jones. Strictness analysis - A practical approach. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 35–49. Springer-Verlag, Berlin, DE, 1985. Lecture Notes in Computer Science 201Proceedings of. Conference at Nancy.

[Cré90] P. Crégut. An abstract machine for lambda- terms normalization. In G. Kahn, editor, *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 333–340, Nice, France, June 1990. ACM Press.

[Cur90] P.-L. Curien. The $\lambda\rho$-calculus: An abstract framework for environment machines. Technical report, LIENS–CNRS, 1990.

[DW90] K. Davis and P. L. Wadler. Strictness analysis: Proved and improved. In *Glasgow functional programming workshop*, August 90.

[Dyb87] P. Dybjer. Inverse image analysis. In T. Ottmann, editor, *Automata, Languages and Programming, 14th International Colloquium*, volume 267 of *Lecture Notes in Computer Science*, pages 21–30, Karlsruhe, Germany, 13–17 July 1987. Springer-Verlag.

[FB92] S. Finne and G. L. Burn. Assessing the evaluation transformer model of reduction on the spineless G-machine. Research Report DoC 92/36, Imperial College, London, UK, 1992.

[FW86] J. Fairbairn and S. C. Wray. Code generation techniques for functional languages. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 94–104. ACM, ACM, August 1986.

[Gol87] B. Goldberg. Detecting sharing of partial applications in functional programs. In G. Kahn, editor, *FPCA'87, Portland, Oregon*, volume 274 of *LNCS*, pages 408–425. Springer-Verlag, September 1987.

[GPP96] L. A. Galán, C. Pareja, and R. Peña. Functional Skeletons Generate Process Topologies in Eden. In *Int. Symp. on Programming Languages, Implementations, Logics and Programs PLILP'96. Aachen, Germany. LNCS Springer-Verlag no. 1140*, pages 289–303, 1996.

[HB85] P. Hudak and A. Bloss. The aggregate update problem in functional programming languages. In *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 300–314. ACM, January 1985. Presents both static and dynamic techniques for avoiding the copying that is ostensibly required by purely functional aggregate update.

[HH92] C. Hankin and S. Hunt. **Approximate Fixed Points in Abstract Interpretation**. In B. Krieg-Brückner, editor, *ESOP '92, 4th European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 219–232, Rennes, France, February 26–28, 1992. Springer, Berlin.

[HL92] R. J. M. Hughes and J. Launchbury. Reversing Abstract Interpretations. In *ESOP'92*, volume 582 of *LNCS*, pages 269–286. Springer-Verlag, 1992.

[HL94a] C. Hankin and D. Le Metayer. Lazy type inference for the strictness analysis of lists. *Lecture Notes in Computer Science*, 788, 1994.

[HL94b] C. Hankin and D. Le Metayer. **A Type-based Framework for Program Analysis**. *Lecture Notes in Computer Science*, 864, 1994.

[HM94] C. Hankin and D. Le Metayer. Deriving algorithms from type inference systems: application to strictness analysis. In ACM, editor, *Proceedings of 21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 202–212, New York, NY, USA, 1994. ACM Press.

[HS91] S. Hunt and D. Sands. Binding time analysis: A new PERspective. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, September 91.

[Hud86] P. Hudak. A Semantic Model of Reference Counting and its Abstraction (Detailed Summary). In *ACM Symposium on Lisp and Functional Programming*, pages 351–363. ACM, 1986.

[Hug86] R. J. M. Hughes. Strictness detection in non-flat domains. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, volume 217 of *LNCS*, pages 112–135. Springer-Verlag, October 1986.

[Hug87] R. J. M. Hughes. Analysing strictness by abstract interpretation of continuations. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 4. Ellis Horwood, 1987.

[Hug88] R. J. M. Hughes. **Backwards Analysis of Functional Programs**. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 187–208. North-Holland, Amsterdam, NL, 1988. Also Report CSC/87/R3, Department of Computing Science, University of Glasgow (1987).

[Hug89] R. J. M. Hughes. **Abstract interpretation of first-order polymorphically typed languages**. In Cordelia Hall, John Hughes, and John T O'Donnell, editors, *1988 Glasgow Workshop on Functional Programming*, volume 89/R4 of *Research Report*, pages 68–86. Glasgow Univ., February 1989.

[Hug90] R. J. M. Hughes. Why functional programming matters. In D. A. Turner, editor, *Research Topics in Functional Programming*, UT Year of Programming Series, chapter 2, pages 17–42. Addison-Wesley, 1990.

[Hun91] S. Hunt. **PERs Generalise Projections for Strictness Analysis**. In S. L. Peyton Jones, G. Hutton, and C. K. Holst, editors, *Functional Programming, Glasgow 1990*, pages 114–125. Springer-Verlag, London, UK, 1991.

[Hun91] S. Hunt. ***Abstract Interpretation of Functional Languages: From Theory to Practice***. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, October 91.

[HW87] C. V. Hall and D. S. Wise. Compiling strictness into stream. In *Conference record of the 14th ACM Symposium on Principles of Programming Languages (POPL)*, pages 132–143, 1987.

[HY85] P. Hudak and J. Young. A set-theoretic characterisation of function strictness in the lambda calculus. Tech. Rep. YALEU/DCS/RR-391, Yale Univ., 1985.

[HY86] P. Hudak and J. Young. Higher-order strictness analysis in untyped lambda calculus. In *Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, FL*, pages 97–109. ACM, New York, NY, 1986.

[JC87] S. L. Peyton Jones and C. Clack. Finding fixpoints in abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 11, pages 246–265. Ellis-Horwood, 1987.

[Jen91] T. P. Jensen. **Strictness Analysis in Logical Form**. In R. J. M. Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 352–366. Springer-Verlag, New York, NY, 1991. Lecture Notes in Computer Science 523.

[Jen92] T. P. Jensen. *Abstract Interpretation in Logical Form.* PhD thesis, Imperial College, University of London, November 1992. Available as DIKU Report 93/11 from DIKU, University of Copenhagen.

[JM86] N. D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, FL*, pages 296–306. ACM, 1986.

[JM90] T. P. Jensen and T. Mogensen. A backwards analysis for compile-time garbage collection. In *ESOP '90, Copenhagen, Denmark (Lecture Notes in Computer Science, vol. 432)*, pages 227–239. Springer-Verlag LNCS 432, 1990.

[Joh81] T. Johnsson. *Detecting when Call-by-Value Can be Used Instead of Call-by-Need.* Programming Methodology Group, Institutionen for Informationsbehandling, Chalmers Tekniska Høgskola, Göteborg, SE, 1981.

[Kam92] S. Kamin. **Head-strictness is not a monotonic abstract property**. *Information Processing Letters*, 41(4):195–198, March 1992.

[KM89] T-M. Kuo and P. Mishra. **Strictness analysis: a new perspective based on type inference**. In *FPCA'89, London, England*, pages 260–272. ACM Press, September 1989.

[KOMP98] U. Klusik, Y. Ortega-Mallén, and R. Peña. Implementing Eden - or: Dreams Become Reality. In *Implementation of Functional Languages, IFL'98, London, Sept. 1998. To appear in LNCS, Springer-Verlag*, pages 1–16, 1998.

[Lan63] P. J. Landin. The mechanical evaluation of expressions. *Computer Jrnl.*, 6(4), 1963.

[Mac71] S. MacLane. *Categories for Working Mathematicians*. Springer-Verlag, New York, 1971.

[MH87] J. C. Martin and C. Hankin. **Finding Fixed Points in Finite Lattices**. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 426–445. Springer-Verlag, Berlin, DE, 1987. Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, OR.

[Mos94] C. Mossin. Polymorphic binding-time analysis. Master's thesis, DIKU, University of Copenhagen, Denmark, July 1994.

[Myc80] A. Mycroft. **The Theory and Practice of Transforming Call-by-Need into Call-by-Value**. In B. Robinet, editor, *Proceedings of the 4th International Symposium on Programming/Colloque International sur la Programmation*, volume 83 of *LNCS*, pages 269–281, Paris, France, April 1980. Springer.

[Myc81] A. Mycroft. **Abstract interpretation and optimising transformations for applicative programs**. Cst-15-81 (phd thesis), Dept Computer Science, University of Edinburgh, December 1981. (LA has).

[NN92] F. Nielson and H. R. Nielson. The Tensor Product in Wadler's Analysis of Lists. In *ESOP'92*, volume 582 of *LNCS*, pages 351–370. Springer-Verlag, 1992.

[Pie91] B. C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, Cambridge, Mass., 1991.

[PP93] S. L. Peyton Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In J. T. O'Donnell, editor, *Glasgow Workshop on Functional Programming 1993*, Lecture Notes in Computer Science. Springer-Verlag, 5–7 July 1993.

[PvE93] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[Rea89] C. Reade. *Elements of Functional Programming*. International Computer Science Series. Addison-Wesley, Wokingham, England, 1989.

[SA89] P. Sestoft and G. Argo. Detecting Unshared Expressions in the Improved Three Instructions Machine. Tech. Rep., Glasgow Univ., 1989.

[San95] A. Santos. *Compilation by Transformation in Non-Strict Functional Languages.* PhD thesis, Glasgow University, Department of Computing Science, 1995.

[Sch85] D. A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[Ses88] P. Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, October 1988. 107 pages.

[Ses89] P. Sestoft. Replacing function parameters by global variables. In *FPCA'89, London, England.* ACM Press, September 1989.

[Ses91] P. Sestoft. ***Analysis and efficient implementation of functional programs***. PhD thesis, DIKU, October 1991.

[Shi88] O. Shivers. Control-flow analysis in Scheme. *ACM SIGPLAN Notices*, 23(7):164–174, July 1988. Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation.

[TWM95] D. N. Turner, P. L. Wadler, and C. Mossin. Once Upon a Type. In *1995 Conf. on Functional Programming and Computer Architecture*, pages 1–11, 1995.

[Wad87] P. L. Wadler. **Strictness analysis on non-flat domains (by abstract interpretation)**. In S Abramsky and C Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12, pages 266–275. Ellis-Horwood, 1987.

[WF89] S. C. Wray and J. Fairbairn. Non-strict languages - programming and implementation. *The Computer Journal*, 32(2), 1989.

[WH87] P. L. Wadler and R. J. M. Hughes. **Projections for Strictness Analysis**. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 385–407. Springer-Verlag, Berlin, DE, 1987. Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, OR.

[Wra85] S. C. Wray. A new strictness detection algorithm. In *Aspenäs Workshop on Implementation of Functional Languages, Göteborg*, 1985.