Análisis de programas en lenguajes funcionales paralelos

TESIS DOCTORAL

Clara María Segura Díaz

Departamento de Sistemas Informáticos y Programación
Facultad de Ciencias Matemáticas
Universidad Complutense de Madrid
Octubre 2001

Análisis de programas en lenguajes funcionales paralelos

Memoria presentada para obtener el grado de
Doctor en Matemáticas

Clara María Segura Díaz

Dirigida por el profesor
Ricardo Peña Marí

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

Octubre 2001

Agradecimientos

Quisiera comenzar esta tesis mostrando mi agradecimiento a mi director Ricardo Peña, sin el cual no hubiera sido posible la realización de este trabajo. Su gran experiencia ha sido de inestimable ayuda en la orientación y desarrollo de mi investigación. Le agradezco también el tiempo y atención que me ha dedicado, sobre todo teniendo en cuenta lo apretado de su agenda.

He de agradecer a todos los miembros que pertenecen o han pertenecido al grupo español de Edén su interés en mi trabajo y su colaboración a todos los niveles: Alberto de la Encina, Luis Antonio Galán, Félix Hernández, Mercedes Hidalgo, Manuel Núñez, Yolanda Ortega, Pedro Palao, Cristóbal Pareja, Ricardo Peña y Fernando Rubio. Quisiera destacar a Fernando Rubio que siempre ha respondido a mis preguntas sobre la implementación de Edén; a Cristóbal Pareja, con quien estudié de forma exhaustiva cómo afectan las transformaciones de GHC a la eficiencia y la semántica de Edén; y a Alberto de la Encina que siempre ha estado dispuesto a echar una mano en todo. Y por supuesto a los miembros del grupo alemán de Edén en Marburg, Silvia Breitinger, Ulrike Klusik, Rita Loogen y Steffen Priebe, cuyo amplio trabajo en Edén ha sido una referencia constante en mi investigación.

No puedo dejar de mencionar al resto de los "pezqueñines" del departamento con quien comparto todos los días de trabajo, y también algunos de diversión: Olga Marroquín y Natalia López. Sin olvidar a mi compañero de despacho Alberto Verdejo que siempre me ha escuchado y me ha ofrecido su ayuda y opinión cuando la he necesitado.

Y finalmente, pero no por ello menos importante, he de agradecer su apoyo a todas las personas que forman parte de mi vida fuera de la Universidad. A mis padres, en quienes he encontrado un respaldo a mi trabajo a lo largo de toda mi vida. A todos mis amigos, en especial a Conchi y a Agustín, con quienes he compartido mis años de carrera y los posteriores, y cuya amistad ha sido siempre un punto de apoyo insustituible. Y a mi novio David, que es indiscutiblemente la persona más importante, la que ha estado presente en todo momento y de quien siempre he recibido un apoyo incondicional en todos aquellos objetivos que me he propuesto.

Índice General

Capítulo 1

Introducción

Las técnicas de análisis estático de programas se han demostrado útiles en todo tipo de lenguajes para determinar de forma estática aproximaciones a las propiedades dinámicas de los programas. Sus aplicaciones incluyen la optimización y transformación de los programas, así como la demostración o verificación de propiedades de los mismos. El desarrollo y aplicación de las técnicas de análisis a los lenguajes funcionales ha sido muy amplio en las últimas dos décadas. Esta tesis se enmarca en el área de análisis estáticos de programas aplicados a los lenguajes funcionales paralelos. La programación funcional paralela es un área de investigación intensa en la actualidad que proporciona nuevos problemas en el campo de los análisis de programas, pero es aún un área poco trabajada donde queda mucho por hacer. En esta tesis se definen tres análisis originales aplicados al lenguaje funcional paralelo Edén (ver Capítulo 3): análisis de conexión directa, análisis de no determinismo, y análisis de terminación y productividad. El primero tiene como objetivo optimizar los programas Edén reduciendo sobrecargas debidas a la creación de procesos y a la comunicación entre ellos. El segundo pretende determinar aquellas partes de los programas Edén donde es posible mantener el razonamiento ecuacional característico de los lenguajes funcionales. El tercero es un sistema de demostración de propiedades de terminación y productividad de los programas Edén que garantiza la ausencia de bloqueos en ellos. Para definir estos análisis se han utilizado algunas de las técnicas usadas en el análisis de los lenguajes funcionales: la interpretación abstracta y los sistemas de tipos anotados. En este capítulo introductorio se describe con más detalle el marco que rodea a esta tesis situando en él al lenguaje Edén sobre el que se aplican los análisis definidos. Después se presentan los objetivos de la tesis y sus aportaciones. Finalmente se proporciona la estructura detallada de los capítulos.

La programación funcional

La programación funcional es un paradigma con un desarrollo espectacular durante la década de los 80, dando lugar a una familia de lenguajes con características muy diferentes a las de los primeros lenguajes funcionales basados en Lisp. Posee una serie de ventajas [Hug90] de las que otro tipo de paradigmas carece. Por ejemplo, la ausencia de efectos laterales permite razonar con mayor facilidad sobre los programas escritos en ellos. Además, los lenguajes funcionales actuales poseen disciplina de tipos, polimorfismo, tipos de datos recursivos que permiten definir estructuras de datos sofisticadas, ajuste de patrones para definir funciones sobre dichas estructuras, listas intensionales y otras características, como el orden superior, que proporcionan mayor modularidad y un nivel de abstracción elevado de programación. Las dos características fundamentales que los distinguen de otro tipo de lenguajes son el orden superior y el polimorfismo, los cuales colaboran en una mayor reutilización del software.

La presencia de pereza en la mayoría de los lenguajes funcionales actuales permite además la utilización de estructuras de datos infinitas sin provocar necesariamente la no terminación de los programas, como sucede en los lenguajes funcionales estrictos.

Con el fin de difundir el uso de la programación funcional se llevó a cabo un esfuerzo de estandarización cuyo resultado fue el lenguaje Haskell [HW88]. Se trata de un lenguaje funcional puro, es decir, sin efectos laterales, que incorpora un sistema sofisticado de tipos, un sistema de módulos con compilación separada y un mecanismo de entrada/salida que respeta la pureza del lenguaje. Además dispone de interfaces con otros lenguajes (como C) o librerías. En la página oficial de Haskell `http://haskell.org` se puede encontrar toda la información sobre este lenguaje: definición, librerías, herramientas, intérpretes, compiladores, etc. Hasta el momento se dispone de un intérprete, *Hugs*, y de varios compiladores de Haskell: el compilador GHC (*Glasgow Haskell Compiler*) [PHH$^+$93] es el mejor hasta la fecha. Otros son el *NHC* (*Nearly Haskell Compiler*) desarrollado en York y el *HBC* (*Chalmers' Haskell-B*) desarrollado en Chalmers.

La ventaja principal de utilizar un lenguaje funcional es la rapidez de desarrollo de los programas. 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. Por ello, se ha invertido un gran esfuerzo en la inclusión de análisis en el compilador para mejorar la eficiencia del código.

A pesar de dichas pegas, la programación funcional ha sido objeto de interés de algunas industrias, como Ericsson, que ha utilizado el lenguaje funcional Erlang [JLAV93] en sus sistemas telefónicos. También es un área en la que se desarrolla una intensa investigación en todo el mundo. En la página http://www.cs.ukc.ac.uk/people/staff/cr3/FP/ se pueden encontrar punteros a grupos de investigación (unos 44 distribuidos por todo el mundo), lenguajes (Haskell, Clean, Lisp, Scheme, ML), congresos (entre 20 y 30 en los últimos 8 años), publicaciones, proyectos y todo tipo de herramientas relacionadas con la programación funcional (compiladores e intérpretes, interfaces con otros lenguajes, herramientas gráficas, perfiladores y bancos de pruebas, trazadores y depuradores).

La programación funcional paralela

Con el objetivo de paliar la ineficiencia de los lenguajes declarativos, lógicos y funcionales, desde mediados de los años 80 se ha intentado integrar el paralelismo en estos lenguajes. El objetivo principal de la programación paralela es la utilización eficiente de máquinas con varios procesadores para acelerar los algoritmos en un factor idealmente igual al número de procesadores empleados. Los lenguajes declarativos en general y los lenguajes funcionales en particular son especialmente aptos para la incorporación de paralelismo implícito, es decir, paralelismo transparente al programador. Debido a la ausencia de efectos laterales, el orden de evaluación de las subexpresiones de las expresiones funcionales y de las cláusulas lógicas es flexible, por lo que en muchas ocasiones se pueden evaluar en paralelo sin afectar al resultado final. Por ejemplo, en el caso de los lenguajes funcionales el argumento de una función puede evaluarse en paralelo con la llamada a la función y con la evaluación de otros argumentos.

Los lenguajes lógicos fueron pioneros en esta integración con el desarrollo de lenguajes como *Concurrent Prolog* [Sha83] y PARLOG [CG83]. En estos lenguajes se pueden distinguir dos tipos de paralelismo: el paralelismo-Y y el paralelismo-O. El primero consiste en la ejecución simultánea de varios subobjetivos de una cláusula. El segundo está relacionado con el no determinismo inherente a los lenguajes lógicos: si un objetivo unifica con varias cláusulas se prueban todas ellas en paralelo. Mientras que el paralelismo-Y investiga las subpartes de una solución particular en paralelo, el paralelismo-O investiga muchas posibles soluciones en paralelo.

La incorporación del paralelismo en los lenguajes funcionales fue más tardía. Las primeras ideas se deben a Hudak [Hud86a], y FACILE [GMP89] es uno de los primeros lenguajes funcionales que incorporan paralelismo. Un programa funcional admite muchas secuencias posibles de ejecución, todas ellas conducentes al mismo resultado.

Ahora bien, tanto en los lenguajes lógicos como en los funcionales es necesario respetar las dependencias establecidas por los datos, lo que requiere la sincronización adecuada.

Sin embargo, los intentos de explotación del paralelismo implícito no proporcionan beneficios significativos cuando se dejan completamente en manos del compilador. La creación de procesos y la comunicación entre ellos suponen sobrecargas en tiempo que deben ser compensadas por el ahorro obtenido por la paralelización de la evaluación de las subexpresiones. Esto supone que sólo vale la pena ejecutar en paralelo tareas de tamaño razonable. El problema de establecer el "tamaño razonable" recibe el nombre de *problema de granularidad* [Loi98]. Lamentablemente, el compilador carece de criterios para decidirlo. El problema se agudiza en los lenguajes funcionales perezosos. En presencia de la pereza se retarda la evaluación de un argumento de una función hasta que se necesita. Por tanto, la creación de procesos para evaluar argumentos en paralelo con la función supone un riesgo de creación de procesos especulativos cuya evaluación no era necesaria, con el consiguiente desperdicio de recursos. En consecuencia, es necesario obtener una garantía de que para la obtención del resultado final la evaluación es efectivamente requerida .

Estas dificultades han conducido a los investigadores a involucrar al programador en la decisión de cuándo han de crearse las tareas paralelas. El grado de implicación del programador puede ser muy variable, desde el uso de simples anotaciones de paralelismo hasta el uso de nuevas construcciones de paralelismo explícitas. Esto ha dado lugar a un amplio espectro de lenguajes funcionales paralelos.

Una forma de indicar cuándo es rentable la creación de tareas en paralelo son las *anotaciones* de paralelismo. En esta línea se enmarcan los lenguajes *Caliban* [Kel89], *Concurrent Clean* [PvE93] y GpH (*Glasgow parallel Haskell*) [THM$^+$96]. Los tres son lenguajes funcionales puros. El primero de ellos sólo permite crear redes estáticas de procesos, es decir, redes cuyo tamaño y configuración se establecen en tiempo de compilación. En el segundo, no existe una noción explícita de proceso y la comunicación/sincronización entre las distintas *hebras* concurrentes es implícita y transparente al programador. El tercero ha cosechado buenos resultados en los últimos años. Está basado en el compilador GHC. GpH extiende Haskell mediante un operador primitivo `par` con el que el programador anota las expresiones que él piensa que pueden ser evaluadas en paralelo. Sin embargo, es en tiempo de ejecución cuando se decide qué expresiones, de entre aquéllas que han sido anotadas, se realizan en paralelo y por tanto la topología que se crea. Se mantiene la semántica perezosa de Haskell pero es posible definir *estrategias* que permiten controlar el grado de evaluación de las expresiones de forma que el programador puede introducir impaciencia allí donde considere necesario para incrementar el paralelismo de los programas. GpH ha destacado por haber sido utilizado para paralelizar programas de gran tamaño, como Naira [Jun98] (un compilador de 5.000 líneas de código fuente) y Lolita [LMT$^+$97] (una herramienta de procesamiento de lenguaje natural de 47.000 líneas de código). Asímismo, se ha desarrollado GranSim [HLP95, Loi96], un simulador de GpH que genera gráficas que permiten visualizar de forma sencilla el comportamiento paralelo de los programas escritos en GpH.

Un paso más allá consiste en introducir nuevas construcciones en los lenguajes para expresar la creación y comunicación entre los procesos. Este tipo de paralelismo recibe el nombre de *paralelismo explícito.* En esta línea se encuentran los lenguajes *Concurrent ML* [Rep91], *Erlang* [JLAV93] y *Concurrent Haskell* [PGF96]. Puesto que las construcciones concurrentes provocan efectos laterales, en este tipo de lenguajes las ventajas de la programación funcional son solamente aplicables a la parte no concurrente de los programas.

Otro problema de los lenguajes funcionales paralelos es la distribución del grafo que representa la expresión en evaluación. No resulta evidente cómo dividir el grafo en partes con pocas interconexiones. En GpH el grafo se va distribuyendo entre los procesadores a medida que se crean nuevos procesos pero nunca se duplican nodos, es decir, el grafo sigue existiendo como una única entidad aunque sus nodos estén distribuidos. Esto implica la existencia de punteros de unos procesadores a otros y la multiplicación de las comunicaciones, cuando se desea acceder repetidamente a una estructura remota.

Para una descripción exhaustiva de los distintos enfoques seguidos en programación funcional paralela en los últimos años y de las líneas de investigación actuales se puede consultar [HM99].

El lenguaje Edén

A partir de 1996 investigadores de la Universidad Complutense y de la Universidad de Marburg (Alemania) han venido desarrollando el lenguaje funcional paralelo Edén [BLOMP96]. Edén extiende el lenguaje funcional Haskell con expresiones que permiten definir al programador sistemas de procesos. Está implementado reutilizando el código fuente de GHC. A continuación se describen las características de Edén que permiten enmarcarlo dentro de la categoría de lenguajes funcionales paralelos.

A diferencia de otros lenguajes, puede expresar a la vez programas **paralelos** y **reactivos** por medio de un número muy reducido de construcciones nuevas con las que el programador indica qué expresiones han de ejecutarse en paralelo. Como consecuencia programar en Edén es bastante parecido a programar en Haskell.

La creación de procesos es explícita. El programador define esquemas genéricos de proceso llamados *abstracciones de procesos* y decide cuándo han de crearse ejemplares a partir de los mismos. Estas abstracciones son semejantes a funciones y, como ellas, son objetos de primera clase. La diferencia es que cuando una abstracción de proceso se aplica a unos datos, se da lugar a un nuevo proceso que se ejecuta en paralelo con el resto del programa. A diferencia de GpH, los lanzamientos de procesos son obligatorios para el sistema en tiempo de ejecución.

La comunicación es **implícita** y **asíncrona**. No existen por tanto, como en otros lenguajes funcionales concurrentes, primitivas para enviar mensajes ni sincronización explícita . Para establecer la comunicación basta con conectar la salida de un proceso a la entrada de otro .

Al crear un nuevo proceso se copia en la memoria local del procesador la parte del grafo que necesita para su ejecución. Esta comunicación se realiza de una sola vez y antes de comenzar la ejecución del proceso. Una vez en ejecución, los procesos solamente recibirán por sus canales de entrada valores completamente evaluados . Esta estrategia puede dar lugar a la duplicación de nodos aún no evaluados y por tanto a la duplicación de trabajo en varios procesadores. Sin embargo, reduce mucho las comunicaciones.

Edén se puede ejecutar en multicomputadores pero también en multiprocesadores con memoria compartida.

Edén es **no determinista**, al contrario que por ejemplo GpH. El no determinismo se introduce por medio del proceso reactivo `merge` que recibe una lista de listas y copia en la lista de salida los valores de las listas de entrada, a medida que están disponibles. Por tanto, el no determinismo es consecuencia de su reactividad y refleja las velocidades relativas de los procesos productores de las listas de entrada.

En Edén se pueden definir **topologías dinámicas** en las que el número de procesos no está determinado a priori, al contrario que en Caliban. El hecho de que las abstracciones de procesos sean valores de primera clase permite además definir las topologías de forma genérica como funciones de orden superior. Estas funciones reciben el nombre de *esqueletos*.

Edén hereda de Haskell su semántica perezosa, lo cual tiene ventajas a la hora de crear topologías circulares de procesos, ya que éstos pueden trabajar en paralelo sobre listas parcialmente producidas. Sin embargo, la semántica perezosa tiene un inconveniente: no exige evaluar completamente las expresiones (hasta forma normal) sino solamente hasta una forma intermedia llamada *forma normal débil*. Esto hace que la granularidad de los procesos paralelos sea muy baja, ya que a éstos no se les permite evaluar más allá de la forma normal débil. Por ello, en Edén se altera la semántica perezosa en dos puntos críticos: el lanzamiento de procesos y la producción de la salida de un proceso. En ambos casos las expresiones se evalúan aunque no haya demanda de sus valores. De esta forma se incrementa la granularidad de los procesos y el grado de paralelismo de los programas.

Análisis estático de programas funcionales

Las técnicas de análisis estático de programas se han demostrado útiles en una gran variedad de lenguajes y para resolver diversos problemas. En concreto y al hilo del tipo de lenguajes que estamos describiendo, pueden utilizarse para mejorar la eficiencia del código generado por los compiladores de los lenguajes funcionales; para resolver algunos de los problemas de los lenguajes funcionales paralelos que se han mencionado, como el problema de la granularidad y del lanzamiento de procesos inútiles; y también, para demostrar propiedades de los programas reactivos, como la terminación y la productividad.

Una forma de mejorar la eficiencia de los lenguajes funcionales es la transformación de programas con el objetivo de reducir su tiempo de ejecución o su consumo de espacio de almacenamiento.

Muchas transformaciones requieren el conocimiento previo de cierta 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 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.

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. En el segundo caso el análisis de programas pretende identificar situaciones de usos restringidos de determinadas construcciones que permitan una implementación menos general, pero más eficiente, de las mismas. El Capítulo 2 realiza un amplio resumen de los logros alcanzados en este área.

En este ámbito 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 que permitan llevar a cabo estas transformaciones. En las últimas dos décadas se han desarrollado numerosos análisis sobre lenguajes funcionales utilizando diversas técnicas. Dichas técnicas pueden dividirse en tres grandes grupos: análisis basados en el flujo, análisis basados en la semántica (interpretación abstracta) y análisis basados en tipos. Se trata de técnicas generales que se han adaptado a las características particulares de los lenguajes funcionales. La más ampliamente utilizada durante la década de los 80 fue la interpretación abstracta, mientras que en la actualidad los análisis basados en tipos se encuentran en pleno auge.

Sin embargo, la aplicación de estas técnicas al análisis de lenguajes funcionales paralelos es aún escaso, aunque puede ofrecer muchas oportunidades tanto para mejorar el comportamiento paralelo de los programas como para estudiar propiedades de los mismos.

El problema por excelencia es el problema de la granularidad mencionado anteriormente. Un análisis de la granularidad que estime de forma automática el tamaño y coste de las expresiones [Loi98] puede servir como base de decisión para paralelizar los programas.

Otro problema es el riesgo de creación de procesos inútiles cuando el lenguaje es perezoso. Antes de crear un proceso es necesaria una garantía de que el valor de la expresión que va a evaluar dicho proceso será efectivamente demandado. Dicha garantía puede ser proporcionada por un análisis, llamado análisis de estrictez, que ya ha sido estudiado exhaustivamente [Myc81, BHA86].

En relación con los lenguajes funcionales empotrados en sistemas reactivos, se han estudiado análisis para determinar propiedades como la terminación, la productividad [HPS96] y el uso acotado de memoria [HP99].

Objetivos y contribuciones de la tesis

Esta tesis pretende no solamente resolver los problemas concretos que surgen en la programación paralela con el lenguaje Edén, sino también ser una aportación al campo de los análisis de programas funcionales paralelos y un ejemplo de la aplicación de las técnicas existentes a los mismos. Los objetivos fundamentales perseguidos con la realización de esta tesis son los siguientes:

- Desarrollar análisis que permitan un **aumento de la aceleración** de los programas funcionales paralelos, ya sea mejorando la distribución o el reparto de carga entre los procesadores o disminuyendo las sobrecargas debidas a la creación de hebras, envío de mensajes, duplicación de la evaluación, etc.

- Estudiar cómo **interfieren** las construcciones que expresan el paralelismo con las transformaciones ya existentes en los lenguajes funcionales.

- Desarrollar análisis para estudiar **propiedades** de los programas funcionales paralelos, como la ausencia de bloqueos.

Para cumplir estos objetivos se abordan tres análisis originales de programas, útiles en el ámbito de los lenguajes funcionales paralelos, y en concreto del lenguaje funcional paralelo Edén.

Análisis de conexión directa. Cuando un proceso Edén crea otros procesos hijo, se comunica con ellos mediante canales. Sin embargo, en muchos casos el padre no interviene posteriormente en las comunicaciones de sus hijos con otros procesos, más que como un mero intermediario que copia los valores que recibe en una entrada a una salida. Esta situación puede repetirse en varios procesos, estableciéndose cadenas de intermediarios. En estos casos es más eficiente conectar directamente a los productores y consumidores de los valores, evitando así la creación de hebras inútiles en el proceso intermediario y el envío de mensajes inútiles de los productores a los intermediarios y de los intermediarios a los consumidores. Llamamos a esta optimización conexión directa (*bypassing*). La solución proporcionada en esta tesis a este problema consiste en la combinación de un análisis en tiempo de compilación seguida de una modificación en el protocolo de comunicación entre procesos. El análisis pretende detectar hebras que solamente copian un valor de entrada a un canal de salida, lo cual tiene una traducción simple en términos sintácticos: encontrar aquellas variables que se usan exactamente una vez como entrada y exactamente una vez como salida de un proceso. La modificación del protocolo de comunicación implica la reducción del número de mensajes enviados en tiempo de ejecución y por lo tanto la reducción de la sobrecarga debida al paralelismo.

Análisis de no determinismo. Algunas de las transformaciones automáticas realizadas por el compilador GHC para Haskell interaccionan de forma negativa con las construcciones paralelas de Edén. Unas afectan a la eficiencia y otras a la semántica de los programas. En esta tesis se hace un estudio de dichas transformaciones. En particular, la presencia de no determinismo invalida algunas de estas transformaciones y adicionalmente afecta a la transparencia referencial de los programas. Una opción posible para resolver este problema sería desactivar estas transformaciones en todos los casos; pero puesto que sus beneficios han sido ampliamente probados, en su lugar se opta por definir un análisis para detectar las porciones de texto que siguen conservando una semántica funcional pura y aquellas que podrían haber perdido tal característica debido al uso directo o indirecto de procesos no deterministas. De esta forma solamente será necesario desactivar las transformaciones conflictivas en las partes afectadas por el no determinismo. En esta tesis se proporcionan tres análisis de no determinismo con distintos grados de eficiencia y potencia, se comparan entre sí y se presenta la implementación de uno de ellos.

Análisis de terminación y productividad. Las extensiones de Edén permiten la definición de esqueletos de una forma sencilla como funciones de orden superior. Sin embargo, el programador puede introducir inadvertidamente bucles infinitos o bloqueos. Con el objetivo de demostrar la ausencia de este comportamiento indeseado en los esqueletos Edén, se presenta en esta tesis una extensión del análisis de terminación y productividad desarrollado en [HPS96, Par00]. Como ilustración, se demuestran estas propiedades para algunos esqueletos Edén.

Estructura de la tesis

A continuación se presenta un resumen del contenido de cada uno de los capítulos:

Capítulo 2 En este capítulo se presenta un resumen de las técnicas utilizadas en el análisis estático de programas en general, y de su aplicación a los lenguajes funcionales, en particular. En primer lugar, se demuestra la utilidad de los análisis enumerando algunos de los más conocidos en lenguajes imperativos y declarativos, y se describen brevemente las tres técnicas de análisis básicas: análisis basados en el flujo, interpretación abstracta y análisis basados en tipos. A continuación, y tras un recorrido histórico por el desarrollo de los análisis en lenguajes funcionales, se describen con más detalle la interpretación abstracta y los análisis basados en tipos. Se proporcionan primero los conceptos generales de estas técnicas y después las características particulares que resultan de la aplicación de ambas técnicas a los lenguajes funcionales, utilizando algunos ejemplos para ilustrarlas. Cuando se considera necesario se proporcionan también algunos preliminares teóricos específicos, como las conexiones de Galois, la categoría de pares inmersión-clausura, los dominios potencia y los operadores de ensanchamiento y estrechamiento. En [Seg99] se describen con más detalle algunas de las técnicas aquí resumidas.

Capítulo 3 En este capítulo se describe el lenguaje funcional paralelo Edén. Se describen primero los fundamentos del lenguaje, su sintaxis y su semántica, esta última de manera informal. A continuación se describen algunos detalles de implementación, como el proceso de compilación del lenguaje y algunas características del funcionamiento del sistema en tiempo de ejecución que serán útiles en capítulos posteriores, como el protocolo de comunicación y de creación de procesos. En [BLOMP98] se puede encontrar una descripción detallada del lenguaje Edén y su semántica, y en [KOMP99] una descripción del sistema en tiempo de ejecución.

Capítulo 4 En este capítulo se presenta el análisis de conexión directa. En primer lugar se describe el análisis enmarcándolo en el proceso de compilación del lenguaje Edén. Posteriormente se describe cómo se modifica el protocolo de comunicación teniendo en cuenta la información producida por el análisis. El contenido de este capítulo se ha presentado en [PS98a, KPS00].

Capítulo 5 En este capítulo se presentan tres análisis de no determinismo. Primero se motiva la necesidad de un análisis de no determinismo presentando un resumen de transformaciones que suponen un riesgo para los programas Edén tanto desde el punto de vista de la eficiencia como de la semántica. En concreto, se describen con más detalle las que afectan al no determinismo. A continuación, se presentan los tres análisis. Se describe cada uno de ellos mostrando sus ventajas e inconvenientes y se estudian las relaciones entre ellos. El resultado de este desarrollo se plasma en la implementación de uno de los análisis.

Los resultados de este capítulo se han presentado en [PS01d, PS01a, PS01c, PS00b, PS01b, PPRS00a, PPRS00b, PS00a].

Capítulo 6 En este capítulo se presenta una extensión de un análisis de terminación y productividad para que pueda ser aplicado a los programas escritos en Edén. Se describen los problemas introducidos por las características propias de Edén, y las extensiones necesarias para resolver dichos problemas. A continuación se aplica el nuevo análisis a algunos programas Edén. El contenido de este capítulo se ha presentado en [PS01e].

La tesis concluye con el Capítulo 7 donde se resumen las principales contribuciones de esta tesis y se presentan posibles líneas futuras de trabajo. Adicionalmente se presentan los siguientes apéndices:

Apéndice A Contiene algunos conceptos básicos de teoría de dominios y teoría de categorías que pueden resultar útiles como recordatorio.

Apéndice B Se presenta el código Haskell del algoritmo que implementa uno de los análisis de no determinismo presentado en el Capítulo 5.

Apéndice C Se muestran los resultados de la aplicación del análisis implementado en el apéndice anterior a algunos programas ejemplo.

Capítulo 2

Técnicas de análisis de programas

Este capítulo proporciona una panorámica de las técnicas utilizadas en el análisis estático de programas en general, y de programas funcionales en particular. En una primera aproximación, se describe la utilidad de los análisis en los distintos tipos de lenguajes enumerando los análisis existentes más conocidos y sus aplicaciones, se resumen y comparan las distintas técnicas utilizadas y se proporciona una perspectiva histórica del desarrollo de los análisis para lenguajes funcionales. A continuación, se describen con más detalle la interpretación abstracta y los análisis basados en tipos, ya que no solamente son las técnicas más ampliamente utilizadas en el análisis de programas funcionales sino que en particular son las empleadas en la definición de los tres análisis tema de esta tesis.

En el trabajo de tercer ciclo [Seg99] se llevó a cabo un recorrido de las distintas técnicas de análisis de programas funcionales. Fue escrito con el objetivo de servir como guía para introducirse en este campo, por lo que allí se describen con más detalle la mayoría de las técnicas aquí mencionadas. Sin embargo, el enfoque que se proporcionó allí a los análisis basados en tipos difiere en cierta medida del adoptado aquí, ya que es en este tipo de análisis donde se han centrado los esfuerzos investigadores en los últimos tiempos, con lo que el estilo se ha modificado.

2.1 Panorámica general

2.1.1 Problemas que resuelven

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 de programas (por ejemplo paralelización, evaluacion parcial), optimizaciones del código (por ejemplo recolección de basura en tiempo de compilación, eliminación del *occur-check* innecesario), depuración (por ejemplo inferencia de tipos) e incluso pruebas de corrección (por ejemplo pruebas de terminación o de productividad).

Los análisis estáticos de programas se han utilizado en todo tipo de lenguajes. A continuación se enumeran algunos de los análisis más conocidos y la utilidad de cada uno de ellos, clasificados en base al tipo de lenguajes sobre los que se definen.

Lenguajes imperativos. Los análisis en lenguajes imperativos están principalmente orientados a estudiar el flujo de datos y de control a lo largo del programa con el objetivo de optimizar el código [UAH74, NNH99]. Se puede conseguir mejoras calculando constantes en tiempo de compilación, evitando recalcular expresiones, eliminando código muerto, trasladando código fuera de bucles o sustituyendo operaciones costosas por otras más baratas. Algunos ejemplos de análisis son los siguientes:

- *Análisis de propagación de constantes*: Encuentra aquellas expresiones de un programa que dan lugar a valores constantes, los cuales pueden calcularse en tiempo de compilación.

- *Análisis de expresiones disponibles*: Detecta para cada punto del programa qué expresiones han sido ya calculadas y no modificadas en cada uno de los caminos que llevan a dicho punto. Esta información puede utilizarse para evitar recalcular una expresión.

- *Análisis de definiciones previas (reaching definitions)*: Detecta para cada punto del programa qué asignaciones han podido llevarse a cabo sin ser modificadas al llegar a dicho punto a lo largo de algún camino. Una de las aplicaciones principales es la construcción de enlaces directos entre los bloques que producen valores y los que los utilizan. Los enlaces que a cada uso de una variable le asocian todas las asignaciones previas reciben el nombre de *cadenas de uso-definición*. También se pueden establecer enlaces tales que a cada asignación se le asocien todos sus usos. Estos reciben el nombre de *cadenas de definición-uso*. Ambos tipos de cadenas pueden utilizarse para eliminar código muerto (*dead code elimination*) o para trasladar código de un punto del programa a otro (*code motion*).

- *Análisis de expresiones "atareadas" (very busy expressions)*: Se dice que una expresión está atareada tras la ejecución de una instrucción si dicha expresión va a ser usada en cualquier camino posible antes de que alguna de las variables que aparecen en ella sea redefinida. Dicha expresión puede evaluarse inmediatamente y guardar su valor para su uso posterior, con lo que se ahorra código generado. Esta optimización recibe el nombre de *hoisting*.

- *Análisis de variables vivas*: Una variable está viva tras la ejecución de una instrucción si hay algún camino en el que se usa la variable sin redefinirla. Esta información se utiliza para eliminar código muerto: si una variable no está viva, se pueden eliminar las asignaciones a dicha variable.

- *Análisis de propagación de copias*: Detecta aquellos conjuntos de variables con un mismo valor. La transformación de propagación de copias sustituye todas las variables con el mismo valor por una de ellas. Esta transformación no produce una mejora inmediata, sino que genera la oportunidad de posteriores eliminaciones de variables muertas.

- *Análisis de variables invariantes*: Identifica las variables invariantes de un bucle, es decir, aquéllas a las que se les asigna el mismo valor en cada vuelta del bucle. El código correspondiente a estas variables puede trasladarse fuera del bucle, lo cual disminuye la cantidad de código en el bucle. Esta transformación se utiliza especialmente para optimizar algoritmos sobre matrices.

- *Análisis de variables de inducción*: Identifica variables de un bucle que varían regularmente en cada iteración. Cuando hay dos o más variables de inducción en un bucle puede llevarse a cabo la sustitución de operaciones costosas por otras más baratas, por ejemplo la sustitución de operaciones de multiplicación por otras de suma o sustracción. Esto recibe el nombre de *reducción de intensidad*. Por ejemplo, supongamos que en un bucle aparecen las instrucciones `j:=j-1; x:=4*j`. Cada vez que el valor de `j` disminuye en 1, el de `x` disminuye en 4, por lo que se podría sustitutir la asignación `x:=4*j` por `x:=x-4`.

Lenguajes lógicos. La unificación de dos términos con variables es una operación fundamental en este tipo de lenguajes, y resulta más costosa que la asignación imperativa o el encaje de patrones en los lenguajes funcionales. Una de las razones es que la unificación implica ligaduras en dos direcciones, es decir, las variables de cada uno de los términos a unificar pueden ligarse a partes del otro término. Por otra parte, hay que tener en cuenta el *occur-check*, esto es, la comprobación de que una variable no quede ligada a un término que la contiene, ya que los programas que contienen términos circulares no tienen interpretación lógica natural. Por lo tanto, algunos análisis en este tipo de lenguajes están orientados a acelerar la unificación, o a evitar de forma segura el *occur-check* [Deb86].

A continuación, se presentan algunos de los análisis más estudiados:

- *Análisis de modo*: Para un objetivo concreto, determina aquellas variables libres que están ligadas cuando se llama a dicho objetivo (variables de entrada) y cuáles quedarán ligadas como resultado de la llamada (variables de salida) [Mel86]. Esta información se utiliza para identificar aquellos puntos del programa en los que pueden utilizarse formas especiales más eficientes de unificación.

- *Análisis de cierre (groundness)*: Identifica aquellas variables que se ligan a un término cerrado (sin variables) al alcanzar un determinado punto del programa [JS87]. Este análisis es complejo debido a la existencia de alias y de subestructuras compartidas: ligar una variable a un término cerrado puede afectar a las variables de otra parte del programa que está siendo analizado.

- *Análisis de circularidad*: Busca aquellas unificaciones que se pueden realizar de forma segura sin llevar a cabo el costoso *occur-check* [Pla84, Søn86].

- *Análisis de dependencias*: Infiere dependencias entre los términos ligados a las variables del programa. Esta información puede utilizarse para optimizar el mecanismo de *backtracking* [CD85]. También puede usarse para predecir en tiempo de compilación la independencia de objetivos en tiempo de ejecución. Esto último permite eliminar costosas comprobaciones en tiempo de ejecución en aquellos sistemas que soportan paralelismo-Y [MH92].

- *Análisis de tipos*: Describe el tipo de los argumentos de un predicado [Mo84, FS91], es decir, el conjunto de términos a los que se liga un argumento de un predicado cuando se le llama o cuando tiene éxito.

Otros análisis determinan cuándo se pueden aplicar algunas transformaciones de forma segura, como por ejemplo la transformación de listas diferencia [MS89].

Lenguajes funcionales. 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 valor. Otras pretenden reducir el consumo de espacio de almacenamiento, como por ejemplo la globalización de parámetros. Algunos análisis llevados a cabo en el ámbito de la programación funcional son los presentados a continuación:

- *Análisis de estrictez*: En los lenguajes funcionales perezosos se utiliza el *paso de parámetros por necesidad*, por el cual un parámetro solamente se evalúa si se utiliza en el cuerpo de la función y, en tal caso, una vez se ha evaluado a forma normal débil el parámetro se actualiza con ese valor, por lo que sucesivos usos de ese parámetro no necesitan evaluarlo de nuevo. 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. El análisis de estrictez [Myc81, BHA86, WH87, Jen91] detecta aquellos parámetros que una función va a evaluar con seguridad. En tal caso se puede evitar la construcción de una clausura para dicho argumento y llevar a cabo su evaluación inmediata. Es decir, se sustituye el paso de parámetros por necesidad por el paso de parámetros por valor.

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 de uso o de compartición*: La compartición de la evaluación es crucial en los lenguajes funcionales perezosos. Esta se consigue mediante la construcción de clausuras que una vez evaluadas se actualizan, de forma que usos posteriores de dicha clausura comparten el valor ya calculado. El análisis de uso [Mar93, LGH$^+$92, TWM95, WJ99, GS01] detecta aquellas clausuras a las que se accede a lo sumo una vez. En tal caso se puede evitar el coste de la actualización de la clausura. 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.

- *Propagación de constantes*: En este análisis [JM86], 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.

- *Análisis de clausuras*: Muchos análisis se definen fácilmente para lenguajes de primer orden. Cuando se desea extenderlos a lenguajes con orden superior se hace necesario conocer, por un lado, a qué funciones están ligadas los parámetros funcionales y, por otro, a qué función se evalúa realmente la expresión que se aplica como función. El objetivo de este análisis [Shi88, Ses89, NN97] es 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 se puede aplicar una determinada lambda abstracción.

- *Análisis de globalización*: 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. En consecuencia, es interesante detectar automáticamente tales parámetros globales y sustituirlos por variables globales con asignación imperativa. Este análisis [Sch85, Ses89] 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 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 [Mog89, HS91, Mos94] 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.

- *Análisis de ausencia*: Mientras que en el análisis de estrictez deseamos saber si una función utiliza un parámetro, el análisis de ausencia [Wra85] 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.

- *Análisis de terminación*: Aunque el problema de parada nos dice que la terminación de los programas no es decidible, sí se puede definir un análisis [Myc81, NN96] que aproxime la respuesta.

- *Análisis de productividad*: Un programa es productivo si siempre que recibe una entrada infinita produce una salida infinita. Los análisis de productividad [Sij89] se basan en demostrar que las llamadas recursivas están guardadas, es decir, aparecen dentro de un constructor.

- *Análisis para la recuperación de espacio*: Este análisis [JM90] detecta cuándo ha sido usada por última vez una determinada celda de memoria, por lo que puede ser liberada para un nuevo uso. Esto reduce sustancialmente el número de recolecciones de basura.

- *Análisis de regiones*: Un aspecto positivo de la programación funcional es que las tareas relacionadas con la gestión de la memoria son llevadas a cabo por el compilador y el sistema de soporte en tiempo de ejecución. Sin embargo, los lenguajes funcionales usan mucha memoria, lo cual es difícil de arreglar por parte del programador. En [HP99, Par00] se desarrolló un lenguaje funcional estricto de primer orden con construcciones explícitas de almacenamiento en *regiones* de memoria llamado *Embedded ML*. Las regiones son montones (*heaps*) de corta vida introducidos y liberados por el programador, y en las que solamente se pueden almacenar tuplas o constructores. En el momento de su creación están vacías, se van llenando con valores a lo largo de su vida y se eliminan de una vez cuando son descartadas. El recolector de basura no actúa sobre ellas. En este lenguaje un análisis de regiones indica en el ámbito de qué regiones se encuentra una expresión y en cuál de ellas se almacenará su valor. Para cada región, determina la cantidad de palabras usadas en el cómputo de la expresión, la altura máxima alcanzada por la pila de valores y por la pila de regiones, y cuál es el tipo de la expresión. Esta información tiene como objetivo evitar que los programas se queden sin espacio en tiempo de ejecución.

Lenguajes lógico-funcionales. La programación lógico-funcional combina la búsqueda de los lenguajes lógicos con características de los lenguajes funcionales como la evaluación perezosa [Han94a]. Tienen una sintaxis funcional y usan el *estrechamiento* como semántica operacional. El estrechamiento es un mecanismo de resolución de objetivos basado en unificación que subsume al principio de reducción de los lenguajes funcionales y el principio de resolución de los lenguajes lógicos. El *estrechamiento necesario* es una estrategia óptima de estrechamiento. Algunos de los análisis desarrollados para estos lenguajes [Zar97, HL99] son variaciones de los ya definidos para lenguajes lógicos, como el análisis de modo, y para lenguajes funcionales, como el análisis de la demanda:

- *Análisis de modo*: Este análisis [HZ94] describe el estado de la vinculación de las variables de una llamada a una función cuando se aplica o bien un paso de estrechamiento o bien un paso de reescritura. Usando este tipo de información se pueden llevar a cabo varias optimizaciones [Han94b] como evitar intentos de reescritura innecesarios, compilar de forma más eficiente derivaciones de reescritura y eliminar reglas de reescritura o estrechamiento innecesarias.

- *Análisis de argumentos redundantes*: La aplicación de transformaciones automáticas a los programas lógico-funcionales puede introducir argumentos redundantes en las funciones, es decir, argumentos cuyos valores son irrelevantes para la evaluación de la función. Este análisis [AEL00] pretende detectar dicho tipo de argumentos con el objetivo de eliminarlos.

- *Análisis de la demanda*: En una implementación eficiente del estrechamiento perezoso es crucial evaluar tan pronto como sea posible, ya que, en caso contrario, los argumentos podrían recalcularse muchas veces. Este análisis [JMMMN93, MNKM$^+$93] detecta qué partes de los argumentos se pueden evaluar de forma segura antes de la llamada a la función. Está relacionado con el análisis de estrictez de los lenguajes funcionales.

- *Análisis de no determinismo*: Una característica esencial de los lenguajes lógico-funcionales es la posibilidad de manejar cómputos no deterministas para calcular soluciones de objetivos parcialmente sustituidos. Sin embargo, su combinación con la entrada/salida de los programas puede dar lugar a problemas. El objetivo de este análisis [HS00] es detectar todas las posibles fuentes de no determinismo. Una vez identificadas se puede transformar el programa para que todos los cómputos no deterministas queden encapsulados, incrementando así la estabilidad y seguridad de los programas.

- *Análisis de residuación*: El principio de residuación pospone la evaluación de las funciones durante el proceso de unificación hasta que los argumentos están lo suficientemente concretados. De esta forma se preserva la naturaleza determinista de las funciones. Sin embargo, si las variables de una función cuya evaluación se ha retrasado no son concretadas por la parte lógica del programa, dicha función no podrá ser evaluada, por lo que se perderán algunas respuestas lógicas. El análisis de residuación [Han95] pretende detectar estas situaciones en tiempo de compilación.

- *Análisis de secuencialidad*: La generación de código eficiente en la implementación de los lenguajes lógico-funcionales depende de la secuencialidad de las reglas de los programas, es decir, de la existencia de un orden óptimo de evaluación para los argumentos. El análisis de secuencialidad [MMN98] es una combinación del análisis de estrictez y del análisis de tipos. La información que obtiene puede ayudar al compilador a generar versiones secuenciales de los programas.

Lenguajes concurrentes y paralelos. En este tipo de lenguajes surgen nuevos problemas relacionados con las comunicaciones y la creación de tareas:

- *Análisis de granularidad*: La granularidad de las tareas es importante en la ejecución eficiente de los programas paralelos. Una granularidad excesivamente fina da lugar a considerables sobrecostes, mientras que una excesivamente gruesa puede provocar un reparto de carga pobre. El análisis de granularidad pretende determinar la granularidad de las tareas, información que puede ser aprovechada por un compilador para un lenguaje con paralelismo implícito para decidir qué tareas son adecuadas para ser ejecutadas en paralelo. Este análisis [Loi98] suele basarse en análisis del coste de la evaluación de las expresiones [RG94] y del tamaño de las estructuras de datos [HLA94].

- *Análisis de corrección (terminación y productividad)*: En un sistema reactivo, el software debe reaccionar continuamente a los estímulos externos. El criterio fundamental para la corrección de los sistemas reactivos es la vitalidad o productividad: el programa debe reaccionar continuamente a la entrada produciendo una salida. Este análisis [HPS96, Par00] pretende detectar algunos comportamientos no deseados en los sistemas reactivos como bloqueos, no terminación y otros errores.

- *Análisis del flujo de control*: Muchas técnicas de optimización de programas dependen de la información del flujo de control que, como ya hemos visto en los lenguajes imperativos, consiste en determinar para cada punto del programa, hacia qué puntos del programa fluye el control. El análisis del flujo de control en los lenguajes concurrentes como *Concurrent ML* [NN94, SNN97] debe tener en cuenta las comunicaciones a través de canales entre los distintos procesos.

2.1.2 Resumen de técnicas

Las técnicas utilizadas en el análisis estático de programas pueden dividirse en tres grandes grupos: análisis basados en el flujo, análisis basados en la semántica (interpretación abstracta) y análisis basados en tipos. A continuación, se describe brevemente cada una de ellas. Las técnicas utilizadas en los análisis definidos en esta tesis son la interpretación abstracta y los sistemas de tipos anotados. Por ello, en las seccíones 2.2 y 2.3 se describen con detalle la interpretación abstracta y los análisis basados en tipos respectivamente, y su aplicación al análisis de lenguajes funcionales.

Análisis basados en el flujo. Incluyen las tradicionales técnicas de análisis del flujo de datos y las más recientes del flujo de control. Se han desarrollado fundamentalmente para lenguajes imperativos y funcionales.

En los análisis basados en el flujo para lenguajes imperativos el programa se visualiza como un grafo, en el que los nodos son los bloques elementales y las aristas describen cómo fluye el control entre ellos. A partir de dicho grafo se genera un conjunto de restricciones (ecuaciones o inecuaciones) posiblemente recursivas, para las que se busca una solución mínima, en cierto sentido. Ejemplos de este tipo de análisis son los mencionados en la sección anterior para lenguajes imperativos. Para más detalles, se puede consultar [UAH74, NNH99].

Los análisis de flujo en los lenguajes funcionales son más complejos, ya que en ellos el flujo de control no es evidente a partir de la sintaxis del programa. También se representan utilizando conjuntos de restricciones. Dependiendo de la cantidad de información contextual que se incluye en el análisis se distinguen los análisis monovariantes o 0-CFA (*Control Flow Analysis*), sin ningún tipo de información contextual, y los polivariantes o k-CFA, en los que se incluye un mecanismo para distinguir diferentes ejemplares dinámicos de las variables y de los puntos del programa. Ejemplos de análisis de este tipo en los lenguajes funcionales son el análisis de clausuras y el análisis de globalización mencionados en la sección anterior. También se han aplicado estas técnica a los lenguajes funcionales concurrentes, como se vió también en la sección anterior.

Análisis basados en la semántica: Interpretación abstracta. Con este tipo de técnicas se pretende definir los análisis de forma similar a como se define la propia semántica de los programas. La mayoría están basados en el uso de la semántica denotacional. A grandes rasgos, la semántica denotacional de un programa describe el conjunto de los posibles resultados del programa cuando se ejecuta sobre cada entrada posible. Por su parte, la semántica operacional describe paso a paso los cómputos de un programa. La interpretación abstracta es una teoría de aproximación de la semántica de los programas usada para la construcción de algoritmos de análisis de programas, la comparación de semánticas formales (por ejemplo, para la construcción de una semántica denotacional a partir de una operacional), diseño de métodos de demostración de propiedades, etc.

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 sobre valores reales, para obtener, a partir del resultado abstracto obtenido, información común de todas las posibles ejecuciones del programa sobre los valores reales. Los valores abstractos representan aproximaciones de aquellas propiedades de los valores reales que nos interesan en el análisis concreto que estemos realizando.

Esta técnica se utiliza en todo tipo de lenguajes, pero es especialmente apropiada para los lenguajes declarativos. Ejemplos de análisis en los lenguajes funcionales son los análisis de estrictez [Myc81, BHA86], de uso [Mar93], de propagación de constantes [JM86], del momento de vinculación [HS91], de ausencia [Wra85] y de terminación [Myc81]. Los análisis mencionados en la sección anterior para los lenguajes lógicos y para la mayoría de los lógico-funcionales son también análisis definidos mediante interpretación abstracta. El análisis del tamaño de las estructuras de datos [HLA94] también se puede definir usando interpretación abstracta.

Interpretación abstracta en los lenguajes funcionales. La interpretación abstracta en los lenguajes funcionales se puede clasificar 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 raíz 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 las funciones de una forma compacta para poder compararlas. En la Sección 2.2.3 se estudiarán diferentes métodos para representar funciones y para reducir el coste del cálculo del punto fijo.

Los **análisis hacia atrás** propagan la información de la raíz 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 el que se evalúan los argumentos. El concepto de contexto se puede formalizar de distintas formas, de las cuales la más habitual el el uso de proyecciones [WH87, HL90]. 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 supone el cálculo de n puntos fijos. Pero, puesto que cada función a calcular es ahora una función unaria, el coste es cuadrático en el tamaño del programa en el peor de los casos. Los análisis hacia atrás tienen una seria limitación: su extensión a orden superior no parece proporcionar información con la precisión deseada.

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 dominios de Scott como dominios semánticos donde toman valores las expresiones del lenguaje. 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 él. 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 al mismo, puesto que todos ellos tienen también la propiedad que posee ese elemento.

Existen algunas propiedades que no se pueden representar utilizando conjuntos Scott–cerrados, como la estrictez de cabeza [Kam92] y la propiedad de que una función es constante. El uso de **relaciones de equivalencia total** en lugar de funciones abstractas 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** [Hun91] resuelve por completo el problema, puesto que permite representar ambas propiedades para funciones de orden superior. Es, por tanto, la técnica más potente conocida hasta el momento.

La interpretación abstracta de un lenguaje funcional presentada en la Sección 2.2.3 es un análisis hacia delante donde las propiedades se representan mediante conjuntos Scott-cerrados. Para más detalles sobre el resto ver [Seg99].

Análisis basados en tipos. Este tipo de análisis se introduce con la esperanza de incrementar la eficiencia de los análisis. Incluyen los sistemas de tipos anotados y los sistemas de efectos. Para poder aplicar esta técnica es necesario que el lenguaje analizado disponga de un sistema de tipos. Dicho sistema de tipos es modificado con anotaciones adicionales que proporcionan información sobre las propiedades deseadas. Solamente es necesario modificar las reglas del sistema de tipos ya existente incorporando la forma de propagar las anotaciones. Pueden anotarse solamente los tipos base, en cuyo caso se habla de un sistema de tipos anotados. Se habla de sistema de efectos cuando se obtiene información lateral adicional asociada a las derivaciones de tipos, información que recibe el nombre de efecto. Como primer paso de un análisis de este tipo, se utiliza el algoritmo de comprobación de tipos o el algoritmo de inferencia de tipos del sistema de tipos subyacente y durante el mismo se recolectan restricciones entre las anotaciones asignadas a los tipos. A continuación hay una fase de resolución de dichas restricciones para obtener los valores de las anotaciones. En algunos casos, de entre las posibles soluciones puede haber algunas óptimas en un cierto sentido. El coste del análisis depende del tipo de restricciones generadas y del algoritmo utilizado para resolverlas. Ejemplos de análisis definidos usando estas técnicas son: el análisis de uso [LGH+92, TWM95, WJ99, GS01], del momento de vinculación [Mog89, Mos94], de terminación [NN96] y de regiones [HP99, Par00] en lenguajes funcionales; el análisis de no determinismo [HS00] en los lenguajes lógico-funcionales; el análisis de coste [RG94] y de corrección [HPS96, Par00] en los lenguajes paralelos.

2.1.3 Historia del análisis de programas funcionales

La interpretación abstracta fue aplicada por primera vez a los lenguajes funcionales por Mycroft [Myc81]. En 1980, publicó un análisis de estrictez para programas de primer orden con tipos de datos monomórficos y planos.

En 1985, Burn, Hankin y Abramsky [BHA86] presentaron una forma de extender la interpretación abstracta a programas de orden superior, todavía monomórficos, desarrollando un análisis de estrictez de orden superior. Posteriormente, Burn extendió estas ideas a cualquier análisis [Bur91] (ver Sección 2.2.3).

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 [Ses89, Shi88]. Esta última 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 describirá con detalle. Recibe el nombre de análisis de pares de estrictez. En él, el valor abstracto de una expresión e que se evalúa 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, si bien con un coste excesivo. Además no existe una seguridad de terminación del análisis si se aplica a lenguajes no tipados. No se han encontrado condiciones necesarias, aunque 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 produzca 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 por Baraki [Bar93] en 1993.

En 1987, Wadler [Wad87] resolvió el problema de analizar estructuras de datos (es decir, tipos de datos no planos). Definió un dominio abstracto finito para las listas de enteros cuyos valores representaban los distintos grados de evaluación que puede llevar a cabo una función sobre un argumento de tipo lista.

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

Esta idea fue extendida a orden superior por Martin y Hankin [MH87]. 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.

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], y posteriormente mediante la noción de proyección [WH87], la cual especifica qué partes de un valor se usarán posteriormente.

Los análisis hacia atrás se han usado en el análisis de estrictez [Hug86, Hug87, WH87, Joh81, Wra85, HW87]. También se han utilizado en el análisis de recuperación de espacio. Jensen y Mogensen [JM90] desarrollaron un análisis en tiempo de compilación para recogida de basura. 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.

En 1981 Jonhsson [Joh81] desarrolló un 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 simbólicamente sistemas de ecuaciones de contextos.

Dybjer [Dyb87], en 1987, proporcionó una fundamentación teórica del análisis hacia atrás basada en imágenes inversas de conjuntos abiertos.

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. Kei Davis y Wadler [DW90] mejoraron este análisis haciéndolo más preciso.

Hughes y Launchbury [HL92] demostraron que las interpretaciones abstractas para las que existe un análisis hacia atrás 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, Hud86b] 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 de qué se evaluará y no se evaluará con seguridad tras la evaluación de una expresión 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].

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 utilizados en la práctica. Un ejemplo es el análisis de **propagación de constantes** en un lenguaje funcional. 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 utilizar **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. 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. 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. Sin embargo, algunas propiedades como la 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.

Posteriormente, los esfuerzos se centraron en la eficiencia de los análisis. Los análisis basados en tipos pretenden alcanzar la misma potencia que la interpretación abstracta pero con mayor eficiencia. Dentro de los análisis basados en tipos se desarrollaron dos estilos diferentes.

Uno de los estilos consiste en definir nuevos tipos que representan las propiedades deseadas, y construir un nuevo sistema de tipos teniendo en cuenta las propiedades descritas. En este sentido, el primer uso explícito de 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 [HL94a, HL94b, HM94] introdujeron una nueva idea, los tipos perezosos, que permiten mantener la potencia de la interpretación abstracta y a la vez desarrollar algoritmos eficientes.

Este primer estilo de análisis basado en tipos ha quedado relegado en la actualidad en favor de una segunda alternativa. En ella, se parte de un sistema de tipos ya existente y se añaden anotaciones que representan la información deseada. Si es necesario, también se puede añadir efectos, para así obtener información lateral. Son los ya mencionados sistemas de tipos anotados y los sistemas de tipos y efectos.

En 1996 se desarrolló un sistema de tipos únicos para Clean [BS96] con el objetivo de identificar aquellos argumentos que se usan exactamente una vez, es decir, aquellos valores que se pueden actualizar *in situ* sin perder la transparencia referencial.

En [LGH$^+$92] se presentó un análisis de uso para un lenguaje funcional perezoso con un sistema de tipos monomórfico. Posteriormente, en [TWM95] se presentó un análisis de uso para un lenguaje funcional perezoso con sistema de tipos Hindley-Milner. En él se detectó el llamado problema del *envenenamiento*, el cual consiste en que cada llamada a una función influye en el tipo de la función y por tanto en los argumentos de las demás llamadas provocando una gran pérdida de información. En [WJ99] se intentó dar una solución a este problema utilizando subtipado. Se demostró la existencia de un tipo principal con respecto al conjunto de restricciones, por lo que podía obtenerse una solución óptima. El tamaño de las restricciones generadas en este caso era lineal con respecto al tamaño del programa. Sin embargo, pronto se demostró que el subtipado no era capaz de proporcionar tipos satisfactorios para programas reales. Debido a la currificación, no se podía expresar dentro de un tipo interdependencias entre las anotaciones de uso. Para ello, en [WJ00] se introdujo polimorfismo en las anotaciones de uso. En [GS01] se desarrolló también un análisis de uso con polimorfismo acotado de anotaciones y recursión polimórfica tanto en los tipos base como en las anotaciones. Este análisis permite llevar a cabo un análisis más preciso de las estructuras de datos. Sin embargo, la introducción del polimorfismo hace que el número de restricciones crezca mucho, pudiendo llegar a ser hasta exponencial. También en [GS01] se presentó una nueva forma de representar restricciones de una forma compacta: las *abstracciones de restricciones*. Usándolas, de nuevo el número de restricciones requerido es proporcional al tamaño del programa.

Los sistemas de tipos anotados se han utilizado también para garantizar la corrección de los programas funcionales reactivos. En este tipo de sistemas tipar una expresión es equivalente a dar una prueba de corrección. Por ejemplo, en [HPS96, Par97, Par00] se presenta un sistema de tipos con tamaño en el que se garantiza que aquellos programas que están bien tipados, o bien terminan o bien son productivos, y por tanto no se bloquean. Usando ideas similares, en [HP99, Par00] se desarrolló un sistema de tipos en el que se garantiza que los programas bien tipados se ejecutan utilizando una cantidad acotada de memoria.

2.2 Interpretación abstracta

2.2.1 Preliminares

En esta sección se presentan algunos conceptos utilizados en secciones y capítulos posteriores: las conexiones de Galois, los conjuntos Scott-cerrados, los dominios potencia y la categoría de pares inmersión-clausura. En el Apéndice A se pueden consultar también algunos conceptos básicos de la teoría de dominios y de la teoría de categorías.

Conexiones de Galois

Definición 1 *Dados dos órdenes parciales $P^b(\sqsubseteq^b)$ y $P^\#(\sqsubseteq^\#)$, una* ***conexión de Galois*** *es un par de funciones $\alpha \in (P^b \to P^\#)$ y $\gamma \in (P^\# \to P^b)$ tales que:*

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

Se puede denotar por $P^b(\sqsubseteq^b) \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} P^\#(\sqsubseteq^\#)$. Habitualmente α recibe el nombre de *función de abstracción* y γ el de *función de concreción*.

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

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

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

$$\forall a \in P^\#.\gamma(a) = \sqcup^b\{c \mid \alpha(c) \sqsubseteq^\# a\},$$

y

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

Además, α es sobreyectiva si y sólo si γ es inyectiva si y sólo si $\forall p^b \in P^b.\alpha(\gamma(p^b)) = p^b$, en cuyo caso se habla de una **sobreyección de Galois** o **inserción de Galois**. Análogamente α es inyectiva si y solo si γ es sobreyectiva, en cuyo caso se habla de una **inyección de Galois**.

Topología de Scott

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 cerrada bajo uniones arbitrarias e intersecciones finitas 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 aquéllas que sean 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** 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.

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

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

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.

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 $x \in S$, (S cerrado inferiormente)
2. $X \subseteq S$ con X dirigido, entonces $\bigsqcup X \in S$.

Si tomamos, por ejemplo, el dominio $\omega^{\top}$ 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.

Dominios potencia.

Dado un dominio D, existen varias formas de definir órdenes sobre el **conjunto potencia de** D (conjunto $P(D)$ 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:

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

  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:

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

  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:

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

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.

En términos de teoría de categorías $\mathcal{P}$ es un funtor, pues de manera natural se puede aplicar a funciones continuas entre dominios para dar funciones continuas entre dominios potencia. En concreto, si $f : D \to E$, tomamos $\mathcal{P}(f) : \mathcal{P}(D) \to \mathcal{P}(E)$ dada por:

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

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

Pares inmersión–clausura

Dada la categoría C de dominios y funciones continuas, consideramos 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:

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

Se trata de un caso particular de inserciones de Galois. Llamamos a este par de morfismos *par inmersión–clausura*, donde e es la inmersión, y c es la clausura. La composición de morfismos en C^{ec} se define como

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

C^{ec} recibe el nombre de **categoría de dominios y pares inmersión–clausura**.

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 $\perp$.
- $h^e \sqsubseteq k^e$ si y solo si $h^c \sqsupseteq k^c$; h^e está determinada únicamente por h^c y viceversa.

Funtores en $(C^{ec})^n$. Algunos ejemplos de funtores son:

1.

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

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.

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

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

Para cada funtor F sobre $(C^{ec})^n$ tenemos dos funtores: $F^e : (C^{ec})^n \to C$ y $F^c : (C^{ec})^n \to C^{op}$ que actúan igual que F sobre los objetos, pero que sobre un morfismo h seleccionan la componente e o 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.

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

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

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

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

Definición 2 *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:*

$$
\begin{array}{rcl}
h_a^e(x) & = & \left\{ \begin{array}{ll} a & \textit{si } x = 0 \\ \top_A & \textit{si } x = 1 \end{array} \right. \\
h_a^c(x) & = & \left\{ \begin{array}{ll} 0 & \textit{si } x \sqsubseteq a \\ 1 & \textit{e.o.c.} \end{array} \right.
\end{array}
$$

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.

2.2.2 Conceptos generales

Introducción

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), posets, 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 (datos, etc.) y S se define composicionalmente por inducción sobre la estructura sintáctica de los programas, usando habitualmente puntos fijos para manejar recursión.

El marco clásico de la interpretación abstracta, introducido por Cousot [CC77, Cou81, CC92a, CC92b], 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 corrección. 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 lo más usual es utilizar **conexiones de Galois**. La razón de ello es que así se asegura la existencia de una aproximación mejor a las propiedades concretas de entre todas las abstractas que son correctas.

Un ejemplo sencillo: 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, para describir los enteros negativos y positivos en lugar de usar enteros podríamos usar los valores abstractos -1 y $+1$ 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 manejarlos, se introduce un valor abstracto $\top$, que representa el hecho de que no sabemos nada acerca del resultado:

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

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 $\top$: 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.

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

Además, tenemos valores que podrían ser aproximados por distintos valores abstractos mínimos. En nuestro caso tenemos el 0, que puede ser aproximado tanto por $+1$ como por -1. En estos casos, o bien se hace una elección adecuada dependiendo de la expresión a analizar, o bien se añade de forma explícita, enriqueciendo el dominio abstracto. Haciendo esto último, obtendríamos:

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

Aproximación de las propiedades de los programas

Como hemos dicho anteriormente, definiremos 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 a 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 habitual es que el dominio de las propiedades concretas sea un orden parcial $P^b(\sqsubseteq^b)$, en el que la relación de orden representa la precisión relativa de las propiedades concretas: $p_1^b \sqsubseteq^b p_2^b$ 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 el proceso para alcanzarlo puede ser transfinito). 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 misma 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 3 (regla de los signos):

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

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

El orden entre los elementos (ver Figura 2.1) viene determinado entonces por la relación de inclusión $\subseteq$ entre los conjuntos de enteros que representan. En cuanto a las propiedades abstractas, una posible elección sería $P^{\#} = \{f^{\#}, -1, 0, +1, t^{\#}\}$, con sus elementos ordenados según se muestra en la Figura 2.1.

$\square$

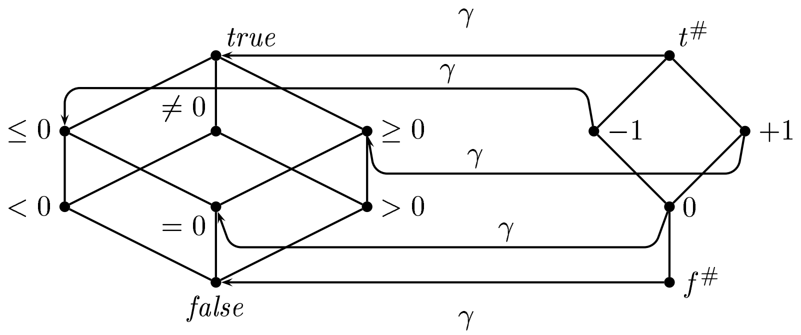


Figura 2.1: Dominios de propiedades concretas y abstractas de los enteros, y función de concreción

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 $P^{\#}$.

Relación de corrección

El tercer paso que debe darse en el diseño de una interpretación abstracta es la especificación de la correspondencia entre las propiedades concretas y abstractas. Esta se puede definir mediante una relación de corrección $\sigma \in P(P^{b} \times P^{\#})$, donde $\langle c, a\rangle \in \sigma$ 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:

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

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 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 devuelve la mejor de las aproximaciones abstractas, $\alpha(p^{b})$, de la propiedad concreta p^{b}.

Ejemplo 4 (regla de los signos)

Continuando con el ejemplo anterior, la función de concreción viene representada en el grafo de la Figura 2.1. Por su parte, la función de abstracción viene dada por la siguiente tabla:

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

□

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

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

El hecho de que a es una aproximación válida de la información dada por c se puede expresar mediante $\alpha(c) \sqsubseteq^{\#} a$, 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 $c \sqsubseteq^{b} \gamma(a)$, 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.

Aproximación del punto fijo

Supongamos que $P^{b}(\sqsubseteq^{b}, \sqcup^{b})$ y $P^{\#}(\sqsubseteq^{\#}, \sqcup^{\#})$ son ordenes parciales completos, que $F^{b} : P^{b} \to P^{b}$ proporciona la semántica concreta *lfp* F^{b} de un programa (donde *lfp* es el operador de menor punto fijo), y que estamos interesados en su abstracción α (*lfp* F^{b}), donde $P^{b}(\sqsubseteq^{b}) \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} P^{\#}(\sqsubseteq^{\#})$.

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

[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 esta sección vamos a estudiar esta forma general de proporcionar la corrección. En la siguiente sección, estos conceptos se particularizarán para los lenguajes funcionales.

intentar calcular α (*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 garanticen 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

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

  (o $F^{\#} = \alpha \circ F^{b} \circ \gamma$ 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, como $\alpha \circ F^{b} = F^{\#} \circ \alpha$ entonces $\alpha(p^{b})$ es un punto fijo de $F^{\#}$ con lo que $\alpha(\bigsqcup^{b}_{n\geq 0} F^{b^n}(\perp^{b})) = \bigsqcup^{\#}_{n\geq 0} F^{\#^n}(\perp^{\#})$ es un punto fijo de $F^{\#}$.

En particular, cuando $F^{\#}$ es monótona, se trata del mínimo punto fijo de $F^{\#}$.

Desgraciadamente, esta situación no es habitual. En general uno debe conformarse con una aproximación abstracta $p^{\#}$ realizada desde arriba, de modo que $\alpha($*lfp* $F^{b}) \sqsubseteq^{\#} p^{\#}$, o equivalentemente *lfp* $F^{b} \sqsubseteq^{b} \gamma(p^{\#})$.

Proposición 5 *Dados dos retículos completos* $P^{b}(\sqsubseteq^{b}, f^{b}, t^{b}, \sqcap^{b}, \sqcup^{b})$ *y* $P^{\#}(\sqsubseteq^{\#}, f^{\#}, t^{\#}, \sqcap^{\#}, \sqcup^{\#})$*, una conexión de Galois* $P^{b}(\sqsubseteq^{b}) \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} P^{\#}(\sqsubseteq^{\#})$ *y* $F^{b} : P^{b} \to P^{b}$ *monótona, entonces*

$$\alpha(\mathit{lfp}\ F^{b}) \sqsubseteq^{\#} \mathit{lfp}\ (\alpha \circ F^{b} \circ \gamma).$$

Una consecuencia de esta proposición 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 a partir de la semántica concreta mediante un cálculo formal que simplifique $\alpha \circ F^{b} \circ \gamma$ para expresarlo usando operadores sobre propiedades abstractas.

2.2.3 Interpretación abstracta en los lenguajes funcionales

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.

Las propiedades de las funciones se pueden representar de formas diversas. Una de ellas consiste en utilizar conjuntos **Scott–cerrados**. En tal caso, se utilizan retículos finitos como dominios abstractos que representan 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 $\{\bot_D\}$ y D, las cuales son a su vez interpretados por los valores abstractos 0 y 1 respectivamente.

Una de las formas más habituales de relacionar la interpretación estándar y la abstracta (o dos interpretaciones abstractas) de un lenguaje funcional consiste en la definición de una función de **concreción** γ de los valores abstractos a las propiedades concretas, demostrando que si la semántica abstracta $tabs[\![\cdot]\!]$ de una expresión en un entorno abstracto ρ^t produce un valor abstracto $t^\#$, entonces la semántica estándar $sem[\![\cdot]\!]$ de dicha expresión en un entorno ρ^s que aproxima superiormente a la concreción de ρ^t tiene la propiedad $\gamma(t^\#)$. Sin embargo, cuando las propiedades estudiadas por los análisis definidos sobre lenguajes funcionales son propiedades referentes a las funciones, la corrección del análisis se enuncia en términos de las expresiones funcionales. Así, si en particular la expresión es una función f, el análisis será correcto si cuando

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

entonces se cumple que

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

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 14 en esta sección). Dicho resultado se apoya en la propiedad de semihomomorfismo de la función de abstracción (ver Proposición 11 en esta sección).

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.

Por su parte, los términos vienen dados por la gramática:

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

donde cada $\mathbf{fix}_{(\sigma\to\sigma)\to\sigma}$ representa al operador de punto fijo, y nos sirve para expresar la recursión[2].

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 a los enteros y los booleanos, y

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

Para definir la semántica necesitamos partir de una interpretación de constantes $K : constantes \to D$, donde $K(c^\sigma) \in D_\sigma$ y de un entorno $\rho^s : Env = Var \to D$ (denotaremos los entornos con un superíndice que nos indica si se trata de un entorno para la semántica estándar, s, o para la abstracta, t):

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

[2]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 ateniéndonos a la sintaxis definida debería hacerse usando el correspondiente operador **fix**.

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

De esta forma, cada término (funcional) sintáctico de tipo $\alpha \to \beta$ se interpreta como una función continua en D, de D_α en D_β, mientras que la aplicación funcional sintáctica se interpreta como la aplicación de una función a un argumento.

Ejemplo 6 En este ejemplo, para simplificar notación, omitiremos los superíndices de las constantes, excepto en el caso del **if**, en el que nos limitamos a indicar su tipo generador σ. En su lugar, precisaremos los tipos de las constantes mediante declaraciones $c : \sigma$. Si tenemos los tipos **int** y **bool** como tipos base, podríamos tener como constantes del lenguaje:

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

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

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

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

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

□

Semántica abstracta. De la misma forma, aunque eligiendo los dominios adecuados para cada caso, se definiría la semántica abstracta adecuada para determinado análisis. En general obtendremos un dominio abstracto

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

donde de nuevo $+$ representa la unión disjunta, y B_A es el retículo finito adecuado para cada tipo base A.

Ejemplo 7 (análisis de estrictez)
Utilizaremos como ejemplo el análisis de estrictez de [BHA86] donde las constantes del lenguaje son las propuestas en el Ejemplo 6.

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 2.2 podemos ver ilustrado parte del dominio abstracto que resulta: se muestra $B_{\mathbf{int}}$, $B_{\mathbf{int}\to\mathbf{int}}$ y $B_{\mathbf{int}\to\mathbf{int}\to\mathbf{int}}$. Puesto que B_A es un retículo, también lo es cada B_σ, y al ser finitos son completos.

La semántica abstracta *tabs* se define de forma análoga a la estándar, con la diferencia de la interpretación de las constantes K'. En el caso del análisis de estrictez estas se interpretan de la siguiente forma:

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

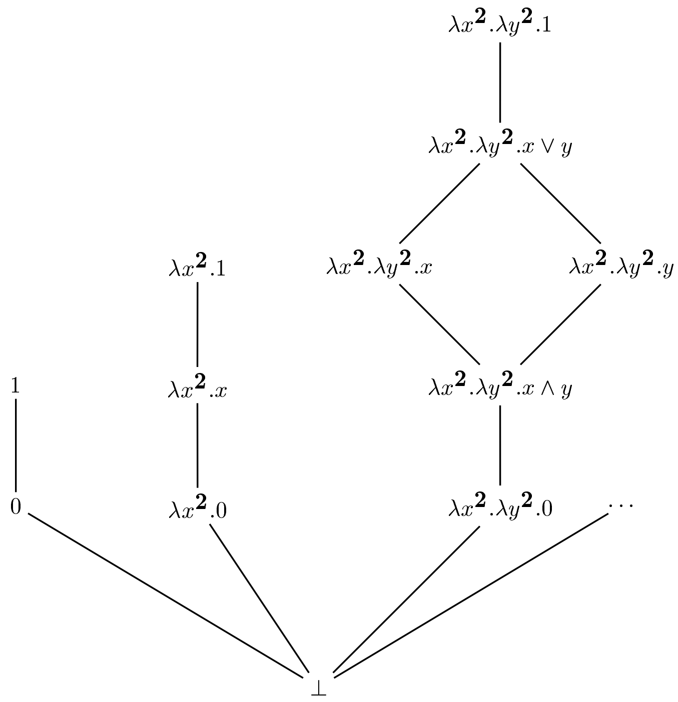


Figura 2.2: El dominio abstracto para el análisis de estrictez

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

$\square$

Relación entre las semánticas. A continuación debemos relacionar la interpretación abstracta y la estándar para garantizar la corrección del análisis. En lugar de definir directamente la función de abstracción $abs : D \to B$ como una única función, la definiremos por partes, 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 8 (análisis de estrictez) En el caso del análisis de estrictez, para cada dominio básico A:

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

$\square$

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 cada dominio D su correspondiente dominio potencia de Hoare $\mathcal{P}(D)$ y a cada 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 (véase definición en página 42).

- $\gamma_\sigma : B_\sigma \to \mathcal{P}(D_\sigma)$ 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 representa 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).

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

Propiedades de las funciones de abstracción y concreción. Las proposiciones de esta sección hacen uso de propiedades del dominio potencia de Hoare. Todas las demostraciones se pueden encontrar en [Bur91].

Proposición 9 *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 quede 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 demostró en [Abr90]que si manejamos retículos completos arbitrarios (posiblemente infinitos), entonces las funciones de abstracción inducidas por las de concreción pueden no ser continuas. Ahora bien, si las funciones de abstracción llevan elementos finitos a elementos finitos, entonces sí que lo son. Y claramente esto se cumple en el caso de que los retículos sean finitos.

A partir de ahora supondremos que disponemos de abs_A continua y estricta y que $abs_{\alpha \to \beta}$ está definida como hemos indicado anteriormente, de modo que se cumplen todas las propiedades de la Proposición 9.

Los siguientes resultados permiten demostrar la corrección del análisis planteado.

Proposición 10 *Tal como están definidas se cumple que γ_σ y Abs_σ forman una conexión de Galois, es decir:*

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

Proposición 11 *abs_σ tiene propiedades semihomomórficas en la aplicación de funciones, es decir:*

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

La demostración se obtiene utilizando la Proposición 10 y propiedades del dominio potencia de Hoare. Como consecuencia de esto, se tiene:

Proposición 12

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

La demostración se obtiene usando las proposiciones 9 y 11.

La corrección del análisis. La propiedad de corrección del análisis para las funciones nos dice que si para toda variable x^τ, $\rho^t(x^\tau) \sqsupseteq abs_\tau(\rho^s(x^\tau))$ y

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

entonces se debe cumplir que:

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

Ejemplo 13 (análisis de estrictez)

La propiedad de corrección del análisis de estrictez es la siguiente:

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

ya que se cumple $\gamma(\bot_{B_\sigma}) = \{\bot_{D_\sigma}\}$.

□

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.

Proposición 14 *Si todas las constantes cumplen la condición de seguridad, entonces para todo ρ^s, ρ^t tales que para todo x^τ, $abs_\tau(\rho^s x^\tau) \sqsubseteq (\rho^t x^\tau)$, y para toda expresión e, se tiene:*

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

Esta proposición se demuestra por inducción estructural sobre e utilizando las proposiciones 11 y 12.

Como consecuencia de esta proposición y de las proposiciones 10 y 11 obtenemos la demostración de la corrección del análisis.

Ejemplo 15 (análisis de estrictez)

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

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

donde 0_{σ_i} y 1_{σ_i} representan el ínfimo y supremo, respectivamente, del retículo B_{σ_i} (ver Figura 2.2).

□

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

Ejemplo 16 Sea la función de orden superior $twice : (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int}$:

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

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

$twice^\#$

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

Puesto que $twice^{\#}$ $(\lambda y.0)$ $1 = 0$ y $twice^{\#}$ $(\lambda y.1)$ $0 = 1$, *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.

□

Cálculo eficiente de puntos fijos

La interpretación de funciones recursivas implica el cálculo de puntos fijos mediante la cadena ascendente de Kleene. Una forma de calcular el punto fijo consiste en iterar sucesivamente un número suficiente de veces hasta estar seguros de haber alcanzado el punto fijo. Puesto que los dominios utilizados son finitos, el número máximo de iteraciones necesarias para calcular el punto fijo viene dado por la profundidad del dominio funcional correspondiente, la cual está definida como la longitud de la cadena más larga estrictamente creciente de valores de dicho dominio. 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 insostenible cuando aquél es grande.

La opción tomada habitualmente es hacer una comparación tras cada iteración para saber si ya se ha alcanzado el punto fijo. En tal caso, la eficiencia de la comparación influye en la eficiencia del cómputo del punto fijo. En el peor de los casos, el coste de la comparación es exponencial en el número de argumentos. El problema de detectar la igualdad de funciones de N argumentos con valores en $\mathbf{2}$ es NP–completo, y por lo tanto no hay algoritmo polinomial que pueda resolverlo con generalidad [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 cuya comparación sea eficiente en los casos más comunes. Clack y Peyton Jones [CP85, PC87] examinaron tres formas distintas de representación para el análisis de estrictez: las **tablas de verdad**, las **expresiones booleanas** y las **fronteras**. El algoritmo de fronteras fue posteriormente optimizado y extendido a orden superior y retículos arbitrarios por Martin y Hankin [MH87]. Posteriormente, Peyton Jones y Partain [PP93] utilizaron una forma de representación aproximada, las **signaturas**. Dichas signaturas permiten definir un operador de ensanchamiento con el objetivo de calcular una aproximación del mínimo punto fijo en un tiempo cuadrático.

Tablas de verdad. Una función queda caracterizada medianteel correspondiente conjunto de pares argumento–resultado (su grafo). Por lo tanto, las funciones abstractas del análisis de estrictez se pueden representar mediante tablas de verdad. Con esta representación la comparación es siempre exponencial en el número de argumentos de la función, independientemente de lo sencillas que sean las funciones a comparar.

Expresiones booleanas. Debido al aspecto del dominio elegido para el análisis de estrictez, para representar las funciones abstractas en dicho análisis se pueden utilizar expresiones booleanas simbólicas. Esta representación es mucho más compacta que la anterior. Para poder compararlas, debemos manipularlas hasta obtener una forma canónica. Las formas más comunes son la **forma normal conjuntiva** y la **forma normal disyuntiva**. En general, ninguna es mejor que la otra, ya que algunas expresiones tienen una representación más compacta en forma conjuntiva y otras en forma disyuntiva. Sin embargo, al pasar a orden superior el uso de expresiones booleanas se hace inmanejable.

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. Si el resultado de aplicar $f^{\#}$ a los argumentos especificados en un nodo es 0 (respectivamente 1), le llamaremos **0–nodo** (respectivamente **1–nodo**). De esta forma podríamos etiquetar los nodos del grafo con el valor correspondiente de $f^{\#}$ sobre ellos.

Se han 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. 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:

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

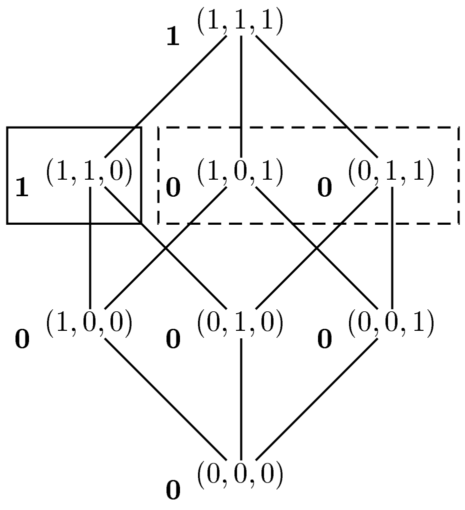


Figura 2.3: Valores de $f_1^\#$ y fronteras (0–frontera en caja punteada y 1–frontera en caja continua)

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 este es el menor de ellos. En la Figura 2.3 se muestra el retículo de posibles argumentos, y cada una de las fronteras. Junto a cada punto del retículo figura en negrita el valor de $f_1^\#$ en ese punto.

□

Resulta inmediato comprobar que **dos funciones con las mismas 0–fronteras son iguales**. En consecuencia, una frontera es una forma **canónica y compacta** de representar una función.

Veamos el coste del cálculo del punto fijo usando esta representación. El número de iteraciones es proporcional a la profundidad del retículo de argumentos. En cada iteración es necesario calcular la nueva 0–frontera y compararla con la de la iteración anterior. La nueva 0–frontera puede calcularse a partir de la anterior, y su coste es proporcional a la profundidad del retículo de argumentos. El coste de comparar dos fronteras es proporcional a la anchura del retículo de argumentos.

Si nos restringimos a funciones de primer orden en el análisis de estrictez, el coste de cada iteración es lineal con respecto al número de argumentos de la función, por lo que el coste del cálculo del punto fijo es cuadrático. Sin embargo, al pasar a orden superior, la profundidad de los retículos de argumentos y por tanto el coste del cálculo del punto fijo se hace de nuevo exponencial. Si se consideran retículos base con un mayor número de elementos, se hace necesario controlar varios conjuntos frontera. Cuando aumenta el número de elementos del retículo base que no son comparables, la eficiencia del algoritmo decrece rápidamente.

En definitiva, 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. Actualmente la atención se centra en el cálculo aproximado de los puntos fijos. Se prefiere renunciar a la exactitud frente a la ineficiencia.

Aproximación de puntos fijos usando ensanchamiento y estrechamiento. Los operadores de ensanchamiento y estrechamiento[3] 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 aquí es el segundo. Naturalmente, el precio a pagar es la pérdida de la exactitud: estas técnicas calculan solamente una aproximación al punto fijo.

Definición 18 *(operador de ensanchamiento) Dado un retículo completo L, un operador $\bigtriangledown \in I\!N \rightarrow ((L \times L) \rightarrow L)$ es un* ***operador de ensanchamiento*** *si satisface las siguientes condiciones:*

- $\forall j > 0, x, y \in L. x \sqcup y \sqsubseteq x \bigtriangledown(j)\ y$.
- *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 $y_0 = x_0, y_1 = y_0 \bigtriangledown(1)\ x_1, \ldots y_n = y_{n-1} \bigtriangledown(n)\ x_n$ es eventualmente estable, es decir, existe un $k \geq 0$ tal que para todo $i \geq k : y_i = y_k$.*

Proposición 19 *Sea f una función monótona sobre L y $\bigtriangledown$ un operador de ensanchamiento. El límite u de la siguiente secuencia:*

$$\begin{array}{lll} x_0 & = \bot & \\ x_{n+1} & = x_n & si\ f(x_n) \sqsubseteq x_n \\ x_{n+1} & = x_n \bigtriangledown(n+1)\ f(x_n) & e.o.c. \end{array}$$

se puede calcular en un número finito de pasos, $fix(f) \sqsubseteq u$ 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 2.4.

[3] En inglés *widening* y *narrowing*.

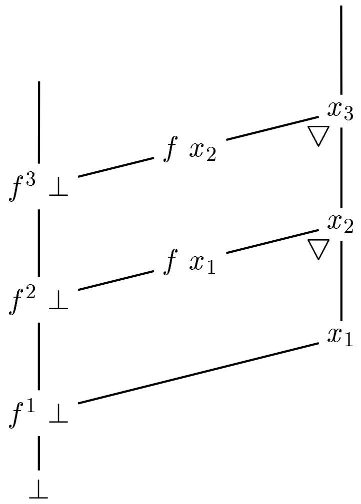


Figura 2.4: Relación entre la cadena de Kleene y la obtenida mediante el operador de ensanchamiento

Definición 20 *(operador de estrechamiento)*
Un operador $\triangle : \mathbb{N} \to ((L \times L) \to L)$ es un ***operador de estrechamiento*** *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 21 *Sea f una función monótona sobre L y $\triangle$ un operador de estrechamiento. Sea $u \in L$ tal que $\mathit{fix}(f) \sqsubseteq u$ 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 $\forall k \geq 0 : \mathit{fix}(f) \sqsubseteq x_k$.

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.

En la Sección 5.6 se usará un operador de ensanchamiento para acelerar el cálculo del punto fijo. Más exactamente se utilizará un operador de cierre superior, es decir, un operador mayor o igual a la identidad $\mathcal{W} \sqsupseteq id$. Ahora bien, dado un operador de cierre superior $\mathcal{W} \sqsupseteq id$, se puede definir un operador de ensanchamiento equivalente $\nabla = \lambda(x, y).x \sqcup \mathcal{W}(y)$, como se hace en [HH92]. Por ello, en la citada sección, al igual que se hace en [PP93], utilizaremos la denominación operador de ensanchamiento.

Signaturas. El uso de signaturas resuelve a la vez el problema de encontrar una representación de las funciones que sea comparable de forma eficiente y la tarea de definir un operador de ensanchamiento para acelerar el cálculo del punto fijo. Una signatura es una representación aproximada de una función abstracta, por lo que varias funciones abstractas pueden estar representadas por una misma signatura. Por lo tanto, en el proceso de obtención de una signatura se pierde información. Dicha pérdida da lugar a un operador de ensanchamiento.

Puesto que en la Sección 5.6 se utilizará esta técnica proporcionando las definiciones formales y demostraciones de todos los resultados obtenidos para el análisis de no determinismo, aquí solamente se proporcionan las intuiciones necesarias para entender el método.

Veamos como ejemplo las signaturas definidas en [PP93] para el análisis de estrictez. La signatura de una función de n argumentos en este análisis es básicamente una secuencia de n "demandas". Dicha secuencia describe la evaluación llevada a cabo por la función de cada uno de los argumentos. Una demanda puede ser L o S. Una demanda L en la posición i-ésima de la signatura representa que no hay garantía de que la función evalúe el argumento i-ésimo. Una demanda S en la posición i-ésima de la signatura indica que la función es estricta en el argumento i-ésimo. Por ejemplo, si $f :: \mathbf{int} \to \mathbf{int} \to \mathbf{bool} \to \mathbf{int}$ fuese la función dada por

$$f\ x\ y\ z = \mathbf{if}\ z\ x\ y,$$

su correspondiente signatura es $L\ L\ S$. Estas signaturas forman un retículo finito.

Para obtener una signatura se aplica la función abstracta a algunas combinaciones de argumentos. Este proceso recibe el nombre de *muestreo*. En el caso del análisis de estrictez, cada función de n argumentos se aplica a las combinaciones: $\bot, \top, \ldots, \top$; $\top, \bot, \top, \ldots, \top$; $\ldots$; $\top, \ldots, \top, \bot$, donde todos los argumentos menos uno reciben como valor el supremo del dominio correspondiente y el restante recibe como valor el ínfimo del dominio correspondiente.

El valor abstracto de la función del ejemplo anterior es $f^\# = \lambda x^\#.\lambda y^\#. \lambda z^\#.z^\# \sqcap (x^\# \sqcup y^\#)$, donde $x^\#$, $y^\#$ y $z^\#$ pertenecen a $\mathbf{2}$. Para obtener la signatura se aplica $f^\#$ a las combinaciones de argumentos $0,1,1$; $1,0,1$ y $1,1,0$. Si el resultado es 0, la demanda es S; si es 1, la demanda es L. Así se obtiene la ya indicada $L\ L\ S$. Hay otras funciones abstractas con la misma signatura, por ejemplo $\lambda x^\#.\lambda y^\#.\lambda z^\#.z^\#$. Sin embargo, se cumple la propiedad de que dada una signatura, existe una función que es mayor o igual que todas las que tienen esa misma signatura. Dada una signatura $d_1, \ldots, d_n$ podemos obtener dicha función de la siguiente manera:

$$r(d_1, \ldots, d_n) = \lambda x_1. \ldots .\lambda x_n. \begin{cases} \perp \text{ si } x_1 \in \theta(d_1) \vee \ldots \vee x_n \in \theta(d_n) \\ \top \text{ e.o.c.,} \end{cases}$$

donde $\theta(d_i)$ devuelve el conjunto de valores del argumento i que hacen divergir con seguridad a la función, es decir, $\theta(L) = \emptyset$ y $\theta(S) = \{\perp\}$.

En el ejemplo se obtiene

$$\lambda x_1.\lambda x_2.\lambda x_3. \begin{cases} 0 \text{ si } x_3 = 0 \\ 1 \text{ e.o.c.} \end{cases}$$

Se cumple que si $f^{\#}$ tiene signatura $d_1 \ldots d_n$, entonces $r(d_1, \ldots, d_n) \sqsupseteq f^{\#}$, es decir la composición del muestreo con la función r da lugar a un operador de ensanchamiento, que se puede utilizar para acelerar el cálculo del punto fijo.

En realidad, el proceso de muestreo se puede ver como una función de abstracción, y la función r como una función de concreción. Ambas forman una inserción de Galois, lo que implica que su composición es un operador de ensanchamiento.

El coste del análisis usando signaturas y el correspondiente operador de ensanchamiento depende del tamaño de las signaturas. Cuanto mayor sea el muestreo realizado mayor será la cantidad de información que contenga una signatura y más caro resultará el cálculo del punto fijo. Por otro lado, si el muestreo es muy pequeño, el análisis perderá demasiada información: es necesario un compromiso. El coste de comparar dos signaturas es proporcional a su tamaño. Así, en el análisis de estrictez estudiado, es lineal con el número de argumentos de la función. El número de iteraciones es también proporcional al tamaño de las signaturas, por lo que en dicho análisis el coste de calcular el punto fijo es cuadrático (sin tener en cuenta el coste del cálculo de la signatura). Es importante notar que las signaturas y el operador de ensanchamiento se utilizan solamente en el cálculo del punto fijo, permaneciendo el resto del análisis inalterado. Es decir, se obtiene un coste aceptable perdiendo información sólo cuando hay que calcular puntos fijos. Por ello, esta técnica será la elegida en el análisis de no determinismo del Capítulo 5, para así obtener una versión del análisis implementable desde el punto de vista práctico.

En la Sección 5.6 las signaturas utilizadas son más complejas que las aquí descritas. Adicionalmente se lleva a cabo un análisis detallado del coste que además tiene en cuenta el proceso de muestreo, obteniéndose un coste cúbico.

Estructuras de datos

Cuando se manejan tipos estructurados, contaremos con dominios abstractos para los tipos base que capturarán las propiedades de interés sobre ellos; por su parte la interpretación abstracta de los constructores de tipo determinará las propiedades que puedan definirse sobre los correspondientes tipos estructurados.

Por ejemplo, en el análisis de estrictez, si el argumento de una función es una lista, hay varias formas en que una función puede usar dicho argumento. Para ilustrar el problema podemos ver algunos ejemplos propuestos por Wadler [Wad87].

Ejemplo 22 Consideremos 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 conocer el constructor en cabeza de la lista, es decir, si se trata de la lista vacía o tiene al menos un elemento. Sin embargo, la segunda, *length*, necesita evaluar toda la lista, aunque no le preocupan los valores contenidos en ella. La tercera, *sum*, no sólo 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 ha de ser más compleja, para así poder 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. Utilizó como dominio abstracto de las listas de enteros el conjunto $\{\perp, \infty, \perp_\epsilon, \top_\epsilon\}$, ordenado como se muestra en la Figura 2.5.

Recordemos que cada uno de los puntos del dominio abstracto representa una propiedad, es decir, un conjunto Scott–cerrado. Por lo tanto, el conjunto de listas representadas por cada punto del dominio contiene a todas las representadas por los puntos inferiores y a algunas más. En la Figura 2.5, junto a cada punto del dominio se presenta un elemento perteneciente al

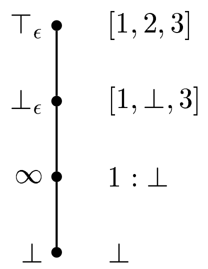


Figura 2.5: Dominio abstracto para las listas de enteros

conjunto representado por dicho punto, que no pertenece a ninguno de 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 capturar los distintos modos de evaluación: evaluar una lista a forma normal débil de cabeza, o evaluar la estructura de la lista, o evaluar la estructura de la lista y cada elemento de ella a forma normal débil de cabeza.

A continuación se definen las abstracciones de los constructores: $nil^\# = \top_\epsilon$, mientras que $cons^\#$ viene definido por la siguiente tabla:

$x^\#/xs^\#$	$\top_\epsilon$	$\perp_\epsilon$	∞	$\perp$
$\top$	$\top_\epsilon$	$\perp_\epsilon$	∞	∞
$\perp$	$\perp_\epsilon$	$\perp_\epsilon$	∞	∞

Si una función h se define por casos de la siguiente manera:

$$\begin{array}{rcl} h\ nil & = & a \\ h\ (cons\ x\ xs) & = & f\ x\ xs \end{array}$$

Wadler obtuvo que su abstracción viene dada por:

$$\begin{array}{lcl} h^\#\ \top_\epsilon & = & a^\# \sqcup (f^\#\ \top\ \top_\epsilon) \\ h^\#\ \perp_\epsilon & = & (f^\#\ \perp\ \top_\epsilon) \sqcup (f^\#\ \top\ \perp_\epsilon) \\ h^\#\ \infty & = & f^\#\ \top\ \infty \\ h^\#\ \perp & = & \perp \end{array}$$

En general, los resultados de la interpretación abstracta de una función h se pueden entender de la siguiente manera:

- Si $h^{\#}\ \perp = \perp$ entonces h es estricta en la lista argumento y éste se puede evaluar a forma normal débil para exponer el constructor en cabeza.
- Si $h^{\#}\ \infty = \perp$ 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 $h^{\#}\ \perp_{\epsilon} = \perp$ 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).

Polimorfismo

La mayoría de los lenguajes funcionales poseen sistemas de tipos polimórficos. Los ejemplares monomórficos de una función polimórfica se pueden analizar usando los métodos definidos anteriormente.

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 incluyen polimorfismo intentan establecer conexiones entre las propiedades e interpretación abstracta de los distintos ejemplares.

La relación entre los ejemplares puede establecerse definiendo un modelo categórico para el 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.

Baraki demostró en [Bar93] que 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, es decir, aquel en el que las variables de tipo se sustituyen por tipos básicos. La semántica de una función polimórfica $f :: \forall t_1. \ldots \forall t_n.\tau$ es la colección de valores $\{f_{D_1 \ldots D_n}\}$ de los distintos ejemplares de la función. Estos ejemplares se relacionan entre sí mediante pares inmersión-clausura; sucede lo mismo con los valores abstractos de dichos ejemplares. Baraki trabajó con la categoría de los retículos finitos y los pares inmersión–clausura A^{ec}.

Su resultado más importante es el **Teorema de Representación**, cuya consecuencia principal es la siguiente: siendo $\{f_A\}_{A \in Obj(A^{ec})}$ un conjunto de funciones continuas indexadas por retículos finitos A (es decir, los valores abstractos de los ejemplares de una función polimórfica), tales que $f_A : F(A) \to G(A)$ para ciertos functores F y G (que modelizan los constructores de datos, ver Sección 2.2.1) sobre A^{ec}, tenemos

$$f_A \sqsubseteq \sqcap_a G^e(h_a) \circ f_{\mathbf{2}} \circ F^c(h_a)$$

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

Esta desigualdad nos proporciona una aproximación al valor abstracto f_A de un ejemplar a partir del valor abstracto $f_{\mathbf{2}}$ del ejemplar más pequeño.

En el Capítulo 5 utilizaremos esta forma de aproximar el valor abstracto de un ejemplar a partir del ejemplar más pequeño, ya que el dominio base *Basic* allí utilizado es semejante al dominio **2** del análisis de estrictez.

Sin embargo, los dominios abstractos pueden ser grandes, por lo que en general no resulta práctico calcular todos los valores implicados en el cálculo de la máxima cota inferior. Puesto que se toma la máxima cota inferior sobre los elementos de A, también se podría elegir como aproximación superior $G^e(h_a) \circ f_2 \circ F^c(h_a)$ para un elemento particular a de A (aunque de esta forma perderíamos precisión). Esta será la opción tomada en el Capítulo 5.

2.3 Análisis basados en tipos

2.3.1 Introducción

En esta sección vamos a desarrollar con más detalle los análisis basados en tipos. Se utilizan como ejemplos el análisis de uso [TWM95, WJ99, GS01] y el análisis de terminación y productividad [HPS96, Par00]. Este último será utilizado como base para desarrollar el análisis del Capítulo 6.

La metodología seguida en los análisis basados en sistemas de tipos anotados y efectos sigue normalmente los siguientes pasos: Primero se define una semántica para el lenguaje tipado. Por ejemplo en los análisis de uso se utiliza una semántica natural de Launchbury [Lau93] para evaluación perezosa. En el análisis de terminación y productividad se define una semántica denotacional en la que los tipos son conjuntos de valores cerrados superiormente y los términos toman valores en dichos conjuntos. Más adelante, en la Sección 2.3.3, se explica con detalle esta semántica.

A continuación se expresa el análisis en términos de un sistema de tipos anotados o de un sistema de efectos. En el siguiente apartado se desarrolla con detalle el análisis de uso. Por su parte, en la Sección 6.1 se estudiará con detalle el sistema de tipos para análisis de terminación y productividad, ya que este será modificado para poder aplicarlo a programas Edén.

Una vez definido el análisis se demuestra su corrección. Si se usa semántica operacional, lo habitual es demostrar la corrección a través de un resultado de reducción del sujeto (*subject reduction*), es decir, demostrando que los tipos anotados se mantienen tras la reducción de la expresión mediante la semántica. Así, se demuestra, por ejemplo, la corrección del análisis de uso. Si se usa semántica denotacional es necesario demostrar que cada afirmación sobre el tipo de un término es correcta, es decir, el valor asignado por la semántica al término pertenece al conjunto de valores denotado por el tipo. Así se demuestra la corrección del análisis de terminación y productividad.

Para poder aplicar el análisis es necesario definir, a continuación, un sistema de comprobación o de inferencia de tipos. Esto suele implicar la generación de restricciones que han de ser resueltas posteriormente. En ocasiones es posible definir un sistema de inferencia que calcule tipos principales, como en el caso del análisis de uso. En otros, sin embargo, es necesario restringir el problema a la comprobación de tipos, como es el caso del análisis de terminación y productividad, en el que el usuario proporciona los tipos anotados y el sistema los comprueba.

2.3.2 El análisis de uso

El análisis de uso pretende determinar cuándo un valor se usa a lo sumo una vez, para evitar la actualización de clausuras cuando resulte innecesaria. Cada tipo es anotado, indicando por medio de un 1 que el correspondiente valor se usa a lo sumo una vez, y con w que puede ser usado muchas veces. Es decir, 1 y w actúan como cotas superiores del uso del valor. El problema de asignar estas anotaciones de uso a los programas no es trivial; no basta con ver si la variable aparece una sola vez en el programa. Consideremos por ejemplo, la siguiente expresión:

$$\begin{array}{l}\textbf{let}\ x = 1+2\\ \quad\ \ f = \lambda z.x+z\\ \textbf{in}\ f\ 3 + f\ 4\end{array}$$

Aunque x aparece sólo una vez en el cuerpo del **let** externo, no es en principio seguro reemplazar x por $1+2$ ya que el programa resultante calculará $1+2$ dos veces, con lo que tanto x como f deben recibir la anotación w.

Presentaremos primero el sistema de [TWM95], discutiendo sus problemas. Se trata de un sistema de tipos monomórficos. Para poder llevar a cabo inferencia se incluyen variables de uso j. Se utiliza el conjunto de restricciones Θ de la forma $j \leq \{k_1,\ldots,k_n\}$ definido por medio de las siguientes reglas:

$$\frac{}{k \leq_\Theta w}\ \textit{Omega} \qquad\qquad \frac{}{1 \leq_\Theta k}\ \textit{One}$$

$$\frac{}{k \leq_\Theta k}\ \textit{Refl} \qquad \frac{j \leq \{k_1,\ldots,k_n\} \in \Theta}{j \leq_\Theta k_i}\ \textit{Taut}$$

Los tipos anotados incluyen a los enteros y a las listas:

$$\tau \ ::= \ a^k \mid \tau \to^k \tau' \mid \mathit{Int}^k \mid [\tau]^k$$

Para devolver el uso asociado al tipo se utiliza $|\tau|$. Para que los tipos $[\tau]^k$ estén bien formados hace falta que $k \leq_\Theta |\tau|$. El lenguaje es un lambda cálculo con expresiones **let**, listas y **case** sobre listas. También dispone de expresiones **letrec** restringidas a ligar valores. Un contexto asocia un tipo anotado a cada variable. Los juicios de tipado son de la forma $\Gamma \vdash_\Theta e : \tau$, indicando que en el contexto Γ, y bajo las restricciones Θ, el término e tiene tipo τ. Las reglas de tipado se muestran en la Figura 2.6. Hay tres reglas estructurales: la regla de contracción $Cont$, la regla de debilitamiento *Weak* y la regla de intercambio $Exch$. Si una variable se utiliza más de una vez, se aplica la regla de contracción asignándole una anotación w. Veamos un ejemplo:

$$\dfrac{\dfrac{\dfrac{}{x : Int^w \vdash_\Theta x : Int^w}\,Var \qquad \dfrac{}{y : Int^w \vdash_\Theta y : Int^w}\,Var}{x : Int^w, y : Int^w \vdash_\Theta x + y : Int^j}\,Plus}{z : Int^w \vdash_\Theta z + z : Int^j}\,Cont$$

Como era de esperar, z adquiere la anotación w. La variable de uso j puede sustituirse por 1 o por w, dependiendo del uso que vaya a hacerse de la suma.

El hecho de que una variable no se usa nunca, se introduce mediante la aplicación de la regla de debilitamiento *Weak*. Esta regla no impone ninguna anotación para la variable, ya que cualquier uso es compatible con el hecho de no ser usada. La regla $Exch$ indica simplemente que el orden no importa dentro de los entornos. Obsérvese que en este sistema se usan en ocasiones contextos distintos en las distintas subexpresiones. Pero en el caso de la regla del **case** para listas hay que tipar las dos ramas con el mismo contexto, ya que solamente se evaluará una de las ramas. Así, por ejemplo, el siguiente tipado es válido:

$$xs : [Int^1]^1, y : Int^1 \vdash_\Theta \textbf{case}\ xs\ \textbf{of}\ nil \to y;\ cons\ x\ xs' \to x + y : Int^1$$

En la regla Abs, la restricción $k \leq |\Gamma|$ refleja el hecho de que, si se puede acceder a una función más de una vez, entonces se puede acceder más de una vez a cualquier variable libre suya. Las reglas Nil y $Cons$ asumen la condición $k \leq_\Theta |\tau|$ para que el tipo de las listas esté bien formado. El operador de suma es estricto, luego el resultado de $e_0 + e_1$ será siempre una constante entera que no tendrá referencias a los resultados de evaluar e_0 o e_1. Por ello, el uso de $e_0 + e_1$ no depende de los usos de e_0 y e_1. Todas las variables en un **letrec** se anotan con w.

Como ya se dijo anteriormente en la Sección 2.1.3, este sistema tiene un *problema de envenenamiento*. Consideremos la expresión

$$\begin{array}{l} \textbf{let}\ f = \lambda x.x + 1 \\ \quad\ \ a = 2 + 3 \\ \quad\ \ b = 5 + 6 \\ \textbf{in}\ a + (f\ a) + (f\ b) \end{array}$$

A simple vista se observa que el valor de a se demanda dos veces, pero el de b solamente una vez. Sin embargo, este sistema asigna tanto a a como a b el tipo Int^w. Esto sucede porque una vez se ha dado a a el tipo Int^w,

$$\frac{\Gamma, x:\tau, y:\tau \vdash_\Theta e:\tau' \quad |\tau| = w}{\Gamma, z:\tau \vdash_\Theta e[z/x, z/y]:\tau'}\ Cont \qquad \frac{\Gamma, x:\tau_0, y:\tau_1 \vdash_\Theta e:\tau}{\Gamma, y:\tau_1, x:\tau_0 \vdash_\Theta e:\tau}\ Exch$$

$$\frac{\Gamma \vdash_\Theta e:\tau'}{\Gamma, x:\tau \vdash_\Theta e:\tau'}\ Weak \qquad \frac{}{x:\tau \vdash_\Theta x:\tau}\ Var$$

$$\frac{\Gamma, x:\tau \vdash_\Theta e:\tau' \quad k \leq |\Gamma|}{\Gamma \vdash_\Theta \lambda x.e:\tau \to^k \tau'}\ Abs \qquad \frac{\Gamma \vdash_\Theta e:\tau \to^k \tau' \quad \Delta \vdash_\Theta x:\tau}{\Gamma, \Delta \vdash e\ x:\tau'}\ App$$

$$\frac{}{\vdash_\Theta n: Int^k}\ Int \qquad \frac{\Gamma \vdash_\Theta e_0: Int^{k_0} \quad \Delta \vdash_\Theta e_1: Int^{k_1}}{\Gamma, \Delta \vdash_\Theta e_0 + e_1: Int^k}\ Plus$$

$$\frac{}{\vdash_\Theta nil:[\tau]^k}\ Nil \qquad \frac{\Gamma \vdash_\Theta x:\tau \quad \Delta \vdash_\Theta y:[\tau]^k}{\Gamma, \Delta \vdash_\Theta cons\ x\ y:[\tau]^k}\ Cons$$

$$\frac{\Gamma \vdash_\Theta e_0:[\tau]^k \quad \Delta \vdash_\Theta e_1:\tau' \quad \Delta, x:\tau, y:[\tau]^k \vdash_\Theta e_2:\tau'}{\Gamma, \Delta \vdash_\Theta \mathbf{case}\ e_0\ \mathbf{of}\ nil \to e_1; cons\ x\ y \to e_2:\tau'}\ Case$$

$$\frac{\Gamma \vdash_\Theta e:\tau \quad \Delta, x:\tau \vdash_\Theta e':\tau'}{\Gamma, \Delta \vdash_\Theta \mathbf{let}\ x = e\ \mathbf{in}\ e':\tau'}\ Let$$

$$\frac{\Gamma, x:v \vdash_\Theta v:\tau \quad \Delta, x:\tau \vdash_\Theta e:\tau' \quad |\tau| = w}{\Gamma, \Delta \vdash_\Theta \mathbf{letrec}\ x = v\ \mathbf{in}\ e:\tau'}\ Letrec$$

Figura 2.6: Reglas de tipado para el análisis de uso la función f toma el tipo $Int^w \to \ldots$ y éste se propaga a b. La solución propuesta en [WJ99] consiste en definir una relación de subtipado (entre esquemas de tipo σ, puesto que este sistema es polimórfico en los tipos) y añadir la regla de subtipado:

$$\frac{\Gamma \vdash e : \sigma' \quad \sigma' \leq \sigma}{\Gamma \vdash e : \sigma}\ Sub$$

Se establece la relación $1 \leq w$ y la relación de subtipado. El orden bajo subtipado es el opuesto al orden de las anotaciones. A continuación se muestran las reglas para las anotaciones y para las funciones:

$$\frac{u_2 \leq u_1 \quad \tau_1 \leq \tau_2}{\tau_1^{u_1} \leq \tau_2^{u_2}}\ Annot \qquad \frac{\sigma_3 \leq \sigma_1 \quad \sigma_2 \leq \sigma_4}{\sigma_1 \to \sigma_2 \leq \sigma_3 \to \sigma_4}\ Arrow$$

Aplicándolas obtenemos, por ejemplo, $Int^w \leq Int^1$.

Este sistema se define además sobre un lenguaje semejante a Core que no solamente maneja listas sino tipos estructurados cualesquiera. Las reglas estructurales se eliminan y se unifican los contextos. En lugar de la regla de contracción se utiliza una función $occur(x, e)$ que cuenta el número de apariciones libres de la variable x en la expresión e. La regla de subtipado hace que las reglas dejen de estar dirigidas por la sintaxis, por lo que suele incorporarse dentro de las propias reglas, como en la siguiente regla para la aplicación:

$$\frac{\Gamma \vdash e_1 : (\sigma_1 \to \sigma_2)^u \quad \Gamma \vdash e_2 : \sigma_1' \quad \sigma_1' \leq \sigma_1}{\Gamma \vdash e_1\ e_2 : \sigma_2}\ App$$

Pero, aún con este sistema, hay problemas debido a la currificación. Por ejemplo, si consideramos la función $g = \lambda x.\lambda y.x + y - 1$, como g evalúa su argumento solamente una vez, esperamos el tipo $g :: (Int^1 \to (Int^1 \to Int^w)^w)^w$. Pero este tipo no es correcto. Supongamos que tenemos la expresión **let** $h = g\ a$ **in** $h\ 3 + h\ 4$. En este caso, a es demandada dos veces, una por cada llamada de h, luego el tipo de g es incorrecto, ya que permitiría proporcionar a a un tipo Int^1 en lugar del apropiado Int^w.

La solución es usar polimorfismo de anotaciones para expresar las dependencias. En el ejemplo, tendríamos $g : (\forall u.Int^u \to (Int^1 \to Int^w)^u)^w$. De esta forma tenemos dos ejemplares válidos: $g_1 :: (Int^w \to (Int^1 \to Int^w)^w)^w$ y $g_2 :: (Int^1 \to (Int^1 \to Int^w)^1)^w$. El primer ejemplar se puede usar para tipar h. El segundo ejemplar se puede utilizar en aquellos casos en los que el argumento se utiliza solamente una vez, obteniendo en tal caso que la aplicación parcial de g no puede ser compartida. En este sistema, la relación de subtipado se establece a través de la relación $w \leq 1$, pero sigue cumpliéndose $\tau^w \leq \tau^1$.

Una opción aún más general sería utilizar polimorfismo acotado y dar a g el tipo:

$$g :: (\forall u, v.u \leq v.Int^u \to (Int^1 \to Int^w)^v)^w$$

donde $w \leq 1$. Este tipo hace explícito que el uso de su primer argumento y su aplicación parcial no tienen por qué ser iguales; solamente hace falta que el uso del argumento sea mayor o igual que el de la aplicación parcial. Es el sistema desarrollado en [GS01], donde además se introduce recursión polimórfica en los tipos y en las anotaciones.

2.3.3 Tipos con tamaño

Describimos ahora la teoría de los tipos con tamaño, su sintaxis y su semántica. Desde un punto de vista semántico, la denotación de un tipo con tamaño es un subconjunto cerrado superiormente de un retículo, que puede no contener $\perp$, donde $\perp$ significa tanto no terminación (para tipos finitos) como bloqueo (para tipos infinitos). Luego, si en este sistema se puede dar un tipo a una función, se puede asegurar que o bien termina (si la función produce un valor finito) o es productiva (si produce un valor infinito). Con este propósito, se deberían distinguir los tipos finitos de los tipos infinitos. Adicionalmente, los tipos pueden tener uno o más *parámetros de tamaño*, que contienen información de tamaño. Estos pueden ser constantes, variables cuantificadas universalmente o, en general, expresiones restringidas. Intuitivamente, el tamaño de un valor de un cierto tipo es el número de iteraciones necesario para construir el valor. Por ejemplo, para una lista finita, es el número de constructores *cons* más uno (este último sumando corresponde al constructor *nil*). Para un árbol binario, es el número de niveles más uno, y así sucesivamente.

Sintaxis de los tipos con tamaño y de los tipos de datos

Sintácticamente, los tipos finitos no recursivos se introducen mediante una declaración **data**, los tipos finitos recursivos mediante una declaración **idata** y los infinitos mediante una declaración **codata**. En la Figura 2.7 se muestra la sintaxis de las signaturas y declaraciones de tipos. Se usan τ, σ, s, k para denotar tipos, esquemas de tipo, expresiones de tamaño y variables de tamaño respectivamente. Para representar variables de tipo utilizaremos t, a, b, c.

Una expresión de tamaño puede ser finita, denotada por i, oinfinita, denotada por ω. En el primero de los casos, obsérvese que están restringidas a ser expresiones sobre números naturales, lineales en un conjunto de variables de tamaño. Las variables de tamaño y de tipo pueden estar cuantificadas universalmente en un esquema de tipo y en una declaración de tipo. Hay algunas restricciones adicionales (por ejemplo que los constructores deben ser únicos, o que no están permitidas las definiciones mutuamente recursivas), las cuales se pueden comprobar estáticamente. Dichas restricciones tienen por objeto que los tipos tengan una semántica correcta. Véase [Par00] para más detalles.

Algunos ejemplos de declaraciones válidas de tipo son:

$\tau ::= t \mid \tau \to \tau \mid T\ \overline{s}\ \overline{\tau}$	$D ::= \textbf{data}\ E \mid \textbf{idata}\ E \mid \textbf{codata}\ E$
$\sigma ::= \forall t.\tau \mid \forall k.\tau \mid \tau$	$E ::= L = R \mid \forall t.E \mid \forall k.E$
$s ::= w \mid i$	$L ::= T\ \overline{s}\ \overline{\tau}$
$i ::= k \mid n \mid p * i \mid i + i$	$R ::= c_1\ \overline{\tau_1} \mid \ \ldots\ \mid c_n\ \overline{\tau_n}$

Figura 2.7: Sintaxis de los tipos con tamaño y de las definiciones de tipos de datos

$$\begin{array}{l}
\textbf{data}\ \texttt{Bool} = \texttt{true} \mid \texttt{false} \\
\textbf{idata}\ \forall a\ .\ \textbf{List}\ \omega\ a = \texttt{nil} \mid \texttt{cons}\ a\ (\textbf{List}\ \omega\ a) \\
\textbf{idata}\ \texttt{Nat}\ \omega = \texttt{zero} \mid \texttt{succ}\ (\texttt{Nat}\ \omega) \\
\textbf{codata}\ \forall a\ .\ \textbf{Strm}\ \omega\ a = \texttt{make}\ a\ (\textbf{Strm}\ \omega\ a)
\end{array}$$

representando el tipo de los booleanos, las listas finitas de cualquier tamaño, los números naturales de cualquier tamaño, y los *streams* de cualquier tamaño respectivamente. Ejemplos de esquemas válidos de tipo son:

$$\begin{array}{l}
\textbf{List}\ 3\ (\texttt{Nat}\ \omega) \\
(a \to b) \to \textbf{Strm}\ k\ a \to \textbf{Strm}\ k\ b \\
\forall a.\forall k\ .\ \textbf{List}\ k\ a \to \texttt{Nat}\ k
\end{array}$$

Semántica de los tipos con tamaño

El primer parámetro de un tipo **idata** o uno **codata** es una expresión de tamaño que acota el tamaño de los valores de ese tipo. Para los tipos **idata** es una *cota superior*, y para los tipos **codata** una *cota inferior*. Así, el tipo `List 2 a` denota a las listas conteniendo cero o un valor (es decir, con a lo sumo dos constructores), mientras que `Strm 2 a` denota los streams de dos o más valores (es decir, que contienen al menos dos constructores `make`). Los streams parciales, como $1; 2; \bot$, y los streams infinitos, como $1; 2; \ldots$, (donde ; es la versión infija de `make`) pertenecen a este tipo. Este parámetro de tamaño puede concretarse con el tamaño infinito ω. Para los tipos **idata**, `List` ω `a` corresponde a listas finitas de *cualquier* tamaño, mientras que para tipos **codata**, como `Strm` ω `a` representa los streams estrictamente *infinitos*.

El análisis se define sobre un lenguaje llamado Haskell Síncrono. Un programa consiste en un conjunto de declaraciones de tipos bien formadas, seguidas de un término escrito en un λ-cálculo enriquecido con aplicaciones de constructores, y expresiones **case** y **letrec**, cada una con un conjunto de ligaduras simples mutuamente recursivas (ver Sección 6.1). Para preservar la corrección de los tipos, se añade la restricción de que todos los correspondientes constructores del tipo deben aparecer una única vez en cada una de las ramas de cada expresión **case**.

El universo de valores $\mathbf{U}$ se define como la solución de la ecuación: $\mathbf{U} \sim [\mathbf{U} \to \mathbf{U}]_\bot \oplus (\mathbf{U} \times \mathbf{U})_\bot \oplus \mathbf{1}_\bot \oplus\ CON_\bot$, donde CON es el conjunto de constructores. Sobre este universo, se proporciona una semántica estándar no estricta del λ-cálculo enriquecido (ver [Par00]). El conjunto de tipos $\mathbf{T} = \{\mathcal{T} \mid \mathcal{T}$ es un subconjunto de $\mathbf{U}$ cerrado superiormente$\}$ forma un retículo completo bajo la relación de inclusión $\subseteq$, en el que $\top_{\mathbf{T}} = \mathbf{U}$ es el supremo, $\perp_{\mathbf{T}} = \emptyset$ el ínfimo, y $\cup$, $\cap$ son los operadores de mínima cota superior y máxima cota inferior respectivamente. Se trata de un cpo pero no es un dominio. Puesto que los tipos son subconjuntos de $\mathbf{U}$ cerrados superiormente, el único tipo que contiene $\perp_{\mathbf{U}}$ es precisamente $\mathbf{U}$.

En este cpo, los tipos recursivos se interpretan como puntos fijos de funcionales $\mathcal{F} : \mathbf{T} \to \mathbf{T}$. Para los tipos inductivos (los definidos por una declaración **idata**) la interpretación es el mínimo punto fijo de una función continua, mientras que para los tipos coinductivos (los definidos por una declaración **codata**) es el mayor punto fijo de una función cocontinua. Dichos puntos fijos se alcanzan por medio de la cadena ascendente $\mathcal{F}^i(\perp_{\mathbf{T}})$ y de la cadena descendente $\mathcal{F}^i(\top_{\mathbf{T}})$. Los funcionales se definen a partir de las declaraciones de tipo. Para ello se definen varios operadores sobre tipos: $\boxed{\times}$ para el producto cartesiano no estricto de tipos, $\boxed{+}$ para la suma de tipos, y $\boxed{\rightarrow}$ para las funciones entre tipos. Se demuestra que dichos operadores preservan las propiedades de cierre superior. Así, por ejemplo, los funcionales correspondientes a los tipos definidos anteriormente son

$$
\begin{array}{l}
F_{Bool} = true \boxed{+} false \\
F_{Nat} = \lambda\mathcal{T} zero \boxed{+} succ \boxed{\times} \mathcal{T} \\
F_{List} = \lambda\mathcal{T}_a.\lambda\mathcal{T}.nil \boxed{+} cons \boxed{\times} \mathcal{T}_a \boxed{\times} \mathcal{T} \\
F_{Strm} = \lambda\mathcal{T}_a.\lambda\mathcal{T}.make \boxed{\times} \mathcal{T}_a \boxed{\times} \mathcal{T}
\end{array}
$$

De aquí en adelante usaremos una línea superior para representar secuencias finitas de elementos. Así, $\overline{s}$ representa una secuencia $s_1 \ldots s_l$ con $l \geq 0$.

La denotación de un constructor de tipo recursivo $T\ \overline{s}\ \overline{\tau}$ es una función que toma como parámetros $l-1$ naturales correspondientes a los tamaños $s_2, \ldots, s_l$, y j tipos correspondientes a los tipos $\tau_1, \ldots, \tau_j$, y devuelve un *iterador de tipo* $\mathcal{F} : \mathbf{T} \to \mathbf{T}$. Este toma como parámetro un tipo correspondiente a las apariciones recursivas del tipo que está siendo definido, y devuelve un nuevo tipo correspondiente al lado derecho de su definición. El primer parámetro de tamaño s_1 determina el número de veces que se aplica el iterador (a $\perp_{\mathbf{T}}$ para los tipos **idata** o a $\top_{\mathbf{T}}$ para los **codata**). Si se aplica k veces, se obtienen valores de tamaño a lo sumo k (resp. al menos k, para los **codata**). En el límite, se obtienen valores de cualquier tamaño, es decir el punto fijo del iterador. Si todas las declaraciones respetan las restricciones semánticas estáticas mencionadas anteriormente, los iteradores de tipo resultantes son continuos para las declaraciones **idata** y cocontinuos para las **codata**.

Así, el significado de **List** 3 *Bool* viene dado por $(F_{List}(F_{Bool}))^3(\perp_{\mathbf{T}})$ y el de **List** ω *Bool* por $(F_{List}(F_{Bool}))^\omega(\perp_{\mathbf{T}}) = \bigcup_{i \in \mathbb{N}} (F_{List}(F_{Bool}))^i(\perp_{\mathbf{T}})$.

El significado de **Strm** 3 *Bool* viene dado por $(F_{Strm}(F_{Bool}))^3(\top_{\mathbf{T}})$ y el de **Strm** w *Bool* por $(F_{Strm}(F_{Bool}))^w(\top_{\mathbf{T}}) = \bigcap_{i \in \mathbb{N}}(F_{Strm}(F_{Bool}))^i(\top_{\mathbf{T}})$.

Se puede definir una relación de subtipado $\tau \rhd \tau'$ entre los tipos con tamaño con el mismo tipo subyacente pero distinto tamaño. Se trata de una relación basada en subconjuntos, es decir, $\tau \rhd \tau'$ se tiene cuando la interpretación de τ es un subconjunto de la interpretación de τ'. Es una relación monótona con respecto a los tamaños para los tipos **idata**, mientras que es antimonótona para los **codata**. Por ejemplo, **List** $2\ a \rhd$ **List** $3\ a$, mientras que **Strm** $3\ a \rhd$ **Strm** $2\ a$. En [Par00] se presentan las reglas para comprobar esta relación de subtipado. Proporcionan una forma de debilitar la información de tamaño. Este debilitamiento se aplicará en las reglas de la aplicación $[APP]$, de la expresión **let** $[LET]$, de la expresión **letrec** $[LETREC]$ y de la expresión **case** $[CASE]$ (ver Figura 6.1 en la Sección 6.1).

Como es habitual, la interpretación del polimorfismo de tipos se obtiene por medio de la intersección de las interpretaciones de cada ejemplar. En términos formales, si $\mathcal{I}$ denota la función de interpretación, γ es un entorno de tipos que asocia tipos en $\mathbf{T}$ a las variables de tipo, y δ es un entorno de tamaños que asocia tamaños en $\mathbb{N}^\omega$ a las variables de tamaño, entonces

$$\mathcal{I}[\![\forall t.\sigma]\!]\ \gamma\ \delta = \bigcap_{\mathcal{T} \in \mathbf{T}} \mathcal{I}[\![\sigma]\!]\ \gamma[\mathcal{T}/t]\ \delta.$$

Análogamente, el polimorfismo de tamaño se interpreta como la intersección de la interpretación para todos los tamaños:

$$\mathcal{I}[\![\forall k.\sigma]\!]\ \gamma\ \delta = \bigcap_{n \in \mathbb{N}} \mathcal{I}[\![\sigma]\!]\ \gamma\ \delta[n/k],$$

si bien, en este caso la intersección está restringida a los tamaños finitos, para poder usar la inducción sobre los números naturales para asignar tipos a las definiciones recursivas (véase la regla $[LETREC]$ en la Sección 6.1). Esta restricción provoca que solamente sea segura la concreción de tipos polimórficos con tamaños finitos; la *concreción omega* no es siempre segura. De hecho, hay algunos tipos para los que la concreción con ω produce un tipo más pequeño que el polimórfico, en lugar de uno mayor. Un ejemplo es $\forall k.($**Strm** $\omega\ (Nat\ k) \to Bool)$, al sustituir k por ω. En [Par00], se proporciona una condición decidible suficiente (que un esquema de tipo σ sea *undershooting* con respecto a una variable de tamaño k, denotado por $\sigma \overset{\cup}{\sim} k$) para que un esquema se pueda concretar con seguridad con ω.

En la Sección 6.1 se presentan las reglas de tipado para Haskell Síncrono. En ellas se controla la concreción de los tipos polimórficos (regla de la variable), se aplica la relación de subtipado (reglas de la aplicación, **let**, **letrec** y **case**) y se utiliza la inducción sobre tamaños para demostrar los tipos de las definiciones recursivas (regla del **letrec**).

Capítulo 3

El lenguaje funcional paralelo Edén

Edén es un lenguaje funcional para la programación concurrente y paralela, definido como una extensión del lenguaje funcional Haskell, mediante construcciones para la especificación explícita de procesos. Puede ser utilizado para programar sistemas tanto transformacionales como reactivos. En este capítulo se describen las características sintácticas y semánticas del lenguaje, así como algunos detalles de su implementación. En [BLOMP98] se puede encontrar una descripción detallada del lenguaje y de su semántica operacional.

3.1 Fundamentos de Edén

Con el objetivo de facilitar la tarea de desarrollar programas concurrentes, en Edén los procesos, la comunicación y la sincronización son conceptos tratados de forma abstracta. Más concretamente, Edén extiende el lenguaje funcional perezoso Haskell [Pe97] mediante construcciones para definir procesos de forma *explícita*. El lenguaje permite definir redes estáticas de procesos pero también sistemas de procesos que evolucionan dinámicamente. Estos procesos se comunican entre sí intercambiando valores a través de canales de comunicación modelados mediante listas perezosas. La comunicación es *asíncrona* e *implícita*, es decir, los datos se envían sin necesidad de esperar la aceptación del receptor y el paso de mensajes es automático, sin ser necesaria su especificación por parte del programador mediante instrucciones del estilo *send* y *receive*.

Por tanto, en Edén se pueden distinguir dos niveles: el nivel de las funciones y el nivel de los procesos. Podemos decir que Haskell constituye el *modelo de cómputo* de Edén, el cual es extendido mediante un *modelo de coordinación*, que incorpora procesos con un estilo funcional.

Edén admite no determinismo dentro de los procesos, siendo el proceso predefinido `merge` la única fuente de no determinismo. Su utilización permite la definición de sistemas reactivos. Como veremos en el Capítulo 5, se puede determinar qué partes de un programa son deterministas y cuáles se pueden ver afectadas por el no determinismo.

Mientras que la mayoría de los lenguajes funcionales concurrentes utilizan memoria compartida, Edén está diseñado para trabajar de forma eficiente sobre arquitecturas distribuidas. Aun así, también se puede implementar sobre memoria compartida, siempre que esté soportada la librería estándar MPI de paso de mensajes.

3.2 Sintaxis

Edén distingue entre *abstracciones de procesos* (que especifican el comportamiento de un proceso con un estilo funcional) y *concreciones de procesos* (en las que se proporcionan valores de entrada a las abstracciones de procesos, con el objetivo de crear nuevos procesos). Así pues, la relación entre las abstracciones y las concreciones de procesos es la misma que entre las λ-abstracciones y la aplicación de las mismas, una vez que reciben sus parámetros reales.

Las abstracciones de procesos son valores de primera clase, es decir, pueden usarse igual que cualquier otro valor: pueden utilizarse como parámetros de una función, almacenarse en estructuras de datos, ser el resultado devuelto por una función, etc.

Un proceso que toma como entradas $in_1, \ldots, in_m$ y produce como salidas $exp_1, \ldots, exp_n$, se especifica mediante una expresión de *abstracción de proceso* de la siguiente forma:

$$\begin{array}{l} \textbf{process}\ (in_1, \ldots, in_m) \rightarrow (exp_1, \ldots, exp_n) \\ \textbf{where}\ ecuacion_1\ \ldots\ ecuacion_r \end{array}$$

Su tipo es $Process\ (t_1, \ldots, t_m)\ (t'_1, \ldots, t'_n)$, donde *Process* es un constructor de tipos binario predefinido, y $t_1, \ldots, t_m$ y $t'_1, \ldots, t'_n$ son los tipos de las entradas y de las salidas respectivamente. La parte **where** es opcional, y se utiliza para definir funciones auxiliares y subexpresiones comunes que se utilizan en la definición del proceso.

No es necesario que se nombren explícitamente todos y cada uno de los canales de entrada y todas y cada una de las expresiones de salida, sino que para especificar los canales de entrada puede utilizarse cualquier ajuste de patrones (incluyendo una única variable), mientras que la salida del proceso puede ser cualquier expresión. La única restricción impuesta es que los tipos sean correctos:

$$\begin{array}{l} \textbf{process}\ entradas \rightarrow expresion \\ \textbf{where}\ ecuaciones \end{array}$$

Un proceso puede tener como entrada (respectivamente, salida) un único canal o una tupla de canales. Supongamos que un proceso tiene tipo *Process* t_1 t_2. Si el tipo t_1 o t_2 es una tupla, entonces cada una de las componentes de dicha tupla se considera un canal independiente. Si un canal tiene tipo $[t_3]$ se tratará como una lista potencialmente infinita, (*stream*) que transmite su contenido elemento a elemento. Por ejemplo, el tipo *Process* $[\mathit{Int}]$ $([\mathit{Int}], \mathit{Int})$ representa un proceso con un canal de entrada, por el que recibe enteros de uno en uno, y dos canales de salida, uno por el que produce enteros de uno en uno, y otro por el que produce un único entero. Sin embargo, si la lista no aparece al nivel más externo, no representa un *stream*, sino que representa un valor que se transmite completo de una sola vez. Por ejemplo, *Process Int* $[[\mathit{Int}]]$ representa un proceso con un canal de entrada por el que recibe un entero y un canal de salida, por el que envía listas completas de enteros, una a una.

Si se desea expresar una estructura de canales, por ejemplo una lista de canales, es necesario utilizar anotaciones de canales. La anotación $\langle a \rangle$ expresa que a es un canal. Por ejemplo, la anotación $[\langle a \rangle]$ significa "lista de canales", que se utiliza para definir abstracciones de procesos con un número de canales que se desconoce en tiempo de compilación. Así, el tipo *Process Int* $[\langle [\mathit{Int}] \rangle]$ representaría un proceso con un canal de entrada por el que recibe un entero, y una lista de canales de salida por los que envía elementos, uno a uno. Dichas anotaciones de canales aún no se han implementado, por lo que excepto en el caso del proceso predefinido `merge`, explicado más abajo, no las tendremos en cuenta.

Ejemplo 23 La siguiente abstracción de proceso especifica un proceso que mezcla dos listas ordenadas para obtener una nueva lista ordenada:

```
merger :: Ord a => Process ([a],[a]) [a]
merger = process (s1,s2) -> smerge s1 s2
    where smerge [] l = l
          smerge l [] = l
          smerge (x:l) (y:t) = if x <= y then x:smerge l (y:t)
                                         else y:smerge (x:l) t
```

□

La creación de procesos en tiempo de ejecución tiene lugar en el momento de aplicación de una abstracción de proceso a una tupla de expresiones de entrada, lo que recibe el nombre de *concreción o lanzamiento de proceso*. El proceso donde se lleva a cabo la concreción recibe el nombre de *proceso padre*, y el nuevo proceso creado recibe el nombre de *proceso hijo*. La aplicación da lugar a una tupla de canales de salida producida por el nuevo proceso:

$$(out_1, \ldots, out_n) = p \ \# \ (input\_exp_1, \ldots, input\_exp_m)$$

Al igual que para las abstracciones de proceso, no es necesario que las concreciones mencionen explícitamente cada uno de los canales de entrada y de salida, y tampoco es necesario que p sea una variable que haya sido declarada directamente como una abstracción de proceso, sino que puede ser una expresión cualquiera. La única restricción que se impone es que los tipos sean correctos. El tipo de # es

$$\#:: (\mathit{Transmissible}\ a, \mathit{Transmissible}\ b) \texttt{ => } \mathit{Process}\ a\ b \texttt{ -> } a \texttt{ -> } b$$

donde la clase `Transmissible` se utiliza para garantizar que existe un método para transmitir a través de canales los valores de entrada y salida del proceso. Por lo tanto se permite cualquier expresión

$$out = e_1 \ \# \ e_2,$$

siempre que esté bien tipada. De hecho, # es un valor de primera clase, por lo que, entre otras cosas, puede pasarse como parámetro de una función de orden superior, como por ejemplo en

$$zipWith\ (\#)\ ps\ ins.$$

Ejemplo 24 Concretando la abstracción de proceso `merger` del ejemplo anterior puede crearse una red dinámica de ordenación:

```
sortNet :: (Ord a, Transmissible a) => Process [a] [a]
sortNet = process list -> sort list
          where sort []  = []
                sort [x] = [x]
                sort xs  = merger # (sortNet # l1, sortNet # l2)
                           where (l1,l2) = unshuffle xs
                unshuffle []  = ([],[])
                unshuffle [x] = ([x],[])
                unshuffle (x:y:t) = (x:t1,y:t2)
                                  where (t1,t2) = unshuffle t
                merger = ...
```

Si la lista en cuestión tiene más de dos elementos, se divide en dos sublistas sobre las que se generan nuevas concreciones de procesos `sortNet` y `merger`. La topología resultante es un árbol binario de procesos.

□

Las construcciones presentadas hasta el momento son suficientes para programar sistemas concurrentes deterministas, pero es necesario introducir nuevos conceptos no-funcionales para extender el poder expresivo del lenguaje de modo que puedan definirse también sistemas reactivos.

En Edén, el *no determinismo* se introduce mediante la abstracción de proceso predefinida `merge`, que representa un proceso que mezcla de forma arbitraria pero *justa* una lista de canales, a través de cada uno de los cuales se recibe una lista:

$$merge :: Process\ [\langle[a]\rangle]\ [a]$$

Se trata de un proceso reactivo, cuya implementación consiste en que cuando aparece un valor disponible en uno de sus canales de entrada, lo copia inmediatamente en la lista de salida.

En el Capítulo 5 las anotaciones de canales son irrelevantes, ya que los procesos se tratan como funciones y no hay distinción entre canales y valores. Por ello consideraremos que el tipo de `merge` es $Process\ [[a]]\ [a]$. Sin embargo, en el Capítulo 6 será necesario tener en cuenta el hecho de que cada componente de la lista interna representa un canal para proporcionar a `merge` un tipo con tamaño. En el resto de casos se considerará siempre un valor de tipo lista.

3.3 Semántica

La semántica operacional de Edén es una extensión de la semántica operacional estándar de Haskell, y refleja la distinción entre los sublenguajes de cómputo y de coordinación que conviven en Edén. Comprende dos niveles de sistemas de transiciones: el nivel inferior maneja aquellos efectos que son locales a un único proceso; mientras que el nivel superior describe los efectos que son globales al sistema completo (la creación de un nuevo proceso, el envío de un dato, la recepción de un dato, y la finalización de una hebra). La interfaz entre ambos niveles está formada por "acciones" que comunican al nivel superior la necesidad de un evento global, como, por ejemplo, la creación de un proceso, el envío de un mensaje o la terminación de un proceso. Los detalles concretos formales de dicha semántica pueden encontrarse en [BLOMP98]. En esta sección se describe de manera menos formal los aspectos fundamentales de la semántica de Edén.

Impaciencia vs. pereza. Aunque el modelo computacional de Edén es un lenguaje perezoso (Haskell), se ha optado por no mantener dicha pereza en el modelo de coordinación, con el objetivo de mejorar el comportamiento paralelo de los programas. Así, la pereza se pierde en dos situaciones:

- Una vez lanzado un proceso, su cómputo está dirigido por la evaluación de sus expresiones de salida, para las cuales existe siempre demanda. De esta forma, se incrementa el grado de paralelismo, puesto que los procesos pueden producir datos de forma independiente, sin necesidad de esperar a que nadie los demande.

- Se permite que un proceso se lance antes de ser demandado, con el objetivo de acelerar la distribución del cómputo. El lanzamiento impaciente de procesos se lleva a cabo mediante una transformación durante el proceso de compilación de Edén presentada en [PPRS00a].

Las dos reglas anteriores pueden conducir a que se realice algún trabajo innecesario para el resultado final, pero éste es un riesgo compensado por la ventaja de aumentar el grado de paralelismo de los programas, si bien el programador deberá tenerlo siempre en mente a la hora de utilizar el lenguaje.

Reparto de trabajo. Con el objeto de poder razonar acerca del comportamiento paralelo de los programas, debe quedar claro qué partes del cómputo deben realizarse en cada proceso. Cuando se evalúa una expresión $e_1\#e_2$ la expresión e_2 se evalúa de forma impaciente en el proceso padre. Si es de tipo tupla se crea una hebra concurrente independiente para evaluar cada componente.

Todo el cómputo necesario para evaluar e_1 se lleva a cabo en el proceso hijo. Esto incluye no sólo la aplicación de e_1 a los valores de entrada correspondientes a e_2, sino también la evaluación de las variables libres de e_1 (en caso de que sea preciso evaluarlas). Por ejemplo, en

```
p :: Int -> Int -> Process Int Int
p x y = process i -> x + y + i
result = p (fib 5) (fib 6) # (fib 7)
```

el padre sólo se encargará de evaluar `fib 7`, mientras que será el proceso hijo quien evalúe no sólo la suma final, sino también `fib 5` y `fib 6`.

Una vez creado, el proceso hijo comienza a producir de forma impaciente su expresión de salida. Para cada una de las expresiones de salida de un proceso, se creará un flujo de ejecución concurrente distinto, puesto que, en principio, dichas evaluaciones son independientes. Por lo tanto, en Edén se distinguen dos niveles de concurrencia: la concurrencia y evaluación paralela de los procesos, y la evaluación concurrente de las distintas hebras de cada proceso.

Debido a la copia de variables libres del proceso padre al proceso hijo, existe riesgo de duplicación de la evaluación. Por ejemplo, en

```
let
  x1 = fib 20
  fib x = ...
in
  (p x1) # x1
```

se copian las clausuras asociadas a `p`, `x1` y `fib`, así como aquellas de las que dependiera `fib`. Es decir, se copian todas las variables libres, y todas las variables libres de dichas variables libres, y así sucesivamente[1]. En este caso `x1` es evaluado tanto por el padre como por el hijo, es decir su evaluación se ha duplicado.

[1]Las variables que se definen en el nivel más externo del programa no se consideran libres, por lo que no es necesario copiarlas. La razón es que residen en todos los procesadores, por lo que basta con indicar cuáles son.

Transmisión de valores. La comunicación se lleva a cabo mediante canales unidireccionales 1:1. Una vez un proceso se encuentra en ejecución, sólo se comunican datos completamente evaluados. La única excepción corresponde a las listas: estas se transmiten en forma de *stream*, es decir, elemento a elemento, de modo que no es necesario evaluar la lista entera antes de enviar sus componentes: se va evaluando cada elemento a forma normal y después se transmite. La transferencia de información a través de los canales de comunicación se realiza de forma automática, sin necesidad de instrucciones explícitas al estilo *send* y *receive*.

Terminación. Dado que el cómputo de un proceso viene dirigido por la evaluación de sus canales de salida, un proceso terminará de forma inmediata en cuanto no tenga ningún canal de salida, bien porque haya terminado el cómputo asociado a dicho canal o porque haya sido cerrado desde el proceso receptor del mismo. Al terminar, sus canales de entrada dejarán de ser útiles, por lo que se eliminarán, y los correspondientes canales de otros procesos que alimentasen dichas entradas serán cerrados, propagándose así la terminación de unos procesos a otros.

Sincronización. Cuando una hebra concurrente de un proceso necesite un determinado valor de un canal de entrada, pero dicho valor aún no haya sido recibido, la evaluación de dicha hebra será suspendida hasta que el correspondiente emisor produzca y envíe el dato. Obsérvese que la comunicación a través de canales se realiza utilizando envío no-bloqueante, pero recepción bloqueante, y que la sincronización entre los procesos se produce exclusivamente mediante el intercambio de información a través de los canales.

3.4 Implementación

El primer compilador de Edén (MEC: Marburg-Madrid Eden Compiler) reutiliza el código fuente de GHC [PHH+93, Jon96], ya que Edén es una extensión de Haskell. De esta forma todas las funcionalidades de GHC quedan cubiertas.

La elección de GHC como punto de partida para el desarrollo de MEC se debe a su eficiencia, fiabilidad y disponibilidad. Su código fuente puede obtenerse en `http://www.haskell.org`.

Actualmente está disponible una primera versión de Edén que implementa casi todas las características del lenguaje, con sólo dos excepciones: aún no se han implementado las anotaciones de canales que permiten generar estructuras de canales; y todavía no se realiza conexión directa (*bypassing*) automática de canales.

A continuación se describen el proceso de compilación de Edén y el de GHC, en el que está basado. En [KOMP99] se puede encontrar una descripción del *RunTime System* (RTS) de MEC a nivel abstracto, y en [Klu98] pueden verse los detalles de su implementaión real.

3.4.1 El proceso de compilación de GHC

El proceso de compilación de GHC está dividido en las fases que pueden apreciarse en la Figura 3.1. El código fuente está escrito por completo en Haskell, con la excepción del analizador sintáctico (escrito en Lex/Yacc y C) y del RTS (escrito en C).

Existe una primera fase para convertir Haskell literario[2] (*literate*) en Haskell, en el caso en el que el programa original estuviese escrito utilizando ese modo. A continuación, el analizador sintáctico genera el árbol de sintaxis abstracta correspondiente al programa de entrada. Dado que el analizador genera C, y el resto del compilador está escrito en Haskell, es necesaria una fase intermedia (*reader*), que se encarga de resolver las precedencias de los operadores infijos que se hayan utilizado en el programa.

La fase de renombramiento (*renamer*) resuelve conflictos debidos al ámbito de visibilidad de los identificadores, especialmente en lo concerniente a las importaciones y exportaciones de módulos.

La fase de inferencia de tipos (*typechecker*) anota el programa con la información de tipos correspondiente, eliminando toda la sobrecarga que pudiera existir.

El "desdulcificador" (*desugarer*) traduce la rica sintaxis abstracta de Haskell a un lenguaje funcional mucho más simple denominado *Core*. Nótese que esta fase es posterior a la inferencia de tipos, de modo que los mensajes de errores de tipos pueden ofrecerse relacionados con el código fuente, haciéndolos más fácilmente comprensibles por el programador.

Posteriormente, se realizan (de forma opcional) un conjunto de transformaciones dentro del lenguaje Core, con el objetivo de mejorar la eficiencia del código generado.

A continuación, el programa Core se traduce a otro lenguaje funcional más simple: *STG*[3], donde se realizan (opcionalmente) otras transformaciones.

Finalmente se realiza la generación de código. GHC genera C como lenguaje final, de modo que posteriormente será necesario procesarlo con un compilador de C. Ahora bien, con el objeto de mejorar la eficiencia sin perder la portabilidad, tanto a distintos compiladores de C como a distintas arquitecturas, en lugar de generarse directamente código C, se genera un

[2]El modo literario da preferencia a los comentarios sobre el código fuente, de modo que, por defecto, todo son comentarios, y para introducir código hay que utilizar notación especial.

[3]Shared Term Graph Language.

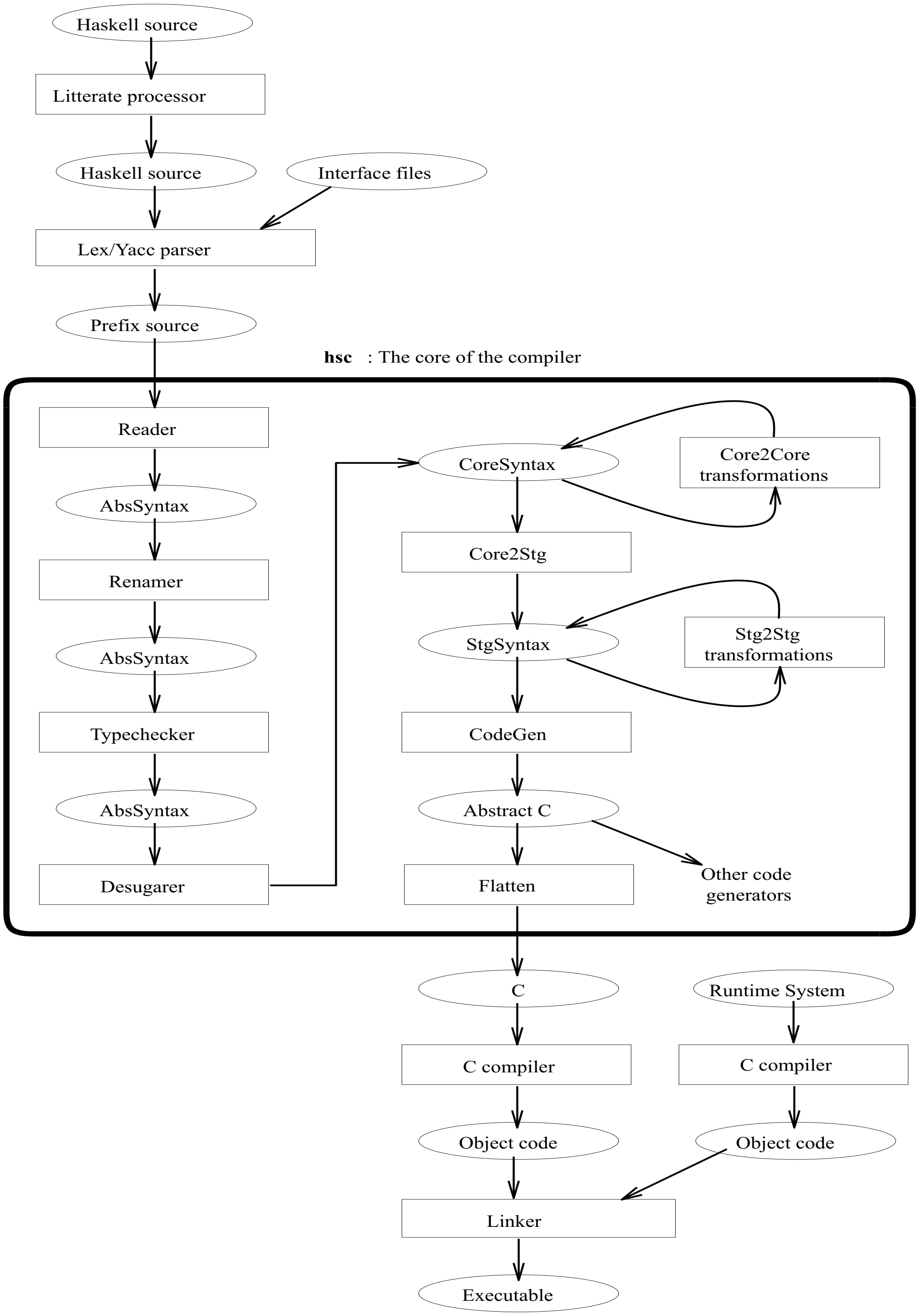


Figura 3.1: Estructura de GHC

árbol de sintaxis abstracta C. Así, una última fase (que puede ser distinta para cada tipo de arquitectura) traduce éste a C, optimizando los recursos de que se disponga.

Una vez generado el código C, se enlaza con el código del RTS, obteniéndose el programa ejecutable.

El código fuente correspondiente al RTS (escrito en C) es, en cierto sentido, independiente del proceso de compilación propiamente dicho. Existen distintos RTS en función del uso que se esté haciendo de GHC. Por ejemplo, existe un RTS para Haskell, otro que incluye además las características necesarias para realizar perfiles secuenciales, un RTS para cuando se utiliza GpH [THM+96], otro para cuando se está trabajando con GranSim [HLP95, Loi96], etc. Estos RTS comparten ciertas partes de su código fuente, mientras que otras son propias de cada uno de ellos.

3.4.2 El proceso de compilación de MEC

La idea fundamental en el desarrollo de MEC es modificar GHC lo mínimo posible. En la Figura 3.2 se muestra el proceso de compilación de MEC.

La primera fase que hace falta modificar es el analizador léxico y sintáctico, ya que es necesario tener en cuenta las nuevas construcciones. Sin embargo, con el objeto de evitar modificaciones en fases posteriores del compilador, no se modifica la sintaxis abstracta generada por el analizador. Para ello, es necesario "esconder" las construcciones Edén dentro de la sintaxis de Haskell. Esto se consigue utilizando funciones predefinidas escritas en Haskell. Así, por ejemplo, cuando el analizador detecta la construcción

```
process patron -> expresion
```

genera como salida la aplicación de la función predefinida `process` a una función

```
process (λ patron -> expresion).
```

Es importante resaltar la gran diferencia existente entre el primer `process` y el segundo. El primero corresponde a una palabra reservada en Edén, mientras que el segundo es el nombre de una función Haskell, que está definida en un módulo que puede considerarse como el preludio de Edén. En dicho módulo, la definición de `process` (y del resto de funciones básicas) se realiza de modo que el sistema de tipos de GHC sea capaz de inferir correctamente los tipos de las expresiones de Edén.

Para ello, basta con seguir las siguientes declaraciones de tipos:

```
data Process a b = Process (a->b)
process :: (a -> b) -> Process a b
(#)::(Transmissible a,Transmissible b) => Process a b -> a -> b
```

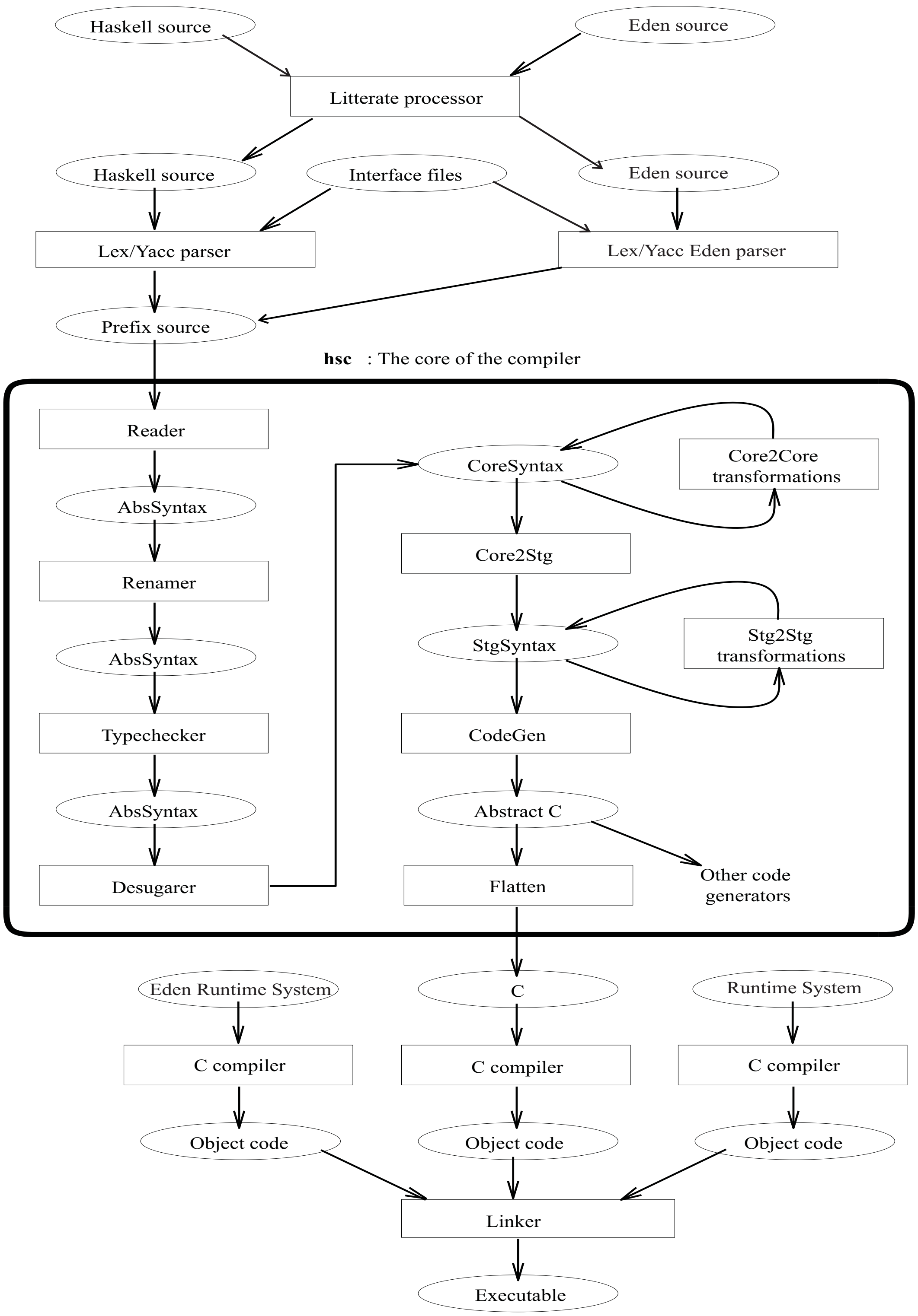


Figura 3.2: Estructura de MEC

Realizando traducciones similares para el resto de las construcciones Edén, se consigue "engañar" a GHC para que las procese.

Las funciones del preludio Edén invocan a otras funciones primitivas (escritas en C) que se introducen directamente en el código del RTS. Así, por ejemplo, cuando se evalúe $e_1\#e_2$ se llamará a una función primitiva, que hará que el RTS cree un nuevo proceso para evaluar e_1, así como que se creen nuevas hebras para evaluar e_2.

Así se consigue un alto grado de reutilización del código fuente de GHC. Ahora bien, existen algunas transformaciones automáticas que serían deseables en Edén, pero que no están presentes en GHC, como es el caso del lanzamiento impaciente de procesos [PPRS00a], por lo que es preciso añadirlas en la fase de transformaciones a nivel Core. Además, desafortunadamente, algunas de las transformaciones automáticas que GHC efectúa al nivel del lenguaje Core no mejoran la eficiencia de los programas Edén sino que, por el contrario, pueden reducirla notablemente. Es más, determinadas transformaciones no sólo pueden empeorar la eficiencia sino que pueden llegar a cambiar la semántica del programa Edén original. En la Sección 5.2 se presentan estas transformaciones. Algunas de ellas se ven afectadas por la presencia del no determinismo, lo que motiva el análisis del Capítulo 5. El objetivo es desactivar las transformaciones de forma selectiva en aquellas situaciones en las que la semántica se ve alterada por dichas transformaciones. Por ello el lenguaje sobre el que se define el análisis de no determinismo es una extensión del lenguaje Core con construcciones para representar de forma explícita las abstracciones y concreciones de procesos, el proceso `merge` y el tipo *Process*.

En esta fase también se puede llevar a cabo el análisis de conexión directa (*bypassing*) presentado en el Capítulo 4. Este análisis detecta hebras innecesarias que copian información de un canal de entrada a uno de salida. Produce anotaciones en las abstracciones y concreciones de procesos para evitar la creación de dichas hebras. Para llevar a cabo este análisis, es necesario no solamente hacer explícitos los procesos, sino también los canales de entrada y salida de los mismos. Se introducirá una nueva extensión de Core llamada CoreEdén con las características apropiadas para llevar a cabo el análisis. La idea es traducir de Core a CoreEdén, llevar a cabo el análisis de conexión directa produciendo un programa CoreEdén anotado con información de bypassing y volver después a Core para continuar con el proceso de compilación. En el Capítulo 4 se proporcionan más detalles.

3.4.3 Protocolo de creación de procesos

El entorno de ejecución de un programa Edén consiste en un conjunto de máquinas abstractas distribuidas, cada una de las cuales ejecuta una abstracción de proceso Edén. Puesto que los procesos Edén son más numerosos que los procesadores (*processing element*, en adelante PE) disponibles, varios procesos Edén pueden compartir un procesador. Cada proceso Edén consiste en una o varias hebras concurrentes de control. Estas evalúan diferentes expresiones de salida de forma independiente y usan una memoria común que contiene información compartida. El destino del valor producido por cada hebra recibe el nombre de *puerto de salida* (*outport*); esencialmente es una dirección global ($PE, port$) que consta de un elemento procesador PE junto con un identificador local único $port$ en dicho procesador llamado *puerto de entrada* (*inport*). El estado de una máquina incluye la información común a todas las hebras y el estado particular de cada hebra individual. Puesto que la entrada de un proceso está compartida entre todas las hebras, en cada máquina se comparte el montón[4] y una tabla de puertos de entrada. En dicha tabla se asigna a los identificadores de puertos de entrada las correspondientes direcciones del montón donde se almacenarán los mensajes de entrada y el identificador global asociado al puerto de salida remoto.

Para poder referenciar los puertos de entrada y salida de otros elementos procesadores, cada PE posee además de la tabla de puertos de entrada, una tabla de puertos de salida y una tabla de procesos. Puesto que varios procesos Edén comparten las tablas de puertos de entrada y salida, cada PE necesita una tabla de procesos conteniendo las listas de puertos de entrada y salida de cada proceso.

En lo sucesivo usaremos los siguientes convenios de notación:

- Supondremos que las abstracciones de proceso tienen n entradas y k salidas, es decir, tienen el tipo $Process\ (t_1, \ldots, t_n)\ (t'_1, \ldots, t'_k)$.
- Los identificadores de los puertos locales de entrada (respectivamente salida) se denotarán por $i_1, \ldots, i_n$ (respectivamente $o_1, \ldots, o_k$) en el proceso hijo y $i'_1, \ldots, i'_k$ (respectivamente $o'_1, \ldots, o'_n$) en el proceso padre.

Los mensajes comunicados a través de los canales pueden ser mensajes de datos o mensajes del sistema. Los mensajes de datos se encargan de la comunicación de valores a través de los canales: $SENDVAL(i, v)$ envía un valor v a un puerto de entrada i.

Los mensajes del sistema implican modificaciones de las tablas de los PE. Este tipo de mensajes siempre incluyen la identificación del PE emisor, para que éste pueda ser usado de forma apropiada por el PE receptor. Los mensajes más importantes son:

- $CREATE\text{-}PROCESS(id_{parent}, pabs, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n)$: Es enviado por el proceso padre id_{parent} a un PE remoto para lanzar allí un nuevo proceso. En el mensaje se incluye la abstracción de proceso $pabs$ y los nuevos puertos locales de entrada $i'_1, \ldots, i'_k$ y de salida $o'_1, \ldots, o'_n$.

[4] En inglés *heap*. El montón es la memoria dinámica utilizada en los lenguajes funcionales para la creación de clausuras.

- $ACK(id_{child}, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n, i_1, \ldots, i_n, o_1, \ldots, o_k)$: Es la respuesta a un mensaje de creación de procesos *CREATE-PROCESS* confirmando la creación del proceso hijo. Incluye la información necesaria para conectar apropiadamente los puertos de entrada y salida del hijo ($i_1, \ldots, i_n$ y $o_1, \ldots, o_k$) con los del padre ($i'_1, \ldots, i'_k$ y $o'_1, \ldots, o'_n$).

- *TERM-THREAD*: Cuando una hebra ha terminado su cómputo y ha enviado su valor de salida, termina cerrando el correspondiente puerto de salida. Si se cierran todos los puertos de salida de un proceso (lo cual se puede consultar en la tabla de procesos), el proceso completo termina cerrando todos sus puertos de entrada. Cuando un puerto de entrada se cierra, la hebra que le envía datos es informada de que sus resultados ya no son necesarios, mediante un mensaje *TERM-THREAD*.

El protocolo de comunicación completo se definió en [KOMP99]. Aquí describimos brevemente la parte correspondiente a la creación de procesos, ya que en el Capítulo 4 esta parte del protocolo será modificada para tener en cuenta la información producida por el análisis de conexión directa. La secuencia de mensajes es la siguiente:

- El proceso padre envía un mensaje

  $$CREATE\text{-}PROCESS(id_{parent}, pabs, o_{tso}, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n)$$

  al PE donde debe crearse el proceso hijo. Cuando se recibe dicho mensaje, se desempaqueta la clausura *pabs* y se copia en el montón local, junto con las clausuras de todas las variables libres referenciadas por ella. En el proceso hijo se crean los puertos de entrada y salida. Las tablas de puertos de entrada y salida del hijo se actualizan con la información enviada por el padre. Se envía un mensaje ACK al padre y se ejecuta la abstracción de proceso. El hijo puede empezar a enviar valores inmediatamente, ya que conoce los puertos de entrada del padre, pero el padre debe esperar.

- Cuando llega un mensaje

  $$ACK(id_{child}, o_{tso}, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n, i_1, \ldots, i_n, o_1, \ldots, o_k)$$

  al PE del proceso padre, pueden completarse las conexiones entre el padre y el hijo, actualizándose las tablas de puertos de entrada y salida del padre con la nueva información. El padre puede empezar a enviar valores al proceso hijo a los puertos de entrada recibidos en el mensaje.

Capítulo 4

Análisis de conexión directa

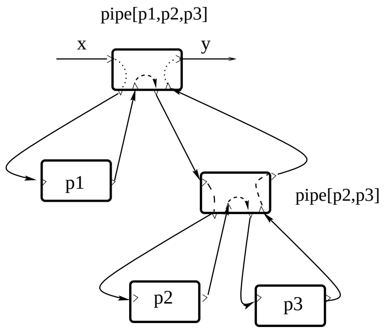
Este capítulo es el primero que forma parte de las contribuciones originales de la tesis. En él se estudia un problema relativamente sencillo pero de gran impacto en la eficiencia de los programas Edén. Se trata de analizar la existencia de hebras y de comunicaciones que no realizan trabajo útil para el programa, en el sentido de que simplemente redirigen mensajes de un procesador a otro. El capítulo explica primeramente el problema en detalle y luego su solución, que resulta ser una combinación de un análisis realizado en tiempo de compilación junto con una modificación del RTS que utiliza los resultados del análisis.

El análisis y el protocolo del RTS que aquí se explican son originales de esta tesis. Este último se presentó en [PS98a]. El protocolo finalmente implementado es una mejora propuesta por Ulrike Klusik, que incluye el tratamiento de *conexiones directas entre ancestros*. La implementación del protocolo en el RTS es también obra de Ulrike Klusik. Un resumen de este capítulo y del protocolo final fue publicado en [KPS00].

La organización del capítulo es la siguiente. En la Sección 4.1 explicamos el problema de la conexión directa en detalle. En la Sección 4.2 se presenta la sintaxis abstracta de un lenguaje intermedio, llamado CoreEdén, antes y después de anotarlo con información de conexión directa. La Sección 4.3 define formalmente el análisis en tiempo de compilación y lo aplica a un ejemplo sencillo. La Sección 4.4 explica el protocolo de conexión directa y el soporte que se le proporciona en tiempo de ejecución. Finalmente, la Sección 4.5 proporciona una cuantificación de las ganancias esperadas y resume el estado de la implementación.

4.1 Introducción

Como se explicó en la Sección 3.3, los procesos en Edén se lanzan dinámicamente al ejecutar definiciones recursivas de funciones y/o procesos. Cuando se crea un nuevo proceso, sus canales se conectan con su proceso padre. El




```
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
```

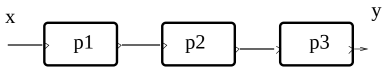


Figura 4.1: Topología creada y deseada para una tubería

padre es responsable de alimentar con valores los canales de entrada del hijo y de recibir valores de los canales de salida del hijo. Esto implica que, en principio, solamente se pueden crear topologías arbóreas de procesos, como se ve en la Figura 4.1 (izquierda). Frecuentemente, sucede que el padre simplemente copia los valores recibidos de un hijo al canal de entrada de otro hijo, y éste a su vez vuelve a enviar dichos valores a otro proceso y así sucesivamente. Detectar esta situación para poder eliminar procesos y hebras intermedios en las transmisiones es una optimización deseable. De esta forma se conectarían directamente las hebras productoras y consumidoras, como se ve en la Figura 4.1 (derecha). Esto ahorraría una gran cantidad de mensajes y muchos cómputos inútiles en tiempo de ejecución. Llamamos a dicha optimización *conexión directa*[1]. La implementación de la misma exige un análisis en tiempo de compilación para detectar conexiones directas *locales* (es decir, dentro de un proceso), junto con un soporte en tiempo de ejecución que encadene varias conexiones directas locales para conseguir una conexión directa *global*.

Para ilustrar el problema, en la Figura 4.1 se muestra la definición y lanzamiento de una tubería. La implementación ingenua de Edén (véase Sección 3.3) desarrolla en tiempo de ejecución la definición recursiva y genera la topología de procesos de la Figura 4.1 (izquierda). Puede observarse que esta estructura no es ni mucho menos la deseada (ver Figura 4.1 (derecha)) ya que se crean procesos no solamente para `p1`, `p2` y `p3`, sino también para `pipe[p1,p2,p3]` y `pipe[p2,p3]`, los cuales simplemente reenvían datos de la entrada a sus hijos y de ellos a su salida. Los canales se conectan siguiendo la cadena de llamadas recursivas.

La conexión directa tratará dichas situaciones de forma que los datos se envíen directamente del productor al consumidor, como en la Figura 4.1 (derecha). Con este enfoque, los procesos intermedios seguirán siendo lan-

[1]En inglés, *automatic bypassing*.

zados, pero terminarán después de crear sus procesos hijos puesto que ni producirán ni consumirán datos.

Este ejemplo ilustra dos de los tres tipos posibles de conexión directa: *conexión directa entre hermanos* (por ejemplo, entre `p2` y `p3`) y *conexión directa entre generaciones* (por ejemplo, entre `pipe[p1,p2,p3]` y `p2`). Hay un tercer tipo, llamado *conexión directa entre ancestros*, en el que un canal de entrada se copia directamente a exactamente un canal de salida, como por ejemplo en **process** $x \rightarrow x$. El análisis detectará los tres casos.

4.2 CoreEdén con y sin anotaciones

La sintaxis de Core[2] se muestra en la Figura 4.2. Actualmente en Core [3], las abstracciones y concreciones (o lanzamientos) de procesos están ocultos dentro de funciones predefinidas.

Para poder llevar a cabo el análisis de conexión directa necesitamos hacer explícitos los canales de entrada y salida. Por ello introducimos un nuevo lenguaje llamado CoreEdén, que es una extensión de Core. Las extensiones básicas son las siguientes:

$$
\begin{array}{l}
\mathit{binds} \rightarrow \mathbf{recpar}\ \mathit{bind}'_1; \ldots ; \mathit{bind}'_n\ [\mathbf{bypass}\ \mathit{channels}] \\
\mathit{bind}' \rightarrow \mathit{var} = \mathit{exp} \\
\qquad\quad \mid\ \mathit{channels} = \mathit{var}\ \#\ \mathit{channels} \\
\mathit{channels} \rightarrow \{\mathit{var}_1, \ldots, \mathit{var}_n\} \\
\mathit{exp} \rightarrow \mathbf{process}\ \mathit{channels} \rightarrow \mathit{body}\ [\mathbf{bypass}\ \mathit{channels}] \\
\mathit{body} \rightarrow [\mathbf{let}\ \mathit{binds}\ \mathbf{in}]\ \mathit{channels}
\end{array}
$$

Se introduce un nuevo tipo de ligaduras **recpar** para las concreciones de procesos, las cuales no son expresiones generales, sino que aparecen siempre dentro de una ligadura de este tipo. Las abstracciones de procesos son explícitas, y además, los canales de entrada y de salida, tanto de las abstracciones como de las concreciones de proceso, son variables explícitas. De este modo resulta relativamente sencillo analizar el uso de las mismas. Las transformaciones necesarias para obtener un programa en CoreEdén a partir de un programa en Core se explican en [PPRS00a].

La conexión directa se detecta cuando una variable que representa un canal de salida es utilizada exactamente una vez como canal de entrada de otro proceso, y además no aparece libre en ninguna otra expresión dentro de su ámbito de visibilidad. Ello supondría que la hebra asociada a esa variable

[2]En Core también se tienen lambdas y aplicaciones de tipo ya que se trata de un lenguaje con polimorfismo de segundo orden, pero aquí ignoramos estas expresiones, ya que no influyen en el análisis.

[3]Recordemos que Core es un lenguaje funcional mínimo al que se traduce Haskell en los primeros pasos de GHC.

$$
\begin{array}{lll}
program & \rightarrow & binds \\
binds & \rightarrow & bind \\
 & | & \mathbf{rec}\ bind_1; \ldots ; bind_n \\
bind & \rightarrow & var = exp \\
exp & \rightarrow & exp\ atom \\
 & | & \lambda\ var \rightarrow exp \\
 & | & \mathbf{case}\ exp\ \mathbf{of}\ alts \\
 & | & \mathbf{let}\ binds\ \mathbf{in}\ exp \\
 & | & C\ var_1 \ldots var_n \\
 & | & prim\ var_1 \ldots var_n \\
 & | & atom \\
alts & \rightarrow & Calt_1; \ldots ; Calt_n; Default \\
 & | & Lalt_1; \ldots ; Lalt_n; Default \\
Calt & \rightarrow & C\ var_1 \ldots var_m \rightarrow exp \\
Lalt & \rightarrow & Literal \rightarrow exp \\
Default & \rightarrow & NoDefault \\
 & | & var \rightarrow exp
\end{array}
$$

Figura 4.2: Sintaxis del lenguaje Core

no realiza ningún trabajo útil, excepto el de copiar los datos de un canal a otro.

La idea es pues introducir una fase intermedia en el compilador de Edén, antes de iniciar las transformaciones de Core a Core (véase Sección 3.4.2) que traduzca primero de Core a CoreEdén. A continuación, se realizaría el análisis de conexión directa. El análisis decora las abstracciones de procesos (en el caso de la conexión directa entre generaciones y entre ancestros) y las ligaduras **recpar** (en el caso de conexión directa entre hermanos y entre generaciones) con cláusulas **bypass**. Finalmente, se traduciría de CoreEdén a Core para continuar con el resto de la compilación. Este proceso se ilustra en la Figura 4.3.

El detalle de estas transformaciones se ha descrito en [PPRS00a] y forma parte de otra tesis [Rub01]. Por otra parte, las anotaciones reales que necesita el RTS son más precisas que las indicadas mediante cláusulas **bypass** en la sintaxis anterior, si bien pueden deducirse a partir de éstas. Para comprender mejor el análisis que describimos aquí son preferibles las anotaciones con cláusulas **bypass**, ya que eliminan detalles irrelevantes a este nivel.

En la Figura 4.5, *Exp* denotará una expresión CoreEdén sin anotar y *DExp* una expresión CoreEdén decorada con información de conexión directa. Lo mismo sucede con las ligaduras (*Bind* y *DBind*).

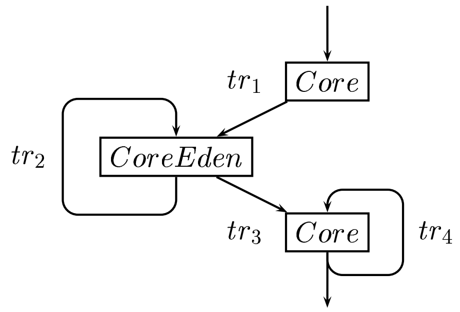
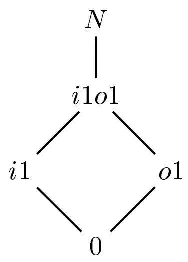


Figura 4.3: Nuevo esquema de transformación

4.3 Análisis de conexión directa

A través del análisis de conexión directa deseamos detectar aquellas situaciones en las que una variable se usa exactamente una vez como canal de entrada y exactamente una vez como canal de salida, y no se usa en ninguna otra expresión. Establecemos el convenio de que *entrada* y *salida* se entenderán siempre desde el punto de vista del proceso padre. Esto significa que, en una abstracción de proceso aquellos canales especificados como entradas del hijo se registrarán como salidas del padre. Este análisis es un problema con cierto parecido al análisis de uso de variables tratado por ejemplo en [Ses91] y [LGH$^+$92], si bien para la optimización que allí se pretende —evitar actualizaciones innecesarias— el análisis ha de detectar usos de a lo sumo una vez de las variables.

Utilizamos un dominio $B^\#$ de valores que se asociarán a las variables libres del programa, mostrado en la Figura 4.4, donde $i1o1$ representa que la variable se usa solamente una vez como canal de entrada y solamente una vez como canal de salida. En tal caso, se incluirá el canal en una cláusula **bypass** indicando una conexión directa local dentro del proceso correspondiente. Si es un proceso padre, la cláusula **bypass** utilizada es la asociada a la expresión **recpar** donde se lanzan los hijos, y la conexión directa sería del tipo entre hermanos o entre generaciones. Si es un proceso hijo, la cláusula **bypass** utilizada es la asociada a la abstracción de proceso, y la conexión directa podría ser entre generaciones o entre ancestros. El valor 0 representa ningún uso de la variable, los valores $i1$ y $o1$ representan respectivamente el uso exclusivo como canal de entrada y como canal de salida, y el valor N representa el hecho de que la variable se usa demasiadas veces como para poder hacer conexión directa, por ejemplo dos veces como entrada, o una o más veces como variable libre dentro de una expresión.



$$
\begin{array}{l}
N + x = N \\
0 + x = x \\
i1o1 + x = N \ (x \neq 0) \\
i1 + o1 = i1o1 \\
i1 + i1 = N \\
o1 + o1 = N
\end{array}
$$

Figura 4.4: Dominio de conexión directa

En algunas funciones del análisis utilizaremos un entorno ρ en el que asociamos valores de conexión directa a las variables ($Env = Var \to B^{\#}$). En las Figuras 4.5 y 4.6 consideramos que las variables que no aparecen en el entorno ρ tienen un valor de conexión directa 0, y denotamos al conjunto de variables con valor de conexión directa distinto de 0 como $dom\ \rho$. Definimos una operación conmutativa $+$ sobre los valores de conexión directa, mostrada en la Figura 4.4, que se utiliza para acumular los usos de una misma variable. La única expresión que produce el valor buscado $i1o1$ es la que acumula un uso $i1$ con un uso $o1$. La mayoría de las restantes combinaciones conduce al valor N, es decir a la imposibilidad de realizar una conexión directa.

La operación $+$ se extiende fácilmente a entornos: $(\rho + \rho')(x) \stackrel{\triangle}{=} \rho(x) + \rho'(x)$. También necesitamos una operación $\boldsymbol{+}$ que es simplemente la operación $+$ extendida a conjuntos de entornos, que se utilizará para acumular las apariciones como canal de entrada de una misma variable. Estas pueden aparecer más de una vez como entrada y entonces la conexión directa tampoco es posible. El análisis usa tres funciones principales: para analizar expresiones (A_{exp}), ligaduras (A_{binds}) y cuerpos de procesos ($A_{bindsbody}$). Se muestran en las Figuras 4.5 y 4.6. Las demás que aparecen en dichas figuras son funciones auxiliares. Usaremos v para denotar variables, e para denotar expresiones, cs para denotar canales y a para denotar átomos. También usamos la función $cs \Cap ds = \{x \mid x \in cs \wedge \exists! y \in ds.y = x\}$, para detectar conexión directa entre ancestros.

La función A_{exp} recibe una expresión CoreEdén e y la decora con información de conexión directa, introduciendo cláusulas **bypass** asociadas a las abstracciones de procesos y a las ligaduras **recpar** que aparecen en e. Tal información es local a la abstracción de proceso, por lo que analizamos su cuerpo con un entorno que contiene toda la información sobre los canales de entrada y salida. Así, en una expresión **process** $cs_1 \to$ **let** $binds$ **in** cs_2, construimos primero un entorno inicial ρ_0 que contiene la información sobre los canales de entrada (del padre) cs_2 y salida (del padre) cs_1. Con ese entorno analizamos las ligaduras $binds$ donde puede que existan más usos de dichos canales, usos que se acumularán en el entorno, dando lugar a un

$$
\begin{array}{l}
A_{exp} :: Exp \to DExp \\
A_{exp}\ (e\ a) = (A_{exp}\ e)\ a \\
A_{exp}\ (\lambda\ v \to e) = \lambda\ v \to A_{exp}\ e \\
A_{exp}\ (\textbf{case}\ e\ \textbf{of}\ alts) = \textbf{case}\ e'\ \textbf{of}\ alts' \\
\qquad \texttt{donde}\ \ e' = A_{exp}\ e \\
\qquad\qquad\quad\ \ alts' = A_{alts}\ alts \\
A_{exp}\ (C\ v_1 \ldots v_n) = C\ v_1 \ldots v_n \\
A_{exp}\ (prim\ v_1 \ldots v_n) = prim\ v_1 \ldots v_n \\
A_{exp}\ (\textbf{let}\ binds\ \textbf{in}\ e) = \textbf{let}\ (A_{binds}\ binds\ (freevars\ e))\ \textbf{in}\ (A_{exp}\ e) \\
A_{exp}\ (\textbf{process}\ cs_1 \to cs_2) = \\
\qquad \textbf{process}\ cs_1 \to cs_2\ (\texttt{if}\ cs_1 \mathbin{\dot\cap} cs_2 = \emptyset\ \texttt{then}\ \epsilon\ \texttt{else}\ \textbf{bypass}\ (cs_1 \mathbin{\dot\cap} cs_2)) \\
A_{exp}\ (\textbf{process}\ cs_1 \to \textbf{let}\ binds\ \textbf{in}\ cs_2) = \\
\qquad \textbf{process}\ cs_1 \to \textbf{let}\ binds'\ \textbf{in}\ cs_2\ byp \\
\qquad \texttt{donde}\ \ \rho_0 = \{c \mapsto o1 \mid c \leftarrow cs_1\} + + \{\{d \mapsto i1\} \mid d \leftarrow cs_2\} \\
\qquad\qquad\quad\ \ (binds', \rho') = A_{bindsbody}\ binds\ \rho_0\ (cs_1 \mathbin{\dot\cap} cs_2) \\
\qquad\qquad\quad\ \ b = \{c \mid c \in cs_1 \cup cs_2, \rho'(c) = i1o1\} \\
\qquad\qquad\quad\ \ byp = \texttt{if}\ b = \emptyset\ \texttt{then}\ \epsilon\ \texttt{else}\ \textbf{bypass}\ b \\
\\
A_{alts}\ (alt_1, \ldots, alt_n; def) = A_{alt}(alt_1); \ldots; A_{alt}(alt_n); A_{alt}(def) \\
A_{alt}(C\ v_1 \ldots v_n \to e) = C\ v_1 \ldots v_n \to A_{exp}(e) \\
A_{alt}(k \to e) = k \to A_{exp}(e) \\
A_{alt}(NoDefault) = NoDefault \\
A_{alt}(v \to e) = v \to A_{exp}(e)
\end{array}
$$

Figura 4.5: La función del análisis para las expresiones

entorno de salida ρ' utilizado para generar la anotación cuando alguno de los canales del proceso tiene un valor $i1o1$. Si no aparecen ligaduras *binds*, basta con mirar los canales de entrada y salida para ver si hay conexiones entre ancestros.

En una expresión **let** *binds* **in** e analizamos e y después las ligaduras, teniendo en cuenta que no se puede hacer conexión directa con las variables libres en e. Por ello, en el entorno usado para analizar las ligaduras asociamos el valor N a todas las variables libres. El resto de los casos son llamadas recursivas triviales a A_{exp}.

Las funciones A_{binds} y $A_{bindsbody}$ combinan la información que proviene de cada ligadura individual (ver Figura 4.6). Llaman a la función auxiliar $A_{propagate}$ con distintos argumentos. Esta función recibe un entorno y construye uno nuevo a partir de él añadiéndole la combinación (dada por $A_{listbinds}$) de información proporcionada por cada ligadura individual (dada por $A_{onebind}$). En el caso de la primera función se recibe un entorno en el que todas las variables libres de la expresión principal del **let** tienen N como valor de conexión directa, y el entorno resultante es descartado. En el caso de la segunda función se recibe un entorno en el que los canales de la abs-

$$
\begin{array}{l}
A_{binds} :: Binds \to Free \to DBinds \\
A_{binds}\ binds\ free = fst\ (A_{propagate}\ binds\ \{x \mapsto N \mid x \leftarrow free\}\ \emptyset) \\
\\
A_{bindsbody} :: Binds \to Env \to (DBinds, Env) \\
A_{bindsbody}\ binds\ \rho\ cs = A_{propagate}\ binds\ \rho\ cs \\
\\
A_{propagate} :: Binds \to Env \to Channels \to (DBinds, Env) \\
A_{propagate}\ (v = e)\ \rho\ cs = (bind', \rho + \rho') \\
\qquad \texttt{donde}\ (bind', \rho') = A_{onebind}\ (v = e) \\
A_{propagate}\ (\mathbf{rec}\ binds)\ \rho\ cs = (\mathbf{rec}\ binds', \rho + \rho') \\
\qquad \texttt{donde}\ (binds', \rho') = A_{listbinds}\ binds \\
A_{propagate}\ (\mathbf{recpar}\ binds)\ \rho\ cs = \\
\qquad (\mathbf{recpar}\ binds'\ byp, \rho'') \\
\qquad \texttt{donde}\quad (binds', \rho') = A_{listbinds}\ binds \\
\qquad\qquad\qquad \rho'' = \rho + \rho' \\
\qquad\qquad\qquad b = \{c \mid c \in dom\ \rho'', c \notin cs, \rho''(c) = i1o1\} \\
\qquad\qquad\qquad byp = \texttt{if}\ b = \emptyset\ \texttt{then}\ \epsilon\ \texttt{else}\ \mathbf{bypass}\ b \\
\\
A_{listbinds} :: [Bind] \to ([DBind], Env) \\
A_{listbinds}\ (bind : binds) = (bind' : binds', \rho' + \rho'') \\
\qquad \texttt{donde}\quad (bind', \rho') = A_{onebind}\ bind \\
\qquad\qquad\qquad (binds', \rho'') = A_{listbinds}\ binds \\
A_{listbinds}\ [] = ([], \emptyset) \\
\\
A_{onebind} :: Bind \to (DBind, Env) \\
A_{onebind}\ (v = e) = \\
\qquad (v = e', \{v \mapsto N\} + \{x \mapsto N \mid x \leftarrow freevars\ e\}) \\
\qquad \texttt{donde}\ e' = A_{exp}\ e \\
A_{onebind}\ (cs_1 = p\ \#\ cs_2) = (cs_1 = p\ \#\ cs_2, \rho') \\
\quad \texttt{donde}\ \rho' = \{c \mapsto o1 \mid c \leftarrow cs_1\} + +\!\!\!+\{\{d \mapsto i1\} \mid d \leftarrow cs_2\} + \{p \mapsto N\}
\end{array}
$$

Figura 4.6: Las funciones del análisis para las ligaduras

tracción de proceso tienen su valor de conexión directa correspondiente $i1$ o $o1$, y el entorno resultante se usa posteriormente para anotar la abstracción de proceso. El tercer argumento de $A_{propagate}$ es el conjunto de canales con los que se puede hacer conexión entre ancestros. En la llamada realizada a $A_{propagate}$ en el cuerpo de un proceso, estos son los canales comunes a la entrada y la salida de una abstracción de proceso, $cs_1 \Cap cs_2$. Queremos que esta información aparezca solamente asociada a la abstracción de proceso y no en las ligaduras internas del proceso, por lo que cuando se llama a $A_{propagate}$ con una ligadura **recpar**, se ignoran los canales contenidos en este parámetro a la hora de generar las anotaciones.

Obsérvese que, en una abstracción de proceso, aquellos canales implicados en una conexión directa entre hermanos aparecen en la anotación **bypass** de las ligaduras, los implicados en una conexión directa entre ancestros aparecen en la anotación **bypass** de la expresión **process** y los implicados en una conexión directa entre generaciones aparecen en ambos lugares (esto es así por razones de implementación).

En una ligadura de la forma $var = e$ y en una expresión **let** *binds* **in** e, no se puede hacer conexión directa con las variables libres en e, por lo que se les da un valor de conexión directa N. La definición de *freevars*, que nos proporciona las variables libres de una expresión, es la usual, extendida como es de esperar para las nuevas construcciones.

Una consecuencia de esta definición es que no se puede hacer conexión directa cuando aparece una expresión **case** como la siguiente:

$$\begin{array}{l} x = p_1\#z \\ y = \mathbf{case}\ e\ \mathbf{of} \\ \qquad C_1\ x_1 \rightarrow \mathbf{let\ recpar}\ u = p_2\#x\ \mathbf{in}\ u \\ \qquad C_2\ x_2\ x_3 \rightarrow 1 \end{array}$$

En este caso x es variable libre del lado derecho de la ligadura para y, por lo que se le da el valor N, siendo por tanto imposible hacer conexión directa. Esto es correcto, puesto que la segunda rama del **case** no contiene una concreción de proceso, por lo que no podemos indicar una conexión directa. Si también en la segunda rama apareciera una concreción de la forma $p_3\#x$, podría indicarse una conexión directa de x. Para poder expresar esto necesitaríamos comprobar que en todas y cada una de las ramas x tiene el valor i_1. Hemos optado por ignorar este caso.

Aplicando este algoritmo al ejemplo de la Sección 4.1 traducido a CoreEdén, obtenemos las siguientes anotaciones para la ligadura de *pipe*:

$$\begin{array}{l} pipe = \lambda\ ps \rightarrow \mathbf{case}\ ps\ \mathbf{of} \\ \qquad\qquad [p] \rightarrow p \\ \qquad\qquad p : pp \rightarrow \mathbf{process}\ z \rightarrow \mathbf{let\ recpar}\ inter = p\ \#\ z \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad p' = pipe\ pp \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad y = p'\ \#\ inter \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{bypass}\ inter, z, y \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ y\ \mathbf{bypass}\ z, y \end{array}$$

Puesto que no hay cálculos de puntos fijos, la eficiencia de este análisis es lineal con el tamaño del código.

4.4 El protocolo de conexión directa

Primeramente estudiaremos los casos de conexión directa por separado, para finalmente presentar el protocolo completo. En este protocolo no se tienen en cuenta las conexiones entre ancestros.

De aquí en adelante, llamaremos *asa* a una conexión directa local a un proceso detectada por el análisis precedente. Será *descendente* si conecta un

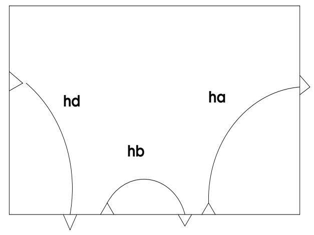


Figura 4.7: Convenio gráfico de asas locales

canal de entrada propio con el canal de entrada de un hijo, *ascendente* si conecta un canal de salida de un hijo con un canal de salida propio y será *entre hermanos* si conecta un canal de salida con un canal de entrada de dos hijos diferentes. Emplearemos el convenio gráfico que se muestra en la Figura 4.7. Asímismo, llamaremos *cadena de reenvío* a una conexión indirecta desde un puerto real de salida (también llamado el *productor*) a un puerto real de entrada (también llamado el *consumidor*) a través de una secuencia de asas locales intermedias. Un puerto de entrada (respectivamente, de salida) es la implementación de un canal de entrada (respectivamente, de salida) de una concreción de proceso. El objetivo del protocolo es hacer que estos puertos finales se conozcan entre sí.

4.4.1 El protocolo sin conexión directa

El protocolo de comunicación original se describió en la Sección 3.4.3 pudiéndose encontrar más detalles en [KOMP99]. Los mensajes relevantes eran:

- *CREATE-PROCESS* $(pabs, ins_p, outs_p)$ que inicia la creación de un proceso hijo en un procesador p usando la abstracción de proceso *pabs* y los puertos de entrada y salida del padre, respectivamente ins_p y $outs_p$, que se conectarán al hijo.

- *ACK* $(outs_c \to ins_p, outs_p \to ins_c)$ que comunica la creación del proceso hijo al padre, incluyendo las conexiones entre padre e hijo. Aquí mostramos los canales de entrada y salida relacionados con $\to$ para que queden claras las conexiones que se establecen.

Mientras que el hijo comienza a mandar inmediatamente valores al padre, el padre debe esperar a recibir el mensaje *ACK* para saber a qué puertos del hijo deben ser enviados los valores. Estos se envían usando un mensaje *SENDVAL (in,value)*.

4.4.2 Nuevas componentes del RTS: asas y tablas de asas de reenvío

Para representar cada asa local utilizaremos un identificador que incluye el número de procesador, que llamaremos también *asa de reenvío* (*forward handle*):

```
type HForward = (PE, Int)
```

Este par (p, h) identifica unívocamente un asa dentro del programa en ejecución ya que siempre es posible conseguir un identificador único h para cada asa existente dentro de un mismo procesador p. Las asas de reenvío pueden tomar el lugar de los puertos de entrada y salida en los mensajes del protocolo de conexión directa. Para almacenar la información de conexión intermedia añadimos al RTS una nueva tabla en tiempo de ejecución llamada *tabla de asas de reenvío*.

4.4.3 Conexión directa entre hermanos

El siguiente programa presenta un caso de conexión directa entre hermanos:

```
... -- El proceso padre es p0
let recpar
    x = p2 # y
    y = p1 # inp
    bypass y
in ...
```

La topología creada puede verse en la Figura 4.8. El proceso padre `p0` tiene un asa entre hermanos hy correspondiente a la variable `y`. El protocolo revisado no debe crear los puertos de entrada y salida para `y` ni enviarlos a sus hijos `p1` y `p2` en el mensaje *CREATE-PROCESS*. En su lugar enviará el identificador hy del asa. Los hijos podrían responder en el mensaje de ACK con los puertos reales de salida y de entrada para hy. Con esta información el padre sería capaz de informar al hijo productor de `y`, es decir a `p1`, que su destino real es un puerto de entrada en el proceso consumidor, es decir en `p2`. Con ello, `p1` podría enviar mensajes directamente a `p2` y habríamos ahorrado la creación de dos puertos y de una hebra innecesaria en `p0`.

En el caso general (Sección 4.4.6), veremos que no siempre es posible para un hijo informar en el mensaje ACK del productor real de una cadena de reenvío, por lo que es necesario considerar más casos.

4.4.4 Conexión directa descendente

El siguiente programa presenta un caso de conexión directa con varias asas descendentes en la cadena de reenvío:

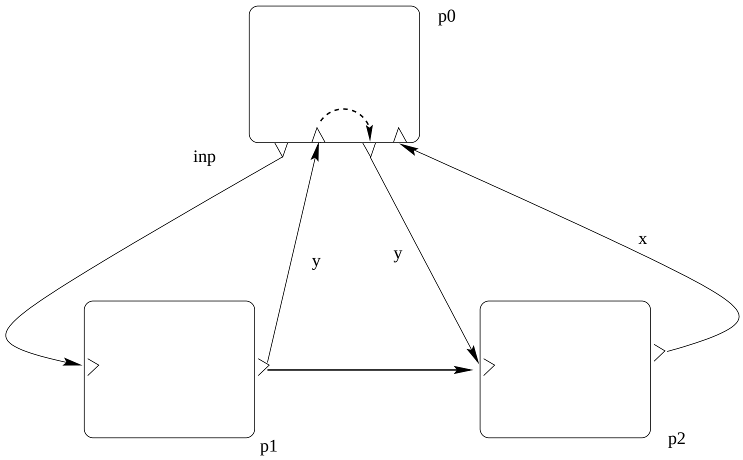


Figura 4.8: Conexión directa entre hermanos

```
... -- El proceso padre es p0
let recpar
    x = p # y
    p = process y'  ->  let recpar
                              h = q # y'
                              v = f h
                              bypass y'
                        in  v
        bypass y'
    q = process y'' ->  let recpar
                              i = r # y''
                              w = g i
                              bypass y''
                        in w
        bypass y''
in ...
```

La topología creada puede verse en la Figura 4.9. Para incluir información de conexión directa en los mensajes de creación, en el protocolo revisado se puede aprovechar el hecho de que los lanzamientos de los procesos `p`, `q` y `r` se producen en estricta secuencia, por ser `q` hijo de `p` y `r` hijo de `q`. En particular, `p` puede informar a `q` de que el origen de su canal de entrada es el puerto de salida recibido de su padre para el asa hy' asociada a `y'`. Esta información puede ser propagada en el mensaje *CREATE-PROCESS* enviado por `q` para crear a `r`, de forma que en lugar de enviarle un asa local hy'' asociada a `y''`, le envía directamente el puerto de entrada que ha recibido de su padre. De esta forma, el consumidor real de la cadena de reenvío —en este ejemplo, `r`— recibe directamente el puerto productor en el proceso `p0`.

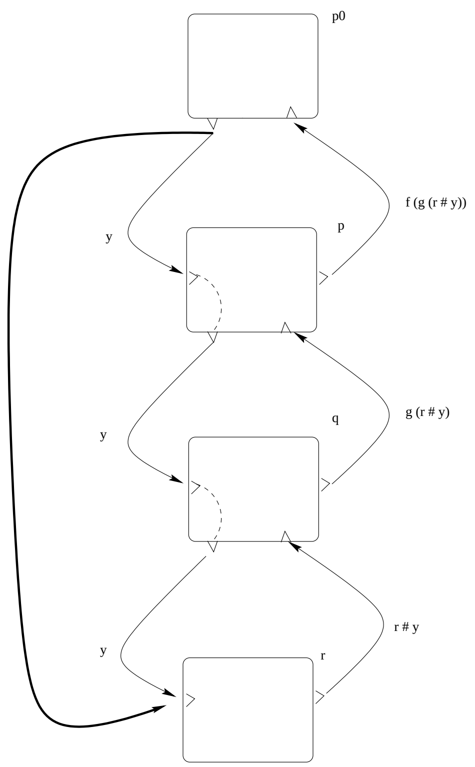


Figura 4.9: Conexión directa descendente

En el caso general (Sección 4.4.6), no siempre el padre de la cadena descendente envía un puerto real, sino que puede enviar también un asa entre hermanos.

4.4.5 Conexión directa ascendente

El siguiente programa presenta un caso de conexión directa con varias asas ascendentes en la cadena de reenvío:

```
... -- El proceso padre es p0
let recpar
    x = p # y
    p = process y'  -> let recpar
                           v = f y'
                           h = q # v
                           bypass h
                       in h
        bypass h
```

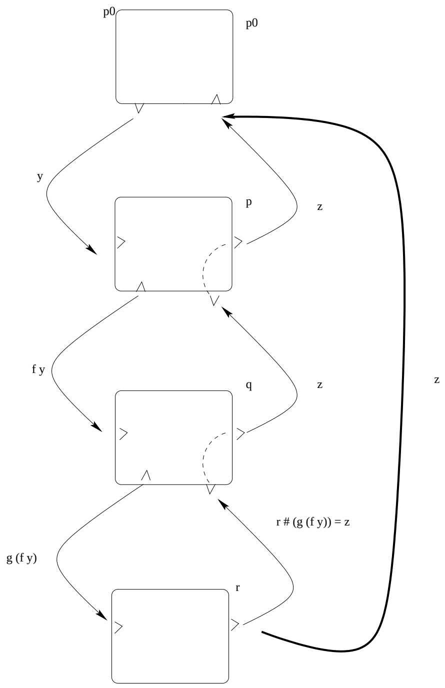


Figura 4.10: Conexión directa ascendente

```
    q = process y'' -> let recpar
                               w = g y''
                               i = r # w
                               bypass i
                           in i
        bypass i
in ...
```

La topología creada puede verse en la Figura 4.10. En el protocolo revisado se puede aprovechar igualmente el hecho de que los lanzamientos de los procesos `p`, `q` y `r` se producen en estricta secuencia, para incluir información de conexión directa en los mensajes de creación. En particular, `p` puede informar a `q` de que el destino del canal de salida de `q` es el puerto de entrada recibido de su padre para el asa ascendente *hh* asociada a `h`. Esta información puede ser propagada en el mensaje *CREATE-PROCESS* enviado por `q` para crear a `r`, de forma que en lugar de enviarle un asa local *hi* asociada a `i`, le envía directamente el puerto de salida que ha recibido de su padre. De esta forma, el productor real de la cadena de reenvío —en este ejemplo, `r`— recibe directamente el puerto consumidor en el proceso `p0`.

En el caso general (Sección 4.4.6), no siempre el padre de la cadena ascendente envía un puerto de entrada real, sino que puede enviar también un asa entre hermanos.

4.4.6 Caso general

El siguiente programa presenta una cadena de reenvío que combina un asa entre hermanos con un asa descendente:

```
... -- El proceso padre es p0
let recpar
    x  = p2 # y
    y  = p1 # inp
    p2 = process i -> let recpar
                          h = p3 # i
                          v = f h
                          bypass i
                      in v
         bypass i
    bypass y
in ...
```

La topología creada puede verse en la Figura 4.11. En este ejemplo puede observarse que la cadena de reenvío incluye un asa entre hermanos en `p0`, una cadena ascendente de longitud 1 cuyo productor real es `p1`, y una cadena descendente de longitud 2 cuyo consumidor real es `p3`. Combinando las revisiones al protocolo descritas en las secciones precedentes, tanto `p1` como `p3` recibirían información de que su destino real (resp. su origen real) es el asa entre hermanos *hy* asociada a `y` en `p0`. Está claro que lo que resta por hacer es:

1. Que tanto `p1` como `p3` informen a `p0` de los puertos reales, respectivamente productor y consumidor, de la cadena de reenvío.

2. Que `p0` informe a `p1` de que el destino real de su puerto de salida es un puerto de entrada en `p3`.

A partir de esta recepción `p1` puede comenzar a enviar mensajes directamente a `p3`. El ahorro ha consistido en una hebra innecesaria en `p0`, otra en `p2` y varios puertos que no ha sido necesario crear, tanto en `p0` como en `p2`.

4.4.7 El protocolo revisado

En el nuevo protocolo, el mensaje *CREATE-PROCESS* llevará los puertos del padre si son conocidos, pero también están permitidas asas de reenvío. Además de los mensaje *CREATE-PROCESS* y *ACK*, hay tres nuevos mensajes de conexión directa:

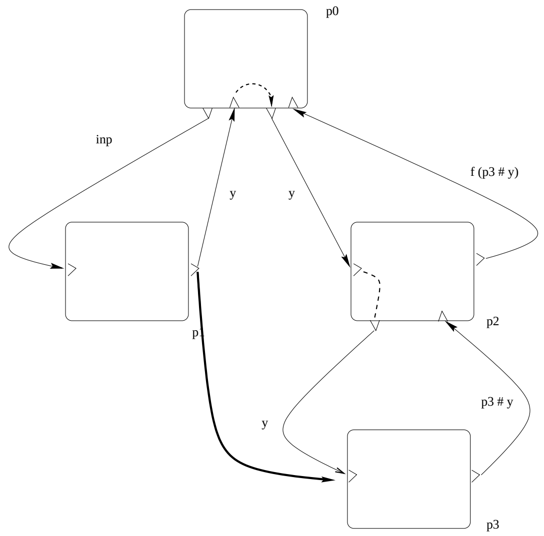


Figura 4.11: Situación general de conexión directa

- *FINAL-ORIGIN (out $\rightarrow$ in)*: Le dice al consumidor quién es el verdadero productor de una cadena ascendente. *out* es siempre un puerto de salida real, mientras que *in* puede ser un puerto de entrada o un asa entre hermanos.

- *FINAL-DESTINATION (out $\rightarrow$ in)*: Le dice al productor quién es el verdadero consumidor de una cadena descendente. *in* es siempre un puerto de entrada real, mientras que *out* puede ser un puerto real o un asa.

- *PRODUCE (out $\rightarrow$ in)*: Le dice al productor real de una cadena general de reenvío quién es el consumidor final. En este caso, tanto *out* como *in* son puertos reales.

El esquema general del protocolo, reflejado en las Figuras 4.12 y 4.13, es el siguiente

Fase 1: Creación de procesos modificada. En presencia de un reenvío, algunos puertos de entrada y/o salida no serán creados. En el mensaje *CREATE-PROCESS*, asas de reenvío sustituyen a dichos puertos ausentes. Esto sucede, por ejemplo, en los mensajes *CREATE-PROCESS* de la Figura 4.13. Si ya se conoce alguna información de conexión, se envía en lugar del asa de reenvío local. Esto sucede en los reenvíos intergeneracionales. Una vez han sido creados todos los procesos, tenemos la siguiente situación: todas las asas intergeneracionales han sido eliminados. Cada proceso del final de la cadena ha

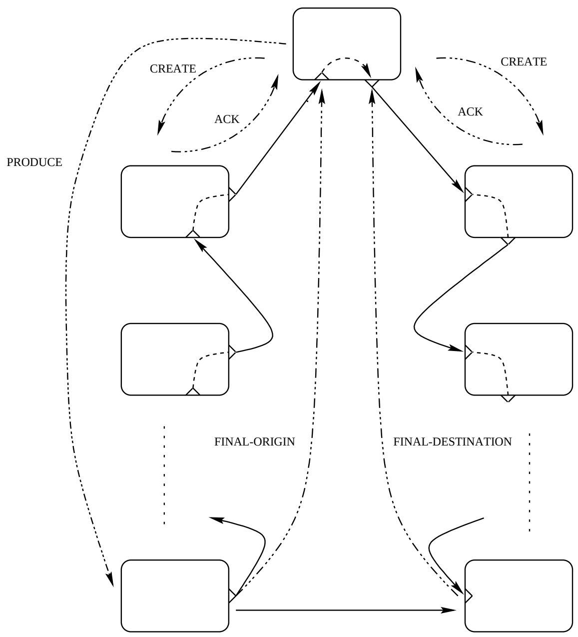


Figura 4.12: Protocolo de conexión directa (1)

recibido como extremo opuesto o bien un asa entre hermanos o bien un puerto real de un proceso ancestro.

Fase 2: Envío hacia arriba de mensajes. Cuando un proceso hijo es el verdadero productor/consumidor de una cadena, debe informar al otro extremo de la cadena. Esto se lleva a cabo a través de un mensaje *FINAL-ORIGIN/FINAL-DESTINATION*. Si un puerto de salida ya conoce su consumidor, puede empezar a enviar datos inmediatamente. En la Figura 4.12 estas situaciones se reflejan en los dos mensajes hacia arriba.

Fase 3: Conexión final. Cuando un asa entre hermanos ha recibido los mensajes de los verdaderos consumidor y productor de la cadena, presenta el consumidor al productor mediante un mensaje *PRODUCE*, para que este último pueda comenzar el envío de datos.

4.5 Costes de comunicación y conclusiones

Se ha indicado que el protocolo realmente implementado trata la conexión directa entre ancestros. Esto introduce algunos problemas porque, en situaciones complicadas, una cadena de reenvío puede entrar y salir hasta dos veces de un mismo procesador, produciéndose un bucle que es necesario

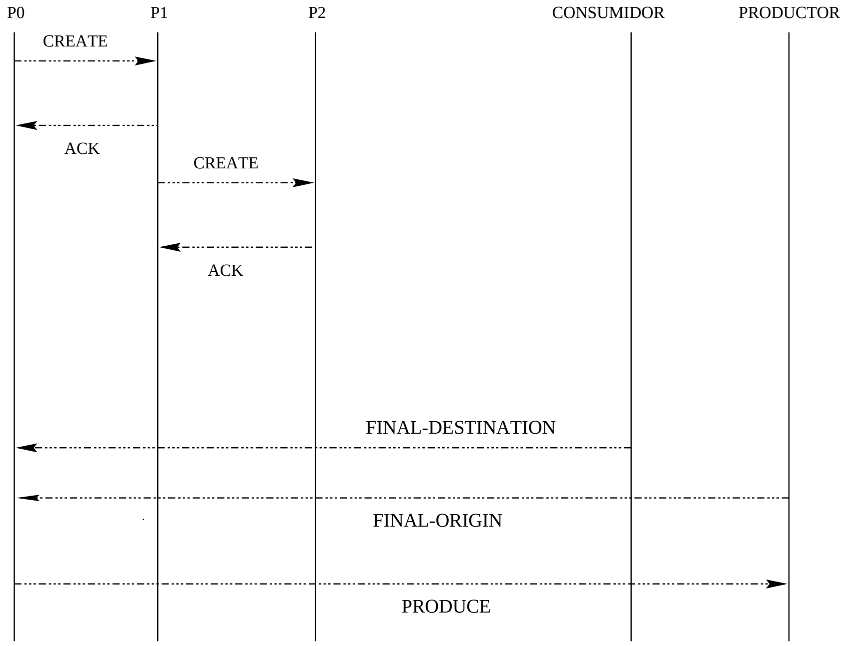


Figura 4.13: Protocolo de conexión directa (2)

eliminar. Ello conlleva un tipo de mensaje adicional llamado *LOOP* y la necesidad de incluir en la información de asas el nivel de la jerarquía de procesos a la que pertenece el asa (el ancestro de una cadena está a nivel 0, los procesos creados por él están a nivel 1, y así sucesivamente).

Es claro que si el programa no contiene asas, los mensajes en ambos protocolos, el viejo y el nuevo, son idénticos, es decir, la forma de crear las topologías es exactamente la misma.

Si el programa contiene asas, entonces el número de mensajes adicionales del nuevo protocolo con respecto al anterior es como mucho tres. El peor de los casos se produce cuando existe un asa entre hermanos en una cadena de longitud mayor que dos, en cuyo caso, hacen falta dos mensajes en la segunda fase, más uno adicional en la fase final de conexión. Es el caso general de la Figura 4.13. Si existe un asa entre hermanos, pero la cadena es de longitud dos, no hacen falta los mensajes *FINAL-ORIGIN* y *FINAL-DESTINATION*, ya que los propios mensajes *ACK* informan al padre, con lo que solamente hace falta un mensaje *PRODUCE* adicional. En las cadenas ascendentes y descendentes que no contienen un asa entre hermanos se puede reducir también el número de mensajes. En una cadena ascendente, el padre es el destino final, por lo que podemos ahorrarnos el mensaje *FINAL-DESTINATION*. Luego en este caso hacen falta solamente dos mensajes adicionales. En una cadena descendente, el padre es el origen de la cadena por lo que podemos ahorrarnos los mensajes *FINAL-ORIGIN* y *PRODUCE*. De modo que en este caso solamente hace falta un mensaje adicional.

El esperado ahorro de la conexión directa proviene del hecho de que el número de mensajes de datos es usualmente mucho mayor que el número de mensajes de protocolo necesitados para crear la conexión directa. Esto es así porque al programar en Edén, muchos de los canales son listas cuya transmisión implica tantos mensajes de datos como elementos hay en la lista.

Para una cadena de reenvío de longitud n, con $n \geq 2$, se ahorran un total de $n-1$ mensajes por cada mensaje de datos enviado a través de la conexión directa. Por otro lado, hemos visto que el número de mensajes adicionales del nuevo protocolo es como mucho tres. Luego, incluso en el caso de que se transmita un único valor, el ahorro se produce en cuanto la longitud de la cadena sea mayor que cuatro.

El estado actual de la implementación es el siguiente: el RTS ha sido ya modificado incluyendo el nuevo protocolo, pero aún está bajo depuración. El análisis en tiempo de compilación y las muchas transformaciones implicadas para producir un "buen" CoreEdén están siendo implementadas aún.

Capítulo 5

Análisis de no determinismo

La presencia de no determinismo produce algunos problemas en Edén: afecta a la transparencia referencial de los programas e invalida algunas optimizaciones realizadas en el GHC. Como solución a este problema, en este capítulo se presentan tres análisis de no determinismo, capaces de establecer con certeza cuándo una expresión Edén es determinista y cuándo no existe tal certeza, es decir, cuándo puede que sea no determinista. El primer análisis es poco costoso (lineal) pero también poco potente. Con el objetivo de superar sus limitaciones, se propone un segundo análisis más potente que resulta ser exponencial. Para mejorar su eficiencia se presenta un tercer análisis intermedio que pretende ser un compromiso entre potencia y eficiencia. La mejora en la eficiencia se consigue acelerando el cálculo del punto fijo mediante un operador de ensanchamiento, representando las funciones mediante signaturas fácilmente comparables. Eligiendo distintos operadores de ensanchamiento se obtienen diferentes variantes de este tercer análisis. Se presenta con detalle una de ellas, y se mencionan otras opciones, con las que se compara. Se comparan formalmente los tres análisis demostrando que el primer análisis es una aproximación segura al tercer análisis y que éste último es una aproximación segura al segundo, con lo que se puede establecer una jerarquía de análisis. Finalmente, se describe un algoritmo que no sólo implementa el tercer análisis sino que además anota las expresiones de programa con información de no determinismo. Dicho algoritmo se ha implementado en Haskell. El código fuente se muestra en el Apéndice B. En el Apéndice C se presentan ejemplos reales de la salida producida por el algoritmo para algunos programas. Para finalizar este capítulo, se lleva a cabo un estudio detallado del coste del algoritmo.

Este trabajo ha dado lugar a varias publicaciones. En [PS00a] se estudió la forma en que las transformaciones realizadas por el GHC se ven afectadas por la semántica de los procesos Edén; en concreto, se estudiaron aquellas transformaciones que se ven afectadas por la presencia de no determinismo. Dicho estudio se incluyó en [PPRS00a, PPRS00b]. Los dos primeros análisis de no determinismo se presentaron en [PS01d] y con mayor detalle en el informe técnico [PS00b]. El primer análisis se presentó primero en forma de *análisis basado en un sistema de tipos anotados*, que aquí no se muestra, y después como *análisis basado en interpretación abstracta*, ya que esta resulta más adecuada para expresar las dependencias funcionales que aparecen en el segundo análisis. El tercer análisis y su implementación se presentaron en [PS01c]. La relación de corrección del primer análisis con respecto a los otros dos se demostró en [PS01a]. El informe técnico [PS01b] pretende ser una recopilación de todos los resultados presentados en los artículos anteriores. En él se demuestra también la relación de corrección del tercer análisis con respecto al segundo.

Este capítulo está estructurado de la siguiente forma. En la Sección 5.1 se motiva la necesidad de un análisis de no determinismo en Edén. En la Sección 5.2 se presenta un resumen de las transformaciones que suponen un riesgo para los programas Edén, ya sea porque modifican la semántica o porque reducen la eficiencia; en concreto, se describen con detalle las que afectan al no determinismo. En la Sección 5.3 se describe el lenguaje en el que se escriben los programas a ser analizados. Se trata de un lenguaje funcional simple con polimorfismo de segundo orden. A continuación, en la Sección 5.4 se presenta el primer análisis y se explica con ejemplos sus limitaciones. En la Sección 5.5 se presenta el segundo análisis, en el que se definen las llamadas funciones de abstracción y concreción, usadas respectivamente en el análisis de aplicaciones de constructores y expresiones **case**, y se estudian sus propiedades. Se estudia también con más detalle el análisis de las expresiónes polimórficas. En la Sección 5.6 se presenta el tercer análisis. Se define el dominio de signaturas utilizado para representar las funciones y el operador de ensanchamiento utilizado para acelerar el cálculo del punto fijo. En la Sección 5.7 se estudian las relaciones de corrección entre los tres análisis. En ella también se presentan otras variantes del tercer análisis. Finalmente, en la Sección 5.8 se desarrolla la implementación del tercer análisis, se muestran ejemplos de su aplicación y se hace un estudio de su coste.

5.1 No determinismo en Edén

La introducción del no determinismo en los lenguajes funcionales tiene una larga tradición y ha sido, asimismo, fuente de controversia. McCarthy [McC63] introdujo el operador `amb :: a -> a -> a` que elige de forma no determinista entre dos valores. Henderson [Hen82] introdujo en su lugar `merge :: [a] -> [a] -> [a]`, que mezcla de forma no determinista dos listas de valores, produciendo una única lista de salida. Ambos operadores violan la transparencia referencial, en el sentido de que una vez introducidos, ya no es posible sustituir iguales por iguales. Por ejemplo,

```
let x = amb 0 1 in x + x ≠ amb 0 1 + amb 0 1
```

ya que la primera expresión solamente puede evaluarse a 0 o a 2, mientras que la segunda puede evaluarse, además, al valor 1.

Hughes y O'Donnell propusieron en [HO90] un lenguaje funcional en el que el no determinismo es compatible con la transparencia referencial. La forma de conseguirlo consiste en introducir el tipo `Set a` de conjuntos de valores, para denotar el resultado de expresiones no deterministas. El programador debe usar explícitamente este tipo siempre que una expresión pueda devolver un valor elegido de entre un conjunto de valores posibles. En la implementación se representa un conjunto mediante un único valor perteneciente al conjunto. Una vez creado un conjunto, el programador no puede regresar a valores únicos. Luego la aplicación de una función determinista `f` a un valor no determinista (un conjunto `S`) debe expresarse como `f * S`, donde `(*) :: (a -> b) -> Set a -> Set b` es la función `map` para los conjuntos. Solamente se permiten una cantidad limitada de operadores. El más importante es $\cup$ (unión de conjuntos), que permite la creación de conjuntos no deterministas y que además puede ser utilizado para simular `amb`. Otros operadores, como `choose :: Set a -> a` o $\cap$ (intersección de conjuntos), no están permitidos, ya sea porque violan la transparencia referencial o porque no pueden implementarse de forma correcta simplemente "recordando" un representante por cada conjunto.

Hughes y O'Donnell dan al lenguaje una semántica denotacional basada en dominios potencia de Hoare, y presentan algunas leyes ecuacionales útiles para razonar sobre la corrección (parcial) de los programas.

Pero la controversia llega aún más allá. En [SS90, SS92], los autores aseguran que lo que hace falta realmente es una definición apropiada de la *transparencia referencial*. Demuestran que varias definiciones aparentemente equivalentes (sustitutividad de iguales por iguales, desplegabilidad de definiciones, ausencia de efectos laterales, definitud de variables, determinismo, etc.) que se han utilizado en distintos contextos, no lo son en presencia de no determinismo. Para situar a Edén en perspectiva, reproducimos aquí los principales conceptos:

Transparencia referencial. Una expresión e es *puramente referencial* en la posición p si y sólo si

$$\forall e_1, e_2 . [\![e_1]\!]\ \rho = [\![e_2]\!]\ \rho \Rightarrow [\![e[e_1/p]]\!]\ \rho = [\![e[e_2/p]]\!]\ \rho$$

Un operador $op :: t_1 \rightarrow \cdots \rightarrow t_n \rightarrow t$ es *referencialmente transparente* si para cada expresión $e = op\ e_1 \cdots e_n$, se cumple que siempre que las expresiones $e_i, 1 \leq i \leq n$, son puramente referenciales en una posición p, la expresión e lo es en la posición $i.p$. Un lenguaje es referencialmente transparente si lo son todos sus operadores.

Definitud. La propiedad de definitud se cumple si una variable denota el mismo valor en todas sus apariciones. Por ejemplo, si las variables son "definidas", la expresión $(\lambda x.x - x)(amb\ 0\ 1)$ se evalúa siempre a 0. Si no lo son, también se puede evaluar a 1 y -1.

Desplegabilidad. La propiedad de desplegabilidad se cumple si $[\![(\lambda x.e)\ e']\!]\ \rho = [\![e[e'/x]]\!]\ \rho$ para cada e y e'. En presencia de no determinismo, esta propiedad no es compatible con la definitud de variables. Por ejemplo, si las variables son definidas

$$[\![(\lambda x.x - x)(amb\ 0\ 1)]\!]\ \rho \neq [\![(amb\ 0\ 1) - (amb\ 0\ 1)]\!]\ \rho.$$

En las definiciones arriba descritas, la semántica de una expresión es un conjunto de valores en el dominio potencia apropiado. Sin embargo, el entorno ρ hace corresponder a cada variable o bien un único valor en el caso de que las variables sean definidas (también llamada *semántica singular*), o un conjunto de valores en el caso de que no lo sean (también llamada *semántica plural*).

Recordemos que en Edén, la única fuente de no determinismo es el proceso predefinido `merge`. Cuando se lanza un nuevo proceso mediante la evaluación de la expresión `e1 # e2`, la clausura `e1`, junto con las clausuras de todas las variables libres allí referenciadas, se copian (posiblemente sin evaluar) en otro procesador, donde el nuevo proceso se lanza. Sin embargo, dentro de un mismo procesador una variable se evalúa a lo sumo una vez y su valor es compartido. Se está desarrollando aún una semántica denotacional para el lenguaje, pero para el propósito de esta discusión, supondremos que la denotación de una expresión de tipo `a` es un conjunto no vacío cerrado inferiormente y cerrado bajo límites de valores de tipo `a` que representa el conjunto de valores posibles devueltos por la expresión[1]. Si la expresión es determinista, su denotación es un conjunto unitario. Esta es la semántica propuesta por [HO90] para el no determinismo angélico, en el que siempre se escoge un valor distinto de $\perp$ si es posible. En el Capítulo 3 ya se dijo que `merge` es un proceso reactivo que cuando tiene un valor disponible en una de las entradas, lo copia en su salida. Es decir, siempre que recibe un valor distinto de $\perp$, lo escoge como valor de salida. Por ello, esta semántica parece la adecuada en nuestro caso.

Bajo estas premisas, podemos caracterizar a Edén de la siguiente manera:

Transparencia referencial. Edén es referencialmente transparente. La única diferencia con respecto a Haskell es que ahora, en un entorno ρ, una expresión denota un conjunto de valores, en lugar de un único valor. Dentro de una expresión, una subexpresión no determinista siempre puede ser reemplazada por su denotación sin afectar al conjunto de valores resultante.

[1]Estos son los conjuntos Scott-cerrados definidos en la Sección 2.2.1.

Definitud. Las variables son definidas dentro de un mismo proceso pero no lo son entre procesos diferentes. Cuando una variable libre no determinista que aún no ha sido evaluada se duplica en dos procesos distintos, puede suceder que el valor real calculado en cada proceso sea diferente. Sin embargo, denotacionalmente ambas variables representan el mismo conjunto de valores, luego la semántica de las expresiones que la contienen no cambiará por el hecho de que la variable se evalúe dos veces.

Desplegabilidad. En general, en Edén no se cumple esta propiedad, excepto en el caso de que la expresión que se despliega sea determinista. Esto es consecuencia de tener variables definidas dentro de un proceso y por tanto incompatibilidad con la desplegabilidad.

El desarrollo de un análisis de no determinismo en Edén surge a raíz de los siguientes hechos:

- En el futuro, los programadores Edén podrían desear tener variables definidas en todas las situaciones. Es sensato pensar en una opción de compilación para seleccionar dicha opción semántica. En dicho caso, el análisis detectaría las variables (posiblemente) no deterministas y el compilador forzaría su evaluación a forma normal antes de ser copiadas a un procesador diferente.

- Actualmente, algunas transformaciones llevadas a cabo por el compilador durante las fases de optimización son semánticamente incorrectas para expresiones no deterministas. La más importante de ellas es la llamada *full laziness* [PPS96]. Otras transformaciones peligrosas son la *transformación de argumentos estáticos* [San95] y la *especialización*. La razón general de su peligrosidad es el incremento en la compartición de clausuras: antes de la transformación, la evaluación repetida de una expresión no determinista puede dar lugar a distintos valores; después de la transformación, una expresión no determinista compartida se evalúa una sola vez, produciendo un único valor. Las tres se describen con detalle en la Sección 5.2.

5.2 Riesgos de las transformaciones en GHC

En esta sección se resumen brevemente los riesgos de algunas de las transformaciones realizadas por GHC sobre Core. En [PS00b, PPRS00a] se proporcionan más detalles.

En primer lugar no se debe realizar una transformación si esta cambia de alguna forma la semántica de los programas (lo que en definitiva es cambiar los deseos del programador). En particular hay tres cambios posibles que son inaceptables: (1) el lanzamiento prematuro o tardío de los procesos; (2)

la modificación del número de procesos lanzados; y (3) la reducción o el incremento del no determinismo. Además, algunas de las transformaciones que en un entorno secuencial son beneficiosas, podrían empeorar la eficiencia de los programas paralelos. La eficiencia puede verse afectada de forma negativa de varias maneras: (1) mediante el cambio de trabajo de un proceso hijo a su padre o viceversa; (2) incrementando el coste de comunicación debido a la copia de clausuras de variables libres; y (3) incrementando la reserva de memoria.

En general, debemos tener en cuenta aquellas transformaciones que incrementan la compartición. Estas incluyen el flotamiento de ligaduras fuera de una lambda o el flotamiento de las mismas fuera del lado derecho de una ligadura `let`. En el primer caso, el número de procesos lanzados puede cambiar, si la ligadura que está siendo flotada encierra alguna concreción de proceso: una vez se ha flotado la ligadura, el lanzamiento tiene lugar solamente la primera vez que se aplica la función. En el segundo caso, se puede cambiar el lugar donde se crean las ligaduras, lo que incrementa el tráfico de clausuras.

También deben tenerse en cuenta las transformaciones que tienen como objetivo una evaluación inmediata de aquellas expresiones que se demuestra son necesarias, ya que pueden dar lugar a un cambio de trabajo entre los procesos.

Veamos ahora un resumen de las transformaciones que se ven afectadas, mostradas en la Figura 5.1. Desarrollamos con detalle las transformaciones que afectan al no determinismo. El resto se pueden ver ampliadas en [PS00a].

5.2.1 Transformaciones que afectan al no determinismo

Full laziness. Esta transformación (Figura 5.1a) flota una ligadura *let* fuera de una lambda abstracción, con el objetivo de compartir su evaluación entre todas las aplicaciones de la función:

$$\begin{array}{l} \mathbf{let} \\ \quad g = \lambda y.\mathbf{let}\ x = e \\ \qquad\qquad\quad \mathbf{in}\ e' \\ \mathbf{in}\ \ldots \end{array} \Rightarrow \begin{array}{l} \mathbf{let} \\ \quad x = e \\ \mathbf{in}\ \mathbf{let} \\ \quad\ g = \lambda y.e' \\ \quad \mathbf{in}\ \ldots \end{array}$$

Es correcta solamente si e no depende de y.

Desde el punto de vista del no determinismo, el problema surge cuando la ligadura a flotar es no determinista. Como ejemplo, consideremos las siguientes dos ligaduras, que representan una función antes y después de la transformación:

```
f = λy.let x = e1          f' = let x = e1
    in x + y                     in λy. x + y
```

transformación	antes	después
(a) Full laziness	$\textbf{let } g = \lambda y.\textbf{let } x = e$ $\textbf{in } e'$ $\textbf{in } \ldots$	$\textbf{let } x = e$ $\textbf{in let } g = \lambda y.e'$ $\textbf{in } \ldots$
(b) Argumentos estáticos	$\mathit{foldr}\ f\ z\ l =$ $\textbf{case } l \textbf{ of}$ $[\,] \to z$ $(a : as) \to \textbf{let } v = \mathit{foldr}\ f\ z\ as$ $\textbf{in } f\ a\ v$	$\mathit{foldr}\ f\ z\ l =$ $\textbf{let } \mathit{foldr}'\ l =$ $\textbf{case } l \textbf{ of}$ $[\,] \to z$ $(a : as) \to \textbf{let } v = \mathit{foldr}\ f\ z\ as$ $\textbf{in } f\ a\ v$ $\textbf{in } \mathit{foldr}'\ l$
(c) Especialización	$g = \Lambda ty.\lambda dict.\lambda y.$ $\textbf{let } f = \Lambda ty.\lambda dict.e$ $\textbf{in } f\ ty\ dict\ (f\ ty\ dict\ y)$	$g = \Lambda ty.\lambda dict.\lambda y.$ $\textbf{let } f = \Lambda ty.\lambda dict.e$ $\textbf{in let } f' = f\ ty\ dict \textbf{ in } f'\ (f'\ y)$
(d) flotamiento de *let* fuera de *let*	$\textbf{let } x = \textbf{let } \mathit{bind}$ $\textbf{in } e$ $\textbf{in } b$	$\textbf{let } \mathit{bind}$ $\textbf{in let } x = e$ $\textbf{in } b$
(e) flotamiento de *case* fuera de *let*	$\textbf{let } v = \textbf{case } e_v \textbf{ of}$ $\ldots$ $C_i\ x_{i1} \ldots x_{ik} \to e_i$ $\ldots$ $\textbf{in } e$	$\textbf{case } e_v \textbf{ of}$ $\ldots$ $C_i\ x_{i1} \ldots x_{ik} \to \textbf{let } v = e_i$ $\textbf{in } e$ $\ldots$
(f) *let* a *case*	$\textbf{let } v = e_v \textbf{ in } e$	$\textbf{case } e_v \textbf{ of } v \to e$
(g) Desencapsular *let* a *case*	$\textbf{let } v = e_v \textbf{ in } e$	$\textbf{case } e_v \textbf{ of}$ $C\ v_1 \ldots v_n \to \textbf{let } v = C\ v_1 \ldots v_n \textbf{ in } e$

Figura 5.1: Transformaciones con riesgos en GHC

Si `e1` es no determinista, la semántica de `f` es una función no determinista. Luego $[\![f\ 5 - f\ 5]\!]\ \rho$ producirá un conjunto no unitario de valores, ya que `x` se evalúa cada vez que se aplica f. La semántica de la expresión ligada a `f'` es sin embargo un conjunto de funciones deterministas, y debido a la definitud de las variables `x` y `f'`, $[\![f'\ 5 - f'\ 5]\!]\ \rho$ se evalúa siempre a $\{0\}$. Es decir, la semántica ha cambiado después de la transformación de *full laziness*.

El compilador detectaría las ligaduras no deterministas y deshabilitaría en dichas situaciones el flotamiento de un `let` fuera de una lambda.

Transformación de argumentos estáticos. Esta transformación (Figura 5.1b) se aplica a definiciones recursivas. Si un argumento tiene el mismo valor en todas las llamadas recursivas de la función, se dice que es un argumento estático. En tal caso se puede definir una función cuyo argumento estático es una variable libre y que se comporta de la misma forma que la función original. Por ejemplo:

$$
\begin{array}{l}
\mathit{foldr}\ f\ z\ l = \\
\quad \textbf{case } l \textbf{ of} \\
\qquad [\,] \to z \qquad\qquad\qquad\qquad\qquad \Rightarrow \\
\qquad (a : as) \to \textbf{let } v = \mathit{foldr}\ f\ z\ as \\
\qquad\qquad\qquad \textbf{in } f\ a\ v
\end{array}
$$

$$
\begin{array}{l}
\mathit{foldr}\ f\ z\ l = \\
\quad \mathbf{let}\ \mathit{foldr'}\ l = \\
\qquad \mathbf{case}\ l\ \mathbf{of} \\
\qquad\quad [\,] \rightarrow z \\
\qquad\quad (a : as) \rightarrow \mathbf{let}\ v = \mathit{foldr}\ f\ z\ as \\
\qquad\qquad\qquad\qquad \mathbf{in}\ f\ a\ v \\
\quad \mathbf{in}\ \mathit{foldr'}\ l
\end{array}
$$

El problema surge cuando la parte no estática de la función, es decir, la aplicación parcial de la función a sus argumentos estáticos (deben ser explícitos para que se pueda aplicar la transformación) no es una forma normal débil de cabeza. Si lo es, no hay problema ya que la nueva función es una lambda abstracción cuyo comportamiento es el mismo que el comportamiento de la aplicación parcial de la función original a sus argumentos estáticos. Pero, si es necesario evaluar una expresión no determinista para obtener la forma normal débil de cabeza, entonces la nueva función y la original podrían no tener el mismo comportamiento: antes de la transformación, la expresión no determinista se evalúa en cada llamada recursiva, mientras que despues de la transformación solamente se evalúa una vez.

Veamos un ejemplo. Consideremos la función recursiva

$$
\begin{array}{l}
f\ x = \mathbf{let} \\
\qquad g = \lambda xs.\mathbf{case}\ xs\ \mathbf{of} \\
\qquad\qquad\qquad [\,] \rightarrow x \\
\qquad\qquad\qquad a : as \rightarrow f\ x\ as \\
\qquad h = \lambda xs.\mathbf{case}\ xs\ \mathbf{of} \\
\qquad\qquad\qquad [\,] \rightarrow 2 * x \\
\qquad\qquad\qquad a : as \rightarrow f\ x\ [\,] \\
\quad \mathbf{in} \\
\qquad hd(merge\ \#\ [[g],[h]])
\end{array}
$$

donde se observa claramente que x es un parámetro que se mantiene en las llamadas recursivas. En esta función, en cada llamada recursiva se escoge una función entre g y h de forma no determinista. La evaluación de una aplicación de esta función puede terminar aunque la lista sea infinita; basta con que en algún momento se escoja h como valor de la función. La función, en caso de que termine, devuelve x o el doble de x. Si la transformamos en

$$
\begin{array}{l}
f\ x = \textbf{let} \\
\qquad f' = \textbf{let} \\
\qquad\qquad g = \lambda xs.\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad [\,] \rightarrow x \\
\qquad\qquad\qquad a : as \rightarrow f'\ as \\
\qquad\qquad h = \lambda xs.\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad [\,] \rightarrow 2 * x \\
\qquad\qquad\qquad a : as \rightarrow f'\ [\,] \\
\qquad\quad \textbf{in} \\
\qquad\qquad hd(merge\ \#\ [[g],[h]]) \\
\qquad \textbf{in}\ f'
\end{array}
$$

perdemos el no determinismo y además la posibilidad de terminación cuando la lista es infinita. Ahora en la primera llamada, f' escogerá entre g y h actualizándose con el valor escogido, fijo ya para todas las llamadas recursivas. Si escoge a g por primera vez y la lista es infinita, con toda seguridad no termina, y si escoge a h va a devolver el doble de x, no pudiendo nunca devolver x.

Especialización. Transforma (Figura 5.1c) aplicaciones parciales a tipos y diccionarios en ligaduras *let*:

$$
\begin{array}{l}
g = \Lambda ty.\lambda dict.\lambda y. \\
\qquad \textbf{let}\ f = \Lambda ty.\lambda dict.e \\
\qquad \textbf{in}\ f\ ty\ dict\ (f\ ty\ dict\ y)
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
g = \Lambda ty.\lambda dict.\lambda y. \\
\qquad \textbf{let}\ f = \Lambda ty.\lambda dict.e \\
\qquad \textbf{in} \\
\qquad\quad \textbf{let}\ f' = f\ ty\ dict \\
\qquad\quad \textbf{in}\ f'\ (f'\ y)
\end{array}
$$

Supongamos que e es una expresión que genera una función de forma no determinista. Esto significa que las dos aplicaciones parciales $f\ ty\ dict$ que aparecen en el cuerpo de g denotan funciones potencialmente diferentes. Una vez aplicada la transformación, la primera vez que se evalúa f', se actualizará con su forma normal débil de cabeza, de forma que las dos apariciones de f' denotan necesariamente la misma función.

5.2.2 Transformaciones que afectan a las concreciones de procesos

Las tres transformaciones anteriores reducen el número de veces que se lanza un proceso. Si, por ejemplo, consideremos la transformación *full laziness* (Figura 5.1a) y suponemos que e contiene una concreción de proceso, antes de aplicar la regla, el proceso sería lanzado varias veces, una por cada aplicación de la función g; después de aplicar la regla, el proceso sería lanzado solamente una vez, cuando la función se aplica por primera vez. Lo mismo sucede en las otras dos transformaciones.

5.2.3 Transformaciones con otros efectos peligrosos

Algunas reglas son correctas con respecto a la semántica de valores calculados por el programa, pero pueden afectar de alguna manera a su semántica operacional, y por tanto a su coste. Las posibles modificaciones son el cambio de trabajo entre un proceso padre y uno hijo (e, f y g de la Figura 5.1), el cambio del coste de comunicación (d, e, f y g de la Figura 5.1) o el cambio en el coste en espacio (d, e, f y g de la Figura 5.1).

Estos cambios a veces son positivos y otras negativos. Adicionalmente, el incremento de oportunidades para la aplicación de otras reglas ha de considerarse un efecto positivo, por lo que se ha decidido dejar estas reglas activas.

5.3 El lenguaje

El lenguaje que va a ser analizado es una extensión de Core-Haskell [PHH+93], es decir un lenguaje funcional simple con polimorfismo de segundo orden, por lo que incluye abstracción de tipo y aplicación a un tipo.

En la Figura 5.2 se muestran la sintaxis del lenguaje y de las expresiones de tipo. Se utiliza v para denotar una variable, k para denotar un literal, x para denotar un átomo (variable o literal), y T para denotar un constructor de tipo. Un programa es una lista de ligaduras posiblemente recursivas de variables a expresiones. Una expresión puede ser una variable, una lambda abstracción, una aplicación de una expresión funcional a un átomo, una aplicación de un constructor, una aplicación de un operador primitivo, una expresión **case** o una expresión **let**. Las aplicaciones de constructores y operadores primitivos son saturadas. Las variables ya contienen información de tipos, por lo que no la escribiremos explícitamente en las expresiones. Cuando sea necesario escribiremos $e :: t$ para hacer explícito el tipo de una expresión.

Un tipo puede ser un tipo básico K, un tipo tupla $(t_1, \ldots, t_m)$, un tipo algebraico $T\ t_1 \ldots t_m$, un tipo funcional $t_1 \rightarrow t_2$ o un tipo polimórfico $\forall \beta.t$.

Las nuevas expresiones que provienen de Edén son la abstracción de proceso **process** $v \rightarrow e$, y la concreción de proceso $v \# x$. También se añade el tipo $Process\ t_1\ t_2$, que representa el tipo de una abstracción de proceso **process** $v \rightarrow e$, donde v tiene tipo t_1 y e tiene tipo t_2. A menudo, t_1 y t_2 son tipos tupla y en tal caso cada elemento de la tupla representa un canal de entrada/salida del proceso.

El polimorfismo de segundo orden de Core. Como ya hemos dicho, Core es un lenguaje con polimorfismo de segundo orden, donde aparecen

$prog$	$\rightarrow$	$bind_1; \ldots; bind_m$		
$bind$	$\rightarrow$	$v = expr$		{ligadura no recursiva}
		$\mid$ **rec** $v_1 = expr_1; \ldots; v_m = expr_m$		{ligadura recursiva}
$expr$	$\rightarrow$	$expr\ x$		{aplicación a un átomo}
		$\mid \lambda v.expr$		{lambda abstracción}
		$\mid$ **case** $expr$ **of** $alts$		{expresión *case*}
		$\mid$ **let** $bind$ **in** $expr$		{expresión *let*}
		$\mid C\ x_1 \ldots x_m$		{aplicación saturada de constructor}
		$\mid op\ x_1 \ldots x_m$		{aplicación saturada de op. primitivo}
		$\mid x$		{átomo}
		$\mid \Lambda\alpha.expr$		{abstracción de tipo}
		$\mid expr\ type$		{aplicación de tipo}
		$\mid v\ \#\ x$		{concreción de proceso}
		$\mid$ **process** $v \rightarrow expr$		{abstracción de proceso}
$alts$	$\rightarrow$	$Calt_1; \ldots; Calt_m; [Deft]$	$m \geq 0$	
		$\mid Lalt_1; \ldots; Lalt_m; [Deft]$	$m \geq 0$	
$Calt$	$\rightarrow$	$C\ v_1 \ldots v_m \rightarrow expr$	$m \geq 0$	{alternativa algebraica}
$Lalt$	$\rightarrow$	$k \rightarrow expr$		{alternativa primitiva}
$Deft$	$\rightarrow$	$v \rightarrow expr$		{alternativa por defecto}
$type$	$\rightarrow$	K		{tipos básicos: enteros, caracteres}
		$\mid \alpha$		{variables de tipo}
		$\mid T\ type_1 \ldots type_m$		{aplicación de constructor de tipo}
		$\mid type_1 \rightarrow type_2$		{tipo función}
		$\mid Process\ type_1\ type_2$		{tipo proceso}
		$\mid \forall\alpha.type$		{tipo polimórfico}

Figura 5.2: Definición del lenguaje y de las expresiones de tipo

abstracciones y aplicaciones de tipos. Sin embargo, el polimorfismo de segundo orden se utiliza solamente como mecanismo para mantener los tipos a lo largo de las transformaciones realizadas en Core [PS98b]. El compilador GHC infiere los tipos (Hindley-Milner) de cada expresión y variable del programa. Si dicha asignación de tipos se mantuviera en la traducción de Haskell a Core y en las subsiguientes transformaciones dentro de Core, tanto algunas transformaciones como el generador de código podrían beneficiarse de la información de tipos para generar mejor código (GHC lo hace). El problema es que la transformación de los programas implica manipulación de los tipos. No es suficiente con anotar cada variable del programa original con su tipo, ya que dicha información no perdura a lo largo de las transformaciones. Por ejemplo, consideremos la función

$$\texttt{compose} :: \forall\alpha\beta\gamma.(\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$$

que podría definirse en un Core sin tipos como

```
compose = \ f g x.let y = g x in f y
```

Supongamos que queremos desarrollar (mediante una transformación llamada *inlining*) una llamada concreta a `compose`, como compose show double v $\Rightarrow$ let y = double v in show y

donde v es un Int, double lo duplica y show convierte su resultado a un String. Ahora queremos identificar el tipo de cada variable y subexpresión por lo que debemos calcular el tipo de y. El problema es que su tipo en compose es una variable de tipo β, por lo que aunque aquí tenga un tipo Int, en otra aplicación podría tener un tipo diferente. Por ejemplo, en

compose toUpper show v $\Rightarrow$ let y = show v in toUpper y

donde toUpper convierte un String en otro, y tiene tipo String.

La solución a este problema es que cada función polimórfica tenga una abstracción de tipo por cada variable cuantificada universalmente en su tipo y cada vez que se llame a una función polimórfica, se le pasan argumentos extra para indicar los tipos con los que se concretan las variables de tipo. Por ejemplo, la definición de compose pasa a ser

```
compose = /\ a b c.\ f::(b->c) g::(a -> b) x::a.
               let y::b = g x in f y
```

La función tiene ahora tres parámetros de tipo (a, b y c) y se puede dar de forma explícita los tipos de los parámetros f, g y x, así como el tipo de la variable local y. Ahora, la primera de las llamadas vistas antes se escribiría

```
compose Int Int String show double v
```

El desarrollo de la llamada produce ahora directamente

```
let y::Int = double v in show y
```

Es decir, ahora a y se le ha asignado automáticamente el tipo correcto.

Por tanto, en Core, el polimorfismo de segundo orden proporciona solamente una notación con la que expresar y transformar programas polimórficos Hindley-Milner. Por esta razón podemos aplicar y aplicaremos a los análisis definidos en este capítulo los resultados obtenidos por Baraki [Bar93] para un lenguaje funcional con polimorfismo Hindley-Milner, descritos en la Sección 2.2.3, para aproximar el valor abstracto de un ejemplar de una función polimórfica a partir del valor abstracto de su ejemplar más pequeño.

5.4 El primer análisis

En esta sección vamos a definir un primer análisis sencillo de no determinismo. Se trata de un análisis lineal, con algunas limitaciones que describiremos más adelante. En primer lugar, definiremos los dominios abstractos utilizados y después la interpretación abstracta. A continuación, mostraremos sus resultados para una versión simplificada de un esqueleto Edén, los trabajadores replicados. Finalmente describiremos sus limitaciones y el motivo de las mismas, el cual nos conducirá a la definición del análisis de la siguiente sección.

$$
\begin{array}{l}
Basic = \{d, n\} \text{ donde } d \sqsubseteq n \\
D1_K = D1_{T\ t_1..t_m} = D1_\alpha = Basic \\
D1_{(t_1,\ldots,t_m)} = \{(b_1,\ldots,b_m) \mid b_i \in Basic\} \\
D1_{t_1 \to t_2} = D1_{Process\ t_1\ t_2} = D1_{t_2} \\
D1_{\forall \alpha.t} = D1_t
\end{array}
$$

Figura 5.3: Dominios abstractos para el primer análisis

$$
\begin{array}{lclclcl}
n \sqcup b & = & n & & \widehat{\sqcup} b & = & b \\
d \sqcup b & = & b & & \widehat{\sqcup}(b_1,\ldots,b_m) & = & \bigsqcup_i b_i \\
\multicolumn{7}{c}{b \sqcup (b_1,\ldots,b_m) = (b_1 \sqcup b, \ldots, b_m \sqcup b)}
\end{array}
$$

Figura 5.4: Operadores de mínima cota superior y de aplanamiento

5.4.1 Dominios abstractos

En la Figura 5.3 se muestran los dominios abstractos del primer análisis. El dominio *Basic* es el dominio abstracto correspondiente a los tipos básicos y algebraicos, exceptuando las tuplas. Está formado por dos valores: d representa *determinismo* y n *posible no determinismo*, con $d \sqsubseteq n$.

Los procesos suelen tener varios canales de entrada y salida, hecho que se representa mediante el uso de tuplas. En la implementación, se crea una hebra concurrente independiente para cada salida de un proceso. Nos gustaría expresar cuáles de ellas son deterministas y cuáles posiblemente no deterministas. Por ejemplo, en la abstracción de proceso

```
process v →  case v of (v1,v2) →  let y1 = v1 in
                                  let y2 = merge # v2 in (y1,y2)
```

tendremos que la primera salida es determinista mientras que la segunda es posiblemente no determinista. Por ello, las tuplas se tratan de forma especial como tuplas de valores abstractos básicos[2]. Puesto que las tuplas internas no representan canales, solamente mantendremos la tupla externa y no se permitirán tuplas anidadas.

El orden entre valores básicos se extiende de forma natural a las tuplas. Se pueden definir varios operadores de mínima cota superior (lub) así como un operador $\widehat{\sqcup}$ para aplanar las tuplas internas (ver Figura 5.4).

Los dominios correspondientes a las funciones y los procesos *se identifican con los dominios de sus rangos*. Esta es la diferencia más importante

[2]Se usa b para denotar un valor abstracto básico y a para denotar un valor básico o una tupla de valores básicos.

$$
\begin{array}{l}
-_t : Basic \rightarrow D_{1t} \\
b_{(t_1,\ldots,t_m)} = (\overset{1}{b}, \ldots, \overset{m}{b}) \\
b_{t_1 \rightarrow t_2} = b_{Process\ t_1\ t_2} = b_{t_2} \\
b_{\forall\alpha.t} = b_t \\
b_t = b \text{ si } t = K, \alpha, T\ t_1 \ldots t_m
\end{array}
$$

Figura 5.5: Definición de la función de adaptación

con el segundo análisis. El dominio abstracto de un tipo polimórfico es el de su ejemplar más pequeño [Bar93], es decir, aquel en el que la variable de tipo se sustituye por un tipo básico K. Luego el dominio correspondiente a una variable de tipo es $Basic$.

5.4.2 Interpretación abstracta

La interpretación abstracta se muestra en la Figura 5.6. La interpretación de una variable se obtiene del entorno. Los literales son deterministas, por lo que su valor abstracto es d. La interpretación de una tupla es una tupla de valores abstractos básicos, luego el valor abstracto de cada componente ha de ser previamente aplanado usando $\widehat{\sqcup}$. En la interpretación de un valor construido se debe ir un paso más allá: una vez se han aplanado las componentes, se debe aplicar el operador de mínima cota superior para obtener finalmente un valor básico. Esto significa que la información de las componentes se pierde.

Puesto que `merge` es la única fuente de no determinismo, diremos a grandes rasgos que una expresión es no determinista cuando "contiene" cualquier concreción de dicho proceso. Luego consideraremos que una función o un proceso es determinista si no genera resultados no deterministas a partir de argumentos deterministas. Por ello, la interpretación de una función y de un proceso es la interpretación de su cuerpo cuando se le da a su argumento un valor abstracto determinista. Dicho valor es en realidad una adaptación del valor abstracto básico d al tipo del argumento (ver Figura 5.5). Dado un tipo t, $-_t$ toma un valor abstracto básico b y devuelve un valor abstracto en D_{1t}. Su comportamiento es el opuesto al del operador de aplanamiento: si t es una m-tupla, replica b para obtener la m-tupla $(b, \ldots, b)$. Obsérvese que se cumple $\widehat{\sqcup} b_t = b$ y $(\widehat{\sqcup} a)_t \sqsupseteq a$.

Volviendo a la interpretación de una función, si el tipo del argumento es una m-tupla, el argumento debería ser a su vez una m-tupla $(d, \ldots, d)$. Si el argumento es no determinista supondremos que el resultado es no determinista. Esto implica que no se expresa la forma en que el resultado de la función depende de su argumento, lo que motivará las limitaciones del análisis. La eliminación de dicha información viene reflejada en la interpre-

$$
\begin{array}{l}
[\![v]\!]_1\ \rho_1 = \rho_1\ v \\
[\![k]\!]_1\ \rho_1 = d \\
[\![(x_1,\ldots,x_m)]\!]_1\ \rho_1 = (\widehat{\sqcup}([\![x_1]\!]_1\ \rho_1),\ldots,\widehat{\sqcup}([\![x_m]\!]_1\ \rho_1)) \\
[\![C\ x_1\ldots x_m]\!]_1\ \rho_1 = \bigsqcup_i \widehat{\sqcup}([\![x_i]\!]_1\ \rho_1) \\
[\![e\ x]\!]_1\ \rho_1 = (\widehat{\sqcup}([\![x]\!]_1\ \rho_1)) \sqcup [\![e]\!]_1\ \rho_1 \\
[\![op\ x_1\ldots x_m]\!]_1\ \rho_1 = (\bigsqcup_i \widehat{\sqcup}([\![x_i]\!]_1\ \rho_1))_t \text{ donde } op :: t_1 \to (t_2 \to \ldots (t_m \to t)) \\
[\![p\#x]\!]_1\ \rho_1 = \widehat{\sqcup}([\![x]\!]_1\ \rho_1) \sqcup [\![p]\!]_1\ \rho_1 \\
[\![\lambda v.e]\!]_1\ \rho_1 = [\![e]\!]_1\ \rho_1\ [v \mapsto d_t] \text{ donde } v :: t \\
[\![\mathbf{process}\ v \to e]\!]_1\ \rho_1 = [\![e]\!]_1\ \rho_1\ [v \mapsto d_t] \text{ donde } v :: t \\
[\![merge]\!]_1\ \rho_1 = n \\
[\![\mathbf{let}\ v = e\ \mathbf{in}\ e']\!]_1\ \rho_1 = [\![e']\!]_1\ \rho_1\ [v \mapsto [\![e]\!]_1\ \rho_1] \\
[\![\mathbf{let\ rec}\ \overline{\{v_i = e_i\}}\ \mathbf{in}\ e']\!]_1\ \rho_1 = [\![e']\!]_1\ (\mathit{fix}\ (\lambda\rho_1'.\rho_1\ \overline{[v_i \mapsto [\![e_i]\!]_1\ \rho_1']})) \\
[\![\mathbf{case}\ e\ \mathbf{of}\ (v_1,\ldots,v_m) \to e']\!]_1\ \rho_1 = [\![e']\!]_1\ \rho_1\ \overline{[v_i \mapsto \pi_i([\![e]\!]_1\ \rho_1)_{t_i}]} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{donde } v_i :: t_i \\
[\![\mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i}]\!]_1\ \rho_1 = b \sqcup (\bigsqcup_i [\![e_i]\!]_1\ \rho_{1i}) \\
\qquad\qquad\qquad\qquad \text{donde } b = [\![e]\!]_1\ \rho_1 \text{ y } \rho_{1i} = \rho_1\ \overline{[v_{ij} \mapsto b_{t_{ij}}]}, v_{ij} :: t_{ij} \\
[\![\mathbf{case}\ e\ \mathbf{of}\ \overline{k_i \to e_i}]\!]_1\ \rho_1 = [\![e]\!]_1\ \rho_1 \sqcup (\bigsqcup_i [\![e_i]\!]_1\ \rho_1) \\
[\![\Lambda\alpha.e]\!]_1\ \rho_1 = [\![e]\!]_1\ \rho_1 \\
[\![e\ t]\!]_1\ \rho_1 = ([\![e]\!]_1\ \rho_1)_{tinst} \text{ donde } (e\ t) :: tinst
\end{array}
$$

Figura 5.6: Interpretación abstracta del primer análisis

tación de la aplicación de una función: el resultado de la aplicación es no determinista o bien cuando la función es no determinista o bien si lo es el argumento, lo cual se expresa usando el operador de mínima cota superior.

Si el argumento es de tipo tupla, debemos aplanar previamente sus componentes, para saber si es determinista o no (lo que implica que alguna de sus componentes no lo es). La información proporcionada por las componentes independientes solamente podrá ser aprovechada cuando estas se usan separadamente en distintas partes del programa.

Veamos un ejemplo. Sea $h = \lambda v_1.\lambda v_2.\mathbf{let}\ v_3 = merge\ \#\ v_2\ \mathbf{in}\ (3, v_3)$. El valor abstracto de esta función es (d, n). Sobre este ejemplo, el análisis es capaz de decirnos que la primera componente es determinista, mientras que la segunda puede ser no determinista. Sin embargo, cuando esta función se aplica a un argumento no determinista, perdemos dicha información: $[\![(h\ x)\ y]\!]_1\ \ [x \mapsto n, y \mapsto d] = (n, n)$. Esto se debe a que no se puede expresar que el hecho de que la primera componente es determinista es independiente de los argumentos. Este problema se resolverá en el segundo análisis.

La interpretación de una abstracción de proceso es similar a la interpretación de una lambda abstracción, y la de una concreción de proceso es similar a la de la aplicación de una función. El proceso `merge` es no determinista por lo que su valor abstracto es n.

Un operador primitivo se considera determinista, por lo que su aplicación sólo será no determinista cuando alguno de sus argumentos lo sea. Por ello se aplanan los valores abstractos de sus argumentos y se toma su mínima cota superior. Finalmente, el valor básico resultante debe adaptarse al tipo del resultado del operador.

En la expresión **let** recursiva, el punto fijo se puede obtener usando la cadena ascendente de Kleene:

$$[\![\mathbf{let\ rec}\ \overline{\{v_i = e_i\}}\ \mathbf{in}\ e']\!]\ \rho = [\![e']\!]\ (\bigsqcup_{n\in\mathbb{N}}(\lambda\rho'.\rho\ \overline{[v_i \rightarrow [\![e_i]\!]\ \rho']})^n(\rho_0)),$$

donde ρ_0 es el entorno en el que todas las variables tienen como valor abstracto el ínfimo $\bot_t$ de su correspondiente dominio abstracto:

$$\begin{array}{l}
\bot_K = \bot_{T\ t_1 \ldots t_n} = \bot_\alpha = d \\
\bot_{(t_1,\ldots,t_n)} = (d,\ldots,d) \\
\bot_{t_1 \rightarrow t_2} = \bot_{Process\ t_1\ t_2} = \bot_{t_2} \\
\bot_{\forall\beta.t'} = \bot_{t'}.
\end{array}$$

Nótese que para cada tipo t, $d_t = \bot_t$ (se puede demostrar por inducción estructural sobre t).

En cada iteración se calculan los valores abstractos de los lados derechos de las ligaduras y el entorno se actualiza con esos nuevos valores. La terminación se produce cuando no hay cambios en el entorno, y está asegurada, ya que los dominios abstractos correspondientes a todos los tipos son finitos. El número de iteraciones es lineal con el número de componentes de tuplas que haya en las ligaduras: una por cada variable que no tiene tipo tupla, y una por cada componente de cada variable de tipo tupla.

Hay tres tipos diferentes de expresiones **case**, para tipos tupla, tipos algebraicos y tipos primitivos. Veamos el caso algebraico, los demás son más sencillos. Una expresión **case** es no determinista si lo es la expresión del discriminante (la elección entre las alternativas es no determinista) o alguna de las expresiones de las alternativas. El valor abstracto b del discriminante e pertenece a *Basic*. Es decir, al interpretar el discriminante, se ha perdido la información sobre sus componentes. Ahora se desea interpretar los lados derechos de las alternativas en entornos extendidos con valores abstractos para las variables $v_{ij} :: t_{ij}$ de los lados izquierdos de las mismas. No se dispone de tales valores abstractos, pero se pueden obtener aproximaciones a ellos usando la función de adaptación definida en la Figura 5.5.

La interpretación del **case** primitivo es análogo al caso algebraico con la salvedad de que no es necesario extender los entornos puesto que no hay variables en los lados izquierdos de las alternativas.

En el caso de que el discriminante sea de tipo tupla, se dispone de valores abstractos básicos para cada una de las componentes, luego es necesario adaptar cada una de ellas al tipo correspondiente. Se mantiene el constructor externo de tuplas, con lo que se pierde menos información que en el caso algebraico.

No hemos mostrado en la Figura 5.6 el caso de las alternativas por defecto para no oscurecer la notación. Si aparece en cualquiera de las expresiones **case** una alternativa por defecto $v \to e$, se obtiene el valor abstracto de e en un nuevo entorno extendido con $[v \to b]$ donde b es el valor abstracto del discriminante, y se calcula la mínima cota superior con el resto de alternativas. Por ejemplo:

$$\begin{array}{l}
[\![\textbf{case } e \textbf{ of } \overline{C_i\ \overline{v_{ij}} \to e_i};\ v \to e_v]\!]_1\ \rho_1 = b \sqcup (\bigsqcup_i [\![e_i]\!]_1\ \rho_{1i}) \sqcup b_v \\
\quad \text{donde } b = [\![e]\!]_1\ \rho_1,\ \rho_{1i} = \rho_1\ \overline{[v_{ij} \mapsto b_{t_{ij}}]}, v_{ij} :: t_{ij} \\
\qquad\quad b_v = [\![e_v]\!]_1\ \rho_1\ [v \to b]
\end{array}$$

El valor abstracto de una abstracción de tipo es el de su cuerpo, que coincide así con el valor abstracto de su ejemplar más pequeño. Para interpretar una aplicación de tipo, es necesario adaptar el valor abstracto de la abstracción de tipo al tipo concretado, ya que puede surgir nueva estructura (tupla) a raíz de la sustitución. Por ejemplo, si $e :: t_e$, donde $t_e = \forall\beta.\beta \to \beta$ y $t = (Int, Int)$, entonces $[\![e]\!]_1\ \rho \in Basic$, mientras que $[\![e\ t]\!]_1\ \rho \in (Basic \times Basic)$. Si no surge nueva estructura (es decir, el dominio correspondiente al ejemplar más pequeño ya es un producto cartesiano), entonces no es necesario llevar a cabo ninguna adaptación. Sobrecargando la notación de la función de adaptación, se puede escribir $(b_1, \ldots, b_m)_{(t_1,\ldots,t_m)} = (b_1, \ldots, b_m)$. Por ejemplo, si $t_e = \forall\beta.\beta \to (\beta,\beta)$ y $t = (Int, Int)$, entonces $[\![e]\!]_1\ \rho \in (Basic \times Basic)$ y también $[\![e\ t]\!]_1\ \rho \in (Basic \times Basic)$.

Un ejemplo: los trabajadores replicados. Este ejemplo es una versión simplificada de una topología de trabajadores replicados (*replicated workers*) [KPR01]. Se dispone de un proceso gestor y n procesos trabajadores. El gestor proporciona tareas a los trabajadores. Cuando alguno de los trabajadores termina su tarea, envía un mensaje al gestor conteniendo los resultados obtenidos y solicita una nueva tarea. Para que el proceso gestor pueda recibir las respuestas de los trabajadores en cualquier orden y asignar inmediatamente nuevas tareas a los procesos ociosos, es necesario utilizar un proceso `merge`.

La función `rw` que representa este esquema para $n = 2$ se muestra en la Figura 5.7, donde `worker` es el proceso trabajador y `ts` es una lista inicial de tareas a realizar por los trabajadores. La salida del proceso gestor `manager` normalmente depende de ambas listas de entrada: la de tareas iniciales `ts` y la producida por los trabajadores `os`. Sin embargo, para comparar la potencia de este análisis con el presentado en la Sección 5.5 supondremos ahora que el proceso `manager` está definido de la siguiente forma:

```
rw = λ worker.λ ts.
       let rec
        t = (ts, is)
        ys = manager # t
        y1 = case ys of (z1,z2,z3) → z1
        y2 = case ys of (u1,u2,u3) → u2
        y3 = case ys of (v1,v2,v3) → v3
        o1 = worker # y1
        o2 = worker # y2
        os = [o1, o2]
        is = merge # os
       in y3

manager = process ts → case ts of
             (t1, t2) → (g t1 t2,h t1 t2,r t1)
```

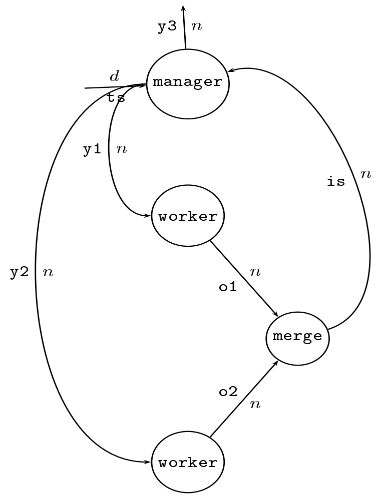


Figura 5.7: Estructura de procesos de los trabajadores replicados para $n = 2$.

```
manager = process ts → case ts of
                         (t1, t2) → (g t1 t2,h t1 t2,r t1)
```

donde `g`, `h` y `r` son funciones deterministas (es decir, tienen como valor abstracto d). Entonces el valor abstracto de `manager` es (d, d, d). Con esta definición, la tercera componente de la salida del proceso sólo depende de la primera lista de entrada, lo que implica que el resultado final de la función `rw` sólo depende de la lista inicial de tareas. En la Figura 5.7 se muestra la topología con los valores abstractos anotados en cada canal. Puesto que las definiciones son mutuamente recursivas, el valor abstracto n se propaga a todos los canales. Sin embargo, sabemos que cuando `ts` es determinista, el resultado de la función también será determinista, ya que solamente depende de la lista inicial, y no de la que procede del proceso `merge`. La respuesta del análisis es segura pero solamente aproximada. En la Sección 5.5 se presentará un análisis más potente. Usando dicho análisis el resultado para este ejemplo será más ajustado.

En las aplicaciones reales de este esquema, aunque el proceso gestor recibe el resultado de los trabajadores en un orden cualquiera, los ordena, de forma que la salida final es totalmente determinista. Sin embargo, esto no puede ser detectado por este análisis ni por ninguno de los aquí presentados, sino que siempre obtendremos un valor abstracto n. Esto es así porque el determinismo final de este esquema viene establecido a partir de la semántica de la función de ordenación, lo que no se puede tener en cuenta en un análisis de este tipo.

5.4.3 Limitaciones

Recordemos que en este análisis no hay dominios abstractos funcionales, sino que el dominio abstracto correspondiente a una función o a un proceso es el correspondiente al tipo de su resultado. Esto significa que el cálculo del punto fijo no es costoso, pero también impone algunas limitaciones al análisis. Por ejemplo, no se puede expresar la dependencia del resultado de una función con respecto a su argumento.

Como ya se ha dicho anteriormente, en la aplicación de una función o en la concreción de un proceso, no se puede usar la información proporcionada por el argumento en toda su extensión. Esto sucede, por ejemplo, cuando el resultado de la función no depende de ninguno de sus argumentos, es decir, se trata de una función constante. Si definimos la función $f\ v = 5$, el análisis nos dirá que la función es determinista, pero cuando se aplica f a un valor posiblemente no determinista, el resultado de la aplicación se establece como posiblemente no determinista. Sin embargo, sabemos que esto no es así, pues la función siempre produce un único valor. Por supuesto, el resultado del análisis es una aproximación segura, pero no es muy ajustada: la función $g\ v = v$ tiene exactamente el mismo comportamiento abstracto. Tanto f como g son funciones deterministas pero tienen distintos niveles de determinismo: f no depende de su argumento, pero g sí lo hace. Si las funciones y los procesos fueran interpretados como funciones, esta limitación desaparecería. La función abstracta $f^{\#}$ correspondiente a f sería $f^{\#} = \lambda z.d$, mientras que la correspondiente a g, $g^{\#}$, sería $\lambda z.z$. Ahora, si se aplicara $f^{\#}$ a n, se obtendría d, mientras que en el caso de $g^{\#}$ obtenemos n.

Sucede lo mismo cuando hay tuplas implicadas; el valor abstracto de

$$h\ v_1\ v_2\ v_3 = \mathbf{let}\ u = \mathit{merge}\ \#\ v_3\ \mathbf{in}\ (v_1, v_2, u)$$

es (d, d, n). Pero cuando la aplicamos, si alguno de sus argumentos tiene n como valor abstracto, el resultado de la aplicación será (n, n, n). Sin embargo, si el valor abstracto de h fuera

$$h^{\#}\ {v_1}^{\#}\ {v_2}^{\#}\ {v_3}^{\#} = ({v_1}^{\#}, {v_2}^{\#}, n)$$

no perderíamos tanta información. Por ejemplo, obtendríamos $h^{\#}\ d\ n\ d = (d, n, n)$.

Luego la solución parece ser interpretar las funciones y los procesos como funciones abstractas, y eso será lo que haremos a continuación.

5.4.4 Tipos algebraicos

El valor abstracto de un valor construido es un valor abstracto básico (excepto en las tuplas). Esto significa que si alguno de los elementos de una lista es posiblemente no determinista, entonces la lista completa es considerada como posiblemente no determinista. Nos preguntamos si se pierde mucha información al interpretar los tipos algebraicos de esta manera y si sería útil utilizar un dominio de cuatro puntos similar al de Wadler [Wad87]. Concentrémonos por ejemplo en las listas de enteros *List Int*. Podríamos diferenciar cuatro casos atendiendo a dos aspectos: la forma en que se han generado los elementos, y la forma en que se ha generado la lista. En nuestro análisis estamos identificando tres de los casos: aquellos en los que o bien los elementos o la lista (o ambos) son generados de forma no determinista. ¿Obtendríamos alguna ventaja de separarlos en tres casos? Por ejemplo, $merge\#[[0,0..],[1,1..]]$ sería una lista de enteros deterministas que ha sido generada de forma no determinista. Sin embargo, al tomar un elemento de la lista, por ejemplo la cabeza, hemos de olvidarnos de la forma en que se han generado los elementos ya que no sabemos qué valor va a aparecer en cada posición. Luego no parece muy útil conocer cómo se han generado los elementos cuando la propia lista es no determinista. Otra de las combinaciones es generar una lista de forma determinista (por ejemplo, aplicando el constructor de las listas) a partir de elementos generados de forma no determinista. En este tipo de listas, la aplicación de aquellas funciones que no hacen uso de los elementos podrían producir un resultado determinista, por ejemplo la función *length*. Por lo tanto, a lo sumo valdría la pena añadir un caso más. Sin embargo, no consideramos que se esté perdiendo demasiada información con dicho caso, por lo que renunciamos a introducir este nivel adicional de detalle.

5.5 El segundo análisis

5.5.1 Introducción

Para superar las limitaciones del análisis anterior definimos ahora un nuevo análisis donde interpretar las funciones y los procesos como funciones abstractas nos permitirá expresar diferentes niveles de determinismo y no determinismo, por ejemplo $\lambda z.d \sqsubseteq \lambda z.z \sqsubseteq \lambda z.n$. Se trata también de un análisis basado en interpretación abstracta y es del estilo de Burn, Hankin y Abramsky [BHA86].

5.5.2 Dominios abstractos

En la Figura 5.8 se muestran los dominios abstractos del segundo análisis. Al igual que en el caso anterior, el dominio abstracto correspondiente a los tipos básicos y a los algebraicos (excepto las tuplas) es el dominio básico *Basic*.

Ahora el dominio abstracto correspondiente a un tipo tupla es el producto cartesiano de los dominios correspondientes a los tipos de las componentes. En consecuencia, podemos tener tuplas anidadas y tuplas conteniendo funciones. De esta manera mantenemos una mayor cantidad de informa-

$$
\begin{array}{l}
Basic = \{d, n\} \text{ donde } d \sqsubseteq n \\
D_{2K} = D_{2T\ t_1 \ldots t_m} = Basic \\
D_{2(t_1, \ldots, t_m)} = D_{2t_1} \times \ldots \times D_{2t_m} \\
D_{2t_1 \to t_2} = D_{2Process\ t_1\ t_2} = [D_{2t_1} \to D_{2t_2}]
\end{array}
$$

Figura 5.8: Dominios abstractos para el segundo análisis

ción que en el análisis previo. Puesto que ahora interpretamos las funciones mediante funciones abstractas, nos gustaría que aquellas que aparecen dentro de una tupla (por ejemplo, si un proceso produce una función como resultado) no se pierdan debido a un aplanamiento.

Los dominios correspondientes a las funciones y los procesos son los dominios de las funciones continuas entre los dominios correspondientes al argumento y al resultado. El polimorfismo se considerará más adelante.

5.5.3 Interpretación abstracta

En la Figura 5.9 se define la interpretación abstracta. La interpretación de una tupla es ahora la tupla de valores abstractos de las componentes. La interpretación de un constructor pertenece a *Basic*. Pero, puesto que ahora cada componente $x_i :: t_i$ tiene un valor abstracto perteneciente a D_{2t_i}, no podemos aplicar directamente el operador de mínima cota superior. Antes hemos de aplanar la información de cada una de las componentes. La función responsable de ello recibe el nombre de *función de abstracción* $\alpha_t : D_{2t} \to Basic$, y se define en la Figura 5.10.

Dado un tipo t, la función de abstracción toma un valor abstracto en D_{2t} y lo aplana para obtener un valor en *Basic*. La idea es aplanar las tuplas aplicando el operador de mínima cota superior y aplicar las funciones a argumentos deterministas.

Puesto que las funciones y los procesos se interpretan como funciones abstractas, las aplicaciones de funciones y concreciones de procesos se interpretan como aplicaciones de funciones abstractas. Al proceso `merge` le damos como valor abstracto el supremo del dominio $Basic \to Basic$, que es el correspondiente a su tipo.

En una expresión **let** recursiva, de nuevo el punto fijo se puede calcular usando la cadena ascendente de Kleene. Puesto que ahora hay dominios funcionales, el número de iteraciones es, en el peor de los casos, exponencial en el número de ligaduras.

En la expresión **case** algebraica de nuevo necesitamos una aproximación segura al valor del discriminante en el dominio $D_{2t_{ij}}$ de cada variable v_{ij}. Estas aproximaciones se obtienen mediante la función $\gamma_t : Basic \to D_{2t}$, a la que llamaremos *función de concreción*, definida en la Figura 5.11.

$$
\begin{array}{l}
[\![v]\!]_2\ \rho_2 = \rho_2(v) \\
[\![k]\!]_2\ \rho_2 = d \\
[\![(x_1,\ldots,x_m)]\!]_2\ \rho_2 = ([\![x_1]\!]_2\ \rho_2,\ldots,[\![x_m]\!]_2\ \rho_2) \\
[\![C\ x_1\ldots x_m]\!]_2\ \rho_2 = \bigsqcup_i \alpha_{t_i}([\![x_i]\!]_2\ \rho_2) \text{ donde } x_i :: t_i \\
[\![e\ x]\!]_2\ \rho_2 = ([\![e]\!]_2\ \rho_2)\ ([\![x]\!]_2\ \rho_2) \\
[\![op\ x_1\ldots x_m]\!]_2\ \rho_2 = (\gamma_{t_{op}}(d))\ ([\![x_1]\!]_2\ \rho_2)\ldots([\![x_m]\!]_2\ \rho_2) \text{ donde } op :: t_{op} \\
[\![p\#x]\!]_2\ \rho_2 = ([\![p]\!]_2\ \rho_2)\ ([\![x]\!]_2\ \rho_2) \\
[\![\lambda v.e]\!]_2\ \rho_2 = \lambda z \in D_{2t_v}.[\![e]\!]_2\ \rho_2\ [v \mapsto z] \text{ donde } v :: t_v \\
[\![\mathbf{process}\ v \to e]\!]_2\ \rho_2 = \lambda z \in D_{2t_v}.[\![e]\!]_2\ \rho_2\ [v \mapsto z] \\
[\![merge]\!]_2\ \rho_2 = \lambda z \in Basic.n \\
[\![\mathbf{let}\ v = e\ \mathbf{in}\ e']\!]_2\ \rho_2 = [\![e']\!]_2\ \rho_2\ [v \mapsto [\![e]\!]_2\ \rho_2] \\
[\![\mathbf{let\ rec}\ \overline{\{v_i = e_i\}}\ \mathbf{in}\ e']\!]_2\ \rho_2 = [\![e']\!]_2\ (\mathit{fix}\ (\lambda\rho'_2.\rho_2\ \overline{[v_i \mapsto [\![e_i]\!]_2\ \rho'_2]})) \\
[\![\mathbf{case}\ e\ \mathbf{of}\ (v_1,\ldots,v_m) \to e']\!]_2\ \rho_2 = [\![e']\!]_2\ \rho_2\ \overline{[v_i \mapsto \pi_i([\![e]\!]_2\ \rho_2)]} \\
[\![\mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i}; [v \to e']]\!]_2\ \rho_2 = \begin{cases} \gamma_t(n) \text{ si } [\![e]\!]_2\ \rho_2 = n \\ \bigsqcup_i [\![e_i]\!]_2\ \rho_{2i}\ [\ \sqcup [\![e']\!]_2\ \rho'_2\ ] \text{ e.o.c.} \end{cases} \\
\qquad\qquad\qquad \text{donde } \rho_{2i} = \rho_2\ \overline{[v_{ij} \mapsto \gamma_{t_{ij}}(d)]}, v_{ij} :: t_{ij}, e_i :: t \\
\qquad\qquad\qquad\qquad \rho'_2 = \rho_2\ [v \mapsto d] \\
[\![\mathbf{case}\ e\ \mathbf{of}\ \overline{k_i \to e_i};\ [v \to e']]\!]_2\ \rho_2 = \begin{cases} \gamma_t(n) \text{ si } [\![e]\!]_2\ \rho_2 = n \\ \bigsqcup_i [\![e_i]\!]_2\ \rho_2\ [\ \sqcup [\![e']\!]_2\ \rho'_2\ ] \text{ e.o.c.} \end{cases} \\
\qquad\qquad\qquad \text{donde } e_i :: t, \rho'_2 = \rho_2\ [v \mapsto d]
\end{array}
$$

Figura 5.9: Interpretación abstracta para el segundo análisis

Las funciones de abstracción y concreción son mutuamente recursivas y en la Sección 5.5.5 veremos algunas de sus propiedades, en particular que forman una inserción de Galois. En la Figura 5.12 se muestra un ejemplo correspondiente a $t = (Int \to Int) \to Int \to Int$.

Dado un tipo t, la función de concreción γ_t *desaplana* un valor abstracto básico y produce un valor abstracto en D_{2t}. La idea detrás de esta función es obtener la mejor aproximación segura a d y n en un dominio dado. En particular, a n se le hace corresponder el supremo del dominio D_{2t}, y a d aquel valor de D_{2t} que refleja nuestra idea original de determinismo (todos los valores por debajo de él tendrán diferentes niveles de determinismo).

El caso funcional necesita explicación, siendo el resto de ellos inmediatos. Consideramos que una función es determinista si produce resultados deterministas a partir de argumentos deterministas; pero si el argumento es no determinista, el único resultado seguro que podemos devolver es no determinista. Luego el desaplanamiento de d para un tipo funcional es una función que toma un argumento, lo aplana para ver si es determinista o no y de nuevo aplica la función de concreción correspondiente al tipo del resultado.

$$\alpha_t : D_{2t} \to Basic$$
$$\alpha_K = \alpha_{T\ t_1 \ldots t_m} = \alpha_\beta = id_{Basic}$$
$$\alpha_{(t_1,\ldots,t_m)}(e_1,\ldots,e_m) = \bigsqcup_i \alpha_{t_i}(e_i)$$
$$\alpha_{Process\ t_1\ t_2}(f) = \alpha_{t_1 \to t_2}(f)$$
$$\alpha_{t_1 \to t_2}(f) = \alpha_{t_2}(f(\gamma_{t_1}(d)))$$
$$\alpha_{\forall\beta.t} = \alpha_t$$

Figura 5.10: Definición de la función de abstracción

$$\gamma_t : Basic \to D_{2t}$$
$$\gamma_K = \gamma_{T\ t_1 \ldots t_m} = \gamma_\beta = id_{Basic}$$
$$\gamma_{(t_1,\ldots,t_m)}(b) = (\gamma_{t_1}(b),\ldots,\gamma_{t_m}(b))$$
$$\gamma_{Process\ t_1\ t_2}(b) = \gamma_{t_1 \to t_2}(b)$$
$$\gamma_{t_1 \to t_2}(b) = \begin{cases} \lambda z \in D_{2t_1}.\gamma_{t_2}(n)\ si\ b = n \\ \lambda z \in D_{2t_1}.\gamma_{t_2}(\alpha_{t_1}(z))\ si\ b = d \end{cases}$$
$$\gamma_{\forall\beta.t} = \gamma_t$$

Figura 5.11: Definición de la función de concreción

El desaplanamiento de n para un tipo funcional es la función que devuelve un resultado no determinista independientemente de su argumento.

Veremos más adelante que $\gamma_t(d)$ es el mayor valor en D_{2t} con la propiedad de preservar el determinismo. Por ejemplo, para $t = Int \to Int$ se tiene que $\gamma_t(d) = \lambda z.z$, mientras que $\lambda z.d$, como es menor, también tiene la propiedad.

Para concluir, si el discriminante tiene valor abstracto n, entonces toda la expresión **case** es no determinista y consecuentemente tiene valor abstracto $\gamma_t(n)$. Si tiene valor abstracto d, se interpretan los lados derechos de las alternativas en entornos extendidos donde cada v_{ij} tiene $\gamma_{t_{ij}}(d)$ como valor abstracto. Después se calcula la mínima cota superior de todas las alternativas.

Los operadores primitivos son de nuevo deterministas. Pero ahora hay varios niveles de no determinismo. Tenemos dos opciones. Podemos asignar a cada operador *op* :: *top* el valor abstracto $\gamma_{top}(d)$. Esta es una opción segura, pero podría no ser todo lo precisa que deseamos. La segunda opción es crear un entorno inicial con los valores abstractos deseados para los operadores primitivos.

5.5.4 Polimorfismo

Incorporamos ahora el polimorfismo al segundo análisis. La Figura 5.13 muestra los dominios abstractos correspondientes a las variables de tipo y a los tipos polimórficos, así como la interpretación abstracta de una abstrac-

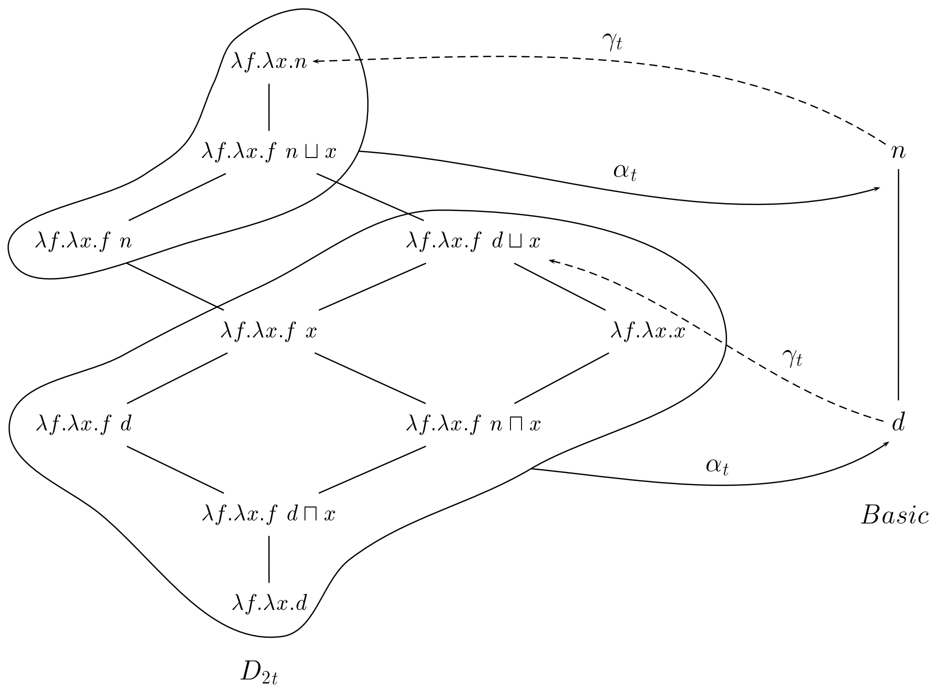


Figura 5.12: Funciones de abstracción y concreción para $t = (Int \rightarrow Int) \rightarrow Int \rightarrow Int$

ción y de una aplicación de tipo.

La interpretación abstracta de una expresión polimórfica es de nuevo la de su ejemplar más pequeño. Por ello, de nuevo el dominio abstracto correspondiente a una variable de tipo β es $Basic$, y el correspondiente a un tipo polimórfico el del tipo sin cualificar.

Cuando se lleva a cabo una aplicación a un tipo t, se debe obtener el valor abstracto del ejemplar apropiado. Dicho valor abstracto se obtiene como una aproximación construida a partir del valor abstracto del ejemplar más pequeño.

De aquí en adelante, el tipo concretado $t'[\beta := t]$ se denotará como $tinst$. La aproximación al valor abstracto del ejemplar se obtiene usando una *función de concreción polimórfica* $\gamma_{t'\,tinst} : D_{2t'} \rightarrow D_{2\,tinst}$. Esta función *adapta* un valor abstracto perteneciente a $D_{2t'}$ para obtener un valor en $D_{2\,tinst}$. Definiremos también otra función $\alpha_{tinst\,t'} : D_{2\,tinst} \rightarrow D_{2t'}$, a la que llamaremos *función de abstracción polimórfica*. Son mutuamente recursivas y se demostrará que forman una inserción de Galois. En la Sección 5.5.6 veremos otras propiedades de estas funciones. La definición de estas funciones corresponde a la aplicación de la consecuencia del Teorema de Representación de Baraki visto en la Sección 2.2.3 (pags. 68-69), tomando como elemento a el valor $\gamma_t(d)$.

Dados dos tipos t, t' y una variable de tipo β, las funciones $\gamma_{t'\,tinst}$ y

$$D_{2\beta} = Basic \qquad D_{2\forall\beta.t} = D_{2t}$$

$$\begin{aligned}
&[\![\Lambda\beta.e]\!]_2\ \rho_2 = [\![e]\!]_2\ \rho_2 \\
&[\![e\ t]\!]_2\ \rho_2 = \gamma_{t'\,tinst}([\![e]\!]_2\ \rho_2)\ \textit{donde}\ e :: \forall\beta.t', \\
&\qquad\qquad\qquad\qquad\qquad\qquad tinst = t'[\beta := t]
\end{aligned}$$

Figura 5.13: Dominios e interpretación abstracta para el polimorfismo

$$\begin{array}{ll}
t' = K, T\ t_1 \ldots t_m & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (id_{Basic}, id_{Basic}) \\
t' = (t_1, \ldots, t_m) & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = \times^m((\gamma_{t_1\,tinst_1}, \alpha_{tinst_1\,t_1}), \ldots, (\gamma_{t_m\,tinst_m}, \alpha_{tinst_m\,t_m})) \\
t' = t_1 \to t_2 & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = \to((\gamma_{t_1\,tinst_1}, \alpha_{tinst_1\,t_1}), (\gamma_{t_2\,tinst_2}, \alpha_{tinst_2\,t_2})) \\
t' = Process\ t_1\ t_2 & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = \to((\gamma_{t_1\,tinst_1}, \alpha_{tinst_1\,t_1}), (\gamma_{t_2\,tinst_2}, \alpha_{tinst_2\,t_2})) \\
t' = \beta & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (\gamma_t, \alpha_t) \\
t' = \beta'\ (\neq \beta) & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (id_{Basic}, id_{Basic}) \\
t' = \forall\beta'.t_1 & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (\gamma_{t_1\,tinst_1}, \alpha_{tinst_1\,t_1})
\end{array}$$

$$\begin{aligned}
\times((f^e, f^c), (g^e, g^c)) &= (f^e \times g^e, f^c \times g^c) \\
\to((f^e, f^c), (g^e, g^c)) &= (\lambda h.g^e \cdot h \cdot f^c, \lambda h'.g^c \cdot h' \cdot f^e)
\end{aligned}$$

Figura 5.14: Definición de las funciones de abstracción y concreción polimórfica

$\alpha_{tinst\,t'}$ se definen formalmente en la Figura 5.14 (donde $tinst_i$ representa $t_i[\beta := t]$). En la Sección 2.2.1 se presentó la categoría de los pares inmersión-clausura. En dicha categoría se definieron dos funtores, $\times$ (y $\times^m$) y $\to$ (ver Figura 5.14 abajo). Ambos construyen un par inmersión-clausura a partir de dos (o más en el caso del producto cartesiano) pares inmersión-clausura. Las funciones $\gamma_{t'\,tinst}$ y $\alpha_{tinst\,t'}$ se definen por medio de estos funtores.

Estas funciones son una generalización de α_t y γ_t; estas operan con valores en $Basic$ y D_{2t}, mientras que aquellas operan con valores en los dominios $D_{2t'}$ y D_{2tinst}, es decir, entre los dominios correspondientes al tipo polimórfico y cada uno de sus ejemplares concretos. Luego, cuando $t' = \beta$, $\alpha_{tinst\,t'}$ y $\gamma_{t'\,tinst}$ coincidirán respectivamente con α_t y γ_t. Por ello mantenemos los mismos nombres, y de aquí en adelante llamaremos función de abstracción indistintamente a α_t y $\alpha_{tinst\,t'}$, y función de concreción a γ_t y $\gamma_{t'\,tinst}$. Cuando sea necesario referirnos explícitamenye a $\alpha_{tinst\,t'}$ y $\gamma_{t'\,tinst}$, usaremos la cualificación "polimórfica".

Como ejemplo, dado el tipo polimórfico $\forall\beta.t'$, donde $t' = (\beta, \beta) \to \beta$, consideremos su aplicación al tipo $t = Int \to Int$. Obtenemos $tinst = t'[t/\beta] = (Int \to Int, Int \to Int) \to Int \to Int$. Para abreviar, denotaremos por E_p a $Basic \times Basic$ y por F_p a $[Basic \to Basic] \times [Basic \to Basic]$. Sea $f \in D_{2t'}$ con $f = \lambda p \in E_p.\pi_1(p)$. Por definición se tiene que $\gamma_{t'\,tinst}(f) = \lambda p \in F_p.\gamma_{t_2\,tinst_2}(f\ (\alpha_{tinst_1\,t_1}(p)))$. En la Figura 5.15 se muestra una tabla para $\gamma_{t'\,tinst}(f)$ y los pasos intermedios para calcularlo.

$p \in F_p$	$\alpha_{tinst_1\,t_1}(p)$	$f\ (\alpha_{tinst_1\,t_1}(p))$	$\gamma_{t_2\,tinst_2}(f\ (\alpha_{tinst_1\,t_1}(p)))$
$(\lambda z \in Basic.d, \lambda z \in Basic.d)$	(d,d)	d	$\lambda u \in Basic.u$
$(\lambda z \in Basic.d, \lambda z \in Basic.z)$	(d,d)	d	$\lambda u \in Basic.u$
$(\lambda z \in Basic.d, \lambda z \in Basic.n)$	(d,n)	d	$\lambda u \in Basic.u$
$(\lambda z \in Basic.z, \lambda z \in Basic.d)$	(d,d)	d	$\lambda u \in Basic.u$
$(\lambda z \in Basic.z, \lambda z \in Basic.z)$	(d,d)	d	$\lambda u \in Basic.u$
$(\lambda z \in Basic.z, \lambda z \in Basic.n)$	(d,n)	d	$\lambda u \in Basic.u$
$(\lambda z \in Basic.n, \lambda z \in Basic.d)$	(n,d)	n	$\lambda u \in Basic.n$
$(\lambda z \in Basic.n, \lambda z \in Basic.z)$	(n,d)	n	$\lambda u \in Basic.n$
$(\lambda z \in Basic.n, \lambda z \in Basic.n)$	(n,n)	n	$\lambda u \in Basic.n$

Figura 5.15: Un ejemplo de polimorfismo

5.5.5 Propiedades de las funciones de abstracción y concreción

En este segundo análisis, hemos utilizado las funciones de abstracción, α_t, y concreción, γ_t, definidas en las Figuras 5.10 y 5.11, las primeras en las aplicaciones de constructores y las segundas en las expresiones **case**.

En esta sección vamos a demostrar que forman una inserción de Galois [NNH99], o equivalentemente una sobreyección de Galois [CC92b], o un par inmersión-clausura [Bar93], donde γ_t es la inmersión y α_t la clausura (ver Proposición 25) y también que $\gamma_{t_1 \to t_2}(d)$ refleja fielmente la noción de función determinista (ver Proposición 27).

En la proposición siguiente se demuestra que α_t y γ_t son monótonas y continuas, y que forman una inserción de Galois. Esto significa que α_t pierde información pero toda de una vez, y que γ_t recupera lo máximo posible de ella, de forma que una nueva aplicación de α_t no pierde más información. Este hecho garantiza que en la interpretación de las expresiones **case**, la información recuperada a partir del discriminante no es solamente segura sino también la mejor que se puede obtener teniendo en cuenta cómo se llevó a cabo la abstracción (proporcionar un valor abstracto no determinista a las variables de las alternativas también es seguro). Además, el punto 25(d) asegura que el desaplanamiento de n es el supremo del dominio correspondiente. En el ejemplo de la Figura 5.12 se reflejan estas ideas. Todos los valores por debajo de $\lambda f.\lambda x.f\ d \sqcup x$ se abstraen a d y el resto a n. Luego la función de concreción devuelve $\lambda f.\lambda x.f\ d \sqcup x$ como la menor (es decir, la más precisa) aproximación segura de d en D_{2t}. Los valores $\lambda f.\lambda x.f\ n \sqcup x$ y $\lambda f.\lambda x.n$ también son aproximaciones seguras, pero menos precisas. La concreción de n es el supremo del dominio $\lambda f.\lambda x.n$.

Proposición 25 *Para cada tipo t:*

(a) *Las funciones α_t y γ_t son monótonas y continuas,*

(b) $\alpha_t \cdot \gamma_t = id_{Basic}$,

(c) $\gamma_t \cdot \alpha_t \sqsupseteq id_{D_{2t}}$,

(d) $\forall e \in D_{2t}.e \sqsubseteq \gamma_t(n)$.

Demostración 1 (Proposición 25) Los puntos 25(a) y 25(d) se demuestran por inducción estructural (i.e.) sobre t, y son necesarios para demostrar, también por i.e., 25(b) y 25(c).

Demostramos primero el punto 25(a). Si demostramos que α_t y γ_t son monótonas, entonces habremos probado que son continuas, ya que sus dominios son finitos (y por tanto satisfacen la Condición de Cadena Ascendente [NNH99]). Dicha monotonía se puede demostrar por inducción estructural sobre t.

Veamos primero que α_t es monótona:

- $t = K$ o $t = T\ t_1 \ldots t_m$ o $t = \beta$. Estos casos son triviales ya que $\alpha_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Por hipótesis de inducción (h.i.), para cada $i = 1, \ldots, m$, α_{t_i}es monótona. Sean $e_i, e'_i \in D_{2t_i}$. Supongamos que $(e_1, \ldots, e_m) \sqsubseteq (e'_1, \ldots, e'_m)$, es decir, para cada i, $e_i \sqsubseteq e'_i$. Entonces,

$$\begin{array}{lcll} \alpha_t(e_1, \ldots, e_m) & = & \bigsqcup\limits_i \alpha_{t_i}(e_i) & \{\text{por definición de } \alpha_t\} \\ & \sqsubseteq & \bigsqcup\limits_i \alpha_{t_i}(e'_i) & \{\text{por h.i. y } e_i \sqsubseteq e'_i\} \\ & = & \alpha_t(e'_1, \ldots, e'_m) & \{\text{por definición de } \alpha_t\}. \end{array}$$

- $t = t_1 \to t_2$ o $t = \mathit{Process}\ t_1\ t_2$. Por h.i. cada α_{t_i} es monótona, para $i = 1, 2$. Sean $f, f' \in [D_{2t_1} \to D_{2t_2}]$. Supongamos que $f \sqsubseteq f'$, es decir, para cada $e \in D_{2t_1}$, $f(e) \sqsubseteq f'(e)$. Entonces,

$$\begin{array}{lcll} \alpha_t(f) & = & \alpha_{t_2}(f(\gamma_{t_1}(d))) & \{\text{por definición de } \alpha_t\} \\ & \sqsubseteq & \alpha_{t_2}(f'(\gamma_{t_1}(d))) & \{\text{por h.i. y } f \sqsubseteq f'\} \\ & = & \alpha_t(f') & \{\text{por definición de } \alpha_t\}. \end{array}$$

- $t = \forall\beta.t'$. Como $\alpha_t = \alpha_{t'}$, se cumple trivialmente por h.i.

Veamos ahora que γ_t es monótona:

- $t = K$ o $t = T\ t_1 \ldots t_m$ o $t = \beta$. Estos casos son triviales, ya que $\gamma_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Por h.i. para cada $i = 1, \ldots, m$, γ_{t_i} es monótona. Sean $b, b' \in \mathit{Basic}$ con $b \sqsubseteq b'$. Entonces,

$$\begin{array}{lcll} \gamma_t(b) & = & (\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) & \{\text{por definición de } \gamma_t\} \\ & \sqsubseteq & (\gamma_{t_1}(b'), \ldots, \gamma_{t_m}(b')) & \{\text{por h.i. y } b \sqsubseteq b'\} \\ & = & \gamma_t(b') & \{\text{por definición de } \gamma_t\}. \end{array}$$

- $t = t_1 \to t_2$ o $t = Process\ t_1\ t_2$. Por h.i. cada γ_{t_i} es monótona, para $i = 1, 2$. Sean $b, b' \in Basic$. Supongamos que $b \sqsubseteq b'$ y demostremos que $\gamma_t(b) \sqsubseteq \gamma_t(b')$.

  El caso $b = b'$ es trivial. El único caso no trivial se da cuando $b = d$ y $b' = n$. En tal caso $\gamma_t(b) = \lambda z.\gamma_{t_2}(\alpha_{t_1}(z))$ y $\gamma_t(b') = \lambda z.\gamma_{t_2}(n)$. Como n es el supremo del dominio $Basic$, se cumple que $\alpha_{t_1}(e) \sqsubseteq n$ para cada $e \in D_{2t_1}$. Luego por h.i. : $\forall e \in D_{2t_1}.\ \gamma_{t_2}(\alpha_{t_1}(e)) \sqsubseteq \gamma_{t_2}(n)$, de donde obtenemos que $\gamma_t(d) \sqsubseteq \gamma_t(n)$.

- $t = \forall\beta.t'$. Como $\gamma_t = \gamma_{t'}$, se cumple trivialmente por h.i.

Demostramos ahora el punto 25(d) ,también por inducción estructural sobre t:

- $t = K$ o $t = T\ t_1 \ldots t_m$ o $t = \beta$. Estos casos son triviales ya que $\gamma_t = id_{Basic}$, y n es el supremo de $Basic$.

- $t = (t_1, \ldots, t_m)$. Si $e \in D_{2t}$, entonces $e = (e_1, \ldots, e_m)$, con $e_i \in D_{2t_i}$ para $i = 1, \ldots, m$. Por h.i. para cada $i = 1, \ldots, m$ se cumple que $e_i \sqsubseteq \gamma_{t_i}(n)$, luego $e = (e_1, \ldots, e_m) \sqsubseteq (\gamma_{t_1}(n), \ldots, \gamma_{t_m}(n)) = \gamma_t(n)$.

- $t = t_1 \to t_2$ o $t = Process\ t_1\ t_2$. Sean $f \in D_{2t}$, y $e \in D_{2t_1}$. Entonces $f(e) \in D_{2t_2}$, y por h.i. $f(e) \sqsubseteq \gamma_{t_2}(n)$. Por otro lado $\gamma_t(n) = \lambda z.\gamma_{t_2}(n)$. Es decir, para cada $e \in D_{2t_1}$, $f(e) \sqsubseteq (\gamma_t(n))(e)$, luego $f \sqsubseteq \gamma_t(n)$.

- $t = \forall\beta.t'$. Como $\gamma_t = \gamma'_t$, se cumple por h.i.

Demostremos ahora el punto 25(b).

- $t = K$ o $t = T\ t_1 \ldots t_m$ o $t = \beta$. Estos casos son triviales, ya que $\alpha_t = \gamma_t = id_{Basic}$.

- $t = (t_1, \ldots, t_m)$. Sea $b \in Basic$. Entonces:

$$\begin{array}{lcll} \alpha_t(\gamma_t(b)) & = & \alpha_t(\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) & \{\text{por definición de } \gamma_t\} \\ & = & \bigsqcup\limits_i \alpha_{t_i}(\gamma_{t_i}(b)) & \{\text{por definición de } \alpha_t\} \\ & = & \bigsqcup\limits_i b & \{\text{por h.i.}\} \\ & = & b. & \end{array}$$

- $t = t_1 \to t_2$ o $t = Process\ t_1\ t_2$. Sea $b \in Basic$. Distinguimos dos casos: $b = d$ y $b = n$.

  Si $b = d$, entonces:

$$\begin{array}{lcll} \alpha_t(\gamma_t(d)) & = & \alpha_t(\lambda z.\gamma_{t_2}(\alpha_{t_1}(z))) & \{\text{por definición de } \gamma_t\} \\ & = & \alpha_{t_2}(\gamma_{t_2}(\alpha_{t_1}(\gamma_{t_1}(d)))) & \{\text{por definición de } \alpha_t\} \\ & = & d & \{\text{por h.i.}\}. \end{array}$$

Si $b = n$, entonces:

$$\begin{array}{rcll} \alpha_t(\gamma_t(n)) & = & \alpha_t(\lambda z.\gamma_{t_2}(n)) & \{\text{por definición de } \gamma_t\} \\ & = & \alpha_{t_2}(\gamma_{t_2}(n)) & \{\text{por definición de } \alpha_t\} \\ & = & n & \{\text{por h.i.}\}. \end{array}$$

- $t = \forall\beta.t'$. Se cumple directamente por h.i. pues $\alpha_t = \alpha_{t'}$ y $\gamma_t = \gamma_{t'}$.

  Falta por demostrar el punto 25(c).

- $t = K$ o $t = T\ t_1 \ldots t_m$ o $t = \beta$. Estos casos son triviales, pues $\alpha_t = \gamma_t = id_{Basic}$.

- $t = (t_1, \ldots, t_m)$. Sean $e_i \in D_{2t_i}$ con $i = 1, \ldots, m$. Entonces:

$$\begin{array}{ll} \gamma_t(\alpha_t(e_1, \ldots, e_m)) & \\ \quad = \gamma_t(\bigsqcup\limits_i \alpha_{t_i}(e_i)) & \{\text{por definición de } \alpha_t\} \\ \quad = (\gamma_{t_1}(\bigsqcup\limits_i \alpha_{t_i}(e_i)), \ldots, \gamma_{t_m}(\bigsqcup\limits_i \alpha_{t_i}(e_i))) & \{\text{por definición de } \gamma_t\} \\ \quad \sqsupseteq (\gamma_{t_1}(\alpha_{t_1}(e_1)), \ldots, \gamma_{t_m}(\alpha_{t_m}(e_m))) & \{\gamma_t \text{ monótona}\} \\ \quad \sqsupseteq (e_1, \ldots, e_m) & \{\text{por h.i.}\}. \end{array}$$

- $t = t_1 \to t_2$ o $t = \mathit{Process}\ t_1\ t_2$. Sea $f \in [D_{2t_1} \to D_{2t_2}]$. Entonces:

$$\gamma_t(\alpha_t(f)) \quad = \quad \gamma_t(\alpha_{t_2}(f(\gamma_{t_1}(d)))) \quad \{\text{por definición de } \alpha_t\}.$$

  Distinguimos dos casos:

  - Si $\alpha_{t_2}(f(\gamma_{t_1}(d))) = d$ entonces, por definición de γ_t, se cumple que $\gamma_t(\alpha_t(f)) = \lambda z.\gamma_{t_2}(\alpha_{t_1}(z))$. Sea $e \in D_{2t_1}$, queremos demostrar que $f(e) \sqsubseteq (\lambda z.\gamma_{t_2}(\alpha_{t_1}(z)))(e)$. Es decir, $f(e) \sqsubseteq \gamma_{t_2}(\alpha_{t_1}(e))$. Distinguimos de nuevo dos casos:

    * $e \sqsubseteq \gamma_{t_1}(d)$. Por monotonía de f y α_t, $\alpha_{t_2}(f(e)) \sqsubseteq \alpha_{t_2}(f(\gamma_{t_1}(d)))$ (que es igual a d), es decir,

    $$\alpha_{t_2}(f(e)) \sqsubseteq d \sqsubseteq \alpha_{t_1}(e),$$

      donde la última desigualdad se cumple porque d es el ínfimo en *Basic*. Luego por monotonía de γ_t, $\gamma_{t_2}(\alpha_{t_2}(f(e))) \sqsubseteq \gamma_{t_2}(\alpha_{t_1}(e))$, y por h.i. tenemos que

    $$f(e) \sqsubseteq \gamma_{t_2}(\alpha_{t_2}(f(e))) \sqsubseteq \gamma_{t_2}(\alpha_{t_1}(e)).$$

    * $e \not\sqsubseteq \gamma_{t_1}(d)$. Veamos que entonces $\alpha_{t_1}(e) = n$. Si no fuera así tendríamos $\alpha_{t_1}(e) = d$. Entonces $\gamma_{t_1}(\alpha_{t_1}(e)) = \gamma_{t_1}(d)$. Por h.i. sabemos que $e \sqsubseteq \gamma_{t_1}(\alpha_{t_1}(e))$, luego $e \sqsubseteq \gamma_{t_1}(d)$, lo que contradice nuestra premisa.

      En consecuencia lo que tenemos que demostrar es que $f(e) \sqsubseteq \gamma_{t_2}(n)$, lo cual se cumple por la Proposición 25(d), ya que $f(e) \in D_{2t_2}$.

– Si $\alpha_{t_2}(f(\gamma_{t_1}(d))) = n$, entonces $\gamma_t(\alpha_t(f)) = \lambda z.\gamma_{t_2}(n)$, luego hemos de probar que dado $e \in D_{2t_1}$, $f(e) \sqsubseteq \gamma_{t_2}(n)$, lo que en efecto se cumple por la Proposición 25(d), pues $f(e) \in D_{2t_2}$.

- $t = \forall\beta.t'$. Se cumple directamente por h.i. pues $\alpha_t = \alpha_{t'}$, $\gamma_t = \gamma_{t'}$ y $D_{2t} = D_{2t'}$.

□

Como consecuencia directa de esta proposición, obtenemos la siguiente (estándar), que describe la relación entre *Basic* y D_{2t} para cada t. Afirma que todos los valores por debajo de $\gamma_t(d)$ se aplanan a d. Es decir, $\gamma_t(d)$ representa a d. También dice que cualquier otro valor, mayor que $\gamma_t(d)$ o incomparable con él, se aplana a n. Esto confirma que $\gamma_t(d)$ es el mayor valor en D_{2t} que representa a d.

Proposición 26 *Para cada tipo t:* $\forall e \in D_{2t}.e \sqsubseteq \gamma_t(d) \Leftrightarrow \alpha_t(e) = d$.

Demostración 2 (Proposición 26) Se puede demostrar usando las proposiciones 25(a), 25(b) y 25(c).

(⇒) Si $e \sqsubseteq \gamma_t(d)$ entonces, como α_t es monótona $\alpha_t(e) \sqsubseteq \alpha_t(\gamma_t(d))$. Por la Proposición 25(b), $\alpha_t(\gamma_t(d)) = d$, y como d es el ínfimo de *Basic*, tenemos que $\alpha_t(e) = d$.

(⇐) Si $\alpha_t(e) = d$, entonces por la Proposición 25(c), $e \sqsubseteq \gamma_t(\alpha_t(e)) = \gamma_t(d)$.

□

Una función/proceso se considera determinista cuando aplicada a argumentos deterministas su resultado es también determinista. Esto significa que, cuando una función (resp. un proceso) de tipo $t_1 \to t_2$ (resp. *Process* t_1 t_2) se aplica a un valor menor o igual que $\gamma_{t_1}(d)$, el resultado es menor o igual que $\gamma_{t_2}(d)$. La siguiente proposición nos dice que una función de tipo $t_1 \to t_2$ es determinista si y sólo si es menor o igual que $\gamma_{t_1 \to t_2}(d)$, es decir, $\gamma_{t_1 \to t_2}(d)$ es el mayor representante de las funciones deterministas de ese tipo. Lo mismo sucede para los procesos.

Proposición 27 *Dada una función $f \in D_{2t}$, donde $t = t_1 \to t_2$ o $t =$ Process t_1 t_2:*

$$f \sqsubseteq \gamma_t(d) \Leftrightarrow \forall e \sqsubseteq \gamma_{t_1}(d).f(e) \sqsubseteq \gamma_{t_2}(d).$$

Demostración 3 (Proposición 27) Se puede demostrar usando las proposiciones 25(d) y 26.

($\Rightarrow$) Sea $f \sqsubseteq \gamma_t(d)$ y $e \sqsubseteq \gamma_{t_1}(d)$. Por la Proposición 26 se cumple que $\alpha_{t_1}(e) = d$. Por otro lado $\gamma_t(d) = \lambda z.\gamma_{t_2}(\alpha_{t_1}(z))$ por definición de γ_t.

Como $f \sqsubseteq \gamma_t(d)$ entonces:

$$f(e) \sqsubseteq \gamma_{t_2}(\alpha_{t_1}(e)) = \gamma_{t_2}(d).$$

($\Leftarrow$) Supongamos que para todo $e \sqsubseteq \gamma_{t_1}(d)$ se cumple que $f(e) \sqsubseteq \gamma_{t_2}(d)$. Hemos de probar que $f \sqsubseteq \gamma_t(d)$. Consideremos $e \in D_{2t_1}$. Distinguimos dos casos:

- $e \sqsubseteq \gamma_{t_1}(d)$. En este caso, por la Proposición 26 tenemos que $\alpha_{t_1}(e) = d$ y:

$$\begin{array}{rcll} f(e) & \sqsubseteq & \gamma_{t_2}(d) & \{\text{por hipótesis}\} \\ & = & \gamma_{t_2}(\alpha_{t_1}(e)) & \{\text{por la Proposición 26}\} \\ & = & (\gamma_t(d))(e) & \{\text{por definición de } \gamma_t\}. \end{array}$$

- En caso contrario, por la Proposición 26 obtenemos que $\alpha_{t_1}(e) = n$. Luego:

$$\begin{array}{rcll} f(e) & \sqsubseteq & \gamma_{t_2}(n) & \{\text{por la Proposición 25(d)}\} \\ & = & \gamma_{t_2}(\alpha_{t_1}(e)) & \{\text{por la Proposición 26}\} \\ & = & (\gamma_t(d))(e) & \{\text{por definición de } \gamma_t\}. \end{array}$$

$\square$

5.5.6 Propiedades del polimorfismo

En esta sección se demuestran algunas propiedades de las versiones polimórficas de las funciones de abstracción y concreción. Tales funciones se definieron en la Figura 5.14 usando los funtores $\times$ y $\rightarrow$ definidos en la categoría de pares inmersión-clausura. Luego la siguiente proposición es trivialmente cierta por la Proposición 25.

Proposición 28 *Dados dos tipos t, t' y una variable de tipo β:*

(a) $\gamma_{t'\,tinst}$ *y* $\alpha_{tinst\,t'}$ *son monótonas y continuas,*

(b) $\alpha_{tinst\,t'} \cdot \gamma_{t'\,tinst} = id_{D_{2t'}}$,

(c) $\gamma_{t'\,tinst} \cdot \alpha_{tinst\,t'} \sqsupseteq id_{D_{2tinst}}$.

En la siguiente proposición se estudia la relación entre las funciones de abstracción y concreción y sus homólogas polimórficas. Tienen algunas propiedades de conmutatividad, mostradas en la Figura 5.16.

Proposición 29 *Dados dos tipos t, t' y una variable de tipo β:*

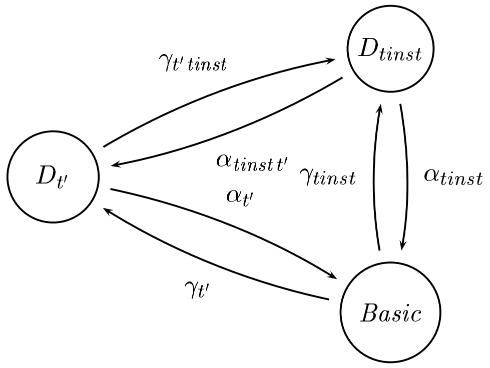


Figura 5.16: Propiedades de conmutatividad de las funciones de abstracción y concreción

(a) $\gamma_{t'\,tinst} \cdot \gamma_{t'} = \gamma_{tinst}$,

(b) $\alpha_{t'} \cdot \alpha_{tinst\,t'} = \alpha_{tinst}$,

(c) $\alpha_{tinst} \cdot \gamma_{t'\,tinst} = \alpha_{t'}$,

(d) $\alpha_{tinst\,t'} \cdot \gamma_{tinst} = \gamma_{t'}$.

Demostración 4 (Proposición 29) Los puntos 29(a) y 29(b) se pueden demostrar por i.e. sobre t'. Los puntos 29(c) y 29(d) se pueden demostrar a partir de los anteriores y de la Proposición 28(b).
Demostramos simultáneamente los puntos 29(a) y 29(b) por inducción estructural sobre t':

- $t' = K$ o $t' = T\ t_1 \dots t_m$ o $t' = \beta'$ $(\neq \beta)$. En estos casos $\gamma_{t'\,tinst} = \gamma_{t'} = \gamma_{tinst} = id_{Basic} = \alpha_{tinst\,t'} = \alpha_{t'} = \alpha_{tinst}$.

- $t' = (t_1, \dots, t_m)$. En este caso $tinst = (tinst_1, \dots, tinst_m)$ donde $tinst_i = t_i[\beta := t]$, para cada $i = 1, \dots, m$. Por h.i. se cumple para cada $i = 1, \dots, m$ que

$$\gamma_{t_i\,tinst_i} \cdot \gamma_{t_i} = \gamma_{tinst_i},$$

y

$$\alpha_{t_i} \cdot \alpha_{tinst_i\,t_i} = \alpha_{tinst_i}.$$

Demostremos primero que $\gamma_{t'\,tinst} \cdot \gamma_{t'} = \gamma_{tinst}$. Sea $b \in Basic$.

$$\begin{array}{ll}
\gamma_{t'\,tinst}(\gamma_{t'}(b)) & \\
= \gamma_{t'\,tinst}(\gamma_{t_1}(b), \dots, \gamma_{t_m}(b)) & \{\text{por definición de } \gamma_t\} \\
= (\gamma_{t_1\,tinst_1}(\gamma_{t_1}(b)), \dots, \gamma_{t_m\,tinst_m}(\gamma_{t_m}(b))) & \{\text{por definición de } \gamma_{t'\,tinst}\} \\
= (\gamma_{tinst_1}(b), \dots, \gamma_{tinst_m}(b)) & \{\text{por h.i.}\} \\
= \gamma_{tinst}(b) & \{\text{por definición de } \gamma_t\}.
\end{array}$$

Demostremos ahora que $\alpha_{t'} \cdot \alpha_{tinst\,t'} = \alpha_{tinst}$. Sea $e_i \in D_{2\,tinst_i}$, para cada $i = 1, \ldots, m$.

$$\begin{array}{ll}
\alpha_{t'}(\alpha_{tinst\,t'}(e_1, \ldots, e_m)) & \\
= \alpha_{t'}(\alpha_{tinst_1\,t_1}(e_1), \ldots, \alpha_{tinst_m\,t_m}(e_m)) & \{\text{por definición de } \alpha_{tinst\,t'}\} \\
= \bigsqcup_i \alpha_{t_i}(\alpha_{tinst_i\,t_i}(e_i)) & \{\text{por definición de } \alpha_t\} \\
= \bigsqcup_i \alpha_{tinst_i}(e_i) & \{\text{por h.i.}\} \\
= \alpha_{tinst}(e_1, \ldots, e_m) & \{\text{por definición de } \alpha_t\}.
\end{array}$$

- $t' = t_1 \to t_2$ or $t' = Process\ t_1\ t_2$. En este caso $tinst = tinst_1 \to tinst_2$ (respectivamente $tinst = Process\ tinst_1\ tinst_2$) donde $tinst_i = t_i[\beta := t]$. Por h.i. tenemos que para cada $i = 1, 2$,

$$\gamma_{t_i\,tinst_i} \cdot \gamma_{t_i} = \gamma_{tinst_i},$$

y

$$\alpha_{t_i} \cdot \alpha_{tinst_i\,t_i} = \alpha_{tinst_i}.$$

Demostremos primero que $\gamma_{t'\,tinst} \cdot \gamma_{t'} = \gamma_{tinst}$. Sea $b \in Basic$. Distinguimos dos casos: $b = d$ y $b = n$.

Si $b = d$:

$$\begin{array}{ll}
\gamma_{t'\,tinst}(\gamma_{t'}(d)) & \\
= \gamma_{t'\,tinst}(\lambda z.\gamma_{t_2}(\alpha_{t_1}(z))) & \{\text{por definición de } \gamma_t\} \\
= \lambda z.\gamma_{t_2\,tinst_2}(\gamma_{t_2}(\alpha_{t_1}(\alpha_{tinst_1\,t_1}(z)))) & \{\text{por definición de } \gamma_{t'\,tinst}\} \\
= \lambda z.\gamma_{tinst_2}(\alpha_{tinst_1}(z)) & \{\text{por h.i.}\} \\
= \gamma_{tinst}(d) & \{\text{por definición de } \gamma_t\}.
\end{array}$$

Si $b = n$ entonces:

$$\begin{array}{ll}
\gamma_{t'\,tinst}(\gamma_{t'}(n)) & \\
= \gamma_{t'\,tinst}(\lambda z.\gamma_{t_2}(n)) & \{\text{por definición de } \gamma_t\} \\
= \lambda z.\gamma_{t_2\,tinst_2}(\gamma_{t_2}(n)) & \{\text{por definición de } \gamma_{t'\,tinst}\} \\
= \lambda z.\gamma_{tinst_2}(n) & \{\text{por h.i.}\} \\
= \gamma_{tinst}(n) & \{\text{por definición de } \gamma_t\}.
\end{array}$$

Demostremos ahora que $\alpha_{t'} \cdot \alpha_{tinst\,t'} = \alpha_{tinst}$. Sea $f \in D_{2\,tinst}$:

$$\begin{array}{ll}
\alpha_{t'}(\alpha_{tinst\,t'}(f)) & \\
= \alpha_{t'}(\lambda z.\alpha_{tinst_2\,t_2}(f(\gamma_{t_1\,tinst_1}(z)))) & \{\text{por definición de } \alpha_{tinst\,t'}\} \\
= \alpha_{t_2}(\alpha_{tinst_2\,t_2}(f(\gamma_{t_1\,tinst_1}(\gamma_{t_1}(d))))) & \{\text{por definición de } \alpha_t\} \\
= \alpha_{tinst_2}(f(\gamma_{tinst_1}(d))) & \{\text{por h.i.}\} \\
= \alpha_{tinst}(f) & \{\text{por definición de } \alpha_t\}.
\end{array}$$

- $t' = \beta$. En este caso $tinst = t$, y por definición tenemos que $\gamma_{t'\,tinst} = \gamma_t$, $\alpha_{tinst\,t'} = \alpha_t$, $\gamma_{t'} = \alpha_{t'} = id_{Basic}$, luego

$$\begin{array}{ll} \gamma_{t'\,tinst} \cdot \gamma_{t'} & \\ \quad = \gamma_t \cdot id_{Basic} & \{\text{por definición de } \gamma_{t'\,tinst} \text{ y } \gamma_t\} \\ \quad = \gamma_{tinst} & \{\text{pues } tinst = t\}, \end{array}$$

y

$$\begin{array}{ll} \alpha_{t'} \cdot \alpha_{tinst\,t'} & \\ \quad = id_{Basic} \cdot \alpha_t & \{\text{por definición de } \alpha_{tinst\,t'} \text{ y } \alpha_t\} \\ \quad = \alpha_{tinst} & \{\text{pues } tinst = t\}. \end{array}$$

- $t' = \forall\beta'.t_1$. En este caso $tinst = tinst_1 = t_1[\beta := t]$ y entonces

$$\begin{array}{ll} \gamma_{t'\,tinst} \cdot \gamma_{t'} & \\ \quad = \gamma_{t_1\,tinst_1} \cdot \gamma_{t_1} & \{\text{por definición de } \gamma_{t'\,tinst} \text{ y } \gamma_t\} \\ \quad = \gamma_{tinst_1} & \{\text{por h.i.}\} \\ \quad = \gamma_{tinst} & \{\text{por definición de } \gamma_t\}, \end{array}$$

y

$$\begin{array}{ll} \alpha_{t'} \cdot \alpha_{tinst\,t'} & \\ \quad = \alpha_{t_1} \cdot \alpha_{tinst_1\,t_1} & \{\text{por definición de } \alpha_{tinst\,t'} \text{ y } \alpha_t\} \\ \quad = \alpha_{tinst_1} & \{\text{por h.i.}\} \\ \quad = \alpha_{tinst} & \{\text{por definición de } \alpha_t\}. \end{array}$$

Los puntos 29(c) y 29(d) se demuestran fácilmente usando los anteriores:

$$\begin{array}{ll} \alpha_{tinst} \cdot \gamma_{t'\,tinst} & \\ \quad = (\alpha_{t'} \cdot \alpha_{tinst\,t'}) \cdot \gamma_{t'\,tinst} & \{\text{por la Proposición 29(b)}\} \\ \quad = \alpha_{t'} & \{\text{por asociatividad y Proposición 28(b)}\} \end{array}$$

$$\begin{array}{ll} \alpha_{tinst\,t'} \cdot \gamma_{tinst} & \\ \quad = \alpha_{tinst\,t'} \cdot (\gamma_{t'\,tinst} \cdot \gamma_{t'}) & \{\text{por la Proposición 29(a)}\} \\ \quad = \gamma_{t'} & \{\text{por asociatividad y Proposición 28(b)}\} \end{array}$$

□

Ya hemos visto que la semántica de una aplicación de tipo se obtiene usando la función $\gamma_{t'\,tinst}$. Pero también se podría haber definido la semántica de un ejemplar de otra manera, primero aplicando la función de abstracción al valor abstracto del ejemplar más pequeño, para obtener un valor en $Basic$, y aplicando después la función de concreción para obtener un valor en $D_{2\,tinst}$. En suma, tendríamos

$$[\![e\ t]\!]_2\ \rho_2 = \gamma_{tinst}(\alpha_{t'}([\![e]\!]_2\ \rho_2)).$$

El primer punto de la siguiente proposición nos dice que ésta hubiera sido una elección peor; es decir, que con ella perderíamos más información. El segundo punto completa las propiedades de conmutatividad con algún interés mostradas en la Figura 5.16.

Proposición 30 *Dados dos tipos t, t' y una variable de tipo β:*

(a) $\gamma_{tinst} \cdot \alpha_{t'} \sqsupseteq \gamma_{t'\,tinst}$,

(b) $\gamma_{t'} \cdot \alpha_{tinst} \sqsupseteq \alpha_{tinst\,t'}$.

Demostración 5 (Proposición 30) El primer punto se puede demostrar a partir de las proposiciones 29(a) y 25(c), y el segundo a partir de las proposiciones 29(b) y 25(c):

$$\begin{array}{ll} \gamma_{tinst} \cdot \alpha_{t'} & \\ \quad = (\gamma_{t'\,tinst} \cdot \gamma_{t'}) \cdot \alpha_{t'} & \{\text{por la Proposición 29(a)}\} \\ \quad \sqsupseteq \gamma_{t'\,tinst} & \{\text{por asociatividad y Proposición 25(c)}\}. \end{array}$$

$$\begin{array}{ll} \gamma_{t'} \cdot \alpha_{tinst} & \\ \quad = \gamma_{t'} \cdot (\alpha_{t'} \cdot \alpha_{tinst\,t'}) & \{\text{por la Proposición 29(b)}\} \\ \quad \sqsupseteq \alpha_{tinst\,t'} & \{\text{por asociatividad y Proposición 25(c)}\}. \end{array}$$

□

Se puede definir una generalización del par $(\gamma_{t'\,tinst}, \alpha_{tinst\,t'})$, donde se concreten en secuencia varias variables de tipo. Esto corresponde a un tipo $\forall\beta_1.\ldots.\forall\beta_m.t'$ y a un ejemplar $tinst = t'[\beta_1 := t_1, \ldots, \beta_m := t_m]$. Los dominios del análisis junto con dichos pares como morfismos forman una categoría (por las proposiciones 29(a) y 29(b)).

5.6 Un análisis intermedio

5.6.1 Introducción

El elevado coste de $[\![\cdot]\!]_2$ es debido al cálculo del punto fijo. En cada iteración se lleva a cabo una comparación entre valores abstractos, la cual resulta ser exponencial en el peor de los casos, cuando hay dominios funcionales implicados. Luego una buena forma de acelerar el cálculo del punto fijo consiste en encontrar una representación rápidamente comparable de las funciones. Se han desarrollado diferentes técnicas en esta dirección, como los ya mencionados algoritmos de fronteras [CP85, PC87, MH87] y los operadores de ensanchamiento y estrechamiento (*widening* y *narrowing*) [CC77, CC79, HH92, PP93]. En nuestro caso representaremos las funciones mediante **signaturas**, de forma similar a la utilizada en [PP93]. Para cada función obtendremos una signatura **muestreando** dicha función con algunas combinaciones explícitamente elegidas de argumentos. Por ejemplo, en el análisis de estrictez de [PP93], una función f con m argumentos era muestreada con m combinaciones de argumentos: aquellas en las que $\perp$ ocupa cada posición de argumento y el resto de ellos recibe un valor $\top$:

$\perp, \top, \ldots, \top$; $\top, \perp, \top, \ldots, \top$; $\ldots$; $\top, \top, \ldots, \perp$. Así, por ejemplo, la función $f = \lambda x :: Int.\lambda y :: Int.y$ tiene una signatura $\top\ \perp$.

Si muestreamos solamente con algunas combinaciones de argumentos, diferentes funciones pueden tener la misma signatura y consecuentemente se pierde información. Esto implica que el cálculo del punto fijo no es exacto sino aproximado. El número de funciones que tienen la misma signatura depende del número de combinaciones de argumentos con los que muestreemos la función. Si se muestrea con muchas combinaciones de argumentos, la signatura contendrá más información, con lo que el número de funciones que tienen la misma signatura será menor. Pero entonces las comparaciones de funciones, y por tanto el análisis, serán más costosos. Si muestreamos la función con todas las combinaciones de argumentos, tendríamos una signatura por función, pero esto sería equivalente a una representación tabulada de la función, lo que no ayudaría en absoluto a reducir el coste del análisis.

Por tanto, debe encontrarse un compromiso entre la cantidad de información que contiene la signatura y el coste de compararla. Aquí nos concentramos en una de las posibilidades, que hemos implementado, y mencionaremos solamente algunas otras. Muestreamos una función de m argumentos con $m+1$ combinaciones de argumentos. En las primeras m combinaciones, cada posición de argumento es ocupada por un valor abstracto no determinista (del tipo correspondiente) $\gamma_{t_i}(n)$, mientras que el resto de ellos recibe un valor determinista: $\gamma_{t_1}(n), \gamma_{t_2}(d), \ldots, \gamma_{t_m}(d)$; $\gamma_{t_1}(d), \gamma_{t_2}(n), \ldots, \gamma_{t_m}(d)$; $\ldots$; $\gamma_{t_1}(d), \gamma_{t_2}(d), \ldots, \gamma_{t_m}(n)$. En la $(m+1)$-ésima combinación, todos los argumentos reciben un valor determinista: $\gamma_{t_1}(d), \gamma_{t_2}(d), \ldots, \gamma_{t_m}(d)$. Incluir esta combinación adicional es importante pues deseamos que el análisis sea más potente que $[\![\cdot]\!]_1$, donde las funciones eran únicamente muestreadas con dicha combinación.

5.6.2 El dominio de las signaturas

En la Figura 5.17 se definen formalmente los dominios de las signaturas S_t. El dominio correspondiente a un tipo básico o algebraico es un dominio de dos elementos, isomorfo a *Basic*. Para distinguirlos, usaremos letras mayúsculas D y N para las signaturas. El dominio correspondiente al tipo tupla es el producto cartesiano de los tipos componente, por ejemplo $(D, N) \in S_{(Int,Int)}$. El orden entre tuplas es el usual, componente a componente.

El dominio correspondiente a un tipo polimórfico es de nuevo el de su ejemplar más pequeño.

Con respecto a las funciones, es necesaria una discusión previa. Diremos que un tipo t es funcional si $t = t_1 \to t_2$, $t = Process\ t_1\ t_2$ o $t = \forall \beta.t'$, donde t' es funcional. Si un tipo t es funcional, escribiremos $fun(t)$; en caso contrario, escribiremos $nonfun(t)$.

Si una función tiene m argumentos, entonces su signatura está compuesta

$$
\begin{array}{l}
S_K = S_{T\ t_1..t_m} = S_\beta = \{D, N\} \text{where } D \preceq N \\
S_{(t_1,\ldots,t_m)} = S_{t_1} \times \ldots \times S_{t_m} \\
S_{\forall\beta.t} = S_t \\
S_t = \{s_1\ s_2\ \ldots\ s_m\ s_{m+1} \mid \\
\qquad \forall\ i \in \{1..(m+1)\}.s_i \in S_{t_r} \wedge s_{m+1} \preceq s_i\} \\
\quad \text{where} \quad t = t_1 \to t_2, Process\ t_1\ t_2 \\
\qquad\qquad m = nArgs(t), t_r = rType(t)
\end{array}
$$

Figura 5.17: El dominio de signaturas S_t

$$
\begin{array}{l}
nArgs :: Type \to \mathbb{N} \\
nArgs(t) = 0 \text{ si } nonfun(t) \\
nArgs(t_1 \to t_2) = 1 + nArgs(t_2) \\
nArgs(Process\ t_1\ t_2) = 1 + nArgs(t_2) \\
nArgs(\forall\beta.t) = nArgs(t) \text{ si } fun(t)
\end{array}
$$

Figura 5.18: Función *nArgs*

por $m + 1$ signaturas, cada una de las cuales corresponderá al tipo (no funcional) del resultado. Por m argumentos entendemos que eliminando todos los cuantificadores de polimorfismo y transformando los tipos proceso en tipos funcionales, el tipo obtenido es de la forma $t_1 \to \ldots \to t_m \to t_r$, donde t_r no es funcional. Llamaremos a este tipo la versión *desenrollada* (unrolled) del tipo funcional:

$$
\begin{array}{l}
unroll(t) = t \text{ si } nonfun(t) \\
unroll(t_1 \to t_2) = t_1 \to unroll(t_2) \\
unroll(Process\ t_1\ t_2) = unroll(t_1 \to t_2) \\
unroll(\forall\beta.t) = unroll(t) \text{ si } fun(t).
\end{array}
$$

Por ejemplo, la versión desenrollada de $Int \to (\forall\beta.Process\ \beta\ (\beta, Int))$ es $Int \to \beta \to (\beta, Int)$.

En las Figuras 5.18, 5.19 y 5.20 se definen tres funciones importantes, *nArgs*, *rType* y *aTypes*. Dado un tipo t, la primera devuelve el número de argumentos de t; la segunda el tipo (no funcional) de su resultado (es la identidad en el resto de casos); y la tercera la lista (de longitud $nArgs(t)$) de los tipos de los argumentos. Con estas definiciones podemos ver que la versión desenrollada de un tipo funcional t tiene $nArgs(t)$ argumentos de tipos $aTypes(t)$, y $rType(t)$ como tipo del resultado.

Para una mayor legibilidad de las signaturas para un tipo funcional,

$$
\begin{array}{l}
rType :: Type \rightarrow Type \\
rType(t) = t \text{ si } nonfun(t) \\
rType(t_1 \rightarrow t_2) = rType(t_2) \\
rType(Process\ t_1\ t_2) = rType(t_2) \\
rType(\forall \beta . t) = rType(t) \text{ si } fun(t)
\end{array}
$$

Figura 5.19: Función $rType$

$$
\begin{array}{l}
aTypes :: Type \rightarrow [Type] \\
aTypes(t) = [\ ] \text{ si } nonfun(t) \\
aTypes(t_1 \rightarrow t_2) = t_1 : aTypes(t_2) \\
aTypes(Process\ t_1\ t_2) = t_1 : aTypes(t_2) \\
aTypes(\forall \beta . t) = aTypes(t) \text{ si } fun(t)
\end{array}
$$

Figura 5.20: Función $aTypes$

en los ejemplos separaremos la última componente con un símbolo $+$. Un ejemplo de signatura para el tipo $Int \rightarrow (Int, Int)$ es $(N, D) + (D, D)$.

No toda secuencia de signaturas es una signatura válida para una función. Recordemos que la última componente se obtiene dando un valor determinista a todos los argumentos de la función, mientras que el resto de las componentes se obtienen dando un valor no determinista a alguno de sus argumentos. Puesto que las funciones son monótonas la última componente debe ser siempre menor o igual que todas las demás componentes. El orden entre signaturas ($\preceq$) se establece componente a componente, al igual que la mínima cota superior y la máxima cota inferior. Es fácil ver que con este orden, el dominio de las signaturas S_t para un tipo t es un retículo completo de altura $\mathcal{H}_t$, definida en la Figura 5.21.

En la Figura 5.22 se muestra el dominio S_t, correspondiente a $t = Int \rightarrow Int \rightarrow (Int, Int)$.

5.6.3 El muestreo

En esta sección se define la función de muestreo $\wp_t :: D_{2t} \rightarrow S_t$, la cual dado un valor abstracto en D_{2t}, obtiene la correspondiente signatura en S_t. La definición formal se muestra en la Figura 5.23. La signatura de un valor básico b es la correspondiente signatura básica B, es decir, si $b = d$ entonces $B = D$ y si $b = n$ entonces $B = N$. La signatura de una tupla es la tupla

$$
\begin{array}{l}
\mathcal{H}_K = \mathcal{H}_{T\ t_1 \ldots t_m} = \mathcal{H}_\beta = 1 \\
\mathcal{H}_{(t_1,\ldots,t_m)} = \sum_{i=1}^{m} \mathcal{H}_{t_i} \\
\mathcal{H}_{\forall \beta . t} = \mathcal{H}_t \\
\mathcal{H}_t = (m+1)\ \mathcal{H}_{t_r} \\
\quad \text{where} \\
\qquad t = t_1 \rightarrow t_2, Process\ t_1\ t_2 \\
\qquad m = nArgs(t), t_r = rType(t)
\end{array}
$$

Figura 5.21: Altura del dominio de signaturas $\mathcal{H}_t$

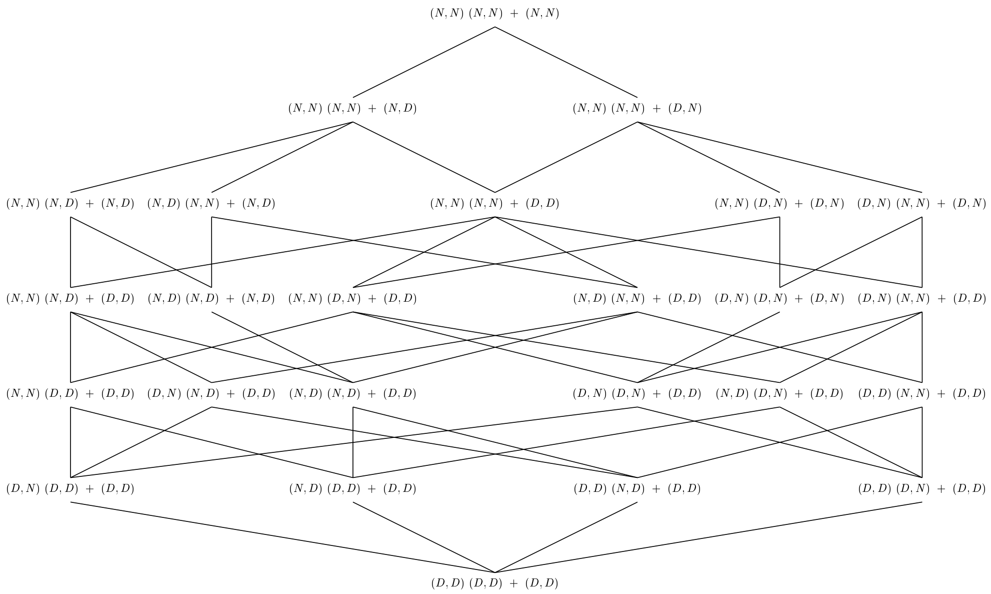


Figura 5.22: Signaturas para $Int \rightarrow Int \rightarrow (Int, Int)$

$$
\begin{array}{l}
\wp_t :: D_{2t} \to S_t \\
\wp_K(b) = \wp_{T\ t_1 \ldots t_m}(b) = \wp_\beta(b) = B \\
\wp_{(t_1,\ldots,t_m)}(e_1,\ldots,e_m) = (\wp_{t_1}(e_1),\ldots,\wp_{t_m}(e_m)) \\
\wp_{\forall\beta.t} = \wp_t \\
\wp_t(f) = \wp_{t_r}(f\ \gamma_{t_1}(n)\ \gamma_{t_2}(d)\ldots\gamma_{t_m}(d))\ \wp_{t_r}(f\ \gamma_{t_1}(d)\ \gamma_{t_2}(n)\ldots\gamma_{t_m}(d))\ldots \\
\qquad\qquad \wp_{t_r}(f\ \gamma_{t_1}(d)\ \gamma_{t_2}(d)\ldots\gamma_{t_m}(n))\ \wp_{t_r}(f\ \gamma_{t_1}(d)\ \gamma_{t_2}(d)\ldots\gamma_{t_m}(d)) \\
\quad \text{donde}\ \ t = t'_1 \to t'_2, \mathit{Process}\ t'_1\ t'_2,\ \ m = \mathit{nArgs}(t),\ \ t_r = \mathit{rType}(t), \\
\qquad\quad [t_1,\ldots,t_m] = \mathit{aTypes}(t)
\end{array}
$$

Figura 5.23: La función de muestreo

$$
\begin{array}{l}
\Re_t :: S_t \to D_{2t} \\
\Re_K(B) = \Re_{T\ t_1 \ldots t_m}(B) = \Re_\beta(B) = b \\
\Re_{(t_1,\ldots,t_m)}(s1,\ldots,s_m) = (\Re_{t_1}(s_1),\ldots,\Re_{t_m}(s_m)) \\
\Re_{\forall\beta.t} = \Re_t \\
\Re_t(s_1\ \ldots\ s_m\ s_{m+1}) = \\
\lambda z_1 \in D_{2t_1}.\ldots.\lambda z_m \in D_{2t_m}.
\begin{cases}
\Re_{t_r}(s_{m+1})\ \text{si}\ \bigwedge_{j=1}^{m} z_j \sqsubseteq \gamma_{t_j}(d) \\
\Re_{t_r}(s_i)\ \text{si}\ \bigwedge_{j=1, j\neq i}^{m} z_j \sqsubseteq \gamma_{t_j}(d) \wedge z_i \not\sqsubseteq \gamma_{t_i}(d)\ i \in \{1..m\} \\
\gamma_{t_r}(n)\ \text{e.o.c.}\ (m > 1)
\end{cases} \\
\quad \text{donde}\ \ t = t'_1 \to t'_2, \mathit{Process}\ t'_1\ t'_2,\ \ m = \mathit{nArgs}(t),\ \ t_r = \mathit{rType}(t), \\
\qquad\quad [t_1,\ldots,t_m] = \mathit{aTypes}(t)
\end{array}
$$

Figura 5.24: Función de concreción correspondiente al muestreo

de las signaturas de las componentes. Y finalmente, la signatura de una función o un proceso $f :: t$ es una secuencia de $m+1$ signaturas, donde $m = nArgs(t)$, que se obtienen mediante muestreo de f con las combinaciones de argumentos mencionados anteriormente.

Ya se ha dicho que en el proceso de muestreo se pierde cierta cantidad de información. Esto significa que una signatura representa varios valores abstractos. Cuando se desee recuperar el valor original, solamente se podrá devolver una aproximación. Una de estas aproximaciones la lleva a cabo la *función de concreción de signaturas* $\Re_t :: S_t \to D_{2t}$, definida en la Figura 5.24. Todos los casos, excepto el funcional, son simples. Dada una signatura $s = s_1 \ldots s_m\ s_{m+1} \in S_t$, $\Re_t(s)$ es una función de m argumentos, cada uno de ellos en el correspondiente D_{2t_i}. Sabemos que el último elemento s_{m+1} se obtuvo muestreando la función con argumentos $\gamma_{t_i}(d)$, $i \in \{1..m\}$. Luego, si todos los argumentos son menores o iguales que el correspondiente $\gamma_{t_i}(d)$, entonces se puede devolver la concreción de s_{m+1} como aproxima-

$$
\begin{array}{l}
\mathcal{W}_t :: D_{2t} \to D_{2t} \\
\mathcal{W}_K = \mathcal{W}_{T\ t_1 \dots t_m} = \mathcal{W}_\beta = id_{Basic} \\
\mathcal{W}_{(t_1,\dots,t_m)}(e_1, \dots, e_m) = (\mathcal{W}_{t_1}(e_1), \dots, \mathcal{W}_{t_m}(e_m)) \\
\mathcal{W}_{\forall \beta . t} = \mathcal{W}_t \\
\mathcal{W}_t(f) = \quad \lambda z_1 \in D_{2t_1}. \dots . \lambda z_m \in D_{2t_m}. \\
\qquad \begin{cases}
\mathcal{W}_{t_r}(f\ \gamma_{t_1}(d) \dots \gamma_{t_m}(d)) \text{ si } \bigwedge\limits_{i=1}^{m} z_i \sqsubseteq \gamma_{t_i}(d) \\
\mathcal{W}_{t_r}(f\ \gamma_{t_1}(d) \dots \gamma_{t_i}(n) \dots \gamma_{t_m}(d)) \text{ si } \bigwedge\limits_{j=1, j \neq i} z_j \sqsubseteq \gamma_{t_j}(d) \wedge z_i \not\sqsubseteq \gamma_{t_i}(d) \quad i \in \{1..m\} \\
\gamma_{t_r}(n) \text{ e.o.c. } (m > 1)
\end{cases} \\
\quad \text{donde } \ t = t_1' \to t_2', Process\ t_1'\ t_2', \quad m = nArgs(t), \quad t_r = rType(t), \\
\qquad\qquad [t_1, \dots, t_m] = aTypes(t)
\end{array}
$$

Figura 5.25: El operador de ensanchamiento

ción segura al resultado. La función original podría tener información más precisa para algunas combinaciones de argumentos por debajo de los $\gamma_{t_i}(d)$, pero ésta se ha perdido. También sabemos que s_i se obtuvo muestreando la función original dando el valor $\gamma_{t_i}(n)$ al i-ésimo argumento y $\gamma_{t_j}(d)$ al resto de ellos ($j \in \{1..m\}$, $j \neq i$). Luego si todos los elementos excepto el i-ésimo son menores o iguales que el correspondiente $\gamma_{t_j}(d)$, entonces se puede devolver la concreción de s_i como aproximación segura al resultado. Si más de uno de los argumentos no es menor o igual que el correspondiente $\gamma_{t_j}(d)$, solamente podemos devolver de forma segura el valor pesimista $\gamma_{t_r}(n)$, ya que carecemos de información sobre dichas combinaciones.

Vamos a usar un operador de ensanchamiento para acelerar el cálculo del punto fijo. El mismo viene dado por $\mathcal{W}_t = \Re_t \cdot \wp_t$, y aplicando las definiciones de ambos obtenemos la definición explícita en la Figura 5.25. En la Proposición 31 demostraremos que se trata de un operador de cierre superior. La definición de operador de ensanchamiento es más general [CC77, NNH99], pero dado un operador de cierre superior $\mathcal{W}_t \sqsupseteq id_{D_{2t}}$, se puede definir un operador de ensanchamiento $\nabla_t = \lambda(x, y).x \sqcup \mathcal{W}_t(y)$, como se hizo en [HH92]. Por ello usaremos de ahora en adelante la denominación operador de ensanchamiento, al igual que se hizo en [PP93].

En la siguiente proposición demostramos que $\wp_t$ y $\Re_t$ forman una inserción de Galois [CC79, NNH99], es decir, $\Re_t$ recupera la mayor cantidad posible de información, considerando la forma en que se construyó la signatura.

Proposición 31 *Para cada tipo t,*

(a) *Las funciones $\wp_t$, $\Re_t$, y $\mathcal{W}_t$ son monótonas y continuas,*

(b) $\mathcal{W}_t \sqsupseteq id_{D_{2t}}$,

(c) $\wp_t \cdot \Re_t = id_{S_t}$.

Demostración 6 (Proposición 31) Demostramos primero 31(a). Para ello demostramos por inducción estructural que $\wp_t$ y $\Re_t$ son monótonas. Puesto que D_{2t} es finito, entonces serán también continuas. Finalmente, puesto que $\mathcal{W}_t = \Re_t \cdot \wp_t$, será igualmente monótona y continua.

- $t = K, T\ t_1 \ldots t_m, \beta$. Es un caso trivial, ya que $\wp_t(b) = B$, $\Re_t(B) = b$, y $d \sqsubset n$, $D \preceq N$.

- $t = (t_1, \ldots, t_m)$. Sean $z, z' \in D_{2t}$, con $z = (z_1, \ldots, z_m) \sqsubseteq (z'_1, \ldots, z'_m) = z'$. Entonces:

$$\begin{array}{lcll} \wp_t(z) & = & (\wp_{t_1}(z_1), \ldots, \wp_{t_m}(z_m)) & \{\text{por definición de } \wp_t\} \\ & \preceq & (\wp_{t_1}(z'_1), \ldots, \wp_{t_m}(z'_m)) & \{\text{por h.i.}\} \\ & = & \wp_t(z') & \{\text{por definición de } \wp_t\}. \end{array}$$

Sean $s, s' \in S_t$, con $s = (s_1, \ldots, s_m) \preceq (s'_1, \ldots, s'_m) = s'$. Entonces:

$$\begin{array}{lcll} \Re_t(s) & = & (\Re_{t_1}(s_1), \ldots, \Re_{t_m}(s_m)) & \{\text{por definición de } \Re_t\} \\ & \sqsubseteq & (\Re_{t_1}(s'_1), \ldots, \Re_{t_m}(s'_m)) & \{\text{por h.i.}\} \\ & = & \Re_t(s') & \{\text{por definición de } \Re_t\}. \end{array}$$

- $t = t_1 \to t_2$ o $t = Process\ t_1\ t_2$. Sean $m = nArgs(t)$, $[t_1, \ldots, t_m] = aTypes(t)$ y $t_r = rType(t)$.

Sean $f, f' \in D_{2t}$ con $f \sqsubseteq f'$. Entonces

$$\begin{array}{lcl} \wp_t(f) & = & \overline{\wp_{t_r}(f\ (\gamma_{t_1}(d) \ldots \gamma_{t_i}(n) \ldots \gamma_{t_m}(d)))}\ \wp_{t_r}(f\ \gamma_{t_1}(d) \ldots \gamma_{t_m}(d)) \\ & & \quad \{\text{por definición de } \wp_t\} \\ & \preceq & \overline{\wp_{t_r}(f'\ (\gamma_{t_1}(d) \ldots \gamma_{t_i}(n) \ldots \gamma_{t_m}(d)))}\ \wp_{t_r}(f'\ \gamma_{t_1}(d) \ldots \gamma_{t_m}(d)) \\ & & \quad \{\text{por h.i. y } f \sqsubseteq f'\} \\ & = & \wp_t(f'). \end{array}$$

Sean $s = s_1 \ldots s_m\ s_{m+1} \in S_t$ y $s' = s'_1 \ldots s'_m\ s'_{m+1} \in S_t$, tales que $s \preceq s'$. Sean $z_i \in D_{2t_i}$ $(i \in \{1..m\})$. Hemos de probar que $\Re_t(s)\ \overline{z_i} \sqsubseteq \Re_t(s')\ \overline{z_i}$. Distinguimos varios casos.

Si $\bigwedge_{i=1}^m z_i \sqsubseteq \gamma_{t_i}(d)$ entonces, por definición de $\Re_t$ e hipótesis de inducción,

$$\Re_t(s)\ \overline{z_i} = \Re_{t_r}(s_{m+1}) \sqsubseteq \Re_{t_r}(s'_{m+1}) = \Re_t(s')\ \overline{z_i}.$$

En caso contrario, hay un $j \in \{1..m\}$ tal que $z_j \not\sqsubseteq \gamma_{t_j}(d)$. Si ahora $\bigwedge_{i=1, i \neq j}^m z_i \sqsubseteq \gamma_{t_i}(d)$, entonces, por definición de $\Re_t$ e hipótesis de inducción,

$$\Re_t(s)\ \overline{z_i} = \Re_{t_r}(s_j) \sqsubseteq \Re_{t_r}(s'_j) = \Re_t(s')\ \overline{z_i}.$$

En caso contrario, por definición de $\Re_t$:

$$\Re_t(s)\ \overline{z_i} = \gamma_{t_r}(n) = \Re_t(s')\ \overline{z_i}.$$

- $t = \forall\beta.t_1$. Este caso es trivial, por hipótesis de inducción, ya que $\wp_t = \wp_{t_1}$ y $\Re_t = \Re_{t_1}$.

Demostramos ahora simultáneamente 31(b) y 31(c), también por inducción estructural.

- $t = K$ o $t = T\ t_1 \ldots t_m$ o $t = \beta$. Son casos triviales, ya que $\mathcal{W}_t = id_{Basic}$ y $\wp_t(\Re_t(B)) = \wp_t(b) = B$.

- $t = (t_1, \ldots, t_m)$. Sean $z_i \in D_{2t_i}$ $(i \in \{1..m\})$. En este caso, por hipótesis de inducción:

$$\mathcal{W}_t(z_1, \ldots, z_m) = (\mathcal{W}_{t_1}(z_1), \ldots \mathcal{W}_{t_m}(z_m)) \sqsupseteq (z_1, \ldots, z_m).$$

  Adicionalmente:

$$\begin{array}{rcl} \wp_t(\Re_t(s_1, \ldots, s_m)) & = & (\wp_{t_1}(\Re_{t_1}(s_1)), \ldots, \wp_{t_m}(\Re_{t_m}(s_m))) \\ & & \quad \{\text{por definición de } \wp_t \text{ y } \Re_t\} \\ & = & (s_1, \ldots, s_m) \\ & & \quad \{\text{por h.i.}\}. \end{array}$$

- $t = t_1 \to t_2$ o $t = Process\ t_1\ t_2$. Sean $m = nArgs(t)$, $[t_1, \ldots, t_m] = aTypes(t)$ y $t_r = rType(t)$.

  Demostramos primero que $\mathcal{W}_t(d) \sqsupseteq id_{D_{2t}}$. Sean $f \in D_{2t}$ y $z_i \in D_{2t_i}$ para $i \in \{1..m\}$. Tenemos que demostrar que $\mathcal{W}_t(f)\ \overline{z_i} \sqsupseteq f\ \overline{z_i}$. Distinguiremos varios casos.

  Si (1) $\bigwedge_{i=1}^{m} z_i \sqsubseteq \gamma_{t_i}(d)$, entonces

$$\begin{array}{rcll} \mathcal{W}_t(f)\ \overline{z_i} & = & \mathcal{W}_{t_r}(f\ \overline{\gamma_{t_i}(d)}) & \{\text{por definición de } \mathcal{W}_t\} \\ & \sqsupseteq & f\ \overline{\gamma_{t_i}} & \{\text{por h.i.}\} \\ & \sqsupseteq & f\ \overline{z_i} & \{\text{por (1) y monotonía de } f\}. \end{array}$$

  En caso contrario, hay un $j \in \{1..m\}$ tal que $z_j \not\sqsubseteq \gamma_{t_j}(d)$. Se ha demostrado que para todo tipo t, (2) $\forall e \in D_{2t}.e \sqsubseteq \gamma_t(n)$.

  Si ahora (3) $\bigwedge_{i=1, i\neq j}^{m} z_i \sqsubseteq \gamma_{t_i}(d)$ entonces:

$$\begin{array}{rcl} \mathcal{W}_t(f)\ \overline{z_i} & = & \mathcal{W}_{t_r}(f\ \gamma_{t_1}(d) \ldots \gamma_{t_j}(n) \ldots \gamma_{t_m}(d)) \\ & & \quad \{\text{por definición de } \mathcal{W}_t\} \\ & \sqsupseteq & f\ \gamma_{t_1}(d) \ldots \gamma_{t_j}(n) \ldots \gamma_{t_m}(d) \\ & & \quad \{\text{por h.i.}\} \\ & \sqsupseteq & f\ \overline{z_i} \\ & & \quad \{\text{por (2), (3) y monotonía de } f\}. \end{array}$$

  En caso contrario, por definición de $\mathcal{W}_t$ y (2):

$$\mathcal{W}_t(f)\ \overline{z_i} = \gamma_{t_r}(n) \sqsupseteq f\ \overline{z_i}.$$

Demostremos ahora que $\wp_t \cdot \Re_t = id_{S_t}$. Sea $s_1 \ldots s_m\ s_{m+1} \in S_t$.

$$\begin{array}{rcl}\wp_t(\Re_t(s_1 \ldots s_m\ s_{m+1})) & = & \wp_{t_r}(\Re_{t_r}(s_1)) \ldots \wp_{t_r}(\Re_{t_r}(s_m))\ \wp_{t_r}(\Re_{t_r}(s_{m+1})) \\ & & \quad \{\text{por definición de } \wp_t \text{ y } \Re_t\} \\ & = & s_1 \ldots s_m\ s_{m+1} \\ & & \quad \{\text{por h.i.}\}.\end{array}$$

- $t = \forall\beta.t_1$. Este caso es trivial, por hipótesis de inducción, ya que $\mathcal{W}_t = \mathcal{W}_{t_1}$, $\wp_t = \wp_{t_1}$ y $\Re_t = \Re_{t_1}$.

$\square$

El análisis

El análisis es muy similar al segundo, presentado en la Sección 5.5. Para distinguirlo usaremos un subíndice 3. Los dominios abstractos son los mismos. La única expresión en la que aparecen diferencias es el **let** recursivo, donde se calcula el punto fijo. En este tercer análisis su interpretación es:

$$[\![\textbf{let rec}\ \overline{\{v_i = e_i\}}\ \textbf{in}\ e']\!]_3\ \rho_3 = [\![e']\!]_3\ (\mathit{fix}\ (\lambda\rho'_3.\rho_3\ \overline{[v_i \mapsto \mathcal{W}_{t_i}([\![e_i]\!]_3\ \rho'_3)]}))$$

donde $e_i :: t_i$. Modificando el operador de ensanchamiento podemos tener diferentes variantes del análisis. Por tanto, podemos expresar el análisis parametrizado por la colección wop_t de operadores de ensanchamiento, obteniendo $[\![\cdot]\!]_3^{wop}$.

5.7 Relación entre los análisis

5.7.1 Introducción

Se pretendía que el tercer análisis fuera un análisis intermedio entre el primero y el segundo. En esta sección mencionaremos además otros operadores de ensanchamiento posibles ($\mathcal{W}_b$, $\mathcal{W}_c$ y $\mathcal{W}_d$) y estudiaremos su relación con el que hemos definido $\mathcal{W}$. La principal diferencia entre ellos radica en el tratamiento de las tuplas, ya sea en los argumentos o en el resultado de las funciones, según los tratemos como entidades indivisibles o componente a componente. Primero demostraremos que el tercer análisis es una aproximación segura al segundo, y posteriormente veremos que el primer análisis es sólo una aproximación segura de algunas de las variantes del tercero. Esto nos permitirá establecer la jerarquía de análisis mostrada en la Figura 5.26.

En la Figura 5.27 se presenta una expresión ejemplo $e :: (Int, Int)$ que pone de relieve la diferencia de potencia entre $[\![\cdot]\!]_1$, $[\![\cdot]\!]_3^{\mathcal{W}}$ y $[\![\cdot]\!]_2$. Para aumentar la claridad la sintaxis está azucarada. Dado un par de enteros, (p_1, p_2) y un tercer entero x, la función $f :: t$, donde $t = (Int, Int) \to Int \to (Int, Int)$, calcula el par $({p_1}^{2*p_2}, x*p_2!)$. La variable q es libre en e. Supongamos que en

$$
\begin{tikzcd}[arrows=dash]
 & \llbracket\cdot\rrbracket_2 \arrow[d] & \\
 & \llbracket\cdot\rrbracket_3^{\mathcal{W}_b} \arrow[dl] \arrow[dr] & \\
\llbracket\cdot\rrbracket_3^{\mathcal{W}_c} \arrow[d] & & \llbracket\cdot\rrbracket_3^{\mathcal{W}} \arrow[dll] \arrow[d] \\
\llbracket\cdot\rrbracket_3^{\mathcal{W}_d} & & \llbracket\cdot\rrbracket_1
\end{tikzcd}
$$

Figura 5.26: La jerarquía de análisis

$$
\begin{array}{l}
e = \mathbf{let\ rec} \\
\qquad f = \lambda p.\lambda x.\mathbf{case}\ p\ \mathbf{of} \\
\qquad\qquad\qquad\qquad (p_1, p_2) \rightarrow \mathbf{case}\ p_2\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad 0 \rightarrow (p_1, x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad z \rightarrow (f\ (p_1 * p_1, p_2 - 1))\ (x * p_2) \\
\quad \mathbf{in\ let} \\
\qquad f_1 = (f\ (q, 3))\ 4 \\
\qquad f_2 = (f\ (1, 2))\ q \\
\qquad x_1 = \mathbf{case}\ f_1\ \mathbf{of}\ (f_{11}, f_{12}) \rightarrow f_{12} \\
\qquad x_2 = \mathbf{case}\ f_2\ \mathbf{of}\ (f_{21}, f_{22}) \rightarrow f_{21} \\
\qquad \mathbf{in}\ (x_1, x_2)
\end{array}
$$

Figura 5.27: Una expresión ejemplo e

nuestro entorno abstracto tiene un valor n, es decir, $\rho = [q \mapsto n]$. Entonces, aplicando las definiciones de cada uno de los análisis obtenemos

$$\llbracket e \rrbracket_1\ \rho = (n, n) \sqsupset \llbracket e \rrbracket_3^{\mathcal{W}} \rho = (n, d) \sqsupset \llbracket e \rrbracket_2\ \rho = (d, d).$$

5.7.2 Relación entre el segundo y tercer análisis

La siguiente proposición nos indica que el tercer análisis es menos preciso que el segundo. Esto es cierto para cualquier variante de este tercer análisis, y en particular para la que hemos descrito con mayor detalle.

Proposición 32 *Sea $\mathcal{W}'_t : D_{2t} \rightarrow D_{2t}$ un operador de ensanchamiento para cada tipo t. Dados entornos ρ_2 y ρ_3 tales que para cada variable $v :: t_v$, $\rho_2(v) \sqsubseteq \rho_3(v)$. Entonces para cada expresión $e :: t_e$, $\llbracket e \rrbracket_2\ \rho_2 \sqsubseteq \llbracket e \rrbracket_3^{\mathcal{W}'} \rho_3$.*

Demostración 7 (Proposición 32) Esta proposición se demuestra por inducción estructural sobre e. Todos los casos, excepto el **let** recursivo, se pueden demostrar fácilmente usando las hipótesis sobre los entornos, la hipótesis de inducción y las propiedades de monotonía de α_t, γ_t, $\alpha_{tinst\,t'}$ y $\gamma_{t'\,tinst}$. Por tanto nos limitamos a estudiar la expresión **let** recursiva.

Sea $e =$ **let rec** $\overline{\{v_i = e_i\}}$ **in** $e' :: t$, donde $e' :: t$, y cada v_i y e_i tienen tipo t_i. El punto fijo se puede calcular usando la cadena ascendente de Kleene:

$$[\![e]\!]_2\ \rho_2 = [\![e']\!]_2\ (\bigsqcup_{n\in\mathbb{N}} (\lambda\rho'_2.\rho_2\ \overline{[v_i \to [\![e_i]\!]_2\ \rho'_2]})^n(\rho_0))$$

donde ρ_0 es el entorno inicial en el que cada variable $y :: t_y$ tiene como valor abstracto $\bot_{t_y}$ (es decir, el ínfimo del dominio correspondiente). Denotaremos por F a la función entre entornos $\lambda\rho'_2.\rho_2\ \overline{[v_i \to [\![e_i]\!]_2\ \rho'_2]}$ y por ρ_2^{fix} a $\bigsqcup_{n\in\mathbb{N}} F^n(\rho_0)$.

De forma similar

$$[\![e]\!]_3^{\mathcal{W}'}\rho_t = [\![e']\!]_3^{\mathcal{W}'}(\bigsqcup_{n\in\mathbb{N}} (\lambda\rho'_3.\rho_3\ \overline{[v_i \to \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}\rho'_3)]})^n(\rho_0)).$$

Denotaremos por G a la función entre entornos $\lambda\rho'_3.\rho_3\ \overline{[v_i \to \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}\rho'_3)]}$ y por ρ_3^{fix} a $\bigsqcup_{n\in\mathbb{N}} G^n(\rho_0)$.

Si demostráramos que para cada variable $y :: t_y$, $\rho_2^{fix}(y) \sqsubseteq \rho_3^{fix}(y)$, entonces por h.i. tendríamos que

$$[\![e']\!]_2\ \rho_2^{fix} \sqsubseteq [\![e']\!]_3^{\mathcal{W}'}\rho_3^{fix}$$

que es lo que debemos demostrar. Para ello, veamos que para todo $n \geq 0$, se tiene

$$\forall y :: t_y.(F^n(\rho_0))(y) \sqsubseteq (G^n(\rho_0))(y)$$

Si fuera cierto entonces

$$\forall y :: t_y.(\bigsqcup_{n\in\mathbb{N}} F^n(\rho_0))(y) \sqsubseteq (\bigsqcup_{n\in\mathbb{N}} G^n(\rho_0))(y)$$

que es lo que deseamos demostrar. Se demuestra por inducción sobre n:

- $n = 0$. Es un caso trivial, ya que $F^0(\rho_0) = \rho_0$ y $G^0(\rho_0) = \rho_0$.

- $n = m + 1$. Entonces

$$\begin{array}{rcll} F^{m+1}(\rho_0) & = & F(F^m(\rho_0)) & \\ & = & \rho_2\ \overline{[v_i \to [\![e_i]\!]_2\ (F^m(\rho_0))]} & \text{por def. de } F, \end{array}$$

y

$$\begin{array}{rcll} G^{m+1}(\rho_0) & = & G(G^m(\rho_0)) & \\ & = & \rho_3\ \overline{[v_i \to \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}(G^m(\rho_0)))]} & \text{por def. de } G. \end{array}$$

Para $y :: t_y$, queremos demostrar que $(F^{m+1}(\rho_0))(y) \sqsubseteq (G^{m+1}(\rho_0))(y)$. Distinguimos dos casos. Si y no es ninguna de las v_i, entonces se cumple por hipótesis sobre los entornos ρ_2 y ρ_3. Si es una de las v_i, tenemos que demostrar que

$$[\![e_i]\!]_2 \ (F^m(\rho_0)) \sqsubseteq \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}(G^m(\rho_0))).$$

Esto se cumple por hipótesis de inducción y por la hipótesis sobre $\mathcal{W}'_t$:

$$[\![e_i]\!]_2 \ (F^m(\rho_0)) \sqsubseteq [\![e_i]\!]_3^{\mathcal{W}'}(G^m(\rho_0)) \sqsubseteq \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}(G^m(\rho_0))).$$

□

5.7.3 Otras variantes del tercer análisis

En esta subsección se presentan otras variantes del tercer análisis. Las relaciones entre ellas y con los análisis $[\![\cdot]\!]_1$ y $[\![\cdot]\!]_2$ se muestran en la Figura 5.26.

En primer lugar definimos una variante $[\![\cdot]\!]_3^{\mathcal{W}_b}$ del tercer análisis más potente que la definida anteriormente $[\![\cdot]\!]_3^{\mathcal{W}}$. La idea es currificar los argumentos de tipo tupla para obtener una mayor cantidad de información. En la versión actual, una función de tipo $(t_1, \ldots, t_m) \rightarrow t$ es muestreada con $\gamma_{(t_1,\ldots,t_m)}(n)$ y $\gamma_{(t_1,\ldots,t_m)}(d)$. En esta nueva variante, se muestrea con $(\gamma_{t_1}(n), \gamma_{t_2}(d), \ldots, \gamma_{t_m}(d)), (\gamma_{t_1}(d), \gamma_{t_2}(n), \ldots, \gamma_{t_m}(d)) \ldots (\gamma_{t_1}(d), \gamma_{t_2}(d), \ldots,$ $\gamma_{t_m}(n))$, es decir, las componentes de las tuplas se consideran como argumentos separados. Los resultados de estos muestreos se agrupan usando una notación especial $\mathcal{U}(A_1, \ldots, A_m)$ para distinguir un muestreo de tupla de un resultado de tipo tupla.

Como ejemplo, sea $f \in D_{2t}$, donde $t = (Int, Int) \rightarrow Int \rightarrow (Int, Int)$, $f = \lambda x \in Basic \times Basic.\lambda x \in Basic.(\pi_1(p), x)$. En este caso, la versión presente devolvería $s = (N, D)(D, N) + (D, D)$, correspondiente a las aplicaciones $f\ (n, n)\ d$, $f\ (d, d)\ n$ y $f\ (d, d)\ d$. El ensanchamiento correspondiente es $\mathcal{W}_t(s) = \lambda p \in Basic \times Basic.\lambda x \in Basic.(\pi_1(p) \sqcup \pi_2(p), x) \sqsupset f$. Con la nueva forma de muestreo $s' = \wp_t^B(f) = \mathcal{U}((N, D), (D, D))\ (D, N) + (D, D)$. La primera componente $\mathcal{U}((N, D), (D, D))$ corresponde a las aplicaciones $f\ (n, d)\ d$ y $f\ (d, n)\ d$, y las otras dos a las aplicaciones $f\ (d, d)\ n$ y $f\ (d, d)\ d$. Obsérvese que el muestreo es mayor, gracias a lo cual el ensanchamiento correspondiente es más preciso que el anterior: $\mathcal{W}_t^B(s') = f$.

También se pueden definir otras variantes menos potentes que la elegida y aún otras incomparables con ella. Una posibilidad consiste en aplanar las tuplas en el resultado de la función, con lo que se pierde información. Denotaremos esta variante con $[\![\cdot]\!]_3^{\mathcal{W}_d}$. En el ejemplo anterior, $s'' = \wp_t^D(f) = N\ N\ +\ D$ y $\mathcal{W}_{dt}(f) = \lambda p \in Basic \times Basic.\lambda x \in Basic.\pi_1(p) \sqcup \pi_2(p) \sqcup x \sqsupset \mathcal{W}_t(f)$.

Otra posibilidad es currificar los argumentos tupla, como en $[\![\cdot]\!]_3^{\mathcal{W}_b}$, pero aplanar las tuplas en el resultado (con lo que será menos preciso que $\mathcal{W}_b$). Denotaremos esta variante con $[\![\cdot]\!]_3^{\mathcal{W}_c}$. Como se muestra en la Figura 5.26, esta variante es incomparable con $[\![\cdot]\!]_3^{\mathcal{W}}$. En aquellas funciones con un tipo tupla como resultado pero sin argumentos de tipo tupla, $[\![\cdot]\!]_3^{\mathcal{W}}$ produce resultados más precisos; mientras que en aquellas funciones con argumentos tuplas pero cuyo resultado no es de tipo tupla, $[\![\cdot]\!]_3^{\mathcal{W}_c}$ es más preciso.

Como ejemplo, sea $f \in D_{2t}$ donde $t = Int \to (Int, Int)$ y $f = \lambda z \in Basic.(d,x)$. En este caso $\wp_t^C(f) = N \ + \ D$ y $\mathcal{W}_t^C(f) = \lambda z' \in Basic.z'$, mientras que $\wp_t(f) = (D,N) \ + \ (D,D)$ y $\mathcal{W}_t(f) = f$. Sea $g \in D_{2t'}$ con $t' = (Int, Int) \to Int$ y $g = \lambda p \in Basic \times Basic.\pi_1(p)$; ahora $\wp_{t'}^C(f) = \mathcal{U}(N,D) \ + \ D$ y $\mathcal{W}_{t'}^C(f) = f$, mientras que $\wp_{t'}(f) = N \ + \ D$ y $\mathcal{W}_{t'}(f) = \lambda p \in Basic \times Basic.\pi_1(p) \sqcup \pi_2(p)$. Finalmente $[\![\cdot]\!]_3^{\mathcal{W}_c}$ es claramente mejor que $[\![\cdot]\!]_3^{\mathcal{W}_d}$.

La siguiente proposición nos dice que dados dos operadores de ensanchamiento comparables, las correspondientes variantes del tercer análisis son también comparables.

Proposición 33 *Sean $\mathcal{W}'_t$, $\mathcal{W}''_t$ operadores de ensanchamiento para cada tipo t. Sean ρ_3, ρ'_3 tales que para cada variable $v :: t_v$, $\rho_3(v) \sqsubseteq \rho'_3(v)$. Entonces si para cada tipo t, $\mathcal{W}'_t \sqsubseteq \mathcal{W}''_t$, para cada expresión $e :: t_e$, $[\![e]\!]_3^{\mathcal{W}'} \rho_3 \sqsubseteq [\![e]\!]_3^{\mathcal{W}''} \rho'_3$.*

Demostración 8 (Proposición 33) Esta demostración es similar a la anterior. En particular, se pueden seguir los mismos pasos en el caso de la expresión **let** recursiva, donde ahora

$$F = \lambda \rho'_3.\rho_3 \ \overline{[v_i \to \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'} \rho'_3)]}),$$

y

$$G = \lambda \rho''_3.\rho_3 \ \overline{[v_i \to \mathcal{W}''_{t_i}([\![e_i]\!]_3^{\mathcal{W}''} \rho''_3)]}).$$

Se puede llevar a cabo la misma inducción sobre n, para demostrar finalmente que

$$\mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}(F^m(\rho_0))) \sqsubseteq \mathcal{W}''_{t_i}([\![e_i]\!]_3^{\mathcal{W}''}(G^m(\rho_0))),$$

lo que es cierto por hipótesis de inducción, dado que $\mathcal{W}'_t \sqsubseteq \mathcal{W}''_t$, y por monotonía de $\mathcal{W}''_t$:

$$\begin{array}{rcl} \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}(F^m(\rho_0))) & \sqsubseteq & \mathcal{W}''_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}(F^m(\rho_0))) \\ & \sqsubseteq & \mathcal{W}''_{t_i}([\![e_i]\!]_3^{\mathcal{W}''}(G^m(\rho_0))). \end{array}$$

$\square$

5.7.4 Relación entre el primer y el tercer análisis

A *grosso modo* veremos que el primer análisis es menos preciso que algunas variantes del tercer análisis, es decir, que es una aproximación segura

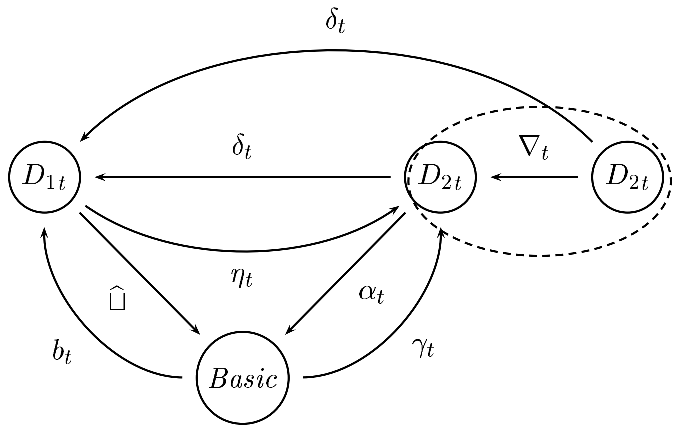


Figura 5.28: Esquema representativo de las funciones de la Sección 5.7

a ellas. En particular, demostraremos que el primer análisis es una aproximación superior a aquellas variantes del tercer análisis que cumplen una determinada propiedad (ver Teorema 43). Demostraremos que el operador de ensanchamiento definido en la Figura 5.25 cumple dicha propiedad.

Puesto que todas las variantes del tercer análisis son aproximaciones superiores al segundo análisis, también se cumple, por tanto, que el primer análisis es una aproximación superior al segundo.

En la Figura 5.28 se muestra un esquema de las funciones definidas en esta sección junto con sus dominios de definición. Utilizándola se pueden visualizar fácilmente muchas de las proposiciones de esta sección. Puesto que los dominios abstractos para el segundo y tercer análisis son los mismos, todas las proposiciones que impliquen solamente a los dominios serán trivialmente ciertas para ambos análisis.

En el primer análisis no hay tuplas anidadas ni funciones abstractas. Sin embargo, el valor abstracto a de una función en dicho análisis representa a una función abstracta. Sabemos que a representa el comportamiento de la función dado un argumento determinista. Si el argumento es no determinista, o tiene cierta “dosis” de no determinismo, indicamos que el resultado de la función siempre es no determinista. Lo que estamos haciendo en realidad es simular el comportamiento de una función abstracta. Supongamos que el valor abstracto de una función en el primer análisis es a. ¿Cual es la función abstracta que representa a? Es una función que toma un valor abstracto: si dicho valor abstracto es no determinista, su resultado es no determinista, y si es determinista, devuelve como resultado el valor abstracto a. Esto nos lleva a definir una *función de expansión*, η_t (ver Figura 5.29) que *expande* los valores abstractos de los dominios del primer análisis en valores abstractos pertenecientes a los dominios abstractos del segundo y tercer análisis. Puesto que en el primer análisis tampoco hay tuplas anidadas también será

$$
\begin{array}{l}
\eta_t : D_{1t} \to D_{2t} \\
\eta_K = \eta_{T\ t_1 \ldots t_m} = \eta_\beta = id_{Basic} \\
\eta_{(t_1,\ldots,t_m)}(b_1,\ldots,b_m) = (\gamma_{t_1}(b_1),\ldots,\gamma_{t_m}(b_m)) \\
\eta_{t_1 \to t_2}(a) = \eta_{Process\ t_1\ t_2}(a) = \\
\qquad \lambda z \in D_{2t_1}. \begin{cases} \eta_{t_2}(a) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ e.o.c.} \end{cases} \\
\eta_{\forall \beta.t} = \eta_t
\end{array}
$$

Figura 5.29: Función de expansión

$$
\begin{array}{l}
\delta_t : D_{2t} \to D_{1t} \\
\delta_{T\ t_1 \ldots t_m} = \delta_\beta = id_{Basic} \\
\delta_{(t_1,\ldots,t_m)}(b_1,\ldots,b_m) = (\alpha_{t_1}(e_1),\ldots,\alpha_{t_m}(e_m)) \\
\delta_{t_1 \to t_2}(f) = \delta_{Process\ t_1\ t_2}(f) = \delta_{t_2}(f(\gamma_{t_1}(d))) \\
\delta_{\forall \beta.t} = \delta_t
\end{array}
$$

Figura 5.30: Función de compresión

necesario adaptar cada uno de los valores básicos a los tipos de las componentes. Para los tipos básicos, tipos algebraicos y variables de tipo η_t es obviamente la función identidad, ya que en ambos análisis los correspondientes dominios abstractos son todos iguales a *Basic*. El caso polimórfico se reduce al tipo sin cualificar.

También podemos mirar el segundo y tercer análisis desde el punto de vista del primero. Simplemente hace falta aplanar las tuplas hasta el primer nivel y aplicar las funciones a $\gamma_t(d)$, para así obtener el comportamiento de la función cuando se aplica a argumentos deterministas. Esto está representado por la *función de compresión* δ_t definida en la Figura 5.30. Los casos de tipos básicos, tipos algebraicos, variables de tipo y tipos polimórficos son similares a los de la función de expansión.

Demostramos ahora que δ_t y η_t forman una inserción de Galois.

Proposición 34 *Para cada tipo t:*

(a) *Las funciones δ_t y η_t son monótonas y continuas,*

(b) $\delta_t \cdot \eta_t = id_{D_{1t}}$,

(c) $\eta_t \cdot \delta_t \sqsupseteq id_{D_{2t}}$.

Demostración 9 (Proposition 34) El punto 34(a) se puede demostrar fácilmente por inducción estructural sobre t aplicando la Proposición 25(a). Los puntos 34(b) y 34(c) se pueden demostrar por inducción estructural sobre t aplicando las Proposiciones 25(d), 25(b) y 25(c).

Demostremos primero el punto 34(a). Si demostramos que son monótonas, puesto que sus dominios son finitos, también serán continuas. La monotonía se demuestra por inducción estructural sobre t.

Demostremos primero que δ_t es monótona:

- $t = K$ o $t = T\ t_1 \dots t_m$ o $t = \beta$. Estos casos son triviales ya que $\delta_t = id_{Basic}$.
- $t = (t_1, \dots, t_m)$. Sean $e_i, e'_i \in D_{2t_i}$ con $(e_1, \dots, e_m) \sqsubseteq (e'_1, \dots, e'_m)$. Entonces,

$$
\begin{array}{ll}
\delta_t(e_1, \dots, e_m) & \\
\quad = (\alpha_{t_1}(e_1), \dots, \alpha_{t_m}(e_m)) & \{\text{por def. de } \delta_t\} \\
\quad \sqsubseteq (\alpha_{t_1}(e'_1), \dots, \alpha_{t_m}(e'_m)) & \{\text{por monotonía de } \alpha_t\} \\
\quad = \delta_t(e'_1, \dots, e'_m) & \{\text{por definición de } \delta_t\}.
\end{array}
$$

- $t = t_1 \to t_2$ o $t = \mathit{Process}\ t_1\ t_2$. Sean $f, f' \in [D_{2t_1} \to D_{2t_2}]$ con $f \sqsubseteq f'$, es decir, para cada $e \in D_{2t_1}$, $f(e) \sqsubseteq f'(e)$. Entonces,

$$
\begin{array}{ll}
\delta_t(f) & \\
\quad = \delta_{t_2}(f(\gamma_{t_1}(d))) & \{\text{por definición de } \delta_t\} \\
\quad \sqsubseteq \delta_{t_2}(f'(\gamma_{t_1}(d))) & \{\text{por h.i. y } f \sqsubseteq f'\} \\
\quad = \delta_t(f') & \{\text{por definición de } \delta_t\}.
\end{array}
$$

- $t = \forall\beta.t'$. Como $\delta_t = \delta_{t'}$, se cumple trivialmente por hipótesis de inducción.

Demostremos ahora la monotonía de η_t.

- $t = K$ o $t = T\ t_1 \dots t_m$ o $t = \beta$. Estos casos son triviales ya que $\eta_t = id_{Basic}$.
- $t = (t_1, \dots, t_m)$. Sean $b_i, b'_i \in \mathit{Basic}$ con $(b_1, \dots, b_m) \sqsubseteq (b'_1, \dots, b'_m)$. Entonces,

$$
\begin{array}{ll}
\eta_t(b_1, \dots, b_m) & \\
\quad = (\gamma_{t_1}(b_1), \dots, \gamma_{t_m}(b_m)) & \\
\quad \sqsubseteq (\gamma_{t_1}(b'_1), \dots, \gamma_{t_m}(b'_m)) & \{\text{por monotonía de } \gamma_t\} \\
\quad = \eta_t(b'_1, \dots, b'_m). &
\end{array}
$$

- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Sean $a, a' \in D_{1t} = D_{1t_2}$ con $a \sqsubseteq a'$. Tenemos que demostrar que para cada $e \in D_{2t_1}$, $(\eta_t(a))(e) \sqsubseteq (\eta_t(a'))(e)$. Sea $e \in D_{2t_1}$. Distinguimos dos casos: $e \sqsubseteq \gamma_{t_1}(d)$ o $e \not\sqsubseteq \gamma_{t_1}(d)$.

Si $e \sqsubseteq \gamma_{t_1}(d)$, entonces

$$\begin{array}{ll} (\eta_t(a))(e) & \\ \quad = \eta_{t_2}(a) & \{\text{por definición de } \eta_t\} \\ \quad \sqsubseteq \eta_{t_2}(a') & \{\text{por h.i. y } a \sqsubseteq a'\} \\ \quad = (\eta_t(a'))(e) & \{\text{por definición de } \eta_t\}. \end{array}$$

En caso contrario, $e \not\sqsubseteq \gamma_{t_1}(d)$, y por definición de η_t: $(\eta_t(a))(e) = \gamma_{t_2}(n) = (\eta_t(a'))(e)$.

- $t = \forall\beta.t'$. Como $\eta_t = \eta_{t'}$, se cumple trivialmente por hipótesis de inducción.

Demostramos ahora los puntos 34(b) y 34(c) por inducción estructural sobre t. Veamos primero que $\delta_t \cdot \eta_t = id_{D_{1t}}$:

- $t = K$ o $t = T\ t_1 \ldots t_m$ o $t = \beta$. Estos casos son triviales, ya que $D_{1t} = Basic$ y $\eta_t = \delta_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Sean $b_i \in Basic$, para cada $i = 1, \ldots, m$. Entonces:

$$\begin{array}{ll} \delta_t(\eta_t(b_1, \ldots, b_m)) & \\ \quad = \delta_t(\gamma_{t_1}(b_1), \ldots, \gamma_{t_m}(b_m)) & \{\text{por definición de } \eta_t\} \\ \quad = (\alpha_{t_1}(\gamma_{t_1}(b_1)), \ldots, \alpha_{t_m}(\gamma_{t_m}(b_m))) & \{\text{por definición de } \delta_t\} \\ \quad = (b_1, \ldots, b_m) & \{\text{por la Proposición 25(b)}\}. \end{array}$$

- $t = t_1 \to t_2$ o $t = Process\ t_1\ t_2$. Sea $a \in D_{1t}$. Entonces

$$\begin{array}{ll} \delta_t(\eta_t(a)) & \\ \quad = \delta_t\left(\lambda z. \left\{ \begin{array}{l} \eta_{t_2}(a) \text{ si } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ e.o.c.} \end{array} \right. \right) & \{\text{por definición de } \eta_t\} \\ \quad = \delta_{t_2}(\eta_{t_2}(a)) & \{\text{por definición de } \delta_t\} \\ \quad = a & \{\text{por h.i.}\}. \end{array}$$

- $t = \forall\beta.t'$. En este caso $D_{1t} = D_{1t'}$, $\eta_t = \eta_{t'}$ y $\delta_t = \delta_{t'}$, luego se cumple trivialmente por h.i.

Demostremos ahora que $\eta_t \cdot \delta_t \sqsupseteq id_{D_{2t}}$:

- $t = K$ o $t = T\ t_1 \ldots t_m$ o $t = \beta$. Estos casos son triviales, ya que $D_{2t} = Basic$ y $\eta_t = \delta_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Sean $e_i \in D_{2t_i}$, para cada $i = 1, \ldots, m$. Entonces,

$$\begin{array}{ll} \eta_t(\delta_t(e_1, \ldots, e_m)) & \\ \quad = \eta_t(\alpha_{t_1}(e_1), \ldots, \alpha_{t_m}(e_m)) & \{\text{por definición de } \delta_t\} \\ \quad = (\gamma_{t_1}(\alpha_{t_1}(e_1)), \ldots, \gamma_{t_m}(\alpha_{t_m}(e_m))) & \{\text{por definición de } \eta_t\} \\ \quad \sqsupseteq (e_1, \ldots, e_m) & \{\text{por la Proposición 25(c)}\}. \end{array}$$

- $t = t_1 \to t_2$ o $t = Process\ t_1\ t_2$. Sea $f \in D_{2t}$. Entonces

$$
\begin{array}{ll}
\eta_t(\delta_t(f)) & \\
= \eta_t(\delta_{t_2}(f(\gamma_{t_1}(d)))) & \{\text{por definición de } \delta_t\} \\
= \lambda z. \left\{ \begin{array}{l} \eta_{t_2}(\delta_{t_2}(f(\gamma_{t_1}(d)))) \text{ si } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ e.o.c.} \end{array} \right. & \{\text{por definición de } \eta_t\}.
\end{array}
$$

Sea $e \in D_{2t_1}$. Tenemos que demostrar que $f(e) \sqsubseteq (\eta_t(\delta_t(f)))(e)$. Distinguimos dos casos:

  - $e \sqsubseteq \gamma_{t_1}(d)$. En este caso

$$(\eta_t(\delta_t(f)))(e) = \eta_{t_2}(\delta_{t_2}(f(\gamma_{t_1}(d)))).$$

  Puesto que f es continua y consecuentemente monótona, se cumple que $f(e) \sqsubseteq f(\gamma_{t_1}(d))$. Luego

$$
\begin{array}{ll}
(\eta_t(\delta_t(f)))(e) & \\
\quad = \eta_{t_2}(\delta_{t_2}(f(\gamma_{t_1}(d)))) & \\
\quad \sqsupseteq f(\gamma_{t_1}(d)) & \{\text{por h.i.}\} \\
\quad \sqsupseteq f(e) & \{\text{por monotonía de } f\}.
\end{array}
$$

  - $e \not\sqsubseteq \gamma_{t_1}(d)$. En este caso $(\eta_t(\delta_t(f)))(e) = \gamma_{t_2}(n)$. Por la Proposición 25(d) $\gamma_{t_2}(n) \sqsupseteq f(e)$, pues $f(e) \in D_{2t_2}$.

- $t = \forall\beta.t'$. En este caso $D_{2t} = D_{2t'}$, $\eta_t = \eta_{t'}$ y $\delta_t = \delta_{t'}$, luego se cumple trivialmente por h.i.

□

La siguiente proposición afirma que si aplicamos δ_t a un valor abstracto en D_{2t} para obtener un valor en D_{1t}, y después aplicamos $\widehat{\sqcup}$ al resultado (podría ser una tupla de valores abstractos básicos), obtenemos el mismo resultado que si le aplicamos directamente α_t para obtener un valor en $Basic$. Esto demuestra una similitud entre la función abstracta α_t usada en las aplicaciones de constructores y la función de compresión δ_t. En efecto, las ideas utilizadas en ambos casos son las mismas.

Proposición 35 *Para cada tipo t:* $\widehat{\sqcup} \cdot \delta_t = \alpha_t$.

Demostración 10 (Proposición 35) Por inducción estructural sobre t:

- $t = K$ o $t = T\ t_1 \ldots t_m$ o $t = \beta$. Estos casos son triviales, ya que $\delta_t = \alpha_t = id_{Basic}$.

$$
\begin{array}{l}
\nabla_t : D_{2t} \to D_{2t} \\
\nabla_K = \nabla_{T\ t_1 \ldots t_m} = \nabla_\beta = id_{Basic} \\
\nabla_{(t_1,\ldots,t_m)}(e_1,\ldots,e_m) = \\
\qquad (\gamma_{t_1}(\alpha_{t_1}(e_1)),\ldots,\gamma_{t_m}(\alpha_{t_m}(e_m))) \\
\nabla_{t_1 \to t_2}(f) = \nabla_{Process\ t_1\ t_2}(f) = \\
\qquad \lambda z \in D_{2t_1}. \left\{ \begin{array}{l} \nabla_{t_2}(f(\gamma_{t_1}(d))) \text{ si } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ e.o.c.} \end{array} \right. \\
\nabla_{\forall \beta . t} = \nabla_t
\end{array}
$$

Figura 5.31: La función de elevación ∇_t

- $t = (t_1, \ldots, t_m)$. Sean $e_i \in D_{2t_i}$, para cada $i = 1, \ldots, m$. Entonces:

$$
\begin{array}{ll}
\widehat{\sqcup}(\delta_t(e_1,\ldots,e_m)) & \\
\quad = \widehat{\sqcup}(\alpha_{t_1}(e_1),\ldots,\alpha_{t_m}(e_m)) & \{\text{por definición de } \delta\} \\
\quad = \bigsqcup_i \alpha_{t_i}(e_i) & \{\text{por definición de } \widehat{\sqcup}\} \\
\quad = \alpha_t(e_1,\ldots,e_m) & \{\text{por definición de } \alpha_t\}.
\end{array}
$$

- $t = t_1 \to t_2$ o $t = Process\ t_1\ t_2$. Sea $f \in D_{2t}$. Entonces:

$$
\begin{array}{ll}
\widehat{\sqcup}(\delta_t(f)) & \\
\quad = \widehat{\sqcup}(\delta_{t_2}(f(\gamma_{t_1}(d)))) & \{\text{por definición de } \delta_t\} \\
\quad = \alpha_{t_2}(f(\gamma_{t_1}(d))) & \{\text{por h.i.}\} \\
\quad = \alpha_t(f) & \{\text{por definición de } \alpha_t\}.
\end{array}
$$

- $t = \forall \beta . t'$. En este caso $\delta_t = \delta'_t$ y $\alpha_t = \alpha_{t'}$, luego se cumple trivialmente por h.i.

□

Una consecuencia directa de la Proposición 34 es que δ_t es estricta, ya que en un par inmersión-clausura, la clausura siempre es estricta.

Proposición 36 *Para cada tipo t: $\delta_t(\perp_{2t}) = d_t$.*

La Proposición 34 nos dice que $\eta_t \cdot \delta_t$ es un operador de ensanchamiento sobre D_{2t}. Esta composición se usará a menudo más adelante, por lo que definimos $\nabla_t = \eta_t \cdot \delta_t$. La llamaremos *función de elevación* (para diferenciarla de $\mathcal{W}_t$). Su definición desplegada se muestra en la Figura 5.31.

También nos dice que para cada tipo t, el rango de ∇_t es isomorfo a D_{1t}: $\nabla_t(D_{2t}) \simeq D_{1t}$. Esto implica que el rango de ∇_t es un subdominio de D_{2t} donde se ha perdido la información adicional proporcionada por el segundo y tercer análisis. Por ejemplo, aún se dispone de tuplas anidadas, pero estas se mantienen de una forma ficticia, pues han sido aplanadas y desaplanadas. Por ello, todas las tuplas internas estarán formadas sólo por valores ns o por valores ds. Por ejemplo, $((n,d),d,(d,d))$ y $((d,n),d,(d,d))$ son transformadas por ∇_t en $((n,n),d,(d,d))$. También tenemos funciones abstractas, pero solamente algunas de ellas: aquellas que pueden representarse con un valor abstracto en los dominios del primer análisis. Estas son las funciones tales que devuelven el mismo resultado para todos los valores por debajo de $\gamma_{t_1}(d)$: el obtenido para $\gamma_{t_1}(d)$, y para el resto de valores devuelven el supremo del correspondiente dominio, $\gamma_{t_2}(n)$.

Hemos visto que $\gamma_t \cdot \alpha_t \sqsupseteq id_{D_{2t}}$ y $\nabla_t \sqsupseteq id_{D_{2t}}$. Pero, ¿cómo se relacionan? La siguiente proposición afirma que ∇_t es menor (es decir, mejor) que $\gamma_t \cdot \alpha_t$.

Proposición 37 *Para cada tipo t:* $\nabla_t \sqsubseteq \gamma_t \cdot \alpha_t$.

Demostración 11 (Proposición 37) Se demostrará por inducción estructural sobre t aplicando las Proposiciones 25(a), 25(d) y 26:

- $t = K$ o $t = T\ t_1 \dots t_m$ o $t = \beta$. Estos casos son triviales, ya que $\nabla_t = \alpha_t = \gamma_t = id_{Basic}$.

- $t = (t_1, \dots, t_m)$. Sean $e_i \in D_{2t_i}$ para cada $i = 1, \dots, m$. Entonces:

$$\begin{array}{l} \nabla_t(e_1,\dots,e_m) \\ \quad = (\gamma_{t_1}(\alpha_{t_1}(e_1)),\dots,\gamma_{t_m}(\alpha_{t_m}(e_m))) \\ \qquad\quad \{\text{por definición de } \nabla_t\} \\ \quad \sqsubseteq (\gamma_{t_1}(\bigsqcup_i \alpha_{t_i}(e_i)),\dots,\gamma_{t_m}(\bigsqcup_i \alpha_{t_i}(e_i))) \\ \qquad\quad \{\text{pues } \alpha_{t_j}(e_j) \sqsubseteq \bigsqcup_i \alpha_{t_i}(e_i) \text{ para cada } j=1,\dots,m \\ \qquad\quad \text{y monotonía de } \gamma_{t_i}\} \\ \quad = \gamma_t(\alpha_t(e_1,\dots,e_m)) \\ \qquad\quad \{\text{por definición de } \alpha_t \text{ y } \gamma_t\}. \end{array}$$

- $t = t_1 \to t_2$ o $t = \mathit{Process}\ t_1\ t_2$. Dada $f \in [D_{2t_1} \to D_{2t_2}]$, por definición de ∇_t,

$$\nabla_t(f) = \lambda z. \begin{cases} \nabla_{t_2}(f(\gamma_{t_1}(d))) \text{ si } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ e.o.c.} \end{cases},$$

  y por definición de γ_t y α_t,

$$\gamma_t(\alpha_t(f)) = \lambda z. \begin{cases} \gamma_{t_2}(\alpha_{t_1}(z)) \text{ si } \alpha_{t_2}(f(\gamma_{t_1}(d))) = d \\ \gamma_{t_2}(n) \ \ \text{e.o.c.} \end{cases}$$

  Distinguimos dos casos:

  - $\alpha_{t_2}(f(\gamma_{t_1}(d))) = n$. En este caso $\gamma_t(\alpha_t(f)) = \lambda z.\gamma_{t_2}(n)$. Sea $e \in D_{2t_1}$. Se cumple que $(\gamma_t(\alpha_t(f)))(e) = \gamma_{t_2}(n)$. Luego, por la Proposición 25(d) tenemos que $\gamma_{t_2}(n) \sqsupseteq (\nabla_t(f))(e)$ ya que $(\nabla_t(f))(e) \in D_{2t_2}$.

– $\alpha_{t_2}(f(\gamma_{t_1}(d))) = d$. Ahora $\gamma_t(\alpha_t(f)) = \lambda z.\gamma_{t_2}(\alpha_{t_1}(z))$. Sea $e \in D_{2t_1}$. Distinguimos de nuevo dos casos: $e \sqsubseteq \gamma_{t_1}(d)$ o no.
Si $e \sqsubseteq \gamma_{t_1}(d)$ entonces $(\nabla_t(f))(e) = \nabla_{t_2}(f(\gamma_{t_1}(d)))$. Por la Proposición 26, $\alpha_{t_1}(e) = d$. Luego lo que en realidad queremos demostrar es que $\nabla_{t_2}(f(\gamma_{t_1}(d))) \sqsubseteq \gamma_{t_2}(d)$. Por h.i. sabemos que $\nabla_{t_2} \sqsubseteq \gamma_{t_2} \cdot \alpha_{t_2}$, luego:

$$\nabla_{t_2}(f(\gamma_{t_1}(d))) \sqsubseteq \gamma_{t_2}(\alpha_{t_2}(f(\gamma_{t_1}(d)))) = \gamma_{t_2}(d).$$

En caso de que $e \not\sqsubseteq \gamma_{t_1}(d)$, entonces $(\nabla_t(f))(e) = \gamma_{t_2}(n)$. Por la Proposición 26, $\alpha_{t_1}(e) = n$, luego $\gamma_{t_2}(\alpha_{t_1}(e)) = \gamma_{t_2}(n)$, cumpliéndose la igualdad deseada.

- $t = \forall\beta.t'$. Como $\alpha_t = \alpha_{t'}$, $\gamma_t = \gamma_{t'}$ y $\nabla_t = \nabla_{t'}$ por definición, se cumple trivialmente por h.i.

□

El Teorema 43 establecerá la corrección del primer análisis con respecto al tercero. Antes veremos algunas propiedades necesarias para demostrarlo.

La siguiente proposición relaciona la función de adaptación del primer análisis con la función de concreción γ_t. Resultan ser iguales a través de la aplicación de δ_t.

Proposición 38 *Para cada tipo t:* $\forall b \in Basic.\ b_t = (\delta_t \cdot \gamma_t)(b)$.

Demostración 12 (Proposición 38) Por inducción estructural sobre t aplicando la Proposición 25(b):

- $t = K$ o $t = T\ t_1 \dots t_m$ o $t = \beta$. Estos casos son triviales, ya que $b_t = b$ y $\delta_t = \gamma_t = id_{Basic}$.

- $t = (t_1, \dots, t_m)$. En este caso tenemos:

$$\begin{array}{ll} \delta_t(\gamma_t(b)) & \\ \quad = \delta_t(\gamma_{t_1}(b), \dots, \gamma_{t_m}(b)) & \{\text{por definición de } \gamma_t\} \\ \quad = (\alpha_{t_1}(\gamma_{t_1}(b)), \dots, \alpha_{t_m}(\gamma_{t_m}(b))) & \{\text{por definición de } \delta_t\} \\ \quad = (b, \dots, b) & \{\text{por la Proposición 25(b)}\} \\ \quad = b_t & \{\text{por definición de } b_t\}. \end{array}$$

- $t = t_1 \to t_2$ o $t = Process\ t_1\ t_2$. Distinguimos dos casos: $b = d$ y $b = n$.

  – $b = d$. En este caso:

$$\begin{array}{ll} \delta_t(\gamma_t(d)) & \\ \quad = \delta_t(\lambda z.\gamma_{t_2}(\alpha_{t_1}(z))) & \{\text{por definición de } \gamma_t\} \\ \quad = \delta_{t_2}(\gamma_{t_2}(\alpha_{t_1}(\gamma_{t_1}(d)))) & \{\text{por definición de } \delta_t\} \\ \quad = \delta_{t_2}(\gamma_{t_2}(d)) & \{\text{por la Proposición 25(b)}\} \\ \quad = d_{t_2} & \{\text{por h.i.}\} \\ \quad = d_t & \{\text{por definición de } b_t\}. \end{array}$$

– $b = n$. Ahora se cumple

$$\begin{array}{rcll} \delta_t(\gamma_t(n)) & = & \delta_t(\lambda z.\gamma_{t_2}(n)) & \{\text{por definición de } \gamma_t\} \\ & = & \delta_{t_2}(\gamma_{t_2}(n)) & \{\text{por definición de } \delta_t\} \\ & = & n_{t_2} & \{\text{por h.i.}\} \\ & = & n_t & \{\text{por definición de } b_t\}. \end{array}$$

- $t = \forall\beta.t'$. En este caso $b_t = b_{t'}$, $\delta_t = \delta'_t$ y $\gamma_t = \gamma_{t'}$, luego se cumple trivialmente por h.i.

□

La siguiente proposición relaciona la función de elevación ∇_t con las funciones de abstracción y concreción α_t y γ_t. La idea básica es que una vez hemos subido en el dominio con ∇_t, las funciones α_t y γ_t se comportan de la misma forma en el subdominio de D_{2t} rango de ∇_t, que en el dominio completo. Esto significa que en realidad α_t y γ_t se mueven dentro de dicho subdominio. En la Figura 5.28 se representó este hecho apareciendo dos dominios D_{2t} rodeados por una línea discontínua. El primero de ellos corresponde al rango de ∇_t, a partir del cual se definen tanto α_t como γ_t.

Proposición 39 *Para cada tipo t:*

(a) $\alpha_t \cdot \nabla_t = \alpha_t$,

(b) $\nabla_t \cdot \gamma_t = \gamma_t$.

Demostración 13 (Proposición 39) El punto 39(a) se demostrará por inducción estructural sobre t aplicando la Proposición 25(b). El punto 39(b) se demostrará por inducción estructural sobre t aplicando las Proposiciones 25(b) y 26. Demostremos primero el punto 39(a):

- $t = K$ o $t = T\ t_1 \ldots t_m$ o $t = \beta$. Estos casos son triviales, ya que $\nabla_t = \alpha_t = id_{Basic}$.

- $t = (t_1, \ldots, t_m)$. Sean $e_i \in D_{2t_i}$ para cada $i = 1, \ldots, m$. Entonces:

$$\begin{array}{ll} \alpha_t(\nabla_t(e_1, \ldots, e_m)) & \\ \quad = \alpha_t(\gamma_{t_1}(\alpha_{t_1}(e_1)), \ldots, \gamma_{t_m}(\alpha_{t_m}(e_m))) & \{\text{por definición de } \nabla_t\} \\ \quad = \bigsqcup_i \alpha_{t_i}(\gamma_{t_i}(\alpha_{t_i}(e_i))) & \{\text{por definición de } \alpha_t\} \\ \quad = \bigsqcup_i \alpha_{t_i}(e_i) & \{\text{por la Proposición 25(b)}\} \\ \quad = \alpha_t(e_1, \ldots, e_m) & \{\text{por definición de } \alpha_t\}. \end{array}$$

- $t = t_1 \to t_2$ o $t = \mathit{Process}\ t_1\ t_2$. Sea $f \in [D_{2t_1} \to D_{2t_2}]$. Entonces:

$$\begin{array}{ll} \alpha_t(\nabla_t(f)) & \\ \quad = \alpha_{t_2}(\nabla_{t_2}(f(\gamma_{t_1}(d)))) & \{\text{por definición de } \alpha_t \text{ y } \nabla_t\} \\ \quad = \alpha_{t_2}(f(\gamma_{t_1}(d))) & \{\text{por h.i.}\} \\ \quad = \alpha_t(f) & \{\text{por definición de } \alpha_t\}. \end{array}$$

- $t = \forall\beta.t'$. Se cumple trivialmente por h.i., pues $\nabla_t = \nabla_{t'}$ y $\alpha_t = \alpha_{t'}$.

Demostremos ahora el punto 39(b):

- $t = K$ o $t = T\ t_1 \dots t_m$ o $t = \beta$. Estos casos son triviales, ya que $\nabla_t = \gamma_t = id_{Basic}$.

- $t = (t_1, \dots, t_m)$. Sea $b \in Basic$. En este caso:

$$\begin{array}{ll} \nabla_t(\gamma_t(b)) & \\ = \nabla_t(\gamma_{t_1}(b), \dots, \gamma_{t_m}(b)) & \{\text{por definición de } \gamma_t\} \\ = (\gamma_{t_1}(\alpha_{t_1}(\gamma_{t_1}(b))), \dots, \gamma_{t_m}(\alpha_{t_m}(\gamma_{t_m}(b)))) & \{\text{por definición de } \nabla_t\} \\ = (\gamma_{t_1}(b), \dots, \gamma_{t_m}(b)) & \{\text{por la Proposición 25(b)}\} \\ = \gamma_t(b) & \{\text{por definición de } \gamma_t\}. \end{array}$$

- $t = t_1 \to t_2$ o $t = Process\ t_1\ t_2$. Sea $b \in Basic$. Distinguimos dos casos: $b = d$ o $b = n$.

  Veamos primero el caso $b = d$. Por un lado tenemos que:

$$\begin{array}{l} \nabla_t(\gamma_t(d)) \\ = \nabla_t(\lambda z.\gamma_{t_2}(\alpha_{t_1}(z))) \\ \qquad \{\text{por definición de } \gamma\} \\ = \lambda z. \begin{cases} \nabla_{t_2}(\gamma_{t_2}(\alpha_{t_1}(\gamma_{t_1}(d)))) \text{ si } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ e.o.c.} \end{cases} \\ \qquad \{\text{por definición de } \nabla_t\} \\ = \lambda z. \begin{cases} \gamma_{t_2}(d) \text{ si } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ e.o.c.} \end{cases} \\ \qquad \{\text{por h.i. y Proposición 25(b)}\}. \end{array}$$

  Por otro lado tenemos que $\gamma_t(d) = \lambda z.\gamma_{t_2}(\alpha_{t_1}(z))$. Tenemos que demostrar que para cada $e \in D_{2t_1}$, $(\nabla_t(\gamma_t(d)))(e) = (\gamma_t(d))(e)$. Distinguimos dos casos:

  - $e \sqsubseteq \gamma_{t_1}(d)$. En este caso

$$(\nabla_t(\gamma_t(d)))(e) = \gamma_{t_2}(d).$$

    Por el Lema 26, si $e \sqsubseteq \gamma_{t_1}(d)$ entonces $\alpha_{t_1}(e) = d$, luego

$$(\gamma_t(d))(e) = \gamma_{t_2}(d).$$

  - En caso contrario, por la Proposición 26, $\alpha_{t_1}(e) = n$, y se cumple:

$$\begin{array}{rcl} (\nabla_t(\gamma_t(d)))(e) & = & \gamma_{t_2}(n) \\ & = & \gamma_{t_2}(\alpha_{t_1}(e)) \\ & = & (\gamma_t(d))(e). \end{array}$$

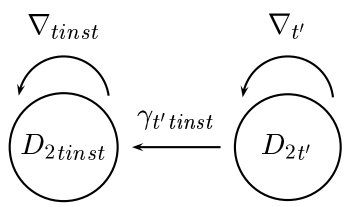


Figura 5.32: Proposición 40(a)

Si $b = n$:

$$\begin{array}{ll}
\nabla_t(\gamma_t(n)) & \\
= \nabla_t(\lambda z.\gamma_{t_2}(n)) & \{\text{por definición de } \gamma_t\} \\
= \lambda z. \left\{ \begin{array}{l} \nabla_{t_2}(\gamma_{t_2}(n)) \text{ si } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ e.o.c.} \end{array} \right. & \{\text{por definición de } \nabla_t\} \\
= \lambda z.\gamma_{t_2}(n) & \{\text{por h.i.}\} \\
= \gamma_t(n) & \{\text{por definición de } \gamma_t\}.
\end{array}$$

- $t = \forall \beta.t'$. Se cumple trivialmente por h.i. pues $\nabla_t = \nabla_{t'}$ y $\gamma_t = \gamma_{t'}$.

□

La siguiente proposición añade algunos resultados correspondientes al estudio de las funciones polimórficas. El primer punto nos dice que si adaptamos al tipo *tinst* un valor abstracto perteneciente a $D_{2t'}$ y después elevamos el resultado, obtenemos el mismo resultado que si primero elevamos y después adaptamos el resultado. Ello se muestra mediante un diagrama en la Figura 5.32.

El segundo punto nos dice que la adaptación de un valor abstracto de $D_{1t'}$ para obtener una aproximación al valor abstracto de un ejemplar *tinst*, es básicamente igual que la adaptación realizada por $\gamma_{t'\,tinst}$ en los dominios del segundo y tercer análisis.

Proposición 40 *Dados dos tipos t, t' y una variable de tipo β:*

(a) $\nabla_{tinst} \cdot \gamma_{t'\,tinst} = \gamma_{t'\,tinst} \cdot \nabla_{t'}$,

(b) $\forall a \in D_{1t'}.\ a_{tinst} = \delta_{tinst}(\gamma_{t'\,tinst}(\eta_{t'}(a)))$.

Demostración 14 (Proposición 40) El punto 40(a) se demostrará por inducción estructural sobre t aplicando las Proposiciones 25(a), 25(b), 26, 28(a), 29, y 39(b). El punto 40(b) se demostrará por inducción estructural sobre t aplicando las Proposiciones 25(b), 29(d), 29(a) y 38.

Demostremos primero el punto 40(a):

- $t' = K$ o $t' = T\ t_1 \dots t_m$ o $t' = \beta'$ $(\neq \beta)$. En estos casos $D_{2\,tinst} = D_{2t'} = \mathit{Basic}$ y $\nabla_{tinst} = \gamma_{t'\,tinst} = \nabla_{t'} = id_{Basic}$, luego se cumple trivialmente.

- $t' = (t_1, \dots, t_m)$. En este caso $tinst = (tinst_1, \dots, tinst_m)$ donde $tinst_i = t_i[\beta := t]$, para cada $i = 1, \dots, m$. Por h.i. se cumple que, para cada $i = 1, \dots, m$, $\nabla_{tinst_i} \cdot \gamma_{t_i\,tinst_i} = \gamma_{t_i\,tinst_i} \cdot \nabla_{t_i}$. Sean $e_i \in D_{2t_i}$ con $i = 1, \dots, m$. Entonces:

$$
\begin{array}{l}
\nabla_{tinst}(\gamma_{t'\,tinst}(e_1, \dots, e_m)) \\
= \nabla_{tinst}(\gamma_{t_1\,tinst_1}(e_1), \dots, \gamma_{t_m\,tinst_m}(e_m)) \\
\qquad \{\text{por definición de } \gamma_{t'\,tinst}\} \\
= (\gamma_{tinst_1}(\alpha_{tinst_1}(\gamma_{t_1\,tinst_1}(e_1))), \dots, \\
\qquad\qquad \gamma_{tinst_m}(\alpha_{tinst_m}(\gamma_{t_m\,tinst_m}(e_m)))) \\
\qquad \{\text{por definición de } \nabla_t\} \\
= (\gamma_{tinst_1}(\alpha_{t_1}(e_1)), \dots, \gamma_{tinst_m}(\alpha_{t_m}(e_m))) \\
\qquad \{\text{por la Proposición 29(c)}\} \\
= (\gamma_{t_1\,tinst_1}(\gamma_{t_1}(\alpha_{t_1}(e_1))), \dots, \gamma_{t_m\,tinst_m}(\gamma_{t_m}(\alpha_{t_m}(e_m)))) \\
\qquad \{\text{por la Proposición 29(a)}\} \\
= \gamma_{t'\,tinst}(\gamma_{t_1}(\alpha_{t_1}(e_1)), \dots, \gamma_{t_m}(\alpha_{t_m}(e_1))) \\
\qquad \{\text{por definición de } \gamma_{t'\,tinst}\} \\
= \gamma_{t'\,tinst}(\nabla_{t'}(e_1, \dots, e_m)) \\
\qquad \{\text{por definición de } \nabla_t\}.
\end{array}
$$

- $t' = t_1 \to t_2$ o $t' = \mathit{Process}\ t_1\ t_2$. En este caso $tinst = tinst_1 \to tinst_2$, o bien $tinst = \mathit{Process}\ tinst_1\ tinst_2$, donde $tinst_i = t_i[\beta := t]$, para $i = 1, 2$. Sea $f \in [D_{2t_1} \to D_{2t_2}]$. Por un lado:

$$
\begin{array}{l}
\nabla_{tinst}(\gamma_{t'\,tinst}(f)) \\
= \nabla_{tinst}(\lambda z.\gamma_{t_2\,tinst_2}(f(\alpha_{tinst_1\,t_1}(z)))) \\
\qquad \{\text{por definición de } \gamma_{t'\,tinst}\} \\[1ex]
= \lambda z. \left\{ \begin{array}{l} \nabla_{tinst_2}(\gamma_{t_2 tinst_2}(f(\alpha_{tinst_1\,t_1}(\gamma_{tinst_1}(d))))) \\ \qquad\qquad\qquad \text{si } z \sqsubseteq \gamma_{tinst_1}(d) \\ \gamma_{tinst_2}(n) \text{ e.o.c.} \end{array} \right. \\
\qquad \{\text{por definición de } \nabla_t\}.
\end{array}
$$

Por otra parte:

$$
\begin{aligned}
&\gamma_{t'\,tinst}(\nabla_{t'}(f))\\
&= \gamma_{t'\,tinst}\left(\lambda z. \begin{cases} \nabla_{t_2}(f(\gamma_{t_1}(d))) \text{ si } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ e.o.c.} \end{cases}\right)\\
&\qquad \{\text{por definición de } \nabla_t\}\\
&= \lambda z. \begin{cases} \gamma_{t_2\,tinst_2}(\nabla_{t_2}(f(\gamma_{t_1}(d)))) \\ \qquad\qquad \text{si } \alpha_{tinst_1\,t_1}(z) \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2\,tinst_2}(\gamma_{t_2}(n)) \text{ e.o.c.} \end{cases}\\
&\qquad \{\text{por definición de } \gamma_{t'\,tinst}\}.
\end{aligned}
$$

Deseamos demostrar la igualdad de estas dos funciones. Sea $e \in D_{2\,tinst_1}$. Distinguimos dos casos:

- $e \sqsubseteq \gamma_{tinst_1}(d)$. Entonces por monotonía de $\alpha_{tinst\,t'}$ (Proposición 28(a)), $\alpha_{tinst_1\,t_1}(e) \sqsubseteq \alpha_{tinst_1\,t_1}(\gamma_{tinst_1}(d))$ y por la Proposición 29(d) $\alpha_{tinst_1\,t_1}(\gamma_{tinst_1}(d)) = \gamma_{t_1}(d)$, luego

$$\alpha_{tinst_1\,t_1}(e) \sqsubseteq \gamma_{t_1}(d).$$

  Así en este caso:

$$
\begin{aligned}
&(\nabla_{tinst}(\gamma_{t'\,tinst}(f)))(e)\\
&= \nabla_{tinst_2}(\gamma_{t_2\,tinst_2}(f(\alpha_{tinst_1\,t_1}(\gamma_{tinst_1}(d))))),
\end{aligned}
$$

  y

$$(\gamma_{t'\,tinst}(\nabla_{t'}(f)))(e) = \gamma_{t_2\,tinst_2}(\nabla_{t_2}(f(\gamma_{t_1}(d)))).$$

  Por tanto:

$$
\begin{aligned}
&(\nabla_{tinst}(\gamma_{t'\,tinst}(f)))(e)\\
&= \nabla_{tinst_2}(\gamma_{t_2\,tinst_2}(f(\alpha_{tinst_1\,t_1}(\gamma_{tinst_1}(d)))))\\
&= \gamma_{t_2\,tinst_2}(\nabla_{t_2}(f(\alpha_{tinst_1\,t_1}(\gamma_{tinst_1}(d)))))\\
&\qquad \{\text{por h.i.}\}\\
&= \gamma_{t_2\,tinst_2}(\nabla_{t_2}(f(\gamma_{t_1}(d)))))\\
&\qquad \{\text{por la Proposición 29(d)}\}\\
&= \gamma_{t'\,tinst}(\nabla_{t'}(f)))(e).
\end{aligned}
$$

- $e \not\sqsubseteq \gamma_{tinst_1}(d)$. En este caso por la Proposición 26, $\alpha_{tinst_1}(e) = n$. Adicionalmente, tenemos que $(\nabla_{tinst}(\gamma_{t'\,tinst}(f)))(e) = \gamma_{tinst_2}(n)$, y además se cumple que $\alpha_{tinst_1\,t_1}(e) \not\sqsubseteq \gamma_{t_1}(d)$. Esto es cierto ya que, en caso contrario, por la Proposición 29(b), $\alpha_{tinst_1}(e) = \alpha_{t_1}(\alpha_{tinst_1\,t_1}(e))$, se cumpliría que $\alpha_{tinst_1}(e) \sqsubseteq \alpha_{t_1}(\gamma_{t_1}(d))$ por monotonía de α_t (Proposición 25(a)), lo que significaría (por la Proposición 25(b)) que $\alpha_{tinst_1}(e) = d$. Esto es imposible pues acabamos de ver que es igual a n. Así $(\gamma_{t'\,tinst}(\nabla_{t'}(f)))(e) = \gamma_{t_2\,tinst_2}(\gamma_{t_2}(n))$, y entonces

$$\begin{array}{ll}
(\nabla_{tinst}(\gamma_{t'\,tinst}(f)))(e) & \\
= \gamma_{tinst_2}(n) & \\
= \gamma_{t_2\,tinst_2}(\gamma_{t_2}(n)) & \{\text{por la Proposición 29(a)}\} \\
= (\gamma_{t'\,tinst}(\nabla_{t'}(f)))(e). &
\end{array}$$

- $t' = \beta$. En este caso:

$$\begin{array}{ll}
\nabla_{tinst} \cdot \gamma_{t'\,tinst} & \\
= \nabla_t \cdot \gamma_t & \{\text{por definición de } \nabla_t \text{ y } \gamma_t\} \\
= \gamma_t & \{\text{por la Proposición 39(b)}\} \\
= \gamma_{t'\,tinst} \cdot \nabla_{t'} & \{\text{por definición de } \gamma_{t'\,tinst} \text{ y } \nabla_t\}.
\end{array}$$

- $t' = \forall\beta'.t_1$. Se cumple directamente por h.i., pues $\nabla_{tinst} = \nabla_{tinst_1}$, $\gamma_{t'\,tinst} = \gamma_{t_1\,tinst_1}$ y $\nabla_{t'} = \nabla_{t_1}$.

Demostremos ahora el punto 40(b):

- $t' = K$ o $t' = T\ t_1 \ldots t_m$ o $t' = \beta'$ ($\neq \beta$). En este caso $D1_{tinst} = D1_{t'} = Basic$. Sea $b \in Basic$. Entonces $b_{tinst} = b$. Por otra parte $\delta_{tinst} = \gamma_{t'\,tinst} = \eta_{t'} = id_{Basic}$, por lo que se cumple trivialmente.

- $t' = (t_1, \ldots, t_m)$. En este caso $tinst = (tinst_1, \ldots, tinst_m)$ donde $tinst_i = t_i[\beta := t]$, para $i = 1, \ldots, m$. Sean $b_i \in Basic$ con $i = 1, \ldots, m$. Entonces

$$\begin{array}{l}
\delta_{tinst}(\gamma_{t'\,tinst}(\eta_{t'}(b_1, \ldots, b_m))) \\
= \delta_{tinst}(\gamma_{t'\,tinst}(\gamma_{t_1}(b_1), \ldots, \gamma_{t_m}(b_m))) \\
\qquad \{\text{por definición de } \eta_t\} \\
= \delta_{tinst}(\gamma_{t_1\,tinst_1}(\gamma_{t_1}(b_1)), \ldots, \gamma_{t_1\,tinst_1}(\gamma_{t_m}(b_m))) \\
\qquad \{\text{por definición de } \gamma_{t'\,tinst}\} \\
= \delta_{tinst}(\gamma_{tinst_1}(b_1), \ldots, \gamma_{tinst_m}(b_m)) \\
\qquad \{\text{por la Proposición 29(a)}\} \\
= (\alpha_{tinst_1}(\gamma_{tinst_1}(b_1)), \ldots, \alpha_{tinst_m}(\gamma_{tinst_m}(b_m))) \\
\qquad \{\text{por definición de } \delta_t\} \\
= (b_1, \ldots, b_m) \\
\qquad \{\text{por la Proposición 25(b)}\} \\
= (b_1, \ldots, b_m)_{tinst} \\
\qquad \{\text{por definición de la adaptación}\}.
\end{array}$$

- $t' = t_1 \to t_2$ o $t' = Process\ t_1\ t_2$. En este caso $tinst = tinst_1 \to tinst_2$, o bien $tinst = Process\ tinst_1\ tinst_2$, donde $tinst_i = t_i[\beta := t]$, para

$i = 1, 2$. Sea $a \in D_{1t'} = D_{1t_2}$. Ahora tenemos:

$$
\begin{array}{l}
\delta_{tinst}(\gamma_{t'\,tinst}(\eta_{t'}(a))) \\
= \delta_{tinst}\left(\gamma_{t'\,tinst}\left(\lambda z. \left\{ \begin{array}{l} \eta_{t_2}(a) \text{ si } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ e.o.c.} \end{array} \right.\right)\right) \\
\quad \{\text{por definición de } \eta_t\} \\
\\
= \delta_{tinst}\left(\lambda z. \left\{ \begin{array}{l} \gamma_{t_2 tinst_2}(\eta_{t_2}(a)) \\ \qquad \text{si } \alpha_{tinst_1\,t_1}(z) \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2 tinst_2}(\gamma_{t_2}(n)) \text{ e.o.c.} \end{array} \right.\right) \\
\quad \{\text{por definición de } \gamma_{t'\,tinst}\} \\
\\
= \delta_{tinst_2}(\gamma_{t_2 tinst_2}(\eta_{t_2}(a))) \\
\quad \{\text{por def. de } \delta_t \text{ y Proposición 29(d)}\} \\
= a_{tinst_2} \\
\quad \{\text{por h.i.}\} \\
= a_{tinst} \\
\quad \{\text{por definición de la adaptación}\}.
\end{array}
$$

- $t' = \beta$. Ahora $D_{1t'} = Basic$ y $D_{1\,tinst} = D_{1t}$. Sea $b \in Basic$. Entonces:

$$
\begin{array}{ll}
\delta_{tinst}(\gamma_{t'\,tinst}(\eta_{t'}(b))) & \\
= \delta_t(\gamma_t(b)) & \{\text{por definición de } \delta_t,\ \gamma_{t'\,tinst} \text{ y } \eta_t\} \\
= b_t & \{\text{por la Proposición 38}\} \\
= b_{tinst}. &
\end{array}
$$

- $t' = \forall\beta'.t_1$. Se cumple directamente por h.i., pues $\delta_{tinst} = \delta_{tinst_1}$, $\gamma_{t'\,tinst} = \gamma_{t_1\,tinst_1}$, $\eta_{tinst} = \eta_{tinst_1}$ y $b_{tinst} = b_{tinst_1}$.

□

Una propiedad muy importante y útil para demostrar la corrección es la propiedad de *semihomomorfismo* de δ_t con respecto a la aplicación de una función. Si bien en el primer análisis no hay funciones, la propiedad se cumple con respecto a la pseudoaplicación usada en tales dominios, es decir, a la forma en que se interpreta la aplicación de una función: $f(x) = (\widehat{\sqcup} x) \sqcup f$ (ver Figura 5.6 en la página 127).

Proposición 41 *Para* $f \in [D_{2t_1} \to D_{2t_2}]$, $e \in D_{2t_1}$ *se tiene:* $\delta_{t_2}(f(e)) \sqsubseteq (\widehat{\sqcup}\delta_{t_1}(e)) \sqcup \delta_{t_1 \to t_2}(f)$.

Demostración 15 (Proposición 41) Utilizaremos las Proposiciones 26, 34(a) y 35. Sean $f \in [D_{2t_1} \to D_{2t_2}]$ y $e \in D_{2t_1}$. Por definición $\delta_{t_1 \to t_2}(f) = \delta_{t_2}(f(\gamma_{t_1}(d)))$, luego tenemos que demostrar que $\delta_{t_2}(f(e)) \sqsubseteq (\widehat{\sqcup}\delta_{t_1}(e)) \sqcup \delta_{t_2}(f(\gamma_{t_1}(d)))$. Distinguimos dos casos:

- $e \sqsubseteq \gamma_{t_1}(d)$. Como f es monótona, $f(e) \sqsubseteq f(\gamma_{t_1}(d))$. Por la Proposición 34(a), δ_{t_2} es monótona, luego

$$\delta_{t_2}(f(e)) \sqsubseteq \delta_{t_2}(f(\gamma_{t_1}(d))),$$

  que es trivialmente menor o igual que

$$(\widehat{\sqcup}\delta_{t_1}(e)) \sqcup \delta_{t_2}(f(\gamma_{t_1}(d))).$$

- $e \not\sqsubseteq \gamma_{t_1}(d)$. En este caso, por la Proposición 26, $\alpha_{t_1}(e) = n$, luego por la Proposición 35, $\widehat{\sqcup}(\delta_{t_1}(e)) = n$.

  Es obvio que si $b \in Basic$, entonces $n \sqcup b \sqsupseteq b'$ para cualquier $b' \in Basic$. Y si $a \in Basic^m$, entonces $n \sqcup a \sqsupseteq a'$, para cualquier $a' \in Basic^m$. Es decir, la mínima cota superior de cualquier valor y n nos conduce al supremo del dominio correspondiente. Luego

$$(\widehat{\sqcup}\delta_{t_1}(e)) \sqcup \delta_{t_1 \to t_2}(f) = n \sqcup \delta_{t_1 \to t_2}(f) \sqsupseteq \delta_{t_2}(f(e)).$$

□

Finalmente, para demostrar el Corolario 44 del Teorema 43 necesitaremos el siguiente lema.

Lema 42 *Para todo tipo t: $\eta_t(d_t) = \gamma_t(d)$.*

Demostración 16 (Lemma 42) Es una consecuencia directa de las proposiciones 38 y 39(b). Por la Proposición 38 sabemos que $b_t = \delta_t(\gamma_t(b))$. Luego:

$$\begin{array}{lcll} \eta_t(b_t) & = & \eta_t(\delta_t(\gamma_t(b))) & \\ & = & \nabla_t(\gamma_t(b)) & \{\text{por definición de } \nabla_t\} \\ & = & \gamma_t(b) & \{\text{por la Proposición 39(b)}\}. \end{array}$$

□

El siguiente teorema, cuya demostración hemos ido preparando por medio de las proposiciones anteriores, establece que el primer análisis es una aproximación segura (superior) a la transformación mediante δ_t de aquellas variantes del tercer análisis $[\![\cdot]\!]_3^{wop}$ tales que $\delta_t \cdot wop_t = \delta_t$.

Teorema 43 *Sea $\mathcal{W}'_t$ un operador de ensanchamiento para cada tipo t, tal que $\delta_t \cdot \mathcal{W}'_t = \delta_t$. Si para cada variable $v :: t_v$, $\rho_1(v) \sqsupseteq \delta_{t_v}(\rho_3(v))$ entonces: $\forall e :: t_e$. $[\![e]\!]_1\ \rho_1 \sqsupseteq \delta_{t_e}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)$, o equivalentemente, en virtud de la Proposición 34: $\forall e :: t_e$. $\eta_{t_e}([\![e]\!]_1\ \rho_1) \sqsupseteq \nabla_{t_e}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)$.*

Demostración 17 (Teorema 43) El teorema se demostrará por inducción estructural sobre e aplicando las Proposiciones 25(a), 25(c), 28(a), 34(a), 35, 36, 38, 40 y 41:

- $v :: t$. En este caso:

$$\begin{array}{rcll} [\![v]\!]\ \rho_1 & = & \rho_1(v) & \{\text{por definición de } [\![\cdot]\!]_1\} \\ & \sqsupseteq & \delta_t(\rho_3(v)) & \{\text{por hipótesis}\} \\ & = & \delta_t([\![v]\!]_3^{\mathcal{W}'}\ \rho_3) & \{\text{por definición de } [\![\cdot]\!]_3^{\mathcal{W}'}\}. \end{array}$$

- $k :: K$.

$$\begin{array}{rcll} [\![k]\!]_1\ \rho_1 & = & d & \{\text{por definición de } [\![\cdot]\!]_1\} \\ & = & [\![k]\!]_3^{\mathcal{W}'}\ \rho_3 & \{\text{por definición de } [\![\cdot]\!]_3^{\mathcal{W}'}\} \\ & = & \delta_K([\![k]\!]_3^{\mathcal{W}'}\ \rho_3) & \{\text{por definición de } \delta_t\}. \end{array}$$

- $C\ x_1 \ldots x_m :: T\ t'_1 \ldots t'_k$, donde $x_i :: t_i$. Por h.i. sabemos que para cada $i = 1, \ldots, m$, se cumple

$$[\![x_i]\!]_1\ \rho_1 \sqsupseteq \delta_{t_i}([\![x_i]\!]_3^{\mathcal{W}'}\ \rho_3).$$

Puesto que el operador $\widehat{\sqcup}$ es monótono, esto implica que para cada $i = 1, \ldots, m$ se tiene

$$\widehat{\sqcup}([\![x_i]\!]_1\ \rho_1) \sqsupseteq \widehat{\sqcup}(\delta_{t_i}([\![x_i]\!]_3^{\mathcal{W}'}\ \rho_3)).$$

Por la Proposición 35 se cumple que $\widehat{\sqcup}(\delta_{t_i}([\![x_i]\!]_3^{\mathcal{W}'}\ \rho_3)) = \alpha_{t_i}([\![x_i]\!]_3^{\mathcal{W}'}\ \rho_3)$, luego

$$\widehat{\sqcup}([\![x_i]\!]_1\ \rho_1) \sqsupseteq \alpha_{t_i}([\![x_i]\!]_3^{\mathcal{W}'}\ \rho_3). \quad (1)$$

Entonces:

$$\begin{array}{ll} [\![C\ x_1 \ldots x_m]\!]_1\ \rho_1 & \\ = \bigsqcup\limits_i(\widehat{\sqcup}([\![x_i]\!]_1\ \rho_1)) & \{\text{por definición de } [\![\cdot]\!]_1\} \\ \sqsupseteq \bigsqcup\limits_i\ (\alpha_{t_i}([\![x_i]\!]_3^{\mathcal{W}'}\ \rho_3)) & \{\text{por } (1)\} \\ = [\![C\ x_1 \ldots x_m]\!]_3^{\mathcal{W}'}\ \rho_3 & \{\text{por definición de } [\![\cdot]\!]_3^{\mathcal{W}'}\}. \end{array}$$

- $(x_1, \ldots, x_m) :: (t_1, \ldots, t_m)$. Denotemos por t a $(t_1, \ldots, t_m)$. En este caso se cumple, por un lado, que:

$$\begin{array}{l} [\![(x_1, \ldots, x_m)]\!]_1\ \rho_1 \\ = (\widehat{\sqcup}([\![x_1]\!]_1\ \rho_1), \ldots, \widehat{\sqcup}([\![x_m]\!]_1\ \rho_1)) \\ \qquad \{\text{por definición de } [\![\cdot]\!]_1\} \\ \sqsupseteq (\widehat{\sqcup}(\delta_{t_1}([\![x_1]\!]_3^{\mathcal{W}'}\ \rho_3)), \ldots, \widehat{\sqcup}(\delta_{t_m}([\![x_m]\!]_3^{\mathcal{W}'}\ \rho_3))) \\ \qquad \{\text{por h.i. y monotonía de } \widehat{\sqcup}\} \\ = (\alpha_{t_1}([\![x_1]\!]_3\ \rho_3), \ldots, \alpha_{t_m}([\![x_m]\!]_3^{\mathcal{W}'}\ \rho_3)) \\ \qquad \{\text{por la Proposición 35}\}, \end{array}$$

y por otro lado:

$$\begin{array}{l}\delta_t(\llbracket(x_1,\ldots,x_m)\rrbracket_3^{\mathcal{W}'}\ \rho_3)\\ \quad = \delta_t(\llbracket x_1\rrbracket_3^{\mathcal{W}'}\ \rho_3,\ldots,\llbracket x_m\rrbracket_3^{\mathcal{W}'}\ \rho_3)\\ \qquad \{\text{por definición de } \llbracket\cdot\rrbracket_3^{\mathcal{W}'}\}\\ \quad = (\alpha_{t_1}(\llbracket x_1\rrbracket_3^{\mathcal{W}'}\ \rho_3),\ldots,\alpha_{t_m}(\llbracket x_m\rrbracket_3^{\mathcal{W}'}\ \rho_3))\\ \qquad \{\text{por definición de } \delta_t\}.\end{array}$$

Luego

$$\llbracket(x_1,\ldots,x_m)\rrbracket_1\ \rho_1 \sqsupseteq \delta_t(\llbracket(x_1,\ldots,x_m)\rrbracket_3^{\mathcal{W}'}\ \rho_3).$$

- $\lambda v.e :: t_1 \to t_2$. (La demostración es exactamente la misma si la expresión fuera **process** $v \to e :: \mathit{Process}\ t_1\ t_2$.)

  En este caso tenemos por un lado :

  $$\llbracket\lambda v.e\rrbracket_1\ \rho_1 = \llbracket e\rrbracket_1\ \rho_1\ [v \mapsto d_{t_1}].$$

  Llamemos ρ_1' a $\rho_1\ [v \mapsto d_{t_1}]$.

  Por un lado tenemos que:

  $$\llbracket\lambda v.e\rrbracket_3^{\mathcal{W}'}\ \rho_3 = \lambda z.\llbracket e\rrbracket_3^{\mathcal{W}'}\ \rho_3\ [v \mapsto z],$$

  así que:

  $$\begin{array}{l}\delta_t(\llbracket\lambda v.e\rrbracket_3^{\mathcal{W}'}\ \rho_3)\\ \quad = \delta_t(\lambda z.\llbracket e\rrbracket_3^{\mathcal{W}'}\ \rho_3\ [v \mapsto z])\\ \quad = \delta_{t_2}(\llbracket e\rrbracket_3^{\mathcal{W}'}\ \rho_3\ [v \mapsto \gamma_{t_1}(d)]) \quad \{\text{por definición de } \delta_t\}.\end{array}$$

  Llamemos ρ_3' a $\rho_3\ [v \mapsto \gamma_{t_1}(d)]$.

  Si demostráramos que para cada variable $y :: t_y$, $\rho_1'(y) \sqsupseteq \delta_{t_y}(\rho_3'(y))$, entonces por h.i. tendríamos que

  $$\llbracket e\rrbracket_1\ \rho_1' \sqsupseteq \delta_{t_2}(\llbracket e\rrbracket_3\ \rho_3'),$$

  que es lo que queremos demostrar.

  Veamos pues que es cierto. Dada una variable $y :: t_y$:

  - Si $y \neq v$, se cumple trivialmente por hipótesis sobre los entornos ρ_1 y ρ_3.
  - Si $y = v$, entonces tenemos que demostrar que $d_{t_1} \sqsupseteq \delta_{t_1}(\gamma_{t_1}(d))$, lo que se cumple trivialmente por la Proposición 38.

- $e\ x :: t_2$, donde $e :: t_1 \to t_2$ y $x :: t_1$. Denotemos por t a $t_1 \to t_2$. (La demostración es similar si la expresión fuera $v\#x$ con $v :: \mathit{Process}\ t_1\ t_2$.) Por h.i. sabemos que

$$[\![x]\!]_1\ \rho_1 \sqsupseteq \delta_{t_1}([\![x]\!]_3^{\mathcal{W}'}\ \rho_3),$$

y

$$[\![e]\!]_1\ \rho_1 \sqsupseteq \delta_t([\![e]\!]_3^{\mathcal{W}'}\ \rho_3).$$

Luego se cumple que:

$$\begin{array}{l} [\![e\ x]\!]_1\ \rho_1 \\ = (\widehat{\sqcup}([\![x]\!]_1\ \rho_1)) \sqcup ([\![e]\!]_1\ \rho_1) \\ \qquad \{\text{por definición de } [\![\cdot]\!]_1\} \\ \sqsupseteq (\widehat{\sqcup}(\delta_{t_1}([\![x]\!]_3^{\mathcal{W}'}\ \rho_3))) \sqcup \delta_t([\![e]\!]_3^{\mathcal{W}'}\ \rho_3) \\ \qquad \{\text{por h.i. y monotonía de } \sqcup \text{ y } \widehat{\sqcup}\} \\ \sqsupseteq \delta_{t_2}(([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)([\![x]\!]_3^{\mathcal{W}'}\ \rho_3)) \\ \qquad \{\text{por la Proposición 41}\} \\ = \delta_{t_2}([\![e\ x]\!]_3^{\mathcal{W}'}\ \rho_3) \\ \qquad \{\text{por definición de } [\![\cdot]\!]_3^{\mathcal{W}'}\}. \end{array}$$

- **let** $v = e$ **in** $e' :: t$, donde v y e tienen tipo t_e, y $e' :: t$. En este caso, tenemos por un lado que:

$$[\![\mathbf{let}\ v = e\ \mathbf{in}\ e']\!]_1\ \rho_1 = [\![e']\!]_1\ \rho_1\ [v \mapsto [\![e]\!]_1\ \rho_1].$$

Denotemos por ρ_1' a $\rho_1\ [v \mapsto [\![e]\!]_1\ \rho_1]$.

Por otro lado tenemos que

$$[\![\mathbf{let}\ v = e\ \mathbf{in}\ e']\!]_3^{\mathcal{W}'}\ \rho_3 = [\![e']\!]_3^{\mathcal{W}'}\ \rho_3\ [v \mapsto [\![e]\!]_3^{\mathcal{W}'}\ \rho_3].$$

Denotemos por ρ_3' a $\rho_3\ [v \mapsto [\![e]\!]_1\ \rho_3]$.

Si demostráramos que para cada variable $y :: t_y$, $\rho_1'(y) \sqsupseteq \delta_{t_y}(\rho_3'(y))$, entonces por h.i. tendríamos que

$$[\![e']\!]_1\ \rho_1' \sqsupseteq \delta_t([\![e']\!]_3^{\mathcal{W}'}\ \rho_3'),$$

que es lo que deseamos demostrar.

Veamos pues que es cierto. Sea $y :: t_y$.

  - Si $y \neq v$, se cumple trivialmente por hipótesis sobre los entornos ρ_1 y ρ_3.
  - Si $y = v$, entonces hace falta demostrar que

$$[\![e]\!]_1\ \rho_1 \sqsupseteq \delta_{t_e}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3),$$

lo que se cumple por h.i.

- **case** e **of** $(v_1, \ldots, v_m) \to e' :: t$, con $e :: t_e = (t_1, \ldots, t_m)$, $v_i :: t_i$ y $e' :: t$. En este caso, tenemos por un lado:

$$\begin{array}{l}[\![\mathbf{case}\ e\ \mathbf{of}\ (v_1, \ldots, v_m) \to e]\!]_1\ \rho_1 \\ = [\![e']\!]_1\ \rho_1\ \overline{[v_i \mapsto (\pi_i([\![e]\!]_1\ \rho_1))_{t_i}]}.\end{array}$$

Y por otra parte

$$\begin{array}{l}[\![\mathbf{case}\ e\ \mathbf{of}\ (v_1, \ldots, v_m) \to e]\!]_3^{\mathcal{W}'}\ \rho_3 \\ = [\![e']\!]_3^{\mathcal{W}'}\ \rho_3\ \overline{[v_i \mapsto \pi_i([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)]}.\end{array}$$

Si demostráramos que para cada $i = 1, \ldots, m$ se tiene

$$(\pi_i([\![e]\!]_1\ \rho_1))_{t_i} \sqsupseteq \delta_{t_i}(\pi_i([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)),$$

entonces por h.i. se cumpliría que

$$\begin{array}{l}[\![e']\!]_1\ \rho_1\ \overline{[v_i \mapsto (\pi_i([\![e]\!]_1\ \rho_1))_{t_i}]} \\ \sqsupseteq \delta_t([\![e']\!]_3^{\mathcal{W}'}\ \rho_3\ \overline{[v_i \mapsto \pi_i([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)]}),\end{array}$$

que es lo que deseamos probar.

Veamos pues que esto es cierto. Por h.i. sabemos que

$$[\![e]\!]_1\ \rho_1 \sqsupseteq \delta_{t_e}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3),$$

lo cual se puede reescribir de la siguiente manera:

$$\begin{array}{l}(\pi_1([\![e]\!]_1\ \rho_1), \ldots, \pi_m([\![e]\!]_1\ \rho_1)) \\ = [\![e]\!]_1\ \rho_1 \\ \qquad \{\text{por ser de tipo tupla}\} \\ \sqsupseteq \delta_{t_e}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3) \\ \qquad \{\text{por h.i.}\} \\ = \delta_{t_e}(\pi_1([\![e]\!]_3^{\mathcal{W}'}\ \rho_3), \ldots, \pi_m([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)) \\ \qquad \{\text{por ser de tipo tupla}\} \\ = (\alpha_{t_1}(\pi_1([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)), \ldots, \alpha_{t_m}(\pi_m([\![e]\!]_3^{\mathcal{W}'}\ \rho_3))) \\ \qquad \{\text{por definición de } \delta_t\}.\end{array}$$

Es decir, para cada $i = 1, \ldots, m$ se cumple lo siguiente:

$$\pi_i([\![e]\!]_1\ \rho_1) \sqsupseteq \alpha_{t_i}(\pi_i([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)). \qquad (2)$$

Luego:

$$\begin{array}{l}(\pi_i([\![e]\!]_1\ \rho_1))_{t_i} \\ = \delta_{t_i}(\gamma_{t_i}(\pi_i([\![e]\!]_1\ \rho_1))) \\ \qquad \{\text{por la Proposición 38}\} \\ \sqsupseteq \delta_{t_i}(\gamma_{t_i}(\alpha_{t_i}(\pi_i([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)))) \\ \qquad \{\text{por (2) y monotonía de } \delta_t \text{ y } \gamma_t\} \\ \sqsupseteq \delta_{t_i}(\pi_i([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)) \\ \qquad \{\text{por la Proposición 25(c) y monotonía de } \delta_t\}.\end{array}$$

como queríamos demostrar.

- **case** e **of** $\overline{C_i\ \overline{v_{ij}} \to e_i} :: t$, con $e :: t_e = T\ t'_1 \dots t'_k$, $v_{ij} :: t_{ij}$ y $e_i :: t$. Denotemos por e' a **case** e **of** $\overline{C_i\ \overline{v_{ij}} \to e_i}$. En este caso tenemos por un lado:

$$[\![e']\!]_1\ \rho_1 = \begin{cases} n_t \text{ si } & [\![e]\!]_1\ \rho_1 = n \\ \bigsqcup_i [\![e_i]\!]_1\ \rho_{1i} & \text{e.o.c.} \end{cases}$$

$$\text{donde } \rho_{1i} = \rho_1\ \overline{[v_{ij} \mapsto d_{t_{ij}}]}, v_{ij} :: t_{ij}, e_i :: t,$$

y por el otro:

$$[\![e']\!]_3^{\mathcal{W}'}\ \rho_3 = \begin{cases} \gamma_t(n) & \text{si } [\![e]\!]_3^{\mathcal{W}'}\ \rho_3 = n \\ \bigsqcup_i [\![e_i]\!]_3^{\mathcal{W}'}\ \rho_{3i} & \text{e.o.c.} \end{cases}$$

$$\text{donde } \rho_{3i} = \rho_3\ \overline{[v_{ij} \mapsto \gamma_{t_{ij}}(d)]}, v_{ij} :: t_{ij}, e_i :: t.$$

Por h.i. sabemos que

$$[\![e]\!]_1\ \rho_1 \sqsupseteq \delta_{t_e}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3),$$

y dado que $\delta_{T\ t'_1,\dots,t'_k} = id_{Basic}$:

$$[\![e]\!]_1\ \rho_1 \sqsupseteq [\![e]\!]_3^{\mathcal{W}'}\ \rho_3.$$

Distinguimos tres casos:

– $[\![e]\!]_1\ \rho_1 = [\![e]\!]_3^{\mathcal{W}'}\ \rho_3 = n$. En tal caso:

$$\begin{array}{rcll} [\![e']\!]_1\ \rho_1 & = & n_t & \{\text{por definición de } [\![\cdot]\!]_1\} \\ & = & \delta_t(\gamma_t(n)) & \{\text{por la Proposición 38}\} \\ & = & \delta_t([\![e']\!]_3^{\mathcal{W}'}\ \rho_3) & \{\text{por definición de } [\![\cdot]\!]_3^{\mathcal{W}'}\}. \end{array}$$

– $[\![e]\!]_1\ \rho_1 = [\![e]\!]_3^{\mathcal{W}'}\ \rho_3 = d$. En este caso:

$$[\![e']\!]_1\ \rho_1 = \bigsqcup_i [\![e_i]\!]_1\ \rho_{1i},$$

donde $\rho_{1i} = \rho_1\ \overline{[v_{ij} \mapsto d_{t_{ij}}]}$, y

$$[\![e']\!]_3^{\mathcal{W}'}\ \rho_3 = \bigsqcup_i [\![e_i]\!]_3^{\mathcal{W}'}\ \rho_{3i},$$

donde $\rho_{3i} = \rho_3\ \overline{[v_{ij} \mapsto \gamma_{t_{ij}}(d)]}$.
Si demostrásemos para cada $i = 1, \dots, m$, que para cada variable $y :: t_y$, $\rho_{1i}(y) \sqsupseteq \delta_{t_y}(\rho_{3i}(y))$, entonces por h.i. se cumpliría lo siguiente para cada i

$$[\![e_i]\!]_1\ \rho_{1i} \sqsupseteq \delta_t([\![e_i]\!]_3^{\mathcal{W}'}\ \rho_{3i}).$$

Y entonces, por continuidad de δ_t:

$$\bigsqcup_i [\![e_i]\!]_1\ \rho_{1i} \sqsupseteq \bigsqcup_i \delta_t([\![e_i]\!]_3^{\mathcal{W}'}\ \rho_{3i}) = \delta_t(\bigsqcup_i [\![e_i]\!]_3^{\mathcal{W}'}\ \rho_{3i}),$$

que es lo que queríamos demostrar.

Veamos pues que para cada variable $y :: t_y$, $\rho_{1i}(y) \sqsupseteq \delta_{t_y}(\rho_{3i}(y))$. Si y no es ninguna de las v_{ij}, se cumple trivialmente por hipótesis sobre ρ_1 y ρ_3. Si y es alguna de las v_{ij}, entonces también es cierto por la Proposición 38.

– $[\![e]\!]_1\ \rho_1 = n \sqsupset [\![e]\!]_3^{\mathcal{W}'}\ \rho_3 = d$. En este caso $[\![e']\!]_1\ \rho_1 = n_t$, y $[\![e']\!]_3^{\mathcal{W}'}\ \rho_3 = \bigsqcup_i [\![e_i]\!]_3^{\mathcal{W}'}\ \rho_{3i}$. Claramente, $n_t \sqsupseteq \bigsqcup_i [\![e_i]\!]_3^{\mathcal{W}'}\ \rho_{3i}$, pues n_t es el supremo de D_{1t}.

El caso de la expresión **case** primitiva es muy similar.

- **let rec** $\overline{\{v_i = e_i\}}$ **in** $e' :: t$, donde $e' :: t$, y cada v_i y e_i tienen tipo t_i. Llamemos e a **let rec** $\overline{\{v_i = e_i\}}$ **in** e'. En este caso:

$$[\![e]\!]_1\ \rho_1 = [\![e']\!]\ (\bigsqcup_{n\in\mathbb{N}} (\lambda\rho'_1.\rho_1\ \overline{[v_i \mapsto [\![e_i]\!]_1\ \rho'_1]})^n(\rho_{01})),$$

donde ρ_{01} es el entorno inicial en el que cada variable $y :: t_y$ tiene d_{t_y} como valor abstracto (es decir, el ínfimo del dominio correspondiente). Denotemos por F a la función entre entornos $\lambda\rho'_1.\rho_1\ \overline{[v_i \mapsto [\![e_i]\!]_1\ \rho'_1]}$ y por ρ_1^{fix} a $\bigsqcup_{n\in\mathbb{N}} F^n(\rho_{01})$.

De forma similar

$$[\![e]\!]_3\ \rho_3 = \\ [\![e']\!]_3^{\mathcal{W}'}\ (\bigsqcup_{n\in\mathbb{N}}(\lambda\rho'_3.\rho_3\ \overline{[v_i \mapsto \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}\ \rho'_3)]})^n(\rho_{03})),$$

donde ρ_{03} es el entorno inicial en el que cada variable $y :: t_y$ tiene $\bot_{2t_y}$ como valor abstracto (es decir, el ínfimo del dominio correspondiente). Denotemos por G a la función entre entornos $\lambda\rho'_3.\rho_3\ \overline{[v_i \to \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}\ \rho'_3)]}$ y por ρ_3^{fix} a $\bigsqcup_{n\in\mathbb{N}} G^n(\rho_{03})$.

Si demostrásemos que para cada variable $y :: t_y$, $\rho_1^{fix}(y) \sqsupseteq \delta_{t_y}(\rho_3^{fix}(y))$, entonces por h.i. tendríamos que

$$[\![e']\!]_1\ \rho_1^{fix} \sqsupseteq \delta_t([\![e']\!]_3^{\mathcal{W}'}\ \rho_3^{fix}),$$

que es lo que queremos demostrar.

Veamos que para cada $n \geq 0$, se cumple lo siguiente:

$$\forall y :: t_y.(F^n(\rho_{01}))(y) \sqsupseteq \delta_{t_y}((G^n(\rho_{03}))(y)).$$

Si esto fuera cierto entonces

$$\forall y :: t_y.(\bigsqcup_{n\in\mathbb{N}} F^n(\rho_{01}))(y) \sqsupseteq \delta_{t_y}((\bigsqcup_{n\in\mathbb{N}} G^n(\rho_{03}))(y)),$$

por continuidad de δ_t, y esto es lo que deseamos demostrar.

El resultado en cuestión se puede demostrar por inducción sobre n:

- $n = 0$. Este es un caso trivial ya que $F^0(\rho_{01}) = \rho_{01}$, $G^0(\rho_{03}) = \rho_{03}$ y $d_t = \delta_t(\perp_{2t})$, por la Proposición 36.
- $n = m + 1$. Entonces

$$\begin{array}{l} F^{m+1}(\rho_{01}) \\ \quad = F(F^m(\rho_{01})) \\ \quad = \rho_1\ \overline{[v_i \mapsto [\![e_i]\!]_1\ (F^m(\rho_{01}))]} \quad \{\text{por definición de } F\}, \end{array}$$

y

$$\begin{array}{l} G^{m+1}(\rho_{03}) \\ \quad = G(G^m(\rho_{03})) \\ \quad = \rho_3\ \overline{[v_i \mapsto \mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}\ (G^m(\rho_{03})))]} \\ \qquad \{\text{por definición de } G\}. \end{array}$$

Sea $y :: t_y$. Queremos demostrar que

$$(F^{m+1}(\rho_{01}))(y) \sqsupseteq \delta_{t_y}((G^{m+1}(\rho_{03}))(y)).$$

Distinguimos dos casos. Si y no es ninguna de las v_i, entonces se cumple por hipótesis sobre los entornos ρ_1 y ρ_3. Si es alguna de las v_i, entonces tenemos que demostrar que

$$[\![e_i]\!]_1\ (F^m(\rho_{01})) \sqsupseteq \delta_{t_i}(\mathcal{W}'_{t_i}([\![e_i]\!]_3^{\mathcal{W}'}\ (G^m(\rho_{03})))), \qquad (3)$$

Por h.i. (interna sobre n)

$$\forall y :: t_y.\ (F^m(\rho_{01}))(y) \sqsupseteq \delta_{t_y}((G^m(\rho_{03}))(y)),$$

de donde obtenemos por h.i. (externa sobre e) que

$$[\![e_i]\!]_1\ (F^m(\rho_{01})) \sqsupseteq \delta_{t_i}([\![e_i]\!]_3\ (G^m(\rho_{03}))).$$

Como $\delta_t \cdot \mathcal{W}'_t = \delta_t$ por hipótesis, obtenemos (3).

- $\Lambda\beta.e :: \forall\beta.t$, con $e :: t$. En este caso:

$$\begin{array}{ll} [\![\Lambda\beta.e]\!]_1\ \rho_1 & \\ \quad = [\![e]\!]_1\ \rho_1 & \{\text{por definición de } [\![\cdot]\!]_1\} \\ \quad \sqsupseteq \delta_t([\![e]\!]_3^{\mathcal{W}'}\ \rho_3) & \{\text{por h.i.}\} \\ \quad = \delta_{\forall\beta.t}([\![\Lambda\beta.e]\!]_3^{\mathcal{W}'}\ \rho_3) & \{\text{por definición de } [\![\cdot]\!]_3^{\mathcal{W}'} \text{ y } \delta_t\}. \end{array}$$

- $e\ t :: tinst$ donde $e :: \forall\beta.t'$ y $tinst = t'[\beta := t]$. En este caso

$$
\begin{array}{l}
[\![e\ t]\!]_1\ \rho_1 \\
= ([\![e]\!]_1\ \rho_1)_{tinst} \\
\qquad \{\text{por definición de } [\![\cdot]\!]_1\} \\
= \delta_{tinst}(\gamma_{t'\,tinst}(\eta_{t'}([\![e]\!]_1\ \rho_1))) \\
\qquad \{\text{por la Proposición 40(b)}\} \\
\sqsupseteq \delta_{tinst}(\gamma_{t'\,tinst}(\eta_{t'}(\delta_{t'}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)))) \\
\qquad \{\text{por h.i. y monotonía de } \delta_t,\ \gamma_{t'\,tinst} \text{ y } \eta_t\} \\
= \delta_{tinst}(\gamma_{t'\,tinst}(\nabla_{t'}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3))) \\
\qquad \{\text{por definición de } \nabla_t\} \\
= \delta_{tinst}(\nabla_{tinst}(\gamma_{t'\,tinst}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3))) \\
\qquad \{\text{por la Proposición 40(a)}\} \\
= \delta_{tinst}(\gamma_{t'\,tinst}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3)) \\
\qquad \{\text{por definición de } \nabla_t \text{ y Proposición 34(b)}\} \\
= \delta_{tinst}([\![e\ t]\!]_3^{\mathcal{W}'}\ \rho_3) \\
\qquad \{\text{por definición de } [\![\cdot]\!]_3^{\mathcal{W}'}\}.
\end{array}
$$

□

Como corolario, obtenemos la corrección del primer análisis con respecto a todas estas variantes: si el primer análisis nos dice que una expresión es determinista entonces la variante correspondiente del tercer análisis también nos dice que es determinista, probablemente con cierto detalle adicional como la independencia de la salida con respecto a la entrada de una función o de un proceso.

Corolario 44 *Sea $\mathcal{W}'_t$ un operador de ensanchamiento para cada tipo t, tal que $\delta_t \cdot \mathcal{W}'_t = \delta_t$. Si para cada variable $v :: t_v$, $\rho_1(v) \sqsupseteq \delta_{t_v}(\rho_3(v))$ entonces: $\forall e :: t_e.\ [\![e]\!]_1\ \rho_1 = d_{t_e} \Rightarrow [\![e]\!]_3^{\mathcal{W}'}\ \rho_3 \sqsubseteq \gamma_{t_e}(d)$.*

Demostración 18 (Corolario 44) Este corolario se demuestra a partir del Teorema 43, el Lema 42 y la Proposición 34(c).

Supongamos que para cada variable $v :: t_v$, $\rho_1(v) \sqsupseteq \delta_{t_v}(\rho_3(v))$. Sea $e :: t_e$ tal que $[\![e]\!]_1\ \rho_1 = d_{t_e}$. Entonces:

$$
\begin{array}{rcll}
[\![e]\!]_3^{\mathcal{W}'}\ \rho_3 & \sqsubseteq & \nabla_{t_e}([\![e]\!]_3^{\mathcal{W}'}\ \rho_3) & \{\text{por la Proposición 34(c)}\} \\
& \sqsubseteq & \eta_{t_e}([\![e]\!]_1\ \rho_1) & \{\text{por Teorema 43}\} \\
& = & \eta_{t_e}(d_{t_e}) & \{\text{por hipótesis}\} \\
& = & \gamma_{t_e}(d) & \{\text{por Lema 42}\}.
\end{array}
$$

□

La Proposición 46 afirma que el operador de ensanchamiento definido en la Figura 5.25 satisface las hipótesis del corolario anterior (el operador $\mathcal{W}_b$ mencionado previamente también las cumple). Para demostrarlo, necesitamos demostrar antes lo siguiente:

Proposición 45 *Para cada tipo* t*,* $\alpha_t \cdot \mathcal{W}_t = \alpha_t$*.*

Demostración 19 (Proposición 45) Se demuestra por inducción estructural sobre t:

- $t = K$ o $t = T\ t_1 \ldots t_m$ o $t = \beta$. Estos casos son triviales pues $\alpha_t = \mathcal{W}_t = id_{Basic}$.

- $t = (t_1, \ldots, t_m)$. Sea $(e_1, \ldots, e_m) \in D_{2t}$. En este caso:

$$\begin{array}{ll} \alpha_t(\mathcal{W}_t(e_1, \ldots, e_m)) & \\ \quad = \bigsqcup\limits_i \alpha_{t_i}(\mathcal{W}_{t_i}(e_i)) & \{\text{por definición de } \alpha_t \text{ y } \mathcal{W}_t\} \\ \quad = \bigsqcup\limits_i \alpha_{t_i}(e_i) & \{\text{por h.i.}\} \\ \quad = \alpha_t(e_1, \ldots, e_m) & \{\text{por definición de } \alpha_t\}. \end{array}$$

- $t = t_1 \to t_2'$ o $t = \mathit{Process}\ t_1\ t_2'$. Para $f \in D_{2t}$, si $m = \mathit{nArgs}(t)$, $[t_1, \ldots, t_m] = \mathit{aTypes}(t)$ y $t_r = \mathit{rType}(t)$, entonces

$$\begin{array}{ll} \alpha_t(\mathcal{W}_t(f)) & \\ \quad = \alpha_{t_r}(\mathcal{W}_{t_r}(f\ \gamma_{t_1}(d) \ldots \gamma_{t_m}(d))) & \{\text{por definición de } \alpha_t \text{ y } \mathcal{W}_t\} \\ \quad = \alpha_{t_r}(f\ \gamma_{t_1}(d) \ldots \gamma_{t_m}(d)) & \{\text{por h.i.}\} \\ \quad = \alpha_t(f) & \{\text{por definición de } \alpha_t\}. \end{array}$$

- $t = \forall \beta . t_1$. En este caso se cumple trivialmente por h.i. pues $\alpha_t = \alpha_{t_1}$ y $\mathcal{W}_t = \mathcal{W}_{t_1}$.

$\square$

Proposición 46 *Para cada tipo* t*,* $\delta_t \cdot \mathcal{W}_t = \delta_t$*.*

Demostración 20 (Proposición 46) Se demuestra por inducción estructural sobre t aplicando la Proposición 45:

- $t = K$ o $t = T\ t_1 \ldots t_m$ o $t = \beta$. Estos casos son triviales ya que $\delta_t = \mathcal{W}_t = id_{Basic}$.

- $t = (t_1, \ldots, t_m)$. Sea $(e_1, \ldots, e_m) \in D_{2t}$. En este caso:

$$\begin{array}{l} \delta_t(\mathcal{W}_t(e_1, \ldots, e_m)) \\ \quad = (\alpha_{t_1}(\mathcal{W}_{t_1}(e_1)), \ldots, \alpha_{t_m}(\mathcal{W}_{t_m}(e_m))) \\ \qquad \{\text{por definición de } \delta_t \text{ y } \mathcal{W}_t\} \\ \quad = (\alpha_{t_1}(e_1), \ldots, \alpha_{t_m}(e_m)) \\ \qquad \{\text{por la Proposición 45}\} \\ \quad = \delta_t(e_1, \ldots, e_m) \\ \qquad \{\text{por definición de } \delta_t\}. \end{array}$$

- $t = t_1 \to t_2'$ o $t = \mathit{Process}\ t_1\ t_2'$. Para $f \in D_{2t}$, si $m = \mathit{nArgs}(t)$, $[t_1, \ldots, t_m] = \mathit{aTypes}(t)$ y $t_r = \mathit{rType}(t)$, entonces

$$\begin{array}{ll}\delta_t(\mathcal{W}_t(f)) & \\ \quad = \delta_{t_r}(\mathcal{W}_{t_r}(f\ \gamma_{t_1}(d) \ldots \gamma_{t_m}(d))) & \{\text{por definición de } \delta_t \text{ y } \mathcal{W}_t\} \\ \quad = \delta_{t_r}(f\ \gamma_{t_1}(d) \ldots \gamma_{t_m}(d)) & \{\text{por h.i.}\} \\ \quad = \delta_t(f) & \{\text{por definición de } \delta_t\}.\end{array}$$

- $t = \forall\beta.t_1$. En este caso se cumple por h.i. pues $\delta_t = \delta_{t_1}$ y $\mathcal{W}_t = \mathcal{W}_{t_1}$.

□

La corrección del primer análisis con respecto al segundo se obtiene trivialmente a partir del Teorema 43, ya que el segundo análisis es, en realidad, una variante del tercero en la que el operador de ensanchamiento es la función identidad.

5.8 Implementación del tercer análisis

5.8.1 Introducción

En esta sección describimos los aspectos principales de la implementación del tercer análisis. El algoritmo aquí descrito no solamente obtiene los valores abstractos de las expresiones, sino que también anota cada expresión y sus subexpresiones con su signatura correspondiente.

Una versión completa de este algoritmo, presentada en el Apéndice B, se ha implementado en Haskell. La implementación del análisis incluye un pequeño analizador sintáctico y un *pretty printer* [Hug95]. En el Apéndice C presentamos algunos ejemplos de aplicación del análisis.

Obsérvese que es importante anotar las subexpresiones, incluso dentro de una lambda abstracción, puesto que, como ya se ha explicado, la transformación de *full laziness* puede modificar la semántica de un programa cuando se ven implicadas expresiones no deterministas. Dada una expresión $f = \lambda y.\mathbf{let}\ x = e_1\ \mathbf{in}\ x + y$, donde e_1 no depende de y, la transformación produciría $f' = \mathbf{let}\ x = e_1\ \mathbf{in}\ \lambda y.x + y$. El problema surge cuando e_1 es no determinista, por lo que anotar expresiones dentro de funciones es necesario.

En el algoritmo se hace uso del hecho de que la implementación está realizada en un lenguaje funcional perezoso. Así, la interpretación de una lambda $\lambda v.e$ en un entorno ρ es una función abstracta. No es necesario usar una notación diferente para construir funciones abstractas, sino que podemos usar las propias construcciones del lenguaje. Usaremos una suspensión $\lambda v.(e, \rho)$ para representar el valor abstracto de $\lambda v.e$. Solamente cuando la función se aplique a un argumento, se interpretará el cuerpo e de la función en el entorno adecuado, emulando de esta forma el comportamiento de la

$$\gamma'_t :: Basic \to S_t$$

$$\gamma'_t(b) = \begin{cases} B \text{ si } t = K, t = T\ t_1 \ldots t_m, t = \beta \\ (\gamma'_{t_1}(b), \ldots, \gamma'_{t_m}(b)) \text{ if } t = (t_1, \ldots, t_m) \\ \gamma'_{t_1}(b) \text{ si } t = \forall t.t_1 \\ \langle t,\ \gamma'_{t_r}(n) \overset{(m)}{\cdots} \gamma'_{t_r}(n)\ +\ \gamma'_{t_r}(b)\rangle \text{ si } t = t_1 \to t_2, Process\ t_1\ t_2 \\ \quad \text{donde } m = nArgs(t),\ \ t_r = rType(t) \end{cases}$$

Figura 5.33: Las signaturas correspondientes a $\gamma_t(n)$ y $\gamma_t(d)$.

	$\gamma'_{t'\,tinst} : D_{2t'} \to D_{2\,tinst}$	$\alpha'_{tinst\,t'} : D_{2\,tinst} \to D_{2t'}$
$t' = K, T\ t_1 \ldots t_m,$ $\beta'(\neq \beta)$	$\gamma'_{t'\,tinst} = id_{Basic}$	$\alpha'_{tinst\,t'} = id_{Basic}$
$t' = (t_1, \ldots, t_m)$	$\gamma'_{t'\,tinst}(av_1, \ldots, av_m) =$ $(\gamma'_{t_1\,tinst_1}(av_1), \ldots, \gamma'_{t_m\,tinst_m}(av_m))$	$\alpha'_{tinst\,t'}(av_1, \ldots, av_m) =$ $(\alpha'_{tinst_1\,t_1}(av_1), \ldots, \alpha'_{tinst_m\,t_m}(av_m))$
$t' = t_1 \to t_2,$ $Process\ t_1\ t_2$	$\gamma'_{t'\,tinst}(av) = G\ t'\ t'[t/\beta]\ av$	$\alpha'_{tinst\,t'}(av) = A\ t'[t/\beta]\ t'\ av$
$t' = \beta$	$\gamma'_{t'\,tinst} = \gamma'_t$	$\alpha'_{tinst\,t'} = \alpha_t$
$t' = \forall\beta'.t_1$	$\gamma'_{t'\,tinst} = \gamma'_{t_1\,tinst_1}$	$\alpha'_{tinst\,t'} = \alpha'_{tinst_1\,t_1}$

Figura 5.34: Implementación de las funciones $\alpha_{tinst\,t'}$ y $\gamma_{t'\,tinst}$

función abstracta. Es decir, usamos la evaluación perezosa de Haskell como maquinaria de interpretación. Si no lo hiciéramos así, sería necesario construir un intérprete, lo que resultaría menos eficiente.

Sin embargo, esta decisión introduce algunos problemas. En ocasiones es necesario construir una función abstracta que no se obtiene a partir de la interpretación de una lambda abstracción del programa. Una de ellas corresponde a la aplicación de $\gamma_t(b)$ cuando t es un tipo función o proceso. En la Proposición 47 veremos que podemos usar la correspondiente signatura para representar $\gamma_t(b)$ sin perder información. Luego, en última instancia, en este caso no hace falta construir una función. Dado un valor básico b, la función $\gamma'_t = \wp_t \cdot \gamma_t$, definida en la Figura 5.33, devuelve la signatura de $\gamma_t(b)$.

Otra situación en la que sucede es cuando se calculan $\alpha_{tinst\,t'}(av)$ y $\gamma_{t'\,tinst}(av)$ cuando t' es un tipo función o proceso. En este caso, como veremos más adelante, no podemos representar estos valores mediante sus signaturas sin perder información, con lo que para representarlos será necesario construir dos nuevas suspensiones $A\ t'[t/\beta]\ t'\ av$ y $G\ t'\ t'[t/\beta]\ av$ (ver Figura 5.35). En la Figura 5.34 se muestra la implementación de las funciones $\gamma_{t'\,tinst}$ y $\alpha_{tinst\,t'}$, llamadas $\gamma'_{t'\,tinst}$ y $\alpha'_{tinst\,t'}$ respectivamente. En el caso funcional se limitan a devolver la correspondiente suspensión. Sólo cuando se aplican a un argumento, se aplican sus definiciones (ver Figura 5.37).

También necesitamos construir una función cuando se calcula la mínima cota superior de funciones en las expresiones **case**. En este caso, usaremos también una nueva suspensión $\boxed{\mathrm{F}}\,[av_1, \ldots, av_m]$ (ver Figura 5.35). Cuando la función representada por esta suspensión se aplica, se llevará a cabo el cálculo de la mínima cota superior (ver Figura 5.37).

Proporcionamos a continuación algunos resultados teóricos que han sido útiles en el desarrollo de la implementación.

5.8.2 Resultados teóricos

La siguiente proposición nos dice que $\gamma_t(d)$ y $\gamma_t(n)$ se pueden representar mediante sus correspondientes signaturas sin perder información, lo cual será muy útil en la implementación del análisis.

Proposición 47 *Para cada tipo t, $\mathcal{W}_t \cdot \gamma_t = \gamma_t$.*

Demostración 21 (Proposición 47) Lo demostramos por inducción estructural sobre t:

- $t = K$ o $t = T\ t_1 \ldots t_m$ o $t = \beta$. Estos casos son triviales, ya que $\mathcal{W}_t = \gamma_t = id_{Basic}$.

- $t = (t_1, \ldots, t_m)$. Sea $b \in Basic$. Entonces

$$\begin{array}{rcl} \mathcal{W}_t(\gamma_t(b)) & = & (\mathcal{W}_{t_1}(\gamma_{t_1}(b)), \ldots \mathcal{W}_{t_m}(\gamma_{t_m}(b))) \\ & & \quad \{\text{por definición de } \mathcal{W}_t \text{ y } \gamma_t\} \\ & = & (\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) \\ & & \quad \{\text{por h.i.}\} \\ & = & \gamma_t(b) \\ & & \quad \{\text{por definición de } \gamma_t\}. \end{array}$$

- $t = t_1 \to t_2$ o $t = Process\ t_1\ t_2$. Sean $m = nArgs(t)$, $[t_1, \ldots, t_m] = aTypes(t)$ y $t_r = rType(t)$. Sean $z_i \in D_{2t_i}$ con $i \in \{1..m\}$. Hemos de demostrar que $\mathcal{W}_t(\gamma_t(b))\ \overline{z_i} = \gamma_t(b)\ \overline{z_i}$. Distinguimos dos casos, $b = n$ y $b = d$.

  Si $b = n$, entonces

$$\begin{array}{rcll} \mathcal{W}_t(\gamma_t(n))\ \overline{z_i} & = & \mathcal{W}_{t_r}(\gamma_{t_r}(n)) & \{\text{por definición de } \mathcal{W}_t \text{ y } \gamma_t\} \\ & = & \gamma_{t_r}(n) & \{\text{por h.i.}\} \\ & = & \gamma_t(n) & \{\text{por definición de } \gamma_t\}. \end{array}$$

  Si $b = d$ tenemos que volver a distinguir dos casos. Si $\bigwedge_{i=1}^{m} z_i \sqsubseteq \gamma_{t_i}(d)$, como α_t y γ_t forman una inserción de Galois tenemos que (4) $\alpha_{t_i}(z_i) = d$ $(i \in \{1..m\})$. Ello implica por definición de γ_t que (5) $\gamma_t(d)\ \overline{z_i} = \gamma_{t_r}(d)$.

$$\begin{array}{rcll} \mathcal{W}_t(\gamma_t(d))\ \overline{z_i} & = & \mathcal{W}_{t_r}(\gamma_{t_r}(d)) & \{\text{por (4) y def. de } \mathcal{W}_t\} \\ & = & \gamma_{t_r}(d) & \{\text{por h.i.}\} \\ & = & \gamma_t(d)\ \overline{z_i} & \{\text{por (5)}\}. \end{array}$$

En caso contrario existe un $j \in \{1..m\}$ tal que $z_j \not\sqsubseteq \gamma_{t_j}(d)$. Esto significa que (6) $\alpha_{t_j}(z_j) = n$, por lo que (7) $\gamma_t(d)\ \overline{z_i} = \gamma_{t_r}(n)$, por definición de γ_t.

Si $\bigwedge_{i=1, i\neq j}^{m} z_i \sqsubseteq \gamma_{t_i}(d)$, entonces

$$\begin{array}{lcll} \mathcal{W}_t(\gamma_t(d))\ \overline{z_i} & = & \mathcal{W}_{t_r}(\gamma_{t_r}(n)) & \{\text{por (6) y def. de } \mathcal{W}_t\} \\ & = & \gamma_{t_r}(n) & \{\text{por h.i.}\} \\ & = & \gamma_t(d)\ \overline{z_i} & \{\text{por (7)}\}. \end{array}$$

En caso contrario,

$$\begin{array}{lcll} \mathcal{W}_t(\gamma_t(d))\ \overline{z_i} & = & \gamma_{t_r}(n) & \{\text{por definición de } \mathcal{W}_t\} \\ & = & \gamma_t(d)\ \overline{z_i} & \{\text{por (7)}\}. \end{array}$$

- $t = \forall\beta.t_1$. Este caso es trivial por hipótesis de inducción, pues $\mathcal{W}_t = \mathcal{W}_{t_1}$ y $\gamma_t = \gamma_{t_1}$.

□

La siguiente proposición nos dice que la comparación entre un valor abstracto y $\gamma_t(d)$ se puede llevar a cabo comparando sus correspondientes signaturas, lo que resulta ser poco costoso. Esto será muy útil en la implementación, ya que dicha comparación se hará muchas veces: en el peor de los casos la comparación hasta $\mathcal{H}_t$ veces ($\mathcal{H}_t$ se definió en la Figura 5.21).

Proposición 48 *Para cada tipo* t, $\forall z \in D_{2t}.z \sqsubseteq \gamma_t(d) \Leftrightarrow \wp_t(z) \preceq \wp_t(\gamma_t(d))$.

Demostración 22 (Proposición 48)

- ($\Rightarrow$) Es trivial, pues $\wp_t$ es monótona.

- ($\Leftarrow$) Supongamos que $\wp_t(z) \preceq \wp_t(\gamma_t(d))$. Entonces

$$\begin{array}{lcll} z & \sqsubseteq & \mathcal{W}_t(z) & \{\text{por la Proposición 31(b)}\} \\ & = & \Re_t(\wp_t(z)) & \{\text{por definición de } \mathcal{W}_t\} \\ & \sqsubseteq & \Re_t(\wp_t(\gamma_t(d))) & \{\text{por hipótesis y Proposición 31(a)}\} \\ & = & \mathcal{W}_t(\gamma_t(d)) & \{\text{por definición de } \mathcal{W}_t\} \\ & = & \gamma_t(d) & \{\text{por la Proposición 47}\}. \end{array}$$

□

$$
\begin{array}{lcllcl}
av & \rightarrow & b & b & \rightarrow & \mid d \\
 & & \mid (av_1, \ldots, av_m) & & & \mid n \\
 & & \mid \lambda v.(e, \rho) & aw & \rightarrow & \mid b \\
 & & \mid G\ t'\ t'[t/\beta]\ av & & & \mid (aw_1, \ldots, aw_m) \\
 & & \mid A\ t'[t/\beta]\ t'\ av & & & \mid \langle t,\ aw_1 \ldots aw_m + aw \rangle \\
 & & \mid \boxed{\mathrm{F}}\,[av_1, \ldots, av_m] & & & \mid \langle t,\ +aw \rangle \\
 & & \mid aw & & &
\end{array}
$$

Figura 5.35: Definición de los valores abstractos

5.8.3 Polimorfismo

Nos gustaría tener una propiedad similar a la Proposición 47, $\mathcal{W}_{tinst} \cdot \gamma_{t'\,tinst} = \gamma_{t'\,tinst}$, de forma que pudiéramos representar los valores abstractos de los ejemplares de un tipo polimórfico mediante sus signaturas, pero no es así. Veamos un contraejemplo. Recordemos el ejemplo de la Sección 5.5.4. Consideramos el tipo polimórfico $\forall \beta.t'$, con $t' = (\beta, \beta) \rightarrow \beta$, aplicado al tipo $t = Int \rightarrow Int$. Entonces $tinst = t'[t/\beta] = (Int \rightarrow Int, Int \rightarrow Int) \rightarrow Int \rightarrow Int$. Para abreviar denotaremos por E_p a $Basic \times Basic$ y por F_p a $[Basic \rightarrow Basic] \times [Basic \rightarrow Basic]$. Sea $f \in D_{2t'}$, $f = \lambda p \in E_p.\pi_1(p)$. Por definición, tenemos que $\gamma_{t'\,tinst}(f) = \lambda p \in F_p.\gamma_{t_2\,tinst_2}(f\ (\alpha_{tinst_1\,t_1}(p)))$.

También por definición se cumple que

$$
\begin{array}{l}
\mathcal{W}_{tinst}(\gamma_{t'\,tinst}(f)) = \\
\quad \lambda p \in F_p. \begin{cases} \lambda u \in Basic.u & \text{si } p \sqsubseteq (\lambda z \in Basic.z, \lambda z \in Basic.z) \\ \lambda u \in Basic.n & \text{e.o.c.} \end{cases}
\end{array}
$$

Y tomando $q = (\lambda z \in Basic.z, \lambda z \in Basic.n)$ obtenemos

$$
\gamma_{t'\,tinst}(f)\ q = \lambda u \in Basic.u \sqsubset \lambda u \in Basic.n = (\mathcal{W}_{tinst}(\gamma_{t'\,tinst}(f)))\ q.
$$

5.8.4 Definición de los valores abstractos

En la Figura 5.35 se definen los valores abstractos usados en la implementación del análisis. En ella se llevan a cabo dos tareas. Por un lado se calculan los valores abstractos de las expresiones, según se han definido en el tercer análisis. Por otro lado, se anotan las expresiones con signaturas correspondientes a dichos valores abstractos, perdiendo en dicho momento cierta información.

En la implementación del análisis las signaturas se considerarán también como valores abstractos, de modo que una signatura $s \in S_t$ es sólo una forma de representar el valor abstracto $\Re_t(s)$. Así, usaremos un único tipo $AbsVal$ para representar tanto los valores abstractos devueltos por el análisis como las anotaciones de las expresiones, ver Figura 5.38.

$$
\begin{array}{l}
n \sqcup b = n \\
d \sqcup b = b \\
(av_1, \ldots, av_m) \sqcup (av'_1, \ldots, av'_m) = (av_1 \sqcup av'_1, \ldots, av_m \sqcup av'_m) \\
\langle t,\ aw_1 \ldots aw_m + aw\rangle \sqcup \langle t,\ aw'_1, \ldots, aw'_m + aw'\rangle = \\
\qquad\qquad\qquad \langle t,\ (aw_1 \sqcup aw'_1) \ldots (aw_m \sqcup aw'_m) + (aw \sqcup aw')\rangle \\
\langle t,\ +aw\rangle \sqcup \langle t,\ +aw'\rangle = \langle t,\ +(aw \sqcup aw')\rangle \\
\langle t,\ aw_1 \ldots aw_m + aw\rangle \sqcup \langle t,\ +aw'\rangle = \langle t,\ +aw'\rangle \sqcup \langle t,\ aw_1 \ldots aw_m + aw\rangle \\
\qquad\qquad\qquad = \langle t,\ +(aw \sqcup aw')\rangle \\
(\boxed{\scriptstyle F}\, avs) \sqcup av = av \sqcup (\boxed{\scriptstyle F}\, avs) = \boxed{\scriptstyle F}\, av : avs \\
av \sqcup \lambda v.(e, \rho) = \lambda v.(e, \rho) \sqcup av = \boxed{\scriptstyle F}\, [av, \lambda v.(e, \rho)] \\
(G\ t'\ t'[t/\beta]\ av) \sqcup av' = av' \sqcup (G\ t'\ t'[t/\beta]\ av) = \boxed{\scriptstyle F}\, [av', G\ t'\ t'[t/\beta]\ av] \\
(A\ t'[t/\beta]\ t'\ av) \sqcup av' = av' \sqcup (A\ t'[t/\beta]\ t'\ av) = \boxed{\scriptstyle F}\, [av', A\ t'[t/\beta]\ t'\ av] \\
\sqcup\ [av_1, \ldots, av_m] = av_1 \sqcup \ldots \sqcup av_m
\end{array}
$$

Figura 5.36: Definición del operador de mínima cota superior

Pueden ser valores abstractos básicos d o n, que se utilizarán tanto para representar un valor abstracto básico verdadero como una signatura básica. Las tuplas de valores abstractos son también valores abstractos. Un valor abstracto funcional puede tener distintas representaciones: puede estar representado por una signatura o por una suspensión.

En la Figura 5.35 una signatura funcional puede ser de dos formas: $\langle t, aw_1 \ldots aw_m + aw\rangle$ o $\langle t, +aw\rangle$. La primera de ellas es una signatura normal, tal y como las hemos visto anteriormente. La signatura $\langle t, +aw\rangle$ representa una función que devuelve aw cuando todos los argumentos son deterministas (es decir, menores o iguales que $\gamma_{t_i}(d)$) y $\gamma_{t_r}(n)$ en caso contrario. Es simplemente un caso particular de $\langle t, aw_1 \ldots aw_m + aw\rangle$ en el que $aw_i = \gamma_{t_r}(n)$ $(i \in \{1..m\})$.

Una función también puede representarse mediante una suspensión. Puede tratarse de una lambda abstracción suspendida $\lambda v.(e, \rho)$, una mínima cota superior suspendida $\boxed{\scriptstyle F}\, [av_1, \ldots, av_m]$ o una suspensión polimórfica de la forma $G\ t'\ t'[t/\beta]\ av$ o $A\ t'[t/\beta]\ t'\ av$.

En la Figura 5.36 se define el operador de mínima cota superior entre valores abstractos. Los casos de valores/signaturas básicos y de las tuplas son simples. En el caso funcional, si ambas funciones están representadas por signaturas, basta con aplicar componente a componente el operador de mínima cota superior. Si una de las funciones es una suspensión, el resultado es una nueva suspensión de mínima cota superior.

5.8.5 Aplicación de una función abstracta

En la Figura 5.37 se muestra la definición de la aplicación de una función abstracta a un argumento abstracto. En el caso de que la función abstracta sea una signatura de la forma $\langle t_1 \to \ldots \to t_m \to t_r,\ aw_1 \ldots aw_m + aw\rangle$ se comprueba si el argumento av' es menor o igual que $\gamma_{t_1}(d)$. Esto se hace, aplicando la Proposición 48, comparando sus signaturas, $\wp_{t_1}(av')$ y $\gamma'_{t_1}(d)$.

$$
\begin{array}{l}
(\lambda v.(e,\rho))\ av = [\![e]\!]'\ \rho[v \mapsto (av, aw, b)] \text{ where } v :: t_v,\quad aw = \wp_{t_v}(av),\quad b = \alpha_{t_v}(av) \\
(\boxed{\mathrm{F}}\,[av_1,\ldots,av_m])\ av' = \bigsqcup\,[av_1\ av',\ldots,av_m\ av'] \\
(G\ t'\ t'[t/\beta]\ av)\ av' = \gamma'_{t_2\,tinst_2}(av\ (\alpha'_{tinst_1\,t_1}(av'))) \\
(A\ t'[t/\beta]\ t'\ av)\ av' = \alpha'_{tinst_2\,t_2}(av\ (\gamma'_{t_1\,tinst_1}(av'))) \\
\langle t_1 \to \ldots \to t_m \to t_r,\ aw_1 \ldots aw_m + aw\rangle\ av'\ (m > 1) \\
\quad \mid \wp_{t_1}(av') \preceq \gamma'_{t_1}(d) = \langle t_2 \to \ldots \to t_m \to t_r,\ aw_2 \ldots aw_m + aw\rangle \\
\quad \mid \text{otherwise} = \langle t_2 \to \ldots \to t_m \to t_r,\ +aw_1\rangle \\
\langle t_1 \to t_r,\ aw_1 + aw\rangle\ av' \\
\quad \mid \wp_{t_1}(av') \preceq \gamma'_{t_1}(d) = aw \\
\quad \mid \text{otherwise} = aw_1 \\
\langle t_1 \to \ldots \to t_m \to t_r,\ +aw\rangle\ av'\ (m > 1) \\
\quad \mid \wp_{t_1}(av') \preceq \gamma'_{t_1}(d) = \langle t_2 \to \ldots \to t_m \to t_r,\ +aw\rangle \\
\quad \mid \text{otherwise} = \gamma'_{t_2 \to \ldots \to t_m \to t_r}(n) \\
\langle t_1 \to t_r,\ +aw\rangle\ av' \\
\quad \mid \wp_{t_1}(av') \preceq \gamma'_{t_1}(d) = aw \\
\quad \mid \text{otherwise} = \gamma'_{t_r}(n)
\end{array}
$$

Figura 5.37: Aplicación de las funciones abstractas

Si se cumple dicha condición, se descarta el primer elemento aw_1 de la signatura y se devuelve $\langle t_2 \to \ldots \to t_m \to t_r, aw_2 \ldots aw_m\ +\ aw\rangle$, ya que estos últimos elementos han sido obtenidos dando al primer argumento un valor $\gamma_{t_1}(d)$. En caso contrario, sólo se puede devolver aw_1 como resultado de la función si el resto de los argumentos son deterministas. Por ello se devuelve una signatura de la forma $\langle t_2 \to \ldots \to t_m \to t_r, +aw_1\rangle$.

Si la función abstracta es una signatura $\langle t_1 \to \ldots \to t_m \to t_r,\ +aw\rangle$, sólo se devuelve el valor aw cuando todos los argumentos son deterministas (es decir, menores o iguales que $\gamma_{t_i}(d)$). Si alguno de ellos no lo es, se devuelve un valor no determinista.

Las suspensiones son solamente una forma de retrasar la evaluación hasta conocer los argumentos. La aplicación de una función suspendida a un argumento provoca la evaluación de la función tanto como sea posible, hasta obtener el resultado de la función, o bien una nueva suspensión. Si se trata de una suspensión $\lambda v.(e, \rho)$, continuamos evaluando el cuerpo e, mediante la función de interpretación $[\![\cdot]\!]'$, presentada en la siguiente sección. El entorno ρ guarda los valores abstractos de todas las variables libres en e excepto v. De modo que añadiremos v al entorno indicando el valor abstracto del argumento.

Si se trata de una mínima cota superior suspendida, primero se aplica cada función al argumento y después se intenta calcular la mínima cota superior de los resultados.

Si se trata de una suspensión de polimorfismo continuamos aplicando la definición de $\gamma_{t'\,tinst}$ o $\alpha_{tinst\,t'}$ (que estaba temporalmente suspendida).

En consecuencia, el algoritmo funciona a saltos suspendiendo y evaluando, una y otra vez.

$$
\begin{array}{l}
[\![\cdot]\!]' :: \mathit{Expr} \rightarrow \mathit{Env} \rightarrow \mathit{AbsVal} \\
[\![e]\!]'\ \rho = \pi_1([\![e]\!]\ \rho) \\
[\![\cdot]\!] :: \mathit{Expr}(\ ) \rightarrow \mathit{Env} \rightarrow (\mathit{AbsVal}, \mathit{Expr}\ \mathit{AbsVal}) \\
[\![v]\!]\ \rho = (av, v@aw) \\
\quad \text{where}\ \ (av, aw, \_) = \rho(v); \\
[\![k]\!]\ \rho = (d, k@d) \\
[\![(x_1, \ldots, x_m)]\!]\ \rho = ((av_1, \ldots, av_m), (x'_1, \ldots, x'_m)@(aw_1, \ldots, aw_m)) \\
\quad \text{where}\ \ (av_i, x'_i) = [\![x_i]\!]\ \rho; \qquad x_i@aw_i = x'_i \\
[\![C\ x_1 \ldots x_m]\!]\ \rho = (aw, C\ x'_1 \ldots x'_m@aw) \qquad \{x_i :: t_i\} \\
\quad \text{where}\ \ (av_i, x'_i) = [\![x_i]\!]\ \rho; \qquad aw = \sqcup_{i=1}^{m}\ b_i; \qquad b_i = \text{if}\ isvar(x_i)\ \text{then}\ (\pi_3(\rho(x_i)))\ \text{else}\ d \\
[\![op\ x_1, \ldots, x_m]\!]\ \rho = (aw, op\ x'_1 \ldots x'_m@aw) \qquad \{op :: top\} \\
\quad \text{where}\ \ (av_i, x'_i) = [\![x_i]\!]\ \rho; \qquad aw = \gamma'_{top}(d)\ aw_1\ \ldots\ aw_m; \\
\qquad\qquad aw_i = \text{if}\ isvar(x_i)\ \text{then}\ (\pi_2(\rho(x_i)))\ \text{else}\ d \\
[\![\lambda v.e]\!]\ \rho = (a, (\lambda v@awv.e')@aw) \qquad \{v :: t_v, (\lambda v.e) :: t\} \\
\quad \text{where}\ \ a = \lambda v.(e, \rho); \qquad aw = \wp_t(a); \qquad awv = \gamma'_{t_v}(n); \\
\qquad\qquad (\_\,, e') = [\![e]\!]\ \rho[v \mapsto (awv, awv, n)] \\
[\![\mathbf{process}\ v \rightarrow e]\!]\ \rho = (a, (\mathbf{process}\ v@awv \rightarrow e')@aw) \qquad \{v :: t_v, (\mathbf{process}\ v \rightarrow e) :: t\} \\
\quad \text{where}\ \ a = \lambda v.(e, \rho); \qquad aw = \wp_t(a); \qquad awv = \gamma'_{t_v}(n); \\
\qquad\qquad (\_\,, e') = [\![e]\!]\ \rho[v \mapsto (awv, awv, n)] \\
[\![e\ x]\!]\ \rho = (a, (e'\ x')@aw) \qquad \{(e\ x) :: t\} \\
\quad \text{where}\ \ (ae, e') = [\![e]\!]\ \rho; \qquad (ax, x') = [\![x]\!]\ \rho; \qquad a = ae\ ax; \qquad aw = \wp_t(a) \\
[\![v\#x]\!]\ \rho = (a, (v'\#x')@aw) \qquad \{(v\#x) :: t\} \\
\quad \text{where}\ \ (av, v') = [\![v]\!]\ \rho; \qquad (ax, x') = [\![x]\!]\ \rho; \qquad a = av\ ax; \qquad aw = \wp_t(a) \\
[\![merge]\!]\ \rho = (aw, merge@aw) \qquad \{merge :: t_{merge}\} \\
\quad \text{where}\ aw = \gamma'_{t_{merge}}(n)
\end{array}
$$

Figura 5.38: Algoritmo de anotación de expresiones (I)

5.8.6 El algoritmo

En el algoritmo hay dos funciones de interpretación diferentes, $[\![\cdot]\!]'$ y $[\![\cdot]\!]$. Dada una expresión e sin anotar y un entorno ρ, $[\![e]\!]\ \rho$ devuelve un par $(av, e'@aw)$ en el que av es el valor abstracto de e, e' es la expresión e en la que todas sus subexpresiones han sido anotadas, y aw es la anotación externa de e. Mientras que las anotaciones de las expresiones son siempre signaturas, se pretende que la primera componente del par mantenga la mayor cantidad de información posible, excepto en el cálculo del punto fijo, donde será sustituida por la signatura correspondiente.

En las Figuras 5.38 y 5.39 se presenta el algoritmo que calcula $[\![\cdot]\!]$ escrito en pseudocódigo. El algoritmo para $[\![\cdot]\!]'$ es muy similar. Dada una expresión e y un entorno ρ, $[\![e]\!]'\ \rho$ se limita a devolver el valor abstracto de la expresión, no calculando ninguno de los valores auxiliares computados por $[\![e]\!]\ \rho$. En la implementación realizada existe un único algoritmo para $[\![\cdot]\!]$ y $[\![\cdot]\!]'$. La evaluación perezosa evita que se realicen cómputos no requeridos.

En un entorno ρ, a cada variable de programa v se le asocia una terna

$$
\begin{array}{l}
[\![\textbf{let}\ bind\ \textbf{in}\ e]\!]\ \rho = (a, (\textbf{let}\ bind'\ \textbf{in}\ e')@aw) \qquad \{e :: t\} \\
\qquad \text{where}\ \ (\rho', bind') = [\![bind]\!]_B\ \rho; \qquad (a, e') = [\![e]\!]\ \rho' \qquad e''@aw = e' \\
[\![\textbf{case}\ e\ \textbf{of}\ (v_1, \ldots, v_m) \to e']\!]\ \rho = \\
\qquad (a, (\textbf{case}\ (e_1@awe)\ \textbf{of}\ (v'_1, \ldots, v'_m) \to (e'_1@aw))@aw) \quad \{v_i :: t_i\} \\
\qquad \text{where}\ \ (ae, e1@awe) = [\![e]\!]\ \rho; \quad aw_i = \pi_i(awe); \quad v'_i = v_i@aw_i; \quad av_i = \pi_i(ae); \\
\qquad\qquad b_i = \alpha_{t_i}(av_i); \qquad (a, e'_1@aw) = [\![e']\!]\ \rho\overline{[v_i \mapsto (av_i, aw_i, b_i)]} \\
[\![\textbf{case}\ e\ \textbf{of}\ \overline{alt_i}]\!]\ \rho = (av, (\textbf{case}\ e'\ \textbf{of}\ \overline{alt'_i})@aw) \qquad \{\textbf{case}\ e\ \textbf{of}\ \overline{alt_i} :: t\} \\
\qquad \text{where}\ \ (ae, e') = [\![e]\!]\ \rho; \qquad (av_i, alt'_i) = [\![alt_i]\!]_A\ ae\ \rho; \\
\qquad\qquad C_i\ v_{i1} \ldots v_{im_i} \to (e_i@wa_i) = alt'_i; \\
\qquad\qquad aw = \text{if}\ ae = n\ \text{then}\ \gamma'_t(n)\ \text{else}\ \sqcup_{i=1}^m\ aw_i; \\
\qquad\qquad av = \text{if}\ ae = n\ \text{then}\ \gamma'_t(n)\ \text{else}\ \sqcup_{i=1}^m av_i \\
[\![\Lambda\beta.e]\!]\ \rho = (av, (\Lambda\beta.e')@awv) \\
\qquad \text{where}\ \ (av, e') = [\![e]\!]\ \rho; \qquad e''@awv = e' \\
[\![(e\ t)]\!]\ \rho = (av, e'\ t@aw) \qquad \{e :: (\forall\beta.t'), tinst = t'[t/\beta]\} \\
\qquad \text{where}\ \ (av', e') = [\![e]\!]\ \rho; \qquad av = \gamma'_{t'\,tinst}(av'); \qquad aw = \wp_{tinst}(av) \\
[\![v = e]\!]_B\ \rho = (\rho[v \mapsto (av, aw, b)], v@aw = e'@aw) \\
\qquad \text{where}\ \ (av, e'@aw) = [\![e]\!]\ \rho; \qquad b = \alpha_{t_v}(av) \\
[\![\textbf{rec}\ \overline{v_i = e_i}]\!]_B\ \rho = (\rho_{fix}, \textbf{rec}\ \overline{v'_i = e''_i}) \qquad \{v_i :: t_i\} \\
\quad \text{where}\ \ \rho_{fix} = fix\ f\ init; \qquad init = \rho\overline{[v_i \mapsto (aw_i, aw_i, d)]}; \qquad aw_i = \gamma'_{t_i}(d) \\
\qquad\quad f\rho' = \rho'\overline{[v_i \mapsto (aw'_i, aw'_i, b_i)]}\ \ \text{where}\ av'_i = [\![e_i]\!]'\ \rho'; \quad aw'_i = \wp_{t_i}(av'_i); \quad b_i = \alpha_{t_i}(av'_i) \\
\qquad\quad (\_\ , e''_i) = [\![e_i]\!]\ \rho_{fix}; \qquad (\_, aw_i, \_) = \rho_{fix}(v_i); \quad v'_i = v_i@aw_i \\
[\![C\ v_1 \ldots v_m \to e]\!]_A\ avd\ \rho = (av, C\ v'_1 \ldots v'_m \to e') \qquad \{v_i :: t_i\} \\
\qquad\quad \text{where}\ \ aw_i = \gamma'_{t_i}(avd); \qquad (av, e') = [\![e]\!]\ \rho\overline{[v_i \mapsto (aw_i, aw_i, avd)]}; \qquad v'_i = v_i@aw_i \\
[\![v \to e]\!]_A\ avd\ \rho = (av, v' \to e') \\
\qquad \text{where}\ \ (av, e') = [\![e]\!]\ \rho[v \mapsto (avd, avd, avd)]; \qquad v' = v@avd
\end{array}
$$

Figura 5.39: Algoritmo de anotación de expresiones (II)

(av, aw, b) de valores abstractos. La primera componente av es el valor abstracto de la variable, aw es la signatura correspondiente $\wp_t(av)$, y b es el valor básico obtenido por $\alpha_t(av)$. Puesto que estos tres valores pueden ser usados varias veces a lo largo de la interpretación, se calculan solamente una vez, cuando se liga la variable, y después se usan allí donde sea necesario.

Describimos ahora la interpretación. La primera componente del resultado av es la definición de $[\![\cdot]\!]_3^{\mathcal{W}}$, por lo que nos limitamos a explicar la parte correspondiente a las anotaciones. En general, para anotar una expresión se anotan primero sus subexpresiones y después se calcula la anotación de la expresión completa mediante muestreo del valor abstracto (primera componente del resultado del algoritmo) de la expresión. Pero, en ocasiones es posible construir la anotación de la expresión completa partiendo sólo de las anotaciones de las subexpresiones, lo cual resulta ser más eficiente.

La anotación de una variable es simple, puesto que su signatura se encuentra en el entorno. Un literal se anota con un valor determinista básico d. Para anotar una tupla, primero anotamos sus componentes; la anotación de la tupla es la tupla formada por dichas anotaciones. Para anotar un valor construido, de nuevo anotamos primero sus componentes. Aquí usamos el hecho de que el entorno contiene el valor básico de cada componente (cuando se trata de una variable), con lo que basta tomar la cota superior de todos ellos para obtener la anotación de la expresión completa.

Para anotar un operador primitivo, anotamos primero cada uno de sus argumentos. Puesto que el valor abstracto del operador primitivo es una signatura $\gamma'_{top}(d)$, el resultado de la aplicación a varios argumentos será directamente una signatura, por lo que en este caso no es necesario ningún muestreo adicional. Para realizar la necesaria aplicación de la signatura a los argumentos hace falta en principio el muestreo de los argumentos, como se vió la Figura 5.37; pero en este punto el entorno nos facilita las signaturas de los argumentos, por lo que podemos proporcionárselos directamente a la aplicación.

Para anotar una lambda abstracción, se anota primero su cuerpo. Para poder hacerlo, debemos dar un valor abstracto al argumento. Dicho valor abstracto depende de los argumentos a los que se aplique la función, pero ahora estamos anotando la función en sí y no conocemos las aplicaciones que aparecerán en el programa. Así que hemos de dar a cada argumento el valor pesimista $\gamma'_{t_v}(n)$.

En una aplicación $e\ x$, primero se anotan las subexpresiones e y x, y después se anota la expresión mediante muestreo del valor abstracto resultante de la aplicación. Las abstracciones y concreciones de proceso son semejantes a las lambda abstracciones y aplicaciones de funciones.

Una expresión **let** recibe la misma anotación que su expresión principal. Si la ligadura $v = e$ es no recursiva, la variable v se añade al entorno. Si es recursiva, para obtener el valor abstracto de cada lado derecho usamos $[\![\cdot]\!]'$ en el cálculo del punto fijo y aplicamos el muestreo para obtener la signatura. Si usáramos $[\![\cdot]\!]$ en su lugar, en cada iteración se anotarían innecesariamente todas las e_i. De esta forma, solamente anotamos los lados derechos de las ligaduras una vez calculado el punto fijo.

Una expresión **case** de tuplas recibe la misma anotación que su alternativa. Cada variable del lado izquierdo de la alternativa se anota con la correspondiente componente de la anotación del discriminante.

En un **case** algebraico, tenemos que considerar que el valor abstracto puede ser una suspensión de mínima cota superior de los valores abstractos de las alternativas. El muestreo de esta suspensión es la mínima cota superior de los muestreos de los valores abstractos de las alternativas, de modo que basta con calcular la mínima cota superior de las anotaciones de las alternativas. Las variables de los lados izquierdos se anotan de acuerdo con el valor del discriminante. Si aparece una alternativa por defecto, la variable del lado izquierdo se anota con el valor abstracto del discriminante.

Los **case** primitivos son semejantes a los algebraicos cuando los constructores no tienen argumentos, por lo que no describimos este caso.

Una abstracción de tipo recibe la misma anotación que su cuerpo. Una aplicación de tipo recibe como anotación el muestreo del correspondiente valor abstracto.

5.8.7 Coste del análisis

Pretendemos estimar el coste de las funciones del algoritmo en términos del tamaño del texto original del programa, es decir, de las expresiones del programa y del de sus tipos, y no en términos de sus valores abstractos.

Analizar el coste del algoritmo de interpretación se ha revelado como una tarea ardua, debido a que muchas de las funciones implicadas—en particular $[\![\cdot]\!]$, $[\![\cdot]\!]'$, la aplicación abstracta, $\gamma'_{t\,tinst}$, $\wp_t$, $\alpha'_{tinst\,t'}$, y α'_t— contienen una fuerte recursión mutua y también a la creación de suspensiones. La interpretación de una expresión se detiene tan pronto como se encuentra una lambda abstracción o una abstracción de proceso (usaremos aquí la palabra "lambda" en ambas ocasiones). La interpretación se reinicia cuando la lambda se aplica a un argumento, pero entonces puede aparecer una nueva lambda con lo que la interpretación se suspendería de nuevo. Este proceso de paradas y arranques continúa hasta que la lambda se aplica a todos los argumentos disponibles.

Afortunadamente, hay otras funciones más simples cuyo coste se puede calcular directamente en términos del tamaño de los tipos implicados. Por ejemplo, una comparación entre dos signaturas en S_t, o el cálculo de su mínima cota superior se puede llevar a cabo en $O(\mathcal{H}_t)$. Luego la mínima cota superior de m valores abstractos de tipo t está en $O((m-1)\ \mathcal{H}_t)$. El coste de $\gamma'_t(b)$ está en $O(m+\mathcal{H}_{t_r})$, siendo $m = nArgs(t)$ y $t_r = rType(t)$. Por su parte, para analizar el coste de las funciones principales de interpretación definimos dos funciones $s, s' : Expr \to Int$ que proporcionan, respectivamente, los "tamaños" de una expresión e al ser interpretada por $[\![\cdot]\!]$ y por $[\![\cdot]\!]'$. Escribimos entonces que $[\![e]\!]'\ \rho \in O(s'(e))$ y que $[\![e]\!]\ \rho \in O(s(e))$. La definición completa de s y s' se da en las Figuras 5.41 y 5.40. Podemos observar que casi siempre $s(e)$ y $s'(e)$ son lineales con respecto al tamaño sintáctico de e, incluyendo el tamaño de los tipos implicados. Pero hay tres excepciones a dicha linealidad:

Aplicaciones: Interpretar una ligadura lambda con $[\![\cdot]\!]'$ cuesta $O(1)$ puesto que inmediatamente se crea una suspensión. Pero el cuerpo de la lambda se interpretará tantas veces como se aplique en el texto. La noción de cuerpo de una lambda es compleja. Implica la sustitución de las variables libres por su definición siempre que las variables aparezcan aplicadas en el cuerpo. Suponiendo que la noción de cuerpo e_λ de una lambda está bien definido (ver función *body* en la Figura 5.45), el algoritmo tiene un coste $O(s'(e_\lambda))$ cada vez que se aplica la lambda.

Muestreo de una función: El muestreo se usa constantemente en $[\![\cdot]\!]$ para anotar expresiones con signaturas, así como por $[\![\cdot]\!]$ y $[\![\cdot]\!]'$ al calcular los puntos fijos. El coste de $\wp_t(e)$ corresponde a $m+1$ aplicaciones abstractas, cada una a m parámetros, siendo $m = nArgs(t)$. De modo que siendo e_λ el cuerpo de e, el coste total estará en $O((m+1)\ s'(e_\lambda))$.

Puntos fijos: Supongamos que hay una única ligadura recursiva $v = e$ de tipo funcional t, donde $m = nArgs(t)$, $t_r = rType(t)$, y e_λ es el cuerpo de e. El algoritmo $[\![\cdot]\!]'$ calculará el mínimo punto fijo tras un máximo de $\mathcal{H}_t = (m+1)\ \mathcal{H}_{t_r}$ iteraciones. Recordemos que $\mathcal{H}_t$ es la altura del dominio de signaturas. En cada iteración, se ha de calcular la signatura de e, con lo que el coste del cálculo de los puntos fijos está en $O(m^2\ \mathcal{H}_{t_r}\ s'(e_\lambda))$. El algoritmo de anotación $[\![\cdot]\!]$ añadirá a este coste el de anotar completamente e, lo que implica m muestreos más, cada uno de ellos con un parámetro menos. En total, el proceso de anotación costará $O(m^2\ s'(e_\lambda))$.

En resumen, podemos decir que el algoritmo de interpretación/anotación es lineal con respecto a e excepto en las aplicaciones—donde la interpretación del cuerpo debe multiplicarse por el número de aplicaciones—, en las anotaciones de las funciones–donde es cuadrático debido al muestreo—y en los puntos fijos donde puede alcanzar un coste cúbico.

Se ha ejecutado el algoritmo con definiciones de esqueletos Edén típicos. Para ficheros de 3.000 líneas netas y 80 segundos de compilación en una SUN 4 250 MHz Ultra Sparc-II, el análisis añade un coste en el rango de 0.5 a 1 segundos, es decir, supone menos de un 1% de sobrecarga.

Funciones de coste

Definiremos ahora con detalle las funciones s y s', así como todas las funciones auxiliares necesarias para hacerlo.

Coste de $[\![e]\!]'\ \rho$. La función s' describe el coste de obtener el valor abstracto de una expresión, ver Figura 5.40. El coste de obtener el valor abstracto de un literal o una variable es constante, ya que en el primer caso es d y en el segundo se obtiene directamente del entorno.

Puesto que tanto las tuplas como los demás valores construidos sólo contienen literales o variables, el coste de obtener su valor abstracto es m, siendo m el número de componentes.

El valor abstracto de un operador primitivo $op :: top$ es una signatura, por lo que solamente hace falta comparar el valor abstracto de cada argumento $x_i :: t_i$ con $\gamma_{t_i}(d)$. Esto se puede hacer a través de las correspondientes signaturas. Puesto que los operadores primitivos se aplican sobre variables o literales, tenemos directamente disponibles sus signaturas. Por tanto tenemos que calcular $\gamma'_{t_i}(d)$, con coste $m_i + \mathcal{H}_{t_{i_r}}$ para cada $i \in \{1..m\}$ donde

$s'(k) = s'(v) = 1$
$s'(x_1, \ldots x_m) = m$
$s'(C\ x_1 \ldots x_m) = m$
$s'(op\ x_1 \ldots x_m) = \sum_i \mathcal{H}_{t_i} + \mathcal{H}_{t_r}$
$\quad \{op :: top, m = nArgs(top), [t_1, \ldots, t_m] = aTypes(top), t_r = rType(top), i \in \{1..m\}\}$
$s'(\lambda v.e) = 1$
$s'(e\ x) = s'(e) + s'(x) + (sum\ (map\ s'\ es) + c)$
$\quad$ donde $(es, c) = body\ e\ x$
$s'(\mathbf{let}\ v = e\ \mathbf{in}\ e') = s'(e) + s'(e') + p_t(e) + a_t(e) \qquad \{e :: t\}$
$s'(\mathbf{let\ rec}\ \overline{v_i = e_i}\ \mathbf{in}\ e') = niter * \sum_i(s'(e_i) + p_{t_i}(e_i) + a_{t_i}(e_i) + niter) + s'(e')$
$\quad$ donde $niter = \sum_i \mathcal{H}_{t_i}$
$\qquad \{i \in \{1..m\}, e_i :: t_i\}$
$s'(\mathbf{case}\ e\ \mathbf{of}\ (v_1, \ldots v_m) \rightarrow e') = s'(e) + s'(e') + (p_t(e) + \sum_i A_{t_i})$
$\quad \{i \in \{1..m\}, v_i :: t_i, e :: t\}$
$s'(\mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{v_{ij}} \rightarrow e_i}) = s'(e) + \sum_{i,j}(m_{ij} + \mathcal{H}_{t_{rij}}) + \sum_i s'(e_i) + (m-1)\ \mathcal{H}_t$
$\qquad \{i \in \{1..m\}, j \in \{1..m_i\}, e_i :: t, v_{ij} :: t_{ij}, t_{rij} = rType(t_{ij}), m_{ij} = nArgs(t_{ij})\}$
$s'(\Lambda\beta.e) = s'(e)$
$s'(e\ t) = s'(e) + \Gamma^{\beta}_{t'\ t}$
$\qquad \{e :: \forall\beta.t'\}$

Figura 5.40: Coste de $[\![e]\!]'\ \rho$

$s(k) = s(v) = 1$
$s(x_1, \ldots x_m) = m$
$s(C\ x_1 \ldots x_m) = m$
$s(op\ x_1 \ldots x_m) = \sum_i \mathcal{H}_{t_i} + \mathcal{H}_{t_r}$
$\quad \{op :: top, m = nArgs(top), [t_1, \ldots, t_m] = aTypes(top), t_r = rType(top), i \in \{1..m\}\}$
$s(\lambda v.e) = s(e) + p_t(\lambda v.e) + (m_1 + \mathcal{H}_{t_{1_r}})$
$\quad \{v :: t_1, t_{1r} = rType(t_1), m_1 = nArgs(t_1), \lambda v.e :: t\}$
$s(e\ x) = s(e) + s(x) + (sum\ (map\ s'\ es) + c) + p_t(e\ x) \qquad \{e\ x :: t\}$
$\quad$ donde $(es, c) = body\ e\ x$
$s(\mathbf{let}\ v = e\ \mathbf{in}\ e') = s(e) + s(e') + a_t(e)$
$s(\mathbf{let\ rec}\ \overline{v_i = e_i}\ \mathbf{in}\ e') = niter * \sum_i(s'(e_i) + p_{t_i}(e_i) + a_{t_i}(e_i) + niter) + \sum_i s(e_i) + s(e')$
$\quad$ donde $niter = \sum_i \mathcal{H}_{t_i}$
$\qquad \{i \in \{1..m\}, e_i :: t_i\}$
$s(\mathbf{case}\ e\ \mathbf{of}\ (v_1, \ldots v_m) \rightarrow e') = s(e) + s(e') + (p_t(e) + \sum_i A_{t_i})$
$\quad \{i \in \{1..m\}, v_i :: t_i, e :: t\}$
$s(\mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{v_{ij}} \rightarrow e_i}) = s(e) + \sum_{i,j}(m_{ij} + \mathcal{H}_{t_{rij}}) + \sum_i s(e_i) + (m-1)\ \mathcal{H}_t$
$\qquad \{i \in \{1..m\}, j \in \{1..m_i\}, e_i :: t, v_{ij} :: t_{ij}, m_{ij} = nArgs(t_{ij}), t_{rij} = rType(t_{ij})\}$
$s(\Lambda\beta.e) = s(e)$
$s(e\ t) = s(e) + \Gamma^{\beta}_{t'\ t} + p_{tinst}(e\ t)$
$\qquad \{e :: \forall\beta.t', e\ t :: tinst\}$

Figura 5.41: Coste de $[\![e]\!]\ \rho$

$m = nArgs(top)$, $m_i = nArgs(t_i)$ y $t_{ir} = rType(t_i)$. Sin embargo, el coste de las comparaciones supera al anterior pues es $\mathcal{H}_{t_i}$ para cada $i \in \{1..m\}$. Si todos los argumentos son deterministas o solamente uno de ellos es no determinista, devolveremos una de las componentes de la signatura. Si hay más de uno no determinista, habrá que devolver un valor no determinista $\gamma'_{t_r}(n)$ donde $t_r = rType(top)$ (tiene 0 argumentos ya que las aplicaciones de operadores primitivos son saturadas), lo que cuesta $\mathcal{H}_{t_r}$.

Para obtener el valor abstracto de una lambda, basta con construir la suspensión, por lo que el coste es constante.

Podemos considerar la evaluación de una aplicación $e\ x$ dividida en dos fases. En la primera se evalúan e y x hasta llegar a un valor abstracto, lo cual cuesta respectivamente $s'(e)$ y $s'(x)$. A continuación ha de evaluarse la propia aplicación. Para calcular su coste se utiliza la función *body*, definida en la Figura 5.45. Dada una expresión e (de tipo funcional) y su argumento, *body* devuelve una lista de expresiones es y un entero c. Los elementos de la lista son aquellas expresiones cuyos valores abstractos son el resultado de la aplicación abstracta. Puede haber más de una debido a la presencia de expresiones **case**. Por tanto hemos de añadir sus costes $sum\ (map\ s'\ es)$ al coste de la aplicación. El entero c representa algunos costes adicionales laterales, detallados más abajo, que también deben ser añadidos al coste de la aplicación.

Para obtener el valor abstracto de una expresión **let** $x = e$ **in** e', donde $e :: t$, tenemos que calcular el valor abstracto de e y e', lo que cuesta respectivamente $s'(e)$ y $s'(e')$. Adicionalmente, se calculan la signatura de e y su correspondiente valor abstracto básico para añadirlos al entorno. El coste de obtener una signatura viene dado por $p_t(e)$, definido en la Figura 5.43 y que se desarrollará más adelante (ver página 201). El coste de calcular el correspondiente valor básico (mediante la aplicación de la función α_t) viene dado por $a_t(e)$, definida más abajo (ver página 204).

En una expresión **let rec** $\overline{v_i = e_i}$ **in** e' debemos multiplicar el coste de hallar el valor abstracto de cada una de las ligaduras por el número de iteraciones. En el peor de los casos, en cada vuelta se modifica la signatura de exactamente una de las ligaduras y ésta solamente en una componente. Por tanto, en el peor de los casos el número de iteraciones $niter$ es la suma de las profundidades de los dominios signatura de los tipos de las ligaduras, es decir, $\sum_i \mathcal{H}_{t_i}$ donde $i \in \{1..m\}$ y m es el número de ligaduras. Además, al final de cada iteración han de compararse las signaturas para saber si ya ha finalizado el cálculo del punto fijo, lo cual supone un coste proporcional a la suma de los tamaños de las signaturas de las ligaduras, es decir, $niter$ (el tamaño de la signatura viene dado por la profundidad del correspondiente dominio de signaturas). Finalmente añadimos el coste de hallar el valor abstracto de e', $s'(e')$.

Para calcular el valor abstracto de una expresión **case** e **of** $(v_1, \ldots v_m) \rightarrow e'$ hace falta calcular el valor abstracto de e y de e', lo cual cuesta respectiva-

$$
\begin{array}{l}
\Gamma^{\beta}_{K,t} = \Gamma^{\beta}_{T\ t_1 \ldots t_m,t} = \Gamma^{\beta}_{\beta',t} = 1 \\
\Gamma^{\beta}_{\beta,t} = nArgs(t) + \mathcal{H}_{rType(t)} \\
\Gamma^{\beta}_{(t_1,\ldots,t_m),t} = \sum_{i=1}^{m} \Gamma^{\beta}_{t_i,t} \\
\Gamma^{\beta}_{t_1 \rightarrow t_2,t} = 1 \\
\Gamma^{\beta}_{Process\ t_1\ t_2,t} = 1 \\
\Gamma^{\beta}_{\forall\beta'.t',t} = \Gamma^{\beta}_{t',t}
\end{array}
$$

Figura 5.42: Coste de $\gamma'_{t'\,tinst}$

mente $s'(e)$ y $s'(e')$. Adicionalmente hay que añadir al entorno las variables v_i, lo cual implica calcular su signatura y sus valores básicos correspondientes. Puesto que la signatura de una tupla es la tupla de las signaturas, esto es equivalente a calcular la signatura de e, y aplanar cada una de sus componentes para obtener los correspondientes valores básicos. Por ello el coste es $p_t(e) + \sum_i \mathcal{A}_{t_i}$, donde $\mathcal{A}_t$ representa el coste de aplanar una signatura en S_t hasta obtener un valor básico (ver Figura 5.46).

Para calcular el valor abstracto de una expresión **case** e **of** $\overline{C_i\ \overline{v_{ij}} \rightarrow e_i}$ con $e_i :: t$ y $v_{ij} :: t_{ij}$, hemos de calcular el valor abstracto de e y de cada una de las alternativas e_i, lo que cuesta respectivamente $s'(e)$ y $s'(e_i)$. Si el valor abstracto del discriminante es determinista, hemos de añadir al entorno las variables v_{ij} con un valor determinista $\gamma'_{t_{ij}}(d)$, lo cual cuesta $\sum_{i,j}(m_{ij} + \mathcal{H}_{t_{rij}})$ donde $m_{ij} = nArgs(t_{ij})$ y $t_{rij} = rType(t_{ij})$. Finalmente, se calcula la mínima cota superior de las alternativas, lo que cuesta $(m{-}1)\ \mathcal{H}_t$ donde m es el número de alternativas. Sin embargo, si el valor abstracto del discriminante es no determinista, devolvemos directamente $\gamma'_t(n)$, lo cual es menos costoso que en el caso anterior.

El coste de obtener el valor abstracto de $\Lambda\beta.e$ es el coste de obtener el valor abstracto de e, $s'(e)$.

Y finalmente, el coste de obtener el valor abstracto de una aplicación de tipos $e\ t$ corresponde al coste de obtener el valor abstracto de e, $s'(e)$, más el coste de calcular $\gamma'_{t'\,tinst}$. Dado un tipo polimórfico $\forall\beta.t'$, otro tipo t, y un valor abstracto $av \in D_{2}{}'_{t}$, el coste de $\gamma'_{t'\,tinst}(av)$ (definido en la Figura 5.42), donde $tinst = t'[t/\beta]$, está en $O(\Gamma^{\beta}_{t'\ t})$.

Coste de $[\![e]\!]\ \rho$. La función s describe el coste de obtener el valor abstracto de una expresión y de anotar la expresión con información de no determinismo (ver Figura 5.41).

Si la expresión es un átomo, una tupla, un valor construido o la aplicación de un operador primitivo, el coste coincide con el dado por s', ya que la anotación de estas expresiones no implica realizar ningún cálculo adicional, sino solamente extraer las signaturas de los entornos. En la aplicación de un constructor hay que calcular la mínima cota superior de m valores básicos, lo que cuesta m.

Para anotar una lambda abstracción $\lambda v.e :: t$, hemos de anotar la variable $v :: t_1$ con el valor pesimista $\gamma'_{t_1}(n)$, lo que cuesta $m_1 + \mathcal{H}_{t_{1r}}$ donde $m_1 = nArgs(t_1)$ y $t_{1r} = rType(t_1)$. También ha de anotarse el cuerpo e, lo que cuesta $s(e)$. Finalmente, la anotación de la lambda se obtiene mediante muestreo de su valor abstracto, cuyo coste es $p_t(\lambda v.e)$.

Para anotar una expresión $e\ x :: t$ han de anotarse e y x, lo que cuesta respectivamente $s(e)$ y $s(x)$. Para calcular la anotación de la expresión, primero se ha de calcular el valor abstracto de la aplicación, cuyo coste se obtiene al igual que antes usando la función *body*. Después se obtiene la signatura mediante muestreo de dicho valor abstracto, lo que cuesta $p_t(e\ x)$.

Para anotar una expresión **let** $x = e$ **in** e' donde $e :: t$, tenemos que anotar e y e', lo que cuesta respectivamente $s(e)$ y $s(e')$. A diferencia de antes, no hace falta calcular la signatura de e, puesto que este coste ya se ha incluido al anotar e en $s(e)$. Sí hace falta en cambio añadir el correspondiente valor básico, lo que cuesta $a_t(e)$.

En una expresión **let rec** $\overline{v_i = e_i}$ **in** e' se ha de tener en cuenta que durante el cálculo del punto fijo no anotamos los lados derechos de las ligaduras, sino que solamente calculamos los valores abstractos de las mismas. Por ello, el coste correspondiente al cálculo del punto fijo sigue siendo $niter * \sum_i (s'(e_i) + p_{t_i}(e_i) + a_{t_i}(e_i) + niter)$ donde $niter = \sum_i \mathcal{H}_{t_i}$. Una vez calculado el punto fijo, procedemos a anotar los lados derechos, con coste $\sum_i s(e_i)$ y la expresión principal, con coste $s(e')$.

Los costes para las expresiones **case** y la abstracción de tipo son análogos a los costes proporcionados para s' siendo la única diferencia el hecho de que las apariciones recursivas son s en lugar de s'.

Finalmente, para anotar una aplicación de tipo $e\ t$, se ha de anotar primero e, con coste $s(e)$. La anotación de la expresión se obtiene calculando primero el valor abstracto, con coste $\Gamma^{\beta}_{t'\ t}$ para después obtener la signatura mediante muestreo, lo que cuesta $p_{tinst}(e\ t)$.

Detallemos ahora algunas de las funciones de coste que han ido apareciendo en las definiciones de s y s'.

Coste de obtener una signatura. En la Figura 5.43 se muestra el coste de obtener una signatura, es decir, de llevar a cabo un muestreo completo.

Explicamos el caso funcional y el caso tupla, ya que los demás son sencillos. Consideremos una expresión $e :: t$, donde $t = t_1 \to t_2$ o $t = Process\ t_1\ t_2$, con valor abstracto av. El coste de obtener la signatura correspondiente a av se denota por $p_t(e)$. Si av es una signatura, el coste de su muestreo es $\mathcal{H}_t$ (lo que se tarda en saber que se trata de una signatura). Si

$$
\begin{array}{l}
p_K(e) = p_{T\ t_1 \ldots t_m}(e) = p_\beta(e) = 1 \\
p_{(t_1,\ldots,t_m)}(e) = \sum_{i,j} p_{t_i}(e_{ij}) + \sum_i n_i\ \mathcal{H}_{t_i} \\
\quad \text{donde} \quad (es, cs) = comps(e), i \in \{1..m\}, n_i = cs!!(i-1), \\
\qquad\qquad\quad j \in \{1..(length\ es_i)\}, e_{ij} = (es!!(i-1))!!(j-1) \\
p_t(e) = \mathcal{H}_t + \sum_{i=1}^{m}(m_i + \mathcal{H}_{t_{r\,i}}) + (m+1)\ \ oneap(e) \\
\quad \{t = t_1 \to t_2, Process\ t_1\ t_2\} \\
p_{\forall\beta.t}(e) = p_t(e)
\end{array}
$$

Figura 5.43: Coste de obtener una signatura para un valor abstracto

$$
\begin{array}{l}
comps :: Expr \to ([[Expr]], [Int]) \\
comps(x_1, \ldots, x_m) = ([[x_1], \ldots, [x_m]], [0, \ldots, 0]) \\
comps(\textbf{let}\ bs\ \textbf{in}\ e) = comps\ (e) \\
comps(\textbf{case}\ e\ \textbf{of}\ \underline{(v_1, \ldots, v_m)} \to e') = comps(e') \\
comps(\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i}) = join(map\ comps\ [e_1, \ldots, e_m]) \quad \{i \in \{1..m\}\} \\
comps(e\ x) = join(map\ comps\ es) \\
\quad \text{donde} \quad (es, \_) = body\ e\ x \\
comps(v) = comps(e) \qquad \{\text{ligada por un } \textbf{let} \text{ no recursivo}\}\ \{v = e\} \\
comps(v) = ([[\ ], \ldots [\ ]], [1, \ldots, 1]) \quad \{\text{ligada por lambda, } \textbf{let} \text{ recursivo, } \textbf{case} \text{ alg.}\} \\
comps(v) = (ves, ncs) \qquad \{\text{ligada por } \textbf{case} \text{ de tupla } k\text{-ésima, discriminante } e\} \\
\quad \text{donde} \quad (es, cs) = comps(e), es_k = es!!(k-1), c_k = cs!!(k-1) \\
\qquad\qquad\ (ves, vcs) = join(map\ comps\ es_k), ncs = zipWith\ (+)\ vcs\ [c_k, \ldots, c_k] \\
\\
join\ [] = ([[\ ], \ldots, [\ ]], [0, \ldots, 0]) \\
join\ xs = foldl1\ g\ xs \\
g\ (xs_1, cs_1)\ (xs_2, cs_2) = (zipWith\ (+\!\!+)\ xs_1\ xs_2, zipWith\ (+)\ cs_1\ cs_2)
\end{array}
$$

Figura 5.44: Definición de la función *comps* no lo es, entonces denotemos por $oneap(e)$ (ver Figura 5.47 en la página 206) al coste de llevar a cabo la aplicación de la función sobre una de las combinaciones de argumentos en que consiste el muestreo. Entonces la obtención de la signatura incluye el cálculo de las combinaciones de argumentos formadas por $\gamma'_{t_i}(d)$ y $\gamma'_{t_i}(n)$, lo que cuesta $\sum_i (m_i + \mathcal{H}_{t_{r_i}})$, y las aplicaciones a dichas $(m+1)$ combinaciones, lo que cuesta $(m+1)$ $oneap(e)$. Luego $p_t(e) = \mathcal{H}_t + \sum_i (m_i + \mathcal{H}_{t_{r_i}}) + (m+1)$ $oneap(e)$ representa el coste del muestreo completo, donde $i \in \{1..m\}, m = nArgs(t)$, $[t_1, \ldots, t_m] = aTypes(t)$, $t_r = rType(t)$, $m_i = nArgs(t_i)$ y $t_{ri} = rType(t_i)$.

Para conocer el coste de obtener una signatura para una tupla, definimos la función auxiliar *comps*, mostrada en la Figura 5.44. Dada una expresión e de tipo tupla $(t_1, \ldots, t_m)$, con valor abstracto $(av_1, \ldots, av_m)$, buscamos la lista de expresiones e_i que dan lugar a los valores abstractos de cada componente av_i. Puesto que tenemos expresiones **case**, podría haber varias posibilidades para cada componente, por lo que *comps* devolverá un par de listas. La primera, *es*, contiene las listas de expresiones posibles para cada componente. La segunda, *cs*, es una lista de enteros cuyo significado explicamos a continuación. Cuando una variable está ligada por una lambda, un **case** algebraico o una expresión **let** recursiva, su valor abstracto es una signatura, por lo que en tal caso, no hay ninguna expresión correspondiente y contamos cuántas variables hay en esta situación para cada componente.

El coste de obtener la signatura para una expresión de tipo tupla e consta por un lado de los costes de obtener las signaturas de todas las posibles expresiones componentes $\sum_{i,j} p_{t_i}(e_{ij})$, con $e_{ij} = (es!!(i-1))!!(j-1)$ (donde $xs!!i$ denota el elemento i-ésimo de la lista xs). Adicionalmente, los n_i representan el número de veces en que el valor abstracto de la componente es ya una signatura, lo que nos cuesta comprobar un tiempo $\mathcal{H}_{t_i}$, por lo hemos de añadir un sumando $\sum_i n_i \; \mathcal{H}_{t_i}$.

La función *body*. La función *body* se define utilizando una función auxiliar *lbody* que toma como argumentos una expresión funcional e, una lista de argumentos xs y un coste acumulado c (ver Figura 5.45). Esta función busca el cuerpo (o cuerpos) de la función una vez aplicada a los argumentos xs y acumula en c los costes adicionales de llevar a cabo las aplicaciones. Por ejemplo, en el caso de las expresiones **case**, el coste de obtener a cabo la correspondiente mínima cota superior.

Si la expresión es una lambda $\lambda v.e$ y aún quedan argumentos por aplicar, llevamos a cabo la aplicación sustituyendo en e la variable v por x. Esto no implica ningún coste adicional, ya que simplemente se añade v al entorno con su nuevo valor.

Si la expresión es una aplicación $e\ x$, buscamos el cuerpo de e añadiendo como nuevo argumento la x. En una expresión **case** $e\ (v_1, \ldots, v_m) \rightarrow e'$ se continúa evaluando e'. En una expresión **case** e **of** $\overline{C_i\ \overline{v_{ij}} \rightarrow e_i}$, cualquiera

$$
\begin{array}{l}
\mathit{body} :: \mathit{Expr} \to \mathit{Var} \to ([\mathit{Expr}], \mathit{Int}) \\
\mathit{body}\ e\ x = \mathit{body}'\ [e]\ [x]\ 0 \\
\mathit{body}' :: [\mathit{Expr}] \to [\mathit{Var}] \to \mathit{Int} \to ([\mathit{Expr}], \mathit{Int}) \\
\mathit{body}'\ [\,]\ xs\ c = ([\,], c) \\
\mathit{body}'\ (e : es)\ xs\ c = (es1 +\!\!+ ess, c_2) \\
\quad \text{donde } (es1, c_1) = \mathit{lbody}\ e\ xs\ c \\
\qquad\qquad (ess, c_2) = \mathit{body}'\ es\ xs\ c_1 \\
\mathit{lbody} :: \mathit{Expr} \to [\mathit{Var}] \to \mathit{Int} \to ([\mathit{Expr}], \mathit{Int}) \\
\mathit{lbody}\ e\ [\,]\ c = ([e], c) \\
\mathit{lbody}\ (\lambda v.e)\ (x : xs)\ c = \mathit{lbody}\ e[x/v]\ xs\ c \\
\mathit{lbody}\ (e\ x)\ xs\ c = \mathit{lbody}\ e\ (x : xs)\ c \\
\mathit{lbody}\ (\mathbf{case}\ e\ \mathbf{of}\ (v_1, \ldots, v_m) \to e')\ xs\ c = \mathit{lbody}\ e'\ xs\ c \\
\mathit{lbody}\ (\mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i})\ xs\ c = \mathit{body}'\ [e_1, \ldots, e_m]\ xs\ (c + (m-1)\ \mathcal{H}_t) \\
\quad \{i \in \{1..m\}, e_i :: t\} \\
\mathit{lbody}\ (\mathbf{let}\ bs\ \mathbf{in}\ e)\ xs\ c = \mathit{lbody}\ e\ xs\ c \\
\mathit{lbody}\ v\ xs\ c = \mathit{lbody}\ e\ xs\ c \quad \{\text{ligada por } \mathbf{let} \text{ no recursivo}\}\ \{v = e\} \\
\mathit{lbody}\ v\ xs\ c = ([\,], c') \quad \{\text{ligada por lambda, } \mathbf{let} \text{ recursivo o } \mathbf{case} \text{ algebraico}\} \\
\quad \text{donde } c' = c + \sum_i p_{t_i}(x_i) + (m-j) + \mathcal{H}_{t_r}, x_i = xs!!(i-1) \\
\qquad \{i \in \{1..j\}, j = \mathit{length}\ xs\ , v :: t, t_r = \mathit{rType}(t)\} \\
\mathit{lbody}\ v\ xs\ c = \mathit{body}'\ es_k\ xs\ c' \quad \{\text{ligada por } \mathbf{case} \text{ de tupla } k\text{-ésima, discriminante } e\} \\
\quad \text{donde} \\
\qquad (es, cs) = \mathit{comps}(e), es_k = es!!(k-1), n_k = cs!!(k-1), \\
\qquad c' = c + n_k\ \sum_i p_{t_i}(x_i) + (m-j) + \mathcal{H}_{t_r}, x_i = xs!!(i-1) \\
\qquad \{v :: t, m = \mathit{nArgs}(t), t_r = \mathit{rType}(t), j = \mathit{length}\ xs, i \in \{1..j\}, x_i :: t_i\}
\end{array}
$$

Figura 5.45: Definición de *body*

de las e_i puede ser el cuerpo de la función. Además acumulamos el coste de calcular la mínima cota superior de las alternativas. En una expresión **let** bs **in** e continuamos evaluando e.

Si la expresión es una variable, distinguimos tres casos. Puede tratarse de una variable ligada por un **let** no recursivo $v = e$, en cuyo caso continuamos buscando el cuerpo en e. Puede ser una variable ligada por una lambda, un **case** algebraico o un **let** recursivo, en cuyo caso el valor abstracto será una signatura. La aplicación de esta signatura a varios argumentos implica calcular el muestreo de los argumentos con coste $\sum_i p_{t_i}(x_i)$ y en el caso de que más de uno de ellos sea no determinista, devolver el correspondiente resultado no determinista, lo que cuesta $(m-j) + \mathcal{H}_{t_r}$ donde $m-j$ es el número de argumentos que faltan por aplicar. Por último, también puede tratarse de una variable ligada en la posición k-ésima por un **case** de tuplas. En tal caso utilizamos de nuevo la función *comps*, que nos devuelve en la componente k-ésima, una lista de expresiones es_k y un entero n_k. Continuamos buscando el cuerpo de la función en los es_k y acumulamos en el coste un sumando n_k $(\sum_i p_{t_i}(x_i) + (m-j) + \mathcal{H}_{t_r})$ correspondiente al coste de aplicar la función en los casos en que su valor abstracto es una signatura.

$$\mathcal{A}_K = \mathcal{A}_{T\ t_1 \ldots t_m} = \mathcal{A}_\beta = 1$$
$$\mathcal{A}_{(t_1,\ldots,t_m)} = \sum_{i=1}^{m} \mathcal{A}_{t_i}$$
$$\mathcal{A}_t = \mathcal{A}_{rType(t)} \ \text{si } t = t_1 \to t_2, Process\ t_1\ t_2$$
$$\mathcal{A}_{\forall \beta . t} = \mathcal{A}_t$$

Figura 5.46: Coste de aplanar un muestreo

Coste de obtener $\alpha_t(av)$. Dada una expresión $e :: t$, cuyo valor abstracto es av, el coste de obtener $\alpha_t(av)$ está en $O(a_t(e))$, siendo $a_t(e) = \mathcal{A}_t + \sum_i (m_i + \mathcal{H}_{t_{r_i}}) + oneap(e)$, donde $i \in \{1..m\}$, $m = nArgs(t)$, $[t_1, \ldots, t_m] = aTypes(t)$, $t_r = rType(t)$, $m_i = nArgs(t_i)$, $t_{ri} = rType(t_i)$ y $\mathcal{A}_t$ está definida en la Figura 5.46. Primero se lleva a cabo el muestreo de av con argumentos deterministas y después se aplana el resultado hasta obtener un valor básico, lo que cuesta $\mathcal{A}_t$.

La definición de $\mathcal{A}_t$ es sencilla. Si se trata de una signatura básica, ya está aplanada, por lo que el coste es constante. Si es una tupla de signaturas de tipo $(t_1, \ldots, t_m)$, hay que aplanar cada componente para tomar después la mínima cota superior, lo que cuesta en total $\sum_{i=1}^{m} \mathcal{A}_{t_i}$. Si es una signatura funcional de tipo t, nos quedamos con la última componente y la aplanamos, con un coste $\mathcal{A}_{rType(t)}$.

La función *oneap*. La Figura 5.47 muestra la función *oneap*. Dada una expresión de tipo funcional e con m argumentos de valor abstracto av, devuelve el coste de aplicar av a m signaturas. *oneap* no tiene en cuenta el coste de las expresiones que hay que evaluar para llegar a av, pues suponemos que ya disponemos de dicho valor abstracto, para lo cual habremos evaluado todo lo necesario, aunque sí ha de tener en cuenta los costes posteriores a la evaluación, como el cálculo de la mínima cota superior en un **case** algebraico. Por ello *oneap* básicamente debe buscar la primera suspensión antes de comenzar las aplicaciones. Si no se trata de una suspensión, sino de una variable ligada por una lambda, un **case** algebraico o un **let** recursivo, con lo que su valor abstracto es una signatura, entonces solamente hay que elegir una componente de la signatura, comparando los argumentos (que sabemos que a su vez son signaturas) con $\gamma'_{t_i}(d)$, lo que cuesta $\mathcal{H}_{t_i}$ para cada $i \in \{1..m\}$ donde m es el número de argumentos de la función.

Pero una vez localizada la aplicación al primer argumento, deberemos calcular el valor de la función auxiliar $skip\lambda$, que se encarga de calcular los costes dentro del cuerpo de la función. Como su nombre en inglés indica, el trabajo de $skip\lambda$ consiste básicamente en evaluar al completo una expresión saltando por encima de las suspensiones.

$oneap\ (\lambda v.e) = skip\lambda(e) + 1$
$oneap\ (e\ x) = c + sum\ (map\ oneap\ es) + (m-1)\ \mathcal{H}_t \quad \{e\ x :: t, m = length\ es\}$
  donde $(es, c) = body\ e\ x$
$oneap\ (\mathbf{case}\ e\ \mathbf{of}\ (v_1, \ldots, v_m) \rightarrow e') = oneap(e')$
$oneap\ (\mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{v_{ij}} \rightarrow e_i}) = \sum_i oneap\ (e_i) + (m-1)\ \mathcal{H}_t \quad i \in \{1..m\}$
$oneap\ (\mathbf{let}\ bs\ \mathbf{in}\ e) = oneap\ (e)$
$oneap\ (v) = oneap\ \ (e)$ {ligada por **let** no recursivo} $\{v = e\}$
$oneap\ (v) = \sum_i \mathcal{H}_{t_i}$ {ligada por lambda, **let** recursivo o **case** algebraico}
  donde $\{v :: t, i \in \{1..m\}, m = nArgs(t), [t_1, \ldots, t_m] = aTypes(t)\}$
$oneap\ (v) = sum\ (map\ oneap\ es) + c'$ {ligada por **case** de tupla k-ésima, discriminante e}
  donde $(es, cs) = comps(e), es_k = es!!(k-1), n_k = cs!!(k-1), c' = n_k\ \sum_j \mathcal{H}_{t_j}$
    $\{v :: t, [t_1, \ldots, t_m] = aTypes(t), j \in \{1..m\}\}$

$skip\lambda\ (\lambda v.e) = skip\lambda\ (e) + 1$
$skip\lambda\ (e\ x) = s'(e\ x) - sum\ (map\ s'\ es) + c'$
  donde $c' = sum\ (map\ skip\lambda\ es)$
    $(es, c) = body\ e\ x, m = length\ es$
$skip\lambda\ (\mathbf{case}\ e\ \mathbf{of}\ (v_1, \ldots, v_m) \rightarrow e') = s'(\mathbf{case}\ e\ \mathbf{of}\ (v_1, \ldots, v_m) \rightarrow e') - s'(e') + skip\lambda(e')$
$skip\lambda\ (\mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{v_{ij}} \rightarrow e_i}) = s'(\mathbf{case}\ e\ \mathbf{of}\ \overline{C_i\ \overline{v_{ij}} \rightarrow e_i}) - \sum_i s'(e_i) + \sum_i skip\lambda\ (e_i)$
$skip\lambda\ (\mathbf{let}\ bs\ \mathbf{in}\ e) = s'(\mathbf{let}\ bs\ \mathbf{in}\ e) - s'(e) + skip\lambda(e)$
$skip\lambda(x_1, \ldots, x_m) = s'(x_1, \ldots, x_m) + p_{(t_1, \ldots, t_m)}(x_1, \ldots, x_m) \quad \{x_i :: t_i\}$
$skip\lambda\ v = s'(v)$

Figura 5.47: La función *oneap*

La función s' calcula el coste de obtener el valor abstracto de una expresión, lo que significa que si este es funcional, se detiene cuando encuentra una suspensión. Ahora no queremos detenernos en ellas, por lo que si encontramos una lambda nos la saltamos. En los casos de aplicación, expresiones **case** y **let**, la definición de $skip\lambda(e)$ se reduce básicamente a $s'(e)$ más el coste de continuar evaluando más allá de la suspensión con nuevos $skip\lambda$. Por ejemplo, si consideramos el cálculo de $skip\lambda$ (**let** bs **in** e) tendríamos $skip\lambda$ (**let** bs **in** e) $= s'$(**let** bs **in** e) $- s'(e) + skip\lambda(e)$. En este caso, s'(**let** bs **in** e) calcula el coste hasta llegar a la suspensión. Como queremos continuar evaluando, llamamos a $skip\lambda(e)$, pero con ello hemos contabilizado de nuevo el coste de evaluar la parte de e anterior a la suspensión, por lo que hemos de restar $s'(e)$. Lo mismo sucede en los demás casos considerados en la Figura 5.47. Cuando llegamos a una tupla, hemos llegado al resultado de la función, por lo que solamente queda obtener su signatura, pues recordemos que estamos obteniendo el coste de calcular una componente de la signatura de la función. Si se trata de una variable, la signatura estará en el entorno, por lo que el coste es 1 (o $s'(e)$).

Polimorfismo. En las definiciones de *lbody*, *comps* y *oneap* no se ha mostrado el caso de la aplicación de tipos. La introducción de este caso resulta compleja, al verse involucradas aplicaciones entremezcladas de $\gamma'_{t'\ tinst}$,

$\alpha'_{tinst\,t'}$ y aplicaciones de valores abstractos. Sin embargo, las operaciones implicadas en estos cálculos son combinaciones de las ya descritas donde los tamaños de los tipos implicados son sucesivamente más pequeños (ver definiciones de $\gamma'_{t'\,tinst}$ y $\alpha'_{tinst\,t'}$), por lo que podemos suponer que los costes de las aplicaciones de tipos son lineales con el tamaño del tipo concretado *tinst* y con el tamaño $s'(e)$ de la expresión aplicada.

5.9 Trabajos relacionados y conclusiones

En este capítulo se han presentado y comparado formalmente tres análisis de no determinismo para un lenguaje funcional con polimorfismo de segundo orden y expresiones no deterministas. Aunque la principal motivación de este trabajo ha sido la compilación correcta de nuestro lenguaje Edén, todo lo aquí presentado se podría aplicar a otros lenguajes funcionales polimórficos no deterministas. Una aplicación posible del análisis sería anotar el programa fuente escrito en uno de estos lenguajes con anotaciones de no determinismo, mostrando al programador dónde sería posible mantener el razonamiento ecuacional.

No hemos encontrado en la literatura análisis previos para resolver este problema en el ámbito de los lenguajes funcionales paralelos. Sí se ha estudiado en lenguajes lógico-funcionales [HS00], pero en ellos la fuente de no determinismo reside en otro tipo de mecanismos, como la unificación. En [BGP00] Baker-Finch, Glynn y Peyton Jones presentan su análisis de *resultado de producto construido* (CPR). El análisis pretende determinar qué funciones pueden devolver resultados múltiples en registros, es decir, qué funciones devuelven una tupla explícitamente construida. Se trata de un análisis basado en interpretación abstracta donde el dominio abstracto correspondiente a un tipo funcional $t_1 \to t_2$ no es el correspondiente dominio funcional, sino que es isomorfo al dominio abstracto del tipo del resultado t_2. Los tipos producto se interpretan como un producto cartesiano de un dominio abstracto básico, es decir, las tuplas anidadas no están permitidas. Nuestro primer análisis sigue estas mismas ideas aunque por diferentes razones, ya explicadas con anterioridad.

El segundo y tercer análisis están basados en interpretación abstracta en el estilo de [BHA86], donde las funciones se interpretan como funciones abstractas. Allí se presentó un análisis de estrictez en el que el dominio básico $\mathbf{2}$ es también un dominio de dos puntos ($\bot \sqsubseteq \top$). Sin embargo, los análisis son diferentes. Por un lado, aunque los dominios semánticos son isomorfos, el significado de un "mismo" elemento es diferente. Así, funciones con el mismo valor abstracto en ambos análisis pueden producir distintos resultados, por ser distintos los muestreos realizados para obtener el resultado del análisis. Por ejemplo, sea $f :: (Int \to Int) \to Int$, donde $f = \lambda g.g(head(merge\#[[0],[1]]))$. En el análisis de estrictez el valor abstracto de f es $f_s^\# = \lambda v \in [\mathbf{2} \to \mathbf{2}].v\ \top$ y en el de no determinismo $f_n^\# = \lambda w \in [Basic \to Basic].w\ n$. Vemos que salvando los nombres de los dominios y de sus supremos, los valores abstractos son los mismos. Sin embargo, el análisis de estrictez nos dice que f **es** estricta en su argumento (pues $f_s^\#\ (\lambda z.\bot) = \bot$), mientras que el análisis de no determinismo nos dice que **puede ser** no determinista (pues $f_n^\#(\lambda z.z) = n$). Por otro lado, hay funciones con distintos valores abstractos en los dos análisis, ya que la interpretación de los operadores primitivos, los constructores y las expresiones **case** es diferente. Por ejemplo, a $h = \lambda x.x + y$, donde el valor de y es una constante no determinista, el análisis de estrictez le daría el valor abstracto $\lambda v \in \mathbf{2}.v$, mientras que el análisis de no determinismo le da el valor abstracto $\lambda v \in Basic.n$.

El primer artículo sobre este tema [PS01d] presentaba nuestras ideas preliminares en la forma de dos análisis no completamente satisfactorios. El primero era eficiente pero no lo suficientemente potente, mientras que el segundo era potente pero no lo suficientemente práctico en términos de eficiencia. Por ello se desarrolló un tercer análisis lo suficientemente potente y práctico en términos de implementación: es polinómico, y comparado con el segundo análisis solamente pierde información en los puntos fijos. Hemos probado el algoritmo con muchos ejemplos obteniendo una traza del número de iteraciones necesarios para alcanzar el punto fijo, y los resultados muestran que la cota superior $\mathcal{H}_t$ casi nunca se alcanza.

En este capítulo se han presentado también los resultados teóricos que soportan la corrección de la implementación del algoritmo. Esta última merece un comentario final: implementar el análisis en un lenguaje funcional perezoso como Haskell nos ha proporcionado bastantes ventajas. La primera de ellas es que una función abstracta se puede representar mediante una interpretación suspendida. Relacionada con ella, gracias a la evaluación perezosa la interpretación abstracta puede hacer uso de la aplicación abstracta y viceversa, sin ningún peligro de no terminación. Además, $[\![\cdot]\!]$ y $[\![\cdot]\!]'$ han sido de hecho implementadas por una única función Haskell. $[\![\cdot]\!]'$ es simplemente una llamada $[\![\cdot]\!]$ tras la que se ignora la segunda componente. La evaluación perezosa no computa esta componente en dichas llamadas. Todo ello no habría sido tan sencillo en un lenguaje impaciente y/o imperativo. Otras características como el orden superior, el polimorfismo y la sobrecarga han contribuido a obtener un algoritmo compacto: la interpretación completa cabe en una veintena de páginas incluyendo comentarios (ver Apéndice B).

Para nuestros propósitos esto cierra el problema inicial. Queda pendiente una demostración de la corrección del segundo análisis con respecto a la semántica denotacional estándar del lenguaje. Desafortunadamente no hay todavía tal semántica formal para Edén. Una forma simplificada de semántica podría usarse para demostrar parte de la corrección del análisis.

Capítulo 6

Análisis de productividad y terminación

Las extensiones presentes en Edén con respecto a Haskell permiten la definición sencilla de esqueletos como funciones de orden superior [PR01]. Sin embargo, el programador puede introducir inadvertidamente en ellos bucles activos o bloqueos. La teoría de los tipos con tamaño, que fue resumida en el Capítulo 2 (Sección 2.3.3), ha sido desarrollada recientemente por Hughes y Pareto [HPS96, Par97, Par00] para proporcionar un marco en el que se puedan realizar análisis automáticos de la terminación y productividad de los programas.

En este capítulo se extiende el sistema de tipos con tamaño para que pueda usarse sobre los programas Edén, de forma que quede garantizado que todos los programas Edén bien tipados, y en concreto los esqueletos, o terminan o son productivos. Se describen con detalle los problemas que surgen debido a las características propias de Edén y sus posibles soluciones. Además se tipan a mano algunos esqueletos en este sistema modificado, el cual aún no ha sido implementado. Este trabajo ha dado lugar a un artículo [PS01e].

En la Sección 2.3.3 se describió la semántica de los tipos con tamaño desarrollada por Hughes y Pareto. En la Sección 6.1 se describe el sistema de tipos de Haskell Síncrono. En la Sección 6.2 se describen algunos de los problemas introducidos por las características de Edén y se extienden consecuentemente las reglas del sistema de tipos. Se presentan dos ejemplos simples, una versión ingenua del esqueleto `map` paralelo y una tubería. Se pueden comprobar sus tipos usando las nuevas reglas. En la Sección 6.3 se chequean esqueletos más complejos, como el esqueleto granja, la topología de trabajadores replicados y dos versiones del esqueleto divide y vencerás. Se discuten los problemas surgidos al intentar tipar estos esqueletos y las posibles soluciones. En la Sección 6.4 concluimos y presentamos trabajo futuro.

6.1 El sistema de tipos de Haskell Síncrono

En [Par00] se define el lenguaje de programación Haskell Síncrono, el cual es un subconjunto de Haskell. Un programa consiste en una serie de declaraciones de tipos seguidas de un término escrito en un λ-cálculo enriquecido:

$$\begin{array}{rl} e ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid c \\ & \mid \textbf{let}\ x :: \sigma = e_1\ \textbf{in}\ e_2 \\ & \mid \textbf{letrec}\ x_1 :: \forall k_1.\sigma_1 = e_1 \ldots x_n :: \forall k_n.\sigma_n = e_n\ \textbf{in}\ e \\ & \mid \textbf{case}\ e\ \textbf{of}\ a_1 \ldots a_m \\ a ::= & c\ x_1 \ldots x_n \to e \end{array}$$

Cada ligadura **let** y **letrec** está anotada con un esquema de tipo que debe ser comprobada por el sistema. En la Figura 6.1 se muestran las reglas de tipos para este sistema.

Se usan índices distintos cuando aparecen secuencias diferentes de elementos en la misma regla. No usaremos i, por estar reservada para representar expresiones de tamaño finitas. Por ejemplo en la regla $[VAR]$, $\tau \overset{\cup}{\sim} k'_j$ significa que si $\overline{k'} = k'_1 \ldots k'_n$, entonces $\forall j \in \{1..n\}.\tau \overset{\cup}{\sim} k'_j$. Utilizamos a para representar variables de tipo y k para representar variables de tamaño.

Veamos con detalle cada una de las reglas. En la regla $[VAR]$ se lleva a cabo la concreción de las variables de tipo y de tamaño. Ya se dijo en la Sección 2.3.3 que la concreción de las variables de tamaño con ω no es siempre segura. Una condición suficiente para que un esquema pueda concretarse con ω de forma segura es que sea *undershooting* con respecto a la variable de tamaño que se va a concretar $\sigma \overset{\cup}{\sim} k$. Por ello, en la regla $[VAR]]$ solamente las variables k'_j con respecto a las cuales τ es *undershooting* se pueden concretar con tamaños s (que son los únicos que incluyen a ω, ver Figura 2.7). El resto sólo se pueden concretar con tamaños finitos i. En [Par00] se proporcionan condiciones suficientes para demostrar esta propiedad, lo que implica la definición de otras relaciones entre los tipos y las variables de tamaño, como la relación de monotonía $\tau \overset{+}{\sim} k$ y antimonotonía $\tau \overset{-}{\sim} k$. Aquí no se muestran las reglas que definen todas estas relaciones ya que esto implicaría incluir una buena parte de la tesis de Pareto [Par00], en la que se explican ampliamente cada una de ellas. Para no alargar excesivamente este capítulo se ha optado por omitirlas.

La regla de los constructores es semejante a la regla de la variable, ya que los tipos algebraicos son también polimórficos.

Las reglas de abstracción $[ABS]$ y de aplicación $[APP]$ son estándar. En la regla de la aplicación se admite que el tipo del argumento sea subtipo del tipo especificado para el argumento formal de la función.

La regla $[LET]$ también es estándar. En este caso se permite que el tipo obtenido para el lado derecho de la ligadura sea un subtipo del tipo (sin cuantificadores) especificado para la variable del lado izquierdo.

$$\frac{\sigma = \forall \overline{a}\ \overline{k}\ \overline{k}'.\tau \quad \tau \overset{\cup}{\sim} k'_j}{\Gamma \cup \{x :: \sigma\} \vdash x :: \tau[\overline{i}/\overline{k}][\overline{s}/\overline{k}'][\overline{\tau}/\overline{t}]}\ \mathit{VAR}$$

$$\frac{c :: \forall \overline{a}\ \overline{k}\ \overline{k}'.\tau \in \mathbf{C} \quad \tau \overset{\cup}{\sim} k'_j}{\Gamma \vdash c :: \tau[\overline{i}/\overline{k}][\overline{s}/\overline{k'}][\overline{\tau}/\overline{t}]}\ \mathit{CONST}$$

$$\frac{\Gamma \cup \{x :: \tau_1\} \vdash e_: : \tau_2}{\Gamma \vdash \lambda x.e :: \tau_1 \to \tau_2}\ \mathit{ABS}$$

$$\frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 :: \tau_3 \quad \tau_3 \rhd \tau_1}{\Gamma \vdash e_1\ e_2 :: \tau_2}\ \mathit{APP}$$

$$\frac{\begin{array}{c}\sigma = \forall \overline{a}\ \overline{k}.\tau \quad \overline{a}, \overline{k} \notin FV(\Gamma) \\ \Gamma \cup \{x :: \sigma\} \vdash e' :: \tau' \quad \Gamma \vdash e :: \tau'' \quad \tau'' \rhd \tau\end{array}}{\Gamma \vdash \mathbf{let}\ x :: \sigma = e\ \mathbf{in}\ e' :: \tau'}\ \mathit{LET}$$

$$\frac{\begin{array}{c} j \in \{1..n\}, \sigma_j[k_j] = \forall \overline{a_j}\ \overline{k_j}.\tau_j[k_j] \quad \overline{a_j}, \overline{k_j}, k_j \notin FV(\Gamma) \\ \Gamma' = \Gamma, x_1 :: \forall \overline{k_1}.\tau_1[k] \ldots x_n :: \forall \overline{k_n}.\tau_n[k] \\ \Gamma'' = \Gamma, x_1 :: \forall k_1.\sigma_1[k_1] \ldots x_n :: \forall k_n.\sigma_n[k_n] \\ \sigma_j[0] = \mathbf{U} \qquad \Gamma' \vdash e_j :: \tau'_j \qquad \tau'_j \rhd \tau_j[k+1] \qquad \Gamma'' \vdash e :: \tau \end{array}}{\Gamma \vdash \mathbf{letrec}\ x_1 :: \forall k_1.\sigma_1[k_1] = e_1 \ldots x_n :: \forall k_n.\sigma_n[k_n] = e_n\ \mathbf{in}\ e :: \tau}\ \mathit{LETREC}$$

$$\frac{\begin{array}{c} \vdash_= \tau = c_1\ \overline{\tau_1} \mid \ldots \mid c_n\ \overline{\tau_n} \\ \Gamma \vdash e :: \tau \quad \Gamma \cup \{x_{j1} :: \tau_{j1}, \ldots, x_{jn_j} :: \tau_{jn_j}\} \vdash e_j :: \tau'_j(\forall j) \quad \tau'_j \rhd \tau'(\forall j) \end{array}}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ c_1\ \overline{x_1} \to e_1 \ldots c_n\ \overline{x_n} \to e_n :: \tau'}\ \mathit{CASE}$$

Figura 6.1: Sistema de tipos para Haskell Síncrono

En la regla del $[CASE]$ se lleva a cabo una destrucción del valor del discriminante. Es necesario ver que el tipo del discriminante corresponde a una definición de tipo algebraico que aparece en el programa, y que los constructores implicados en dicha definición son exactamente los que aparecen en las ramas del **case**, es decir, que este es exhaustivo. Una vez encontrada la definición, si se trata de un tipo **idata** o **codata**, es imprescindible que el tipo obtenido para el discriminante contenga al menos un constructor. Ambas cosas son comprobadas por la relación $\vdash_=$, presentada en [Par00]. Las componentes recursivas tendrán el mismo tipo con un constructor menos, tipo que será asignado a las variables de los lados izquierdos de las alternativas. Así, por ejemplo sea una expresión **case** xs **of** $nil \rightarrow e_1; cons\ x\ xs' \rightarrow e_2$. Encontramos una definición donde aparecen los constructores nil y $cons$. Se trata de la definición de las listas **idata List** $w\ a = nil \mid cons\ a$ (**List** $w\ a$). Como es necesario extraer un constructor para hacer el encaje de patrones, la relación $\vdash_=$ exige que xs tenga el tipo xs :: **List** $(k+1)\ a$, dando a xs' el tipo **List** $k\ a$. Es decir, $\vdash_=$ **List** $(k+1)\ a = nil \mid cons\ a$ (**List** $k\ a$).

La regla $[LETREC]$ corresponde a una aplicación de mecanismos de inducción sobre la primera variable natural k_j en cada ligadura. En consecuencia, cuando preveamos que será necesario aplicar esta regla, deberemos elegir dicha variable de modo que el razonamiento inductivo sea posible. Para ilustrar cómo trabaja esta regla explicaremos el tipado de la función $map :: \forall a, b, k.(a \rightarrow b) \rightarrow$ **List** $k\ a \rightarrow$ **List** $k\ b$, donde $map = \lambda f.\lambda xs.$**case** xs **of** $nil \rightarrow nil; cons\ x\ xs' \rightarrow cons\ (f\ x)\ (map\ f\ xs')$.

A continuación, se estudia el caso base $\sigma_j[0] = \mathbf{U}$. Este recibe el nombre de *comprobación del ínfimo* (*bottom check*). Las reglas para comprobarlo hacen necesaria la definición de otras relaciones sobre los tipos como comprobar si un tipo es vacío ($= \mathbf{E}$), o no vacío ($\neq \mathbf{E}$); y otras sobre tamaños, como $= 0$ y $\neq 0$. La totalidad de las reglas se encuentran en [Par00]; aquí solo mostramos en la Figura 6.2 aquellas que consideramos las más interesantes. En general, un tipo **codata** (T_C) de tamaño 0 denota el universo, mientras que un tipo **idata** (T_I) de tamaño 0 denota el tipo vacío. Algunos ejemplos en los que se cumple la comprobación del ínfimo son: $\forall k.$**Strm** $k\ a$, $\forall a, b, k.(a \rightarrow b) \rightarrow$ **List** $k\ a \rightarrow$ **List** $k\ b$ y $\forall a, k.$ **Strm** $k\ a \rightarrow$ **Strm** $k\ a$.

Después, suponiendo la hipótesis de inducción para cada $j \in \{1..n\}$, $x_j :: \forall \overline{k_j}.\tau_j[k]$, se debe probar que los tipos de los lados derechos de las ligaduras e_j son subtipos de $\tau_j[k+1]$. En el ejemplo, suponiendo que map :: $(a \rightarrow b) \rightarrow$ **List** $k\ a \rightarrow$ **List** $k\ b$ debemos demostrar que su cuerpo tiene tipo $(a \rightarrow b) \rightarrow$ **List** $(k+1)\ a \rightarrow$ **List** $(k+1)\ b$. Esto implica que $f :: a \rightarrow b$ y que xs :: **List** $(k+1)\ a$. En la expresión **case**, si xs es una lista vacía, se devuelve nil :: **List** $1\ a$, que es un subtipo de **List** $(k+1)\ a$, ya que $1 \leq k+1$. Si no es vacía, el valor es destruido, por lo que xs' tiene un constructor menos, xs' :: **List** $k\ a$. Por hipótesis de inducción $map\ f\ xs'$ :: **List** $k\ b$, luego, añadir un nuevo elemento hace que $cons\ (f\ x)\ (map\ f\ xs')$ :: **List** $(k+1)\ b$, que es el tipo deseado. Se permite recursión polimórfica en todas las variables

$$\frac{\tau_2 = \mathbf{U}}{\tau_1 \to \tau_2 = \mathbf{U}} \qquad \frac{\tau_1 = \mathbf{E}}{\tau_1 \to \tau_2 = \mathbf{U}} \qquad \frac{s_1 = 0}{T_C\ \overline{s}\ \overline{\tau} = \mathbf{U}}$$

$$\frac{\sigma = \mathbf{U}}{\forall t.\sigma = \mathbf{U}} \qquad \frac{\sigma = \mathbf{U}}{\forall k.\sigma = \mathbf{U}} \qquad \frac{s_1 = 0}{T_I\ \overline{s}\ \overline{\tau} = \mathbf{E}}$$

Figura 6.2: Reglas para comprobación del ínfimo y una regla para comprobar si un tipo es vacío

de tamaño, excepto en la inductiva. Esto es bastante útil; por ejemplo para tipar la función $reverse$, como se muestra en [Par00].

6.2 Nuevos problemas introducidos por Edén

Para abarcar las características de Edén, son necesarias algunas extensiones al sistema de tipos. Las características más importantes son la forma en que se transmiten los valores a través de los canales, la evaluación impaciente de algunas expresiones y el uso de listas para representar tanto listas Haskell como la transmisión en forma de *stream* de los valores.

Recordemos los detalles del protocolo de lanzamiento de $e_1 \# e_2$ que requerirán ahora cierta atención (ver Sección 3.3):

- (1) la clausura e_1 junto con todas las que de ella dependen son *copiadas*, sin evaluar, a un nuevo procesador y allí se crea un proceso hijo para evaluarla;
- (2) una vez creado, el proceso hijo comienza a producir de forma impaciente su expresión de salida;
- (3) la expresión e_2 se evalúa de forma impaciente en el proceso padre. Si es una tupla, se crea una hebra concurrente independiente para evaluar cada componente (cada elemento de la tupla es un *canal*).

Una vez que un proceso se encuentra en ejecución, solamente se comunican datos completamente evaluados. La única excepción la representan las listas, que se transmiten en forma de *stream*, es decir, elemento a elemento. Primero se evalúa cada elemento de la lista a forma normal y después se transmite.

6.2.1 Transmisión de valores

La comunicación de datos a través de canales produce dos problemas diferentes. Primero, puesto que los valores se evalúan a forma normal antes de enviarlos, es necesario que los tipos de los valores comunicados sean finitos:

tuplas, tipos **data** o **idata** con componentes finitas, y tipos funcionales. Los tipos funcionales se consideran finitos pues su envío por un canal se limita a copiar el puntero a la clausura correspondiente. En segundo lugar, esto implica que la concreción de las variables de tipo debe restringirse a tipos finitos en algunos lugares. Proponemos un mecanismo similar al sistema de clases de Haskell, lo cual implica extender el lenguaje. Definimos una *relación de finitud* $\Gamma_T \vdash_F \tau$ (ver Figura 6.3), donde Γ_T es un conjunto de variables de tipo que pueden aparecer en τ. Este aserto significa que, suponiendo que las variables de tipo en Γ_T solamente pueden concretarse con tipos finitos, τ es también un tipo finito. Para demostrar que un tipo recursivo es finito procedemos por inducción estructural: en la Figura 6.3 usamos una pseudo-variable **F**, siempre finita, para sustituir las apariciones recursivas de un tipo **idata** por un tipo finito. Esta sustitución aparece representada por $S[\cdot]$ como un contexto, es decir, un lado derecho de definición de tipo con un agujero. Si ese agujero es el correspondiente a la aparición recursiva del tipo, colocamos en su lugar **F**.

Usamos esta relación para controlar la concreción de las variables. Puesto que hay formas diferentes de transmitir valores, dependiendo del tipo del canal, podemos usar tipos distintos para representar un canal: si es un canal por el que se envía un único valor, su tipo viene representado por el tipo del valor; si es un canal tipo *stream*, podemos usar o bien un tipo **List** o un tipo **Strm** (ver discusión más adelante). Esta separación impone distintas restricciones: si se trata de un canal de un único valor, su tipo debe ser finito, pero si es de tipo *stream*, solamente ha de ser finito el tipo de sus elementos. Esto nos lleva a introducir dos clases distintas de valores, F (de finito) y T (de interfaz de transmisión). La primera indica que la variable de tipo solamente puede concretarse con tipos finitos, y la segunda que solamente puede concretarse con tipos *interfaz*. Un tipo interfaz es, o bien un tipo finito (lo que incluye al tipo **List** con componentes finitas) o un tipo **Strm** con componentes finitas. Un proceso tiene normalmente varios canales de entrada y salida, representados por una tupla de canales, luego los tipos interfaz deben incluir también tuplas de los tipos anteriores. Los tipos se extienden con las nuevas clases: $\tau' ::= \tau \mid [T\ \overline{a}], [F\ \overline{b}] \Rightarrow \tau$; y hace falta una nueva regla para comprobación del ínfimo, expresando que dicha comprobación no se ve afectada por los contextos:

$$\frac{\tau = \mathbf{U}}{[T\ \overline{a}], [F\ \overline{b}] \Rightarrow \tau = \mathbf{U}}\ .$$

Además, se introducen en las reglas dos nuevos entornos, Γ_T^F y Γ_T^T, que contienen las variables de tipo que aparecen respectivamente en un contexto F o en uno T. Así, nuestros asertos son de la forma $\Gamma_T^F, \Gamma_T^T, \Gamma \vdash e :: \tau$. El predicado $P(\Gamma_T^F, \Gamma_T^T, \tau)$, definido en la Figura 6.4, nos dice cuándo τ es un tipo interfaz, según la definición dada.

$$\frac{a \in \Gamma_T}{\Gamma_T \vdash_F a} \qquad \frac{\Gamma_T \vdash_F \tau_1 \qquad \Gamma_T \vdash_F \tau_2}{\Gamma_T \vdash_F (\tau_1, \tau_2)} \qquad \frac{}{\Gamma_T \vdash_F \tau \rightarrow \tau'} \qquad \frac{}{\Gamma_T \vdash_F F}$$

$$\frac{\Gamma_T \vdash_F R[\overline{\tau}/\overline{t}][\overline{s}/\overline{k}]}{\Gamma_T \vdash_F T_d\ \overline{s}\ \overline{\tau}} \qquad \frac{\Gamma_T \vdash_F S[\mathbf{F}][\overline{\tau}/\overline{t}][\overline{s}/\overline{k}]}{\Gamma_T \vdash_F T_i\ s_1\ \overline{s}\ \overline{\tau}} \qquad \frac{\forall j \in \{1..n\}, l \in \{1..m_j\} \quad \Gamma_T \vdash_F \tau_{jl}}{\Gamma_T \vdash_F c_1\ \overline{\tau_1} \mid \ldots \mid c_n\ \overline{\tau_n}}$$

donde **data** $\forall \overline{k}\ \overline{t}.T_d\ \overline{k}\ \overline{t} = R$ $\qquad R = c_1\ \overline{\tau_1} \mid \ldots \mid c_n\ \overline{\tau_n}$

idata $\forall \overline{k}\ \overline{t}.T_i\ w\ \overline{k}\ \overline{t} = S[T_i\ w\ \overline{k}\ \overline{t}]$ $\qquad S = c_1\ \overline{\tau_1} \mid \ldots \mid c_n\ \overline{\tau_n}$

Figura 6.3: Relación de finitud

En la Figura 6.4 se presentan las reglas de tipo modificadas. Describimos aquí solamente los elementos nuevos que no pertenecen al sistema original de [Par00]. En la regla $[VAR]$ se controla la concreción de las variables de tipo: las que aparecen en un contexto F se concretan con tipos finitos, y las que aparecen en un contexto T se concretan con tipos interfaz.

En un tipo **Process** $\tau\ \tau'$, τ y τ' representan las interfaces de comunicación, por los que en las reglas $[PABS]$ y $[PINST]$ se debe comprobar que son en efecto tipos interfaz. En la regla $[MERGE]$, los valores transmitidos por los canales deben ser finitos. Usamos $\langle$ y $\rangle$ para representar tuplas estrictas, explicadas en la Sección 6.3.3.

En la regla $[LET]$ (la regla $[LETREC]$ es similar), se anotan las ligaduras con sus tipos. Si una variable de tipo cuantificada universalmente está cualificada por una clase F o T, obligamos al programador a indicar la misma cuantificación en todas las anotaciones donde dicha variable aparece libre ($\overline{t^T} \subseteq \Gamma_T^T$ y $\overline{t^F} \subseteq \Gamma_T^F$). De esta forma, la información de clases en el lado derecho de una ligadura está totalmente contenida en la anotación y no es necesario buscarla allí donde está ligada ($\Gamma_T^{T'} = \overline{b}$ y $\Gamma_T^{F'} = \overline{c}$). El resto de reglas ($\lambda$-abstracción, aplicación y **case**) son similares a las originales.

6.2.2 Evaluación impaciente

En Edén, la evaluación se vuelve impaciente en dos casos: (1) los procesos se lanzan impacientemente cuando la expresión bajo evaluación demanda la creación de una clausura de la forma $o = e_1\#e_2$, y (2) los procesos, una vez lanzados, producen su salida incluso aunque esta no se demande. Estas modificaciones semánticas tienen como objetivo incrementar el grado de paralelismo y acelerar la distribución del cómputo. Desde el punto de vista del sistema de tipos, esto significa que algunos valores de tipos interfaz, como o en $o = e_1\#e_2$, se producen sin ser demandados. Si o es finito, su tipo proporciona una cota superior de su tamaño. Si bien con evaluación perezosa, este tamaño no se alcanzará en muchos casos, con evaluación impaciente sí se alcanzará casi siempre, con lo que estimar el tamaño mediante el tipo

$$\dfrac{\begin{array}{c}\sigma = \forall\overline{a}\ \overline{k}\ \overline{k}'.T\ \overline{b}, F\ \overline{c} \Rightarrow \tau \\ \overline{t^T} = \overline{a}\cap\overline{b} \qquad \overline{t^F} = \overline{a}\cap\overline{c} \qquad \overline{t} = \overline{a}\backslash(\overline{t^T}\cup\overline{t^F}) \\ \tau\overset{\cup}{\sim}k'_j \qquad \Gamma_T^F\vdash_F\tau_m^F \qquad P(\Gamma_T^F,\Gamma_T^T,\tau_l^T)\end{array}}{\Gamma_T^F,\ \Gamma_T^T,\ \Gamma\cup\{x::\sigma\}\vdash x::\tau[\overline{i}/\overline{k}][\overline{s}/\overline{k}'][\overline{\tau}/\overline{t}][\overline{\tau^T}/\overline{t^T}][\overline{\tau^F}/\overline{t^F}]}\ VAR$$

$$\dfrac{\Gamma_T^F,\Gamma_T^T,\Gamma\cup\{x::\tau\}\vdash e::\tau' \quad P(\Gamma_T^F,\Gamma_T^T,\tau) \quad P(\Gamma_T^F,\Gamma_T^T,\tau')}{\Gamma_T^T,\Gamma_T^T,\Gamma\vdash \mathbf{process}\ x\rightarrow e::\mathbf{Process}\ \tau\ \tau'}\ PABS$$

$$\dfrac{\begin{array}{c}\Gamma_T^F,\Gamma_T^T,\Gamma\vdash e_1::\mathbf{Process}\ \tau\ \tau' \qquad \Gamma_T^F,\Gamma_T^T,\Gamma\vdash e_2::\tau'' \qquad \tau''\rhd\tau \\ P(\Gamma_T^F,\Gamma_T^T,\tau) \qquad P(\Gamma_T^F,\Gamma_T^T,\tau')\end{array}}{\Gamma_T^F,\Gamma_T^T,\Gamma\vdash e_1\#e_2::\tau'}\ PINST$$

$$\dfrac{\Gamma_T^F,\Gamma_T^T,\Gamma\vdash e::<\mathbf{Strm}\ k_1\ \tau,\ldots,\mathbf{Strm}\ k_n\ \tau> \quad \Gamma_T^F\vdash_F\tau}{\Gamma_T^F,\Gamma_T^T,\Gamma\vdash merge\#e::\mathbf{Strm}\ (\sum_{j=1}^n\ k_j)\ \tau}\ MERGE$$

$$\dfrac{\begin{array}{c}\sigma = \forall\overline{a}\ \overline{k}.T\ \overline{b}, F\ \overline{c}\Rightarrow\tau \qquad \overline{a},\overline{k}\notin FV(\Gamma) \\ \overline{t^T} = \overline{b}\backslash\overline{a} \qquad \overline{t^F} = \overline{c}\backslash\overline{a} \qquad \Gamma_T^{T}{}' = \overline{b} \qquad \Gamma_T^{F}{}' = \overline{c} \\ \Gamma_T^F,\Gamma_T^T,\Gamma\cup\{x::\sigma\}\vdash e'::\tau' \qquad \Gamma_T^{F}{}',\Gamma_T^{T}{}',\Gamma\vdash e::\tau'' \qquad \tau''\rhd\tau \\ \overline{t^T}\subseteq\Gamma_T^T \qquad \overline{t^F}\subseteq\Gamma_T^F\end{array}}{\Gamma_T^F,\Gamma_T^T,\Gamma\vdash \mathbf{let}\ x::\sigma = e\ \mathbf{in}\ e'::\tau'}\ LET$$

$$\dfrac{\begin{array}{c}j\in\{1..n\}, \sigma_j[k_j] = \forall\overline{a_j}\ \overline{k_j}.T\ \overline{b_j}, F\ \overline{c_j}\Rightarrow\tau_j[k_j]\ \ \overline{a_j},\overline{k_j},k_j\notin FV(\Gamma) \\ \Gamma' = \Gamma, x_1::\forall\overline{k_1}.\tau_1[k]\ldots x_n::\forall\overline{k_n}.\tau_n[k] \\ \Gamma'' = \Gamma, x_1::\forall k_1.\sigma_1[k_1]\ldots x_n::\forall k_n.\sigma_n[k_n] \\ \Gamma_T^{T}{}' = \overline{b_1}\cup\ldots\cup\overline{b_n} \qquad \Gamma_T^{F}{}' = \overline{c_1}\cup\ldots\cup\overline{c_n} \qquad \overline{t} = \overline{a_1}\cup\ldots\cup\overline{a_n} \\ \overline{t^T} = \Gamma_T^T\backslash\overline{t} \qquad \overline{t^F} = \Gamma_T^F\backslash\overline{t} \qquad \overline{t^T}\subseteq\Gamma_T^T \qquad \overline{t^F}\subseteq\Gamma_T^F \\ \sigma_j[0] = \mathbf{U} \qquad \Gamma_T^{F}{}',\Gamma_T^{T}{}',\Gamma'\vdash e_j::\tau'_j \qquad \tau'_j\rhd\tau_j[k+1] \qquad \Gamma_T^F,\Gamma_T^T,\Gamma''\vdash e::\tau\end{array}}{\Gamma_T^F,\Gamma_T^T,\Gamma\vdash \mathbf{letrec}\ x_1::\forall k_1.\sigma_1[k_1] = e_1\ldots x_n::\forall k_n.\sigma_n[k_n] = e_n\ \mathbf{in}\ e::\tau}\ LETREC$$

$$\dfrac{\Gamma_T^F,\Gamma_T^T,\Gamma\cup\{x::\tau_1\}\vdash e_:\ :\tau_2}{\Gamma_T^F,\Gamma_T^T,\Gamma\vdash\lambda x.e::\tau_1\rightarrow\tau_2}\ ABS$$

$$\dfrac{\Gamma_T^F,\Gamma_T^T,\Gamma\vdash e_1::\tau_1\rightarrow\tau_2 \qquad \Gamma_T^F,\Gamma_T^T,\Gamma\vdash e_2::\tau_3 \quad \tau_3\rhd\tau_1}{\Gamma_T^F,\Gamma_T^T,\Gamma\vdash::\ e_1\ e_2::\tau_2}\ APP$$

$$\dfrac{\begin{array}{c}\vdash_=\tau = c_1\ \overline{\tau_1}\mid\ldots\mid c_n\ \overline{\tau_n} \\ \Gamma_T^F,\Gamma_T^T,\Gamma\vdash e::\tau \qquad \Gamma_T^F,\Gamma_T^T,\Gamma\cup\{x_{j1}::\tau_{j1},\ldots,x_{jn_j}::\tau_{jn_j}\}\vdash e_j::\tau'_j(\forall j) \qquad \tau'_j\rhd\tau'(\forall j)\end{array}}{\Gamma_T^F,\Gamma_T^T,\Gamma\vdash \mathbf{case}\ e\ \mathbf{of}\ c_1\ \overline{x_1}\rightarrow e_1\ldots c_n\ \overline{x_n}\rightarrow e_n::\tau'}\ CASE$$

$$\begin{array}{lll} P(\Gamma_T^F,\Gamma_T^T,\tau) = & \multicolumn{2}{l}{\mathit{if}\ (\tau = \mathbf{Strm}\ s\ \tau')\ \mathit{then}\ \Gamma_T^F\vdash_F\tau'} \\ & \multicolumn{2}{l}{\mathit{else\ if}\ (\tau = (\tau_1,\ldots,\tau_n))\ \mathit{then}\ \forall j.} \\ & & \mathit{if}\ \tau_j = \mathbf{Strm}\ s_j\ \tau'_j\ \mathit{then}\ \Gamma_T^F\vdash_F\tau'_j \\ & & \mathit{else}\quad \Gamma_T^F\cup\Gamma_T^T\vdash_F\tau_j \\ & \multicolumn{2}{l}{\mathit{else}\quad \Gamma_T^F\cup\Gamma_T^T\vdash_F\tau} \end{array}$$

Figura 6.4: Reglas de tipado

será ajustado en la mayoría de los casos. Si o es un *stream*, su tipo proporciona una cota inferior del número de elementos producidos, siempre que haya demanda para ellos. Con evaluación impaciente, la única diferencia es que dicha demanda está garantizada, con lo cual de nuevo obtendríamos una cota más ajustada. En ambos casos, la evaluación impaciente no obliga a modificar en modo alguno las reglas de tipo, pues estas siguen siendo correctas.

6.2.3 Tipos List y Strm

En Edén, el tipo lista $[\tau]$ se usa para listas Haskell y para canales tipo *stream*, y se transforman uno en el otro de forma transparente al programador. Sin embargo, en este sistema de tipos es necesario dividir las listas Haskell en finitas (tipo **List**) y parciales o infinitas (tipo **Strm**). Este es un problema heredado de Haskell. Un tipo **List** nos proporciona una prueba de terminación, mientras que un tipo **Strm** nos da una prueba de productividad.

Adicionalmente, es necesario identificar los canales tipo *stream*. Normalmente queremos demostrar la productividad de nuestros esqueletos, por lo que usaremos principalmente el tipo **Strm** en tales casos. Pero hay algunos esqueletos que trabajan con tipos finitos y requieren una versión con **List** de otro esqueleto. Por ejemplo, la versión con árboles *dcT* del esqueleto divide y vencerás de la Sección 6.3.4 se define normalmente usando una versión con listas de la topología de trabajadores replicados. En tales casos nos gustaría disponer de ambas versiones del esqueleto, una con listas y otra con *streams*, de forma que habríamos demostrado terminación y productividad. En algunos casos obtenemos ambas versiones gracias al polimorfismo, como en caso del esqueleto `map` ingenuo y la tubería, presentados a continuación.

6.2.4 Dos ejemplos sencillos

Estudiamos ahora dos ejemplos sencillos de esqueletos que ilustran algunas de las ideas desarrolladas en esta sección. En la siguiente sección estudiaremos esqueletos más complejos y los problemas que introducen. En todos ellos, mostraremos primero el esqueleto Edén tal y como se presenta en [PR01] y después veremos la versión con tipos con tamaño. En ocasiones, por razones técnicas (que se explicarán en cada caso), esta última tendrá ligeras modificaciones en su sintaxis con respecto a la versión original. A continuación procedemos a realizar la comprobación de tipos. Si bien no se muestran las demostraciones completas por ser muy pesadas, en aquellas funciones y esqueletos donde se ha usado la inducción para tiparlos, para ayudarnos a visualizar la demostración, escribiremos como subíndice los tamaños de aquellas variables de programa cuyos tipos contienen la variable de tamaño sobre la que se está haciendo inducción. En ocasiones escribiremos también el tamaño de expresiones completas para facilitar la comprensión.

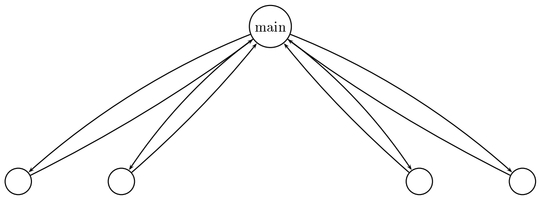


Figura 6.5: Topología map paralela

Cuando se use un tipo compuesto, como en **List** k (**Strm** l a), usaremos corchetes para representar el tipo; como en $k[l]$. En el texto original [PR01] se utiliza una clase `Transmissible` (abreviada `Tr`) que subsume a las clases T y F usadas en el sistema de tipos.

Una implementación ingenua de un esqueleto map paralelo

Comenzamos mostrando una implementación ingenua de un esqueleto `map` (ver Figura 6.5). Un esqueleto `map` toma una función y una lista de valores, y aplica la función a cada uno de los elementos de la lista:

```
map_naive :: (Tr a, Tr b) => (a -> b) -> [a] -> [b]
map_naive f xs = [pf # x | x <- xs]    where pf = process x -> f x
```

En esta versión ingenua, se crea un proceso para cada elemento de la lista, que simplemente aplica la función `f` al elemento correspondiente.

La notación ZF se reescribe como un simple map y la cláusula **where** como un **let**. El tipo se obtiene mediante composición de funciones:

$$\begin{aligned}
&map_naiveL :: \forall a,b,k.T\ a,b \Rightarrow (a \to b) \to \mathbf{List}\ k\ a \to \mathbf{List}\ k\ b\\
&map_naiveL = \lambda f.\lambda xs.\mathbf{let}\ g :: T\ a,b \Rightarrow a \to b\\
&\qquad\qquad\qquad\qquad\quad g = \lambda x.(\mathbf{process}\ y \to f\ y)\#x\\
&\qquad\qquad\qquad\qquad \mathbf{in}\ map\ g\ xs
\end{aligned}$$

Un esqueleto tubería

Una tubería consiste en una secuencia de pasos, de modo que en cada paso se aplica una función distinta al resultado obtenido por el paso anterior.

En el siguiente esqueleto tubería se lanza un nuevo proceso para evaluar cada uno de los pasos de la tubería. En la Figura 6.6 se muestra la topología de procesos generada. Cada proceso de la tubería crea su proceso sucesor:

```
map_pipe :: Tr a => [[a] -> [a]] -> [a] -> [a]
map_pipe fs xs  = (ppipe fs) # xs
ppipe :: Tr a => [[a] -> [a]] -> Process [a] [a]
ppipe [f]    = process xs -> f xs
ppipe (f:fs) = process xs -> (ppipe fs) # (f xs)
```

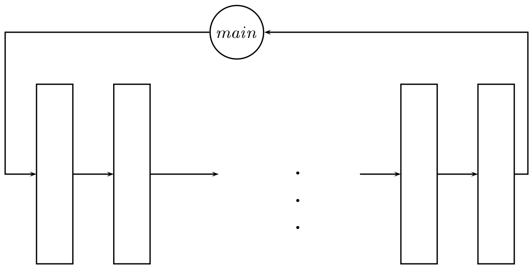


Figura 6.6: Topología generada para una tubería de procesos

El siguiente tipo se puede comprobar por inducción sobre la longitud k de fs:

$$\begin{array}{l} ppipe :: \forall a,k,l.F\ a \Rightarrow \textbf{List}\ k\ (\textbf{Strm}\ l\ a \rightarrow \textbf{Strm}\ l\ a) \rightarrow \textbf{Process}\ (\textbf{Strm}\ l\ a)\ (\textbf{Strm}\ l\ a) \\ ppipe = \lambda fs_{k+1}.\textbf{case}\ fs_{k+1}\ \textbf{of} \\ \qquad\qquad\qquad nil \qquad\quad \rightarrow \textbf{process}\ s \rightarrow s \\ \qquad\qquad\qquad cons\ f\ fs'_k \rightarrow \textbf{process}\ s \rightarrow (ppipe\ fs'_k)\#(f\ s) \end{array}$$

$$\begin{array}{l} map\_pipe :: \forall a,k,l.F\ a \Rightarrow \textbf{List}\ k\ (\textbf{Strm}\ l\ a \rightarrow \textbf{Strm}\ l\ a) \rightarrow \textbf{Strm}\ l\ a \rightarrow \textbf{Strm}\ l\ a \\ map\_pipe = \lambda fs.\lambda s.(ppipe\ fs)\#s \end{array}$$

Suponiendo que $ppipe$ :: **List** k (**Strm** l $a \rightarrow$ **Strm** l a) $\rightarrow$ **Process** (**Strm** l a) (**Strm** l a) se demuestra que el lado derecho tiene tipo **List** $(k+1)$ (**Strm** l a $\rightarrow$ **Strm** l a) $\rightarrow$ **Process** (**Strm** l a) (**Strm** l a).

En este ejemplo nos hemos encontrado con un par de problemas. En primer lugar, el caso de la lista vacía no está incluido en la definición de `ppipe` en el programa original, lo que viola las restricciones del **case**. De modo que para poder dar al esqueleto un tipo, añadimos la cláusula que en tal caso devuelve el proceso identidad.

En segundo lugar, hemos decidido representar los procesos de forma que consuman y produzcan un *stream* de datos para demostrar su productividad. Esto significa que el tipo a no puede ser un tipo interfaz, sino que debe ser un tipo finito, luego el contexto en este caso es F y no T. Sin embargo, podríamos proporcionar tipos menos restrictivos como $ppipe :: \forall a,k,l.T\ a \Rightarrow$ **List** k $(a \rightarrow a) \rightarrow$ **Process** a a y $map\_pipe :: \forall a,k,l.T\ a \Rightarrow$ **List** k $(a \rightarrow a) \rightarrow a \rightarrow a$, en donde sí hemos podido usar la clase T. Observamos en tal caso que si concretamos a con $F\ a \Rightarrow$ **Strm** l a obtenemos el tipo derivado anteriormente.

6.3 Algunos esqueletos en Edén

6.3.1 Esqueleto divide y vencerás ingenuo

Un esquema de programación muy conocido es *divide y vencerás*. En él, si el problema a resolver es lo suficientemente simple (caso base), su solución se calcula directamente. En caso contrario, el problema se divide en varios subproblemas de la misma naturaleza, pero menor tamaño, que se resuelven independientemente y cuyos resultados se combinan para obtener la solución final para el problema original. Utilizando un lenguaje funcional, este esquema se puede expresar de la siguiente forma:

```
dc::(a -> Bool) -> (a -> b) -> (a -> [a]) -> (a-> [b] -> b) -> a -> b
dc trivial solve split combine x
   | trivial x    = solve x
   | otherwise    = combine x children
   where children = map (dc trivial solve split combine) (split x)
```

donde `trivial` es la función que indica si estamos en un caso base, `solve` indica cómo resolver el problema en tales casos, `split` divide el problema en subproblemas y `combine` combina las soluciones de los subproblemas para obtener una solución del problema dividido.

En una versión paralela ingenua de divide y vencerás, se crea un árbol dinámico de procesos en el que cada proceso está conectado con su padre (ver Figura 6.7). Un parámetro entero determina el máximo nivel tras el cual no se generan más procesos hijo, en cuyo caso se usa la versión secuencial `dc` en su lugar. La implementación en Edén es la siguiente:

```
dc_naive::(Tr a, Tr b) => Int -> (a -> Bool) -> (a -> b) -> (a -> [a]) ->
          (a -> [b] -> b) -> a -> b
dc_naive 0 trivial solve split combine   = dc trivial solve split combine
dc_naive depth trivial solve split combine x
   | trivial x    = solve x
   | otherwise    = combine x children
   where children = map_naive
                     (dc_naive (depth-1) trivial solve split combine)
                     (split x)
```

Obsérvese que el árbol de llamadas resultante no es necesariamente homogéneo y que pueden aparecer soluciones triviales a cualquier profundidad. Esta versión no es satisfactoria, pues se crean demasiados procesos. En la Sección 6.3.4 se presentará una versión mejorada.

La traducción correspondiente en el sistema de tipos es muy similar:

$$
\begin{array}{l}
\mathit{dc\_naive} :: \forall a,b,k,l.T\ a,b \Rightarrow \mathit{Nat}\ k \to (a \to \mathit{Bool}) \to (a \to b) \to (a \to \mathbf{List}\ l\ a) \to \\
\qquad\qquad\qquad\qquad (a \to \mathbf{List}\ l\ b \to b) \to a \to b \\
\mathit{dc\_naive} = \lambda n_{k+1}.\lambda \mathit{trivial}.\lambda \mathit{solve}.\lambda \mathit{split}.\lambda \mathit{combine}.\lambda x. \\
\qquad\quad \mathbf{case}\ n\ \mathbf{of} \\
\qquad\qquad \mathit{zero} \quad \to \mathit{dc\ trivial\ solve\ split\ combine}\ x \\
\qquad\qquad \mathit{succ}\ n'_k \to \mathbf{case}\ (\mathit{trivial}\ x)\ \mathbf{of} \\
\qquad\qquad\qquad \mathit{True} \to \mathit{solve}\ x \\
\qquad\qquad\qquad \mathit{False} \to \mathit{combine}\ x\ (\mathit{map\_naiveL} \\
\qquad\qquad\qquad\qquad\qquad (\mathit{dc\_naive}\ n'_k\ \mathit{trivial\ solve\ split\ combine})\ (\mathit{split}\ x))
\end{array}
$$

El tipo obtenido demuestra que el programa termina. Para obtener este tipo (suponiendo que el tipo del algoritmo secuencial ha sido previamente demostrado) se ha aplicado inducción sobre la profundidad k del árbol:

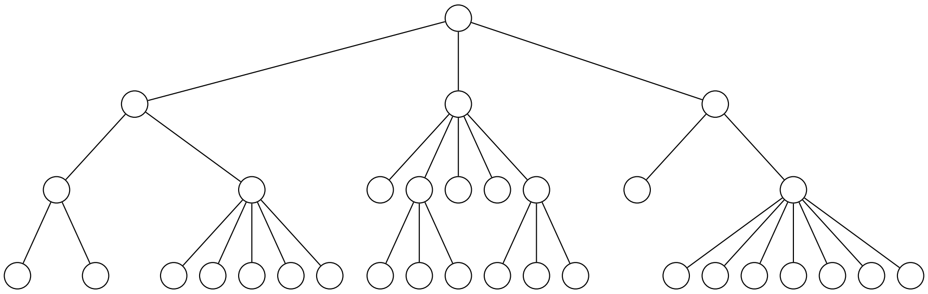


Figura 6.7: Ejemplo de una topología de procesos generada por divide y vencerás

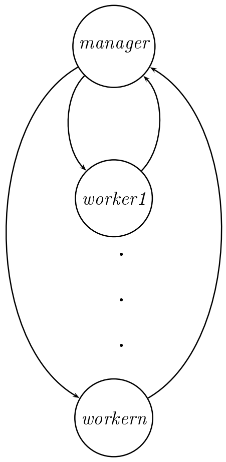


Figura 6.8: Topología de procesos generada por una granja

suponemos que $dc\_naive :: Nat\ k \to (a \to Bool) \to (a \to b) \to (a \to \mathbf{List}\ l\ a) \to (a \to \mathbf{List}\ l\ b \to b) \to a \to b$ y demostramos que el lado derecho tiene tipo $Nat\ (k+1) \to (a \to Bool) \to (a \to b) \to (a \to \mathbf{List}\ l\ a) \to (a \to \mathbf{List}\ l\ b \to b) \to a \to b$.

6.3.2 El esqueleto map paralelo implementado como granja

La versión ingenua `map_naive` se puede mejorar reduciendo el número de procesos trabajadores que se crean.

En una granja `map_farm` se crea un proceso por cada procesador y las tareas se distribuyen equitativamente entre los procesadores. Tendremos un proceso idéntico en cada uno de los procesadores disponibles. Cada trabajador aplica la función f a cada tarea de su sublista y el gestor recolecta los resultados de todos los trabajadores. La topología generada por este esqueleto se muestra en la Figura 6.8. En ella, el proceso `manager` distribuye las tareas entre los procesos trabajadores `worker`.

Es un esquema apropiado cuando la granularidad de las tareas es uniforme y se desea distribuir un cierto número de tareas entre los procesadores. Para mejorar el reparto de carga la longitud de la lista debe ser mucho mayor que el número de procesadores disponibles.

La implementación, en términos de `map_naive`, es la siguiente:

```
map_farm :: (Tr a,Tr b) => (a -> b) -> [a] -> [b]
map_farm = farm noPe unshuffle shuffle
farm :: (Tr a, Tr b) => Int -> (Int->[a]->[[a]]) -> ([[b]]->[b]) ->
        (a -> b) -> [a] -> [b]
farm np unshuffle shuffle f tasks =
                      shuffle (map_naive (map f) (unshuffle np tasks))
```

donde `noPe` es una constante Edén que devuelve el número de procesadores disponibles, la función `unshuffle` se encarga de dividir las tareas entre los procesos y la función `shuffle` recolecta los resultados obtenidos y los mezcla de la forma adecuada. Se pueden utilizar distintas estrategias para dividir el trabajo entre los distintos procesos, siempre que se cumpla que, para toda lista `xs`, `(shuffle . unshuffle) xs == xs`. Por ejemplo, el siguiente esquema distribuye las tareas usando una estrategia *round-robin*:

```
unshuffle :: Int -> [a] -> [[a]]
unshuffle n ins
   | length firsts < n  = take n (map (:[]) firsts ++ repeat [])
   | otherwise          = zipWith (:) firsts (unshuffle n rest)
  where (firsts, rest) = splitAt n ins
shuffle :: [[a]] -> [a]
shuffle = concat . transpose
```

En la Figura 6.10 se muestran los tipos de las funciones auxiliares usadas en el esqueleto granja. Los tipos de *zipWiths*, *takes*, *drops* y $(++)_s$ se demuestran por inducción sobre k. Obsérvese el uso de la relación de subtipado $\rhd$. En la Figura 6.11 se presenta el esqueleto granja modificado con su tipo. Los tipos de las funciones $\mathit{unshuffles}^n$ y *shuffles* se demuestran por inducción sobre l.

Primero es necesario decidir qué tipos lista son listas finitas y cuáles son *streams*. Para estudiar la productivdad hemos elegido tipos *stream* para las listas de entrada `[a]` y salida `[b]` del esqueleto. Puesto que el número de procesos es finito, la distribución de tareas entre procesos se considera como una lista de *streams*. Esta decisión nos lleva a cambiar ligeramente las definiciones de `shuffle` y `unshuffle`, por lo que ahora tienen un tipo diferente. En particular, `shuffle` necesita un *stream* auxiliar para poder tratar el caso en que la lista de canales es vacía, a pesar de que esta situación nunca se dará en la práctica, pues correspondería a tener cero procesadores.

Se han encontrado algunas dificultades al tipar este esqueleto: el primero surge al tipar `unshuffle`. Se está dividiendo un *stream* de elementos en

$$\frac{i = 0}{i * i' = 0} \qquad \frac{i' = 0}{i * i' = 0} \qquad \frac{i_1 = 0 \quad i_2 = 0}{i_1 + i_2 = 0} \qquad \frac{i_1 \neq 0 \quad i_2 \neq 0}{i_1 * i_2 \neq 0} \qquad \frac{i_1 \neq 0}{i_1 + i_2 \neq 0} \qquad \frac{i_2 \neq 0}{i_1 + i_2 \neq 0}$$

$$\frac{i_1 \overset{+}{\sim} k \quad i_2 \overset{+}{\sim} k}{i_1 * i_2 \overset{+}{\sim} k} \qquad \frac{i_1 \overset{+}{\sim} k \quad \imath_2 \overset{-}{\sim} k}{i_1 * i_2 \overset{+}{\sim} k} \qquad \frac{i_1 \overset{-}{\sim} k \quad \imath_2 \overset{+}{\sim} k}{i_1 * i_2 \overset{+}{\sim} k} \qquad \frac{i_1 \overset{-}{\sim} k \quad i_2 \overset{-}{\sim} k}{i_1 * i_2 \overset{-}{\sim} k}$$

Figura 6.9: Reglas adicionales para $= 0$, $\neq 0$, $\overset{+}{\sim}k$ y $\overset{-}{\sim}k$.

n listas de *streams*. Esto significa que el *stream* original debería tener al menos $n * k$ elementos, de forma que se puedan obtener n listas de al menos k elementos. Esto conlleva utilizar productos de variables de tamaño, lo que no está permitido en el sistema de tipos. Hay dos posibles soluciones a este problema. Una consiste en definir una familia *unshuffles*^{n} de funciones, una para cada número fijo de procesos n, de forma que $n * k$ sea el producto de una constante por una variable. Así el parámetro natural desaparecería, y consecuentemente *farm*^{n} y *map_farm*^{n} también serían familias de funciones. El inconveniente de esta alternativa es que hace falta definir muchas versiones del mismo esqueleto, una para cada n que pretendamos utilizar en nuestros programas. Otra posibilidad es permitir productos de dos (o más) variables, o de expresiones de tamaño, para obtener así un esqueleto paramétrico. En este caso se deberían añadir nuevas reglas para comprobar las relaciones $= 0$, $\neq 0$, $\overset{+}{\sim}k$ y $\overset{-}{\sim}k$, mostradas en la Figura 6.9. El inconveniente de esta alternativa es que los productos de variables no están permitidos en el sistema de tipos, e incluso puede que hagan indecidible el algoritmo de chequeo de tipos.

El segundo problema también está relacionado con esta $unshuffle$. El tipo de su resultado es una lista de *streams*. La comprobación de ínfimo falla cuando $l = 0$, ya que **List** n $\mathbf{U} \neq \mathbf{U}$. Este problema surge cuando usamos una lista o una tupla para representar varios canales que salen de un proceso. Un ejemplo más simple con tuplas es la siguiente versión de *unshuffle* para dos *streams*:

$$\begin{array}{l}
\textit{unshuffle2} :: \forall a, k.\mathbf{Strm}\ (2k)\ a \rightarrow < \mathbf{Strm}\ k\ a, \mathbf{Strm}\ k\ a > \\
\textit{unshuffle2} = \lambda s_{2(k+1)}.\mathbf{case}\ s_{2(k+1)}\ \mathbf{of} \\
\qquad\qquad\qquad x; {s'}_{2k+1} \rightarrow \mathbf{case}\ {s'}_{2k+1}\ \mathbf{of} \\
\qquad\qquad\qquad\quad y; {s''}_{2k} \rightarrow \mathbf{let}\ < s1_k, s2_k > = \textit{unshuffle2}\ {s''}_{2k}\ \mathbf{in}\ \ ((x; s_1)_{k+1}, (y; s_2)_{k+1})
\end{array}$$

En [Par00] surgió un problema similar al intentar tipar definiciones mutuamente recursivas. Fue solucionado construyendo una definición especial de punto fijo para un conjunto de ecuaciones simultáneas. Una función de tuplas en tuplas no funcionaría en principio ya que la comprobación de ínfimo fallaría puesto que $(\mathbf{U}, \mathbf{U}) \neq \mathbf{U}$. La solución que nosotros proponemos es eliminar la marca de las tuplas, es decir, definir tuplas estrictas. Definimos los tipos de datos estrictos **sdata** (T_s) y **sidata** (T_{si}), y añadimos

$$
\begin{array}{l}
\textbf{sidata sList}\ w\ a = \mathit{snil} \mid \mathit{scons}\ a\ (\textbf{sList}\ w\ a)\\
\mathit{zipWiths} :: \forall a,b,c,k.(a \to b \to c) \to \textbf{sList}\ k\ a \to \textbf{sList}\ k\ b \to \textbf{sList}\ k\ c\\
\mathit{zipWiths} = \lambda f.\lambda xs_{k+1}.\lambda ys_{k+1}.\textbf{case}\ xs_{k+1}\ \textbf{of}\\
\qquad\qquad \mathit{snil} \to \mathit{snil}_{1\rhd(k+1)}\\
\qquad\qquad \mathit{scons}\ x\ xs'_k \to \textbf{case}\ ys_{k+1}\ \textbf{of}\\
\qquad\qquad\qquad \mathit{snil} \to \mathit{snil}_{1\rhd(k+1)}\\
\qquad\qquad\qquad \mathit{scons}\ y\ ys'_k \to (\mathit{scons}\ (f\ x\ y)\ (\mathit{zipWiths}\ f\ xs'_k\ ys'_k)_k)_{k+1}
\end{array}
$$

$$
\begin{array}{l}
\mathit{takes} :: \forall a,k,l.Nat\ k \to \textbf{Strm}\ (k+l)\ a \to \textbf{sList}\ k\ a\\
\mathit{takes} = \lambda n_{k+1}.\lambda s_{k+l+1}.\textbf{case}\ n_{k+1}\ \textbf{of}\ \mathit{zero} \to \mathit{snil}_{1\rhd k+1}\\
\qquad\qquad \mathit{succ}\ n'_k \to \textbf{case}\ s_{k+l+1}\ \textbf{of}\\
\qquad\qquad\qquad x; s'_{k+l} \to (\mathit{scons}\ x\ (\mathit{takes}\ n'_k\ s'_{k+l})_k)_{k+1}\\
\mathit{drops} :: \forall a,k,l.Nat\ k \to \textbf{Strm}\ (k+l)\ a \to \textbf{Strm}\ l\ a\\
\mathit{drops} = \lambda n_{k+1}.\lambda s_{k+l+1}.\textbf{case}\ n_{k+1}\ \textbf{of}\ \mathit{zero} \to s_{k+l+1\rhd l}\\
\qquad\qquad \mathit{succ}\ n'_k \to \textbf{case}\ s_{k+l+1}\ \textbf{of}\\
\qquad\qquad\qquad x; s'_{k+l} \to (\mathit{drops}\ n'_k\ s'_{k+l})_l\\
\mathit{splitAts} :: \forall a,k,l.Nat\ k \to \textbf{Strm}\ (k+l)\ a \to (\textbf{sList}\ k\ a, \textbf{Strm}\ l\ a)\\
\mathit{splitAts} = \lambda n.\lambda s.(\mathit{takes}\ n\ s, \mathit{drops}\ n\ s)\\
(+\!\!+_s) :: \forall a,k,l.\textbf{sList}\ k\ a \to \textbf{Strm}\ l\ a \to \textbf{Strm}\ l\ a\\
xs_{k+1}\ +\!\!+_s\ s = \textbf{case}\ xs_{k+1}\ \textbf{of}\ \mathit{snil} \to s_l;\quad \mathit{scons}\ x\ xs'_k \to (x; (xs'_k\ +\!\!+_s\ s)_l)_{l+1\rhd l}
\end{array}
$$

Figura 6.10: Funciones auxiliares para el esqueleto granja

$$
\begin{array}{l}
\mathit{farm}^n :: \forall a,b,l.F\ a,b \Rightarrow (\textbf{Strm}\ (n*l)\ a \to \textbf{sList}\ n\ (\textbf{Strm}\ l\ a)) \to\\
\qquad\qquad (\textbf{sList}\ n\ (\textbf{Strm}\ l\ a) \to \textbf{Strm}\ l\ b \to \textbf{Strm}\ l\ b) \to\\
\qquad\qquad (a \to b) \to \textbf{Strm}\ l\ b \to \textbf{Strm}\ (n*l)\ a \to \textbf{Strm}\ l\ b\\
\mathit{farm}^n = \lambda \mathit{unshuffle}^n.\lambda \mathit{shuffle}.\lambda f.\lambda \mathit{aux}.\lambda s.\\
\qquad \mathit{shuffle}\ (\mathit{map\_naiveLs}\ (\mathit{mapS}\ f)\ (\mathit{unshuffle}^n\ s))\ \mathit{aux}
\end{array}
$$

$$
\begin{array}{l}
\mathit{map\_farm}^n :: \forall a,b,k.F\ a,b \Rightarrow (a \to b) \to \textbf{Strm}\ k\ b \to \textbf{Strm}\ (n*k)\ a \to \textbf{Strm}\ k\ b\\
\mathit{map\_farm}^n = \mathit{farm}^n\ \mathit{unshuffles}^n\ \mathit{shuffles}
\end{array}
$$

$$
\begin{array}{l}
\mathit{unshuffles}^n :: \forall a,l.\textbf{Strm}\ (n*l)\ a \to \textbf{sList}\ n\ (\textbf{Strm}\ l\ a)\\
\mathit{unshuffles}^n = \lambda s_{n(l+1)}.\textbf{let}\ (\mathit{firsts}_n, \mathit{rest}_{nl}) = \mathit{splitAts}\ n\ s_{n(l+1)}\\
\qquad\qquad \textbf{in}\ (\mathit{zipWiths}\ (;)\ \mathit{firsts}_n\ (\mathit{unshuffles}^n\ \mathit{rest}_{nl})_{n[l]})_{n[l+1]}
\end{array}
$$

$$
\begin{array}{l}
\mathit{shuffles} :: \forall a,l,k.\textbf{sList}\ (k+1)\ (\textbf{Strm}\ l\ a) \to \textbf{Strm}\ l\ b \to \textbf{Strm}\ l\ b\\
\mathit{shuffles} = \lambda xs_{k+1[l+1]}.\lambda \mathit{aux}_{l+1}.\textbf{case}\ xs_{k+1[l+1]}\ \textbf{of}\\
\qquad\qquad \mathit{snil} \to \mathit{aux}_{l+1}\\
\qquad\qquad \mathit{scons}\ s_{l+1}\ xs'_{k[l+1]} \to \textbf{case}\ s_{l+1}\ \textbf{of}\\
\qquad\qquad\qquad x; s'_l \to \textbf{let}\ \mathit{heads}_k = \mathit{map}\ \mathit{hdS}\ xs'_{k[l+1]}\\
\qquad\qquad\qquad\quad \textbf{in let}\ \mathit{tls}_{k+1[l]} = \mathit{map}\ \mathit{tlS}\ xs_{k+1[l+1]}\\
\qquad\qquad\qquad\quad \textbf{in}\ (x; (\mathit{heads}_k\ +\!\!+_s\ (\mathit{shuffles}\ \mathit{tls}_{k+1[l]}\ \mathit{aux})_l)_l)_{l+1}
\end{array}
$$

Figura 6.11: El esqueleto granja

una nueva regla para representar la estrictez, y otra para establecer cuándo un **sidata** es vacío:

$$\frac{\exists i.\tau_i = \mathbf{U}}{T_s/T_{si}\ \overline{s}\ \overline{\tau} = \mathbf{U}} \qquad\qquad \frac{s_1 = 0}{T_{si}\ \overline{s}\ \overline{\tau} = E}$$

De aquí en adelante se usará $\langle\ \rangle$ para representar tuplas estrictas. Estas se utilizaron también en la regla $[MERGE]$ (ver Figura 6.4). Las listas estrictas de definen en la Figura 6.10. Para que las reglas de estrictez sean semánticamente correctas, definimos un nuevo operador de tipos $\boxed{\times'}$ utilizado para interpretar esta clase de tipos: $\mathcal{T}_1 \boxed{\times'} \mathcal{T}_2 = \begin{cases} \mathcal{T}_1 \boxed{\times} \mathcal{T}_2 \text{ si } \mathcal{T}_1, \mathcal{T}_2 \neq \mathbf{U} \\ \mathbf{U} \text{ e.o.c.} \end{cases}$.

6.3.3 Topología de trabajadores replicados

El reparto de carga entre los procesos puede ser pobre si usamos el esqueleto granja en tres situaciones: (1) cuando la granularidad de las tareas no es uniforme, (2) cuando la arquitectura de los procesadores no es regular, y (3) cuando el programa debe compartir la CPU con otros programas que están ejecutándose en el mismo procesador. La solución a las tres situaciones es repartir el trabajo bajo demanda, es decir, se asigna una tarea a un proceso solamente cuando se sabe que ha terminado su trabajo previo. Esta es la idea de la topología de trabajadores replicados [KPR01]. En esta topología se crea un único proceso repartidor, que llamaremos `manager` y muchos procesos trabajadores o `worker`. El número de trabajadores depende de un parámetro (suele ser el número de procesadores disponibles). Inicialmente, el `manager` asigna una o más tareas a cada uno de los trabajadores. Cada vez que un trabajador termina su tarea, envía un mensaje, incluyendo el resultado parcial, a `manager` y a continuación `manager` le asigna una nueva tarea. El cómputo termina cuando `manager` recibe todos los resultados parciales.

Asignar inicialmente sólo una tarea a cada proceso tiene una desventaja, especialmente cuando la latencia es alta: cuando un trabajador termina un trabajo parcial, envía un mensaje al manager y debe esperar a recibir una nueva tarea. Si el manager enviara inicialmente dos tareas, no haría falta que el proceso esperara sino que inmediatamente podría empezar a trabajar con las tareas pendientes, con lo que los periodos de espera quedan minimizados.

El programador no puede predecir con anterioridad el orden en que los procesos van a terminar sus trabajos, puesto que esto depende de lo que suceda en tiempo de ejecución.

Usando el proceso reactivo (y no determinista) `merge`, el manager puede recibir las contestaciones de distintos procesos tan pronto como estas se producen. Luego, si cada contestación contiene la identidad del proceso emisor, se puede saber quién ha enviado el primer mensaje y asignarle una nueva tarea. La Figura 6.12 muestra la topología de procesos generada.

El esqueleto recibe como parámetros de entrada (1) el número de procesos trabajadores a utilizar; (2) el tamaño de la cola de tareas adelantadas de

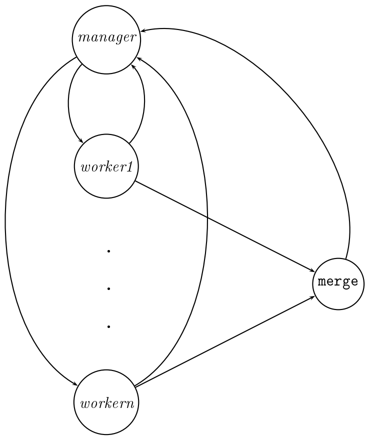


Figura 6.12: Topología de procesos generada para los trabajadores replicados

los trabajadores; (3) la función del trabajador que lleva a cabo el verdadero trabajo sobre las tareas; y (4) la lista de tareas en las que se ha dividido el problema. El esqueleto es el siguiente:

```
rw :: (Tr a, Tr b) => Int -> (a->b) -> [a] -> [b]
rw np prefetch fw tasks =   results    where
  results = sortMerge outputsChildren
  outputsChildren =  [(worker fw i) # inputs
                        |(i,inputs) <- zip [0..np-1] inputss]
  inputss          = distribute tasksAndIds
                        (initReqs ++ (map owner unorderedResult))
  tasksAndIds      = zip [1..] tasks
  initReqs         = concat (generate prefetch [0..np-1])
  unorderedResult = merge # outputsChildren   -- Non-deterministic!!

  distribute [] _          = generate np []
  distribute (e:es) (i:is) = insert i e (distribute es is)
    where insert 0 e ~(x:xs) = (e:x):xs
          insert (n+1) e ~(x:xs) = x:(insert n e xs)

worker :: (Tr a, Tr b) => (a->b) -> Int -> Process [(Int,a)]  [ACK b]
worker f i = process ts -> map (\(id_t,t).ACK i id_t (f t)) ts

data ACK b =  ACK Int Int b
owner (ACK i _ _ )  = i
```

La lista `tasksAndIds` genera los números que se asignan a cada tarea. Dichos números serán utilizados por la función `sortMerge` (que aquí no se muestra) para ordenar los resultados en el mismo orden que las correspondientes tareas de entrada.

Se define un tipo de datos `ACK` para los mensajes de contestación de los trabajadores al manager. Incluye no solamente el resultado calculado

$$
\begin{array}{l}
\textbf{data } ACK\ k\ b = ACK\ (Nat\ k)\ b \\
generate :: \forall k.Nat\ k \rightarrow \textbf{Strm}\ k\ (Nat\ k) \\
generate = \lambda n_{k+1}.\textbf{case } n_{k+1} \textbf{ of } zero \quad \rightarrow zeros_{\omega[1] \triangleright k+1[k+1]}; \\
\qquad\qquad\qquad\qquad\qquad\qquad succ\ n'_k \rightarrow (n_{k+1}; (generate\ n'_k)_{k[k]})_{k+1[k+1]}
\end{array}
$$

$$
\begin{array}{l}
zipS :: \forall a, b, k.\textbf{Strm}\ k\ a \rightarrow \textbf{Strm}\ k\ b \rightarrow \textbf{Strm}\ k\ (a, b) \\
zipS = \lambda s_{k+1}.\lambda t_{k+1}.\textbf{case } s_{k+1} \textbf{ of } x; s'_k \rightarrow \\
\qquad\qquad\qquad \textbf{case } t_{k+1} \textbf{ of } y; t'_k \rightarrow ((x, y); (zipS\ s'_k\ t'_k)_k)_{k+1}
\end{array}
$$

$$
\begin{array}{l}
owner :: \forall b, k.ACK\ k\ b \rightarrow Nat\ k \\
owner\ (ACK\ i\ \_) = i \\
result :: \forall b, k.ACK\ k\ b \rightarrow b \\
result\ (ACK\ \_\ b) = b
\end{array}
$$

$$
\begin{array}{l}
worker :: \forall a, b, k, l.F\ a, b \Rightarrow (a \rightarrow b) \rightarrow Nat\ k \rightarrow \textbf{Process}\ (\textbf{Strm}\ l\ a)\ (\textbf{Strm}\ l\ (ACK\ k\ b)) \\
worker = \lambda f.\lambda n.\textbf{process}\ ts \rightarrow \textbf{let } f' :: F\ a, b \Rightarrow a \rightarrow ACK\ k\ b \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad f' = \lambda t.ACK\ n\ (f\ t) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } (mapS\ f'\ ts)
\end{array}
$$

$$
\begin{array}{l}
mapt^n :: \forall a, b, k_1, \ldots, k_n.F\ a, b \Rightarrow (a \rightarrow b) \rightarrow < \textbf{Strm}\ k_1\ a, \ldots, \textbf{Strm}\ k_n\ a > \rightarrow \\
\qquad\qquad\qquad\qquad\qquad < \textbf{Strm}\ k_1\ (ACK\ n\ b), \ldots, \textbf{Strm}\ k_n\ (ACK\ n\ b) > \\
mapt^n = \lambda w.\lambda < s_1, \ldots, s_n > . < (worker\ w\ 0)\#s_1, \ldots, (worker\ w\ (n-1))\#s_n >
\end{array}
$$

Figura 6.13: Funciones auxiliares para la topología de trabajadores replicados

por el trabajador sino también la identidad del proceso y de la tarea. La lista `unorderedResult` es la lista no determinista generada por `merge`, que es utilizada por `distribute` para distribuir nuevas tareas tan pronto como llegan mensajes de respuesta de los trabajadores. En [KPR01] se pueden encontrar más detalles.

En la Figura 6.13 se muestran los tipos de las funciones auxiliares usadas en esta topología. Los tipos de las funciones *generate* y *zipS* se demuestran por inducción sobre k. Obsérvese que en *generate* hacemos uso de la relación de subtipado. Estamos suponiendo que $mapS :: \forall a, b, k.(a \rightarrow b) \rightarrow \textbf{Strm}\ k\ a \rightarrow \textbf{Strm}\ k\ b$.

En la Figura 6.14 se muestra el tipo de una versión modificada de la topología. Hemos simplificado algunos aspectos. Ha sido eliminado el parámetro `prefetch`. El identificador de tarea también se ha eliminado del tipo *ACK*, por lo que eliminamos la función de ordenación devolviendo el resultado desordenado.

Se han encontrado varios problemas al tipar este esqueleto. El primero es el siguiente: puesto que esta es una topología en la que el trabajo se distribuye bajo demanda, los tamaños de los *streams* que comunican los procesos no son necesariamente los mismos, por lo que trabajar con listas de *streams* no parece apropiado, ya que en tal caso se perdería mucha información. Por ello hemos decidido usar tuplas estrictas de *streams* de diferentes tamaños:

$$
\begin{array}{l}
rw^n :: \forall a, b.F\ a, b \Rightarrow (a \rightarrow b) \rightarrow \mathbf{Strm}\ w\ a \rightarrow \mathbf{Strm}\ w\ b \\
rw^n = \lambda f.\lambda tasks. \\
\quad \mathbf{let}\ initReqs :: \mathbf{Strm}\ n\ (Nat\ n) \\
\quad\quad initReqs = generate\ n \\
\quad \mathbf{in\ let\ rec} \\
\quad\quad inputss :: \forall k_1 \ldots k_n.F\ a \Rightarrow < \mathbf{Strm}\ k_1\ a, \ldots, \mathbf{Strm}\ k_n\ a > \\
\quad\quad inputss = \mathbf{let}\ \ oChildren :: F\ b \Rightarrow < \mathbf{Strm}\ k_1\ (ACK\ n\ b), \ldots, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{Strm}\ k_n\ (ACK\ n\ b) > \\
\quad\quad\quad\quad\quad oChildren = mapt^n\ f\ inputss \\
\quad\quad\quad\quad \mathbf{in\ let}\ unordered :: F\ b \Rightarrow \mathbf{Strm}\ (\sum_{i=1}^{n} k_i)\ (ACK\ n\ b) \\
\quad\quad\quad\quad\quad unordered = merge\#oChildren \\
\quad\quad\quad\quad \mathbf{in\ let}\ restReqs :: \mathbf{Strm}\ (\sum_{i=1}^{n} k_i)\ (Nat\ n) \\
\quad\quad\quad\quad\quad restReqs = mapS\ owner\ unordered \\
\quad\quad\quad\quad \mathbf{in\ let}\ requests :: \mathbf{Strm}\ (\sum_{i=1}^{n} k_i + n)\ (Nat\ n) \\
\quad\quad\quad\quad\quad requests = initReqs{+\!\!+}^n restReqs \\
\quad\quad\quad\quad \mathbf{in}\ distribute^n\ tasks\ requests \\
\quad \mathbf{in\ let} \\
\quad\quad outputChildren :: \forall k_1 \ldots k_n.F\ b \Rightarrow < \mathbf{Strm}\ k_1\ (ACK\ n\ b), \ldots, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{Strm}\ k_n\ (ACK\ n\ b) > \\
\quad\quad outputChildren = mapt^n\ f\ inputss \\
\quad \mathbf{in}\ mapS\ result\ (merge\#outputChildren)
\end{array}
$$

$$
\begin{array}{l}
distribute^n :: \forall a, k_1, \ldots k_n.\mathbf{Strm}\ w\ a \rightarrow \mathbf{Strm}(\sum_{k=1}^{n} k_i)\ (Nat\ n) \rightarrow \\
\quad\quad\quad\quad\quad\quad < \mathbf{Strm}\ k_1\ a, \ldots, \mathbf{Strm}\ k_n\ a > \\
distribute^n = \lambda s_1.\lambda s_2.\mathbf{case}\ s_1\ \mathbf{of}\ x_1; s_1' \rightarrow \mathbf{case}\ s_2\ \mathbf{of}\ x_2; s_2' \rightarrow \\
\quad\quad\quad\quad\quad\quad \mathbf{case}\ (distribute^n\ s_1'\ s_2')\ \mathbf{of}\ < t_1, \ldots, t_n > \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{if}\ (x_2 == 0)\ then\ (x_1; t_1, \ldots, t_n) \\
\quad\quad\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{else}\ \{-(x_2 == n-1)-\}\ (t_1, \ldots, x_1; t_n)
\end{array}
$$

Figura 6.14: La topología de trabajadores replicados

$\langle \mathbf{Strm}\ k_1\ a, \ldots, \mathbf{Strm}\ k_n\ a\rangle$. Esto conlleva que la topología pase a ser una familia de funciones, una para cada cantidad de procesadores. Luego eliminamos el parámetro natural.

El segundo problema es que tipar el **let** recursivo implica demostrar que todos los *streams* crecen al menos en un elemento, y esto no es cierto siempre. Por ejemplo, para `outputChildren`. Sin embargo, se puede observar que una vez se aplica `distribute`, se dará al menos una tarea a un proceso en `inputss` por lo que la topología continuará funcionando. Redefinimos el **let** recursivo definiendo una única ligadura recursiva para *inputss* en términos del resto de ligaduras. De modo que ahora sólo hace falta demostrar que *inputss* crece. Esto es cierto al comienzo de la ejecución gracias a la lista inicial de peticiones *initReqs*. El resto de peticiones se concatenan tras ellas: `initReqs ++ map owner unorderedResult`.

El tercer problema surge precisamente en la función de concatenación. Hemos de concatenar una lista finita de longitud n a un *stream* de al menos k elementos. En el sistema de tipos **List** n a significa que la lista tiene a lo sumo una longitud de n, por lo que no podemos decir con seguridad que el *stream* resultante tenga al menos $n + k$ elementos. Necesitamos usar un *stream* también para el primer parámetro. Dicho *stream* es *generate n* en la Figura 6.14. Genera las primeras n peticiones y después añade al final un número infinito de 0's. Luego, dados dos *streams*, la nueva función de concatenación toma n elementos de la primera y los pone al principio de la segunda. Esto sigue sin resolver el problema del todo, puesto que tenemos que tomar n elementos del primer *stream* y de nuevo los números naturales son inductivos. Luego necesitamos definir (inductivamente) una familia de funciones de concatenación $\{+\!\!+^j :: \forall a,k.\mathbf{Strm}\ j\ a \to \mathbf{Strm}\ k\ a \to \mathbf{Strm}\ (k+j)\ a\}_{j\in\mathbb{N}^+}$, donde ahora cada función se define en términos de la anterior:

$$+\!\!+^1 :: \forall a,k.\mathbf{Strm}\ 1\ a \to \mathbf{Strm}\ k\ a \to \mathbf{Strm}\ (k+1)\ a$$
$$s +\!\!+^1 s' = \mathbf{case}\ s\ \mathbf{of}\ x; s'' \to x; s'$$

$$+\!\!+^{j+1} :: \mathbf{Strm}\ (j+1)\ a \to \mathbf{Strm}\ k\ a \to \mathbf{Strm}\ (k+j+1)\ a$$
$$s +\!\!+^{j+1} s' = \mathbf{case}\ s\ \mathbf{of}\ x; s'' \to x; (s'' +\!\!+^j s')$$

El último problema surge en la función `distribute`. Esta función proporciona nuevas tareas a procesos ociosos. Esto significa que los *streams* no se incrementan uniformemente. Luego no es posible tipar la función $distribute^n$ de la Figura 6.14, ya que desconocemos cuál de los *streams* será incrementado. Sin embargo, sabemos con seguridad que uno de ellos lo hará. Necesitamos, por tanto, hacer *inducción sobre la suma de los tamaños de los streams* y no separadamente sobre cada uno de ellos. Esto no está soportado por el sistema. El proceso sería de forma intuitiva el siguiente. Primero, la comprobación de ínfimo: $\sum_{j=1}^{n} k_j = 0$ implica que para cada $j = 1,\ldots,n$, $k_j = 0$. Lo que significa que $\mathbf{Strm}\ k_j\ a = \mathbf{U}$ y el uso de tuplas estrictas nos lleva a $<\mathbf{Strm}\ k_1\ a,\ldots,\mathbf{Strm}\ k_n\ a> = \mathbf{U}$. La hipótesis de inducción es que $distribute^n :: \mathbf{Strm}\ w\ a \to \mathbf{Strm}(\sum_{j=1}^{n} k_j)\ (Nat\ n) \to <\mathbf{Strm}\ k_1\ a,\ldots,\mathbf{Strm}\ k_n\ a>$ con $\sum_{j=1}^{n} k_j = k$. Hemos de probar que el tipo de $distribute^n$ es $\mathbf{Strm}\ w\ a \to \mathbf{Strm}(\sum_{j=1}^{n} k'_j)\ (Nat\ n) \to <\mathbf{Strm}\ k'_1\ a,\ldots,\mathbf{Strm}\ k'_n\ a>$ con $\sum_{j=1}^{n} k'_j = \sum_{j=1}^{n} k_j +1 = k+1$. Tomamos $s_1 :: \mathbf{Strm}\ w\ a$ y $s_2 :: \mathbf{Strm}\ (\sum_{j=1}^{n} k_j + 1)\ (Nat\ n)$. Entonces, aplicando la regla **case**, $s'_1 :: \mathbf{Strm}\ w\ a$ y $s'_2 :: \mathbf{Strm}\ (\sum_{j=1}^{n} k_j)\ (Nat\ n)$, por lo que, aplicando la hipótesis de inducción, obtenemos que $distribute^n\ s'_1\ s'_2 :: <\mathbf{Strm}\ k_1\ a,\ \ldots,\mathbf{Strm}\ k_n\ a>$; y entonces, $t_j :: \mathbf{Strm}\ k_j\ a$, para cada $j = 1,\ldots,n$. En cada rama de la expresión **if** se incrementa en uno una de las componentes, es decir, cada rama tiene un tipo en el que la suma de los tamaños es $k+1$, por lo que la expresión completa tiene un tipo en el que la suma de los tamaños es $k+1$, como deseábamos demostrar. Este tipo de inducción debe aplicarse también a *inputss*.

Este nuevo tipo inducción no está incluido en forma de regla en el sistema de tipos extendido ya que la expresión de tamaño sobre la que hacer inducción depende en gran medida del programa concreto que se está tipando. De alguna manera el programador debería indicar no solamente el tipo del programa sino también cual es la expresión de tamaño sobre la que desea hacer inducción.

6.3.4 Esqueleto divide y vencerás con un árbol

Definimos ahora una implementación diferente del esqueleto divide y vencerás como ejemplo de inducción sobre un tipo distinto de las listas y los *streams*. Ahora usamos un árbol y para tipar la función `combineTop'` definida debajo hacemos inducción sobre su profundidad. La tarea original se divide hasta una profundidad dada y para cada subárbol a dicha profundidad se crea una subtarea. La lista de subtareas se proporciona a un esqueleto `map_naive` (o mejor a uno `rw`) en el que la función transformadora es exactamente el algoritmo secuencial. Al dividir la tarea inicial se debe generar un árbol explícito de argumentos para poder combinar los resultados obtenidos por los procesos paralelos:

```
dcT :: (Tr a,Tr b) => Int -> (a -> Bool) -> (a -> b) -> (a -> [a]) ->
                      (a -> [b] -> b) -> a -> b
dcT depth trivial solve split combine x =
        combineTop combine levels results
        where (tasks,levels) = generateTasks depth trivial split x
              results = map_naive thr (dc trivial solve split combine) tasks
data Tree a = Node a [Tree a] | Leaf a
-- Generates a list of tasks at level n
generateTasks :: Int -> (a -> Bool) -> (a -> [a]) -> a -> ([a], Tree a)
generateTasks 0 _ _ a = ([a],Leaf a)
generateTasks n trivial split a
 | trivial a = ([a],Leaf a)
 | otherwise = (concat ass,Tree a ts)
 where assts = map (generateTasks (n-1) trivial split) (split a)
       (ass,ts) = unzip assts
-- Combines all the results obtained by the farm/rw skeleton
combineTop :: (a -> [b] -> b) -> (Tree a) -> [b] -> b
combineTop c t bs = fst (combineTop' c t bs)

combineTop' :: (a->[b]->b) -> (Tree a) -> [b] -> (b,[b])
combineTop' _ (Leaf a) (b:bs) = (b,bs)
combineTop' combine (Tree a ts) bs = (combine a (reverse res),bs')
 where (bs',res) = foldl f (bs,[]) ts
       f (olds,news) t = (remaining,b:news)
         where (b,remaining) = combineTop' combine t olds
```

En la Figura 6.15 se define la versión tipada. El tipo del árbol se define como un tipo **idata** con dos parámetros de tamaño. El primero (recursivo) establece la profundidad del árbol, y el segundo es una cota superior de la aridad de los nodos en el árbol. También es necesario añadir un caso especial en $combineTop'$ para cuando bs sea vacía. Se añade un nuevo parámetro b, de forma que cuando bs es vacía se devuelve el par $(b, [\,])$. También podríamos devolver un par auxiliar. Este parámetro debe añadirse también en $combineTop$ y dcT, para poder proporcionárselo a la función $combineTop'$ cuando sea necesario. Los tipos de $generateTasks$ y $combineTop'$ se demuestran por inducción sobre la profundidad del árbol. Para no oscurecer el programa, en este esqueleto omitimos los subíndices.

```
idata Tree w k a = Leaf a | Node a (List k (Tree w k a))
dcT :: ∀a, b.F a, b ⇒ (a → Bool) → (a → b) → (a → List w a) → (a → List w b → b)
                       → b → a → b
dcT = λtrivial.λsolve.λsplit.λcombine.λb.λx.
      let
        tasksLevels :: ∀k.F a ⇒ (List w a, Tree k w a)
        tasksLevels = generateTasks n trivial split x
      in let
        results :: F b ⇒ List w b
        results = map_naiveL (dc trivial solve split combine) (fst tasksLevels)
      in
        combineTop combine (snd tasksAndLevels) results b

generateTasks :: ∀a, k, l.Nat k → (a → Bool) → (a → List l a) → a →
                 (List w a, Tree k l a)
generateTasks = λn.λtrivial.λsplit.λx.
  case n of
    zero → (cons x nil, Leaf x)
    succ n′ → case (trivial x) of
           True → (cons x nil, Leaf x)
           False → let
                    assts :: (List l (List w a), List l (Tree k l a))
                    assts = unzip(map (generateTasks n′ trivial split) (split x))
                   in (concat (fst assts), Node x (snd assts))

combineTop :: ∀a, b, k, l.(a → List w b → b) → Tree k l a → List (l + 1) b → b → b
combineTop = λf.λt.λbs.λb.fst(combineTop′ c t bs b)

combineTop′ :: ∀a, b, k, l.(a → List w b → b) → Tree k l a → List (l + 1) b → b →
                (b, List (l + 1) b)
combineTop′ = λc.λa.λbs.λb.
              case a of
                 Leaf a → case bs of
                              nil → (b, nil)
                              cons b′ bs′ → (b′, bs′)
                 Node x ts →
                   let
                     f :: (List (l + 1) b, List w b) → Tree k l a →
                        (List (l + 1) b, List w b)
                     f = λ(os, ns).λt.
                                 let br :: (b, List (l + 1) b)
                                     br = combineTop′ c t os
                                 in (snd br, cons (fst br) ns)
                   in let
                     bsrest :: (List (l + 1) b, List w b)
                     bsrest = foldl f (bs, nil) ts
                   in
                     (combine x (reverse (snd bsrest)), fst bsrest)

concat :: ∀a, k, l.List k (List w a) → List w a
concat = λxss.case xss of
              nil → nil
              cons xs xss′ → xs++(concat xss′)
```

Figura 6.15: Esqueleto divide y vencerás con un árbol

Lo habitual en este esqueleto es lanzar una topología de trabajadores replicados, en lugar de un `map` ingenuo, para así mejorar el reparto de carga de los procesos. Pero para hacerlo necesitaríamos una versión con listas, de la que aún no disponemos. Es trabajo futuro.

6.4 Conclusiones y trabajo futuro

Edén es un lenguaje funcional paralelo lo suficientemente expresivo como para que el programador pueda introducir bloqueos y no terminación en los programas. Disponer de una prueba de que tales efectos no están presentes es altamente deseable. Tales pruebas son particularmente interesantes para los esqueletos, puesto que éstos representan topologías paralelas reutilizadas muchas veces en distintas aplicaciones.

Hemos extendido en distintas direcciones el sistema de tipos con tamaño de Hughes y Pareto para hacerlo más útil para tipar nuestros esqueletos. Hemos demostrado que algunos esqueletos están libres de terminación anormal y de bucles infinitos. En concreto, hemos introducido clases de tipos para restringir la concreción de algunas variables de tipo. De esta forma, se puede demostrar que solamente se transmiten valores finitos entre procesos. Además, hemos añadido al sistema tipos estrictos, y sus correspondientes reglas de tipado, para poder hacer demostraciones por inducción para procesos que reciben y/o producen tuplas o listas de canales. Por otra parte nos hemos encontrado con la necesidad de incorporar productos de variables de tamaño a las expresiones de tamaño, de ser ello posible, lo que haría al sistema de tipos mucho más expresivo de forma que se podrían tipar muchos más algoritmos. En el caso general la extensión haría probablemente indecidible el chequeo de tipos, pero probablemente se encontrarán muchos subcasos útiles en los que el algoritmo siga siendo decidible. Nuestros esqueletos `map_farm` y `rw` proporcionan ejemplos concretos de que esta extensión es realmente necesaria. Finalmente, proponemos extender la demostración por inducción incluida en la regla $[LETREC]$ para poder llevar a cabo inducción sobre expresiones, como la suma de variables de tamaño, en lugar de solamente sobre variables independientes.

Aun en el caso de que las extensiones propuestas permitan automatizar las demostraciones, el programador será aún responsable de conjeturar un tipo para su programa y de reformularlo para facilitar su demostración.

Como trabajo futuro, planeamos en primer lugar continuar con el tipado de otros esqueletos más complejos como el anillo, el toro y otros. Nos gustaría trabajar en la formalización de las nuevas características y en la extensión de la implementación actual de los tipos con tamaño con ellas para chequear automáticamente las demostraciones presentadas aquí.

Capítulo 7

Conclusiones y trabajo futuro

En esta tesis se han presentado tres análisis del lenguaje funcional paralelo Edén utilizando distintas técnicas. Los primeros capítulos de esta tesis han pretendido situar este trabajo en el contexto adecuado y describir los conceptos utilizados en los capítulos principales. En el Capítulo 2 se ha motivado la utilidad de los análisis de programas en los lenguajes de programación y se han resumido las principales técnicas utilizadas para definirlos. Se ha puesto especial interés en describir la forma en que se aplican a los lenguajes funcionales la interpretación abstracta y los sistemas de tipos anotados, pues son las técnicas utilizadas en los análisis presentados en esta tesis. En el Capítulo 3 se ha descrito el lenguaje funcional Edén: su sintaxis, semántica e implementación. Sin embargo, en los capítulos posteriores, se recuerdan las características particularmente relevantes que es necesario tener en cuenta en cada uno de los análisis.

En los siguientes capítulos se presentan los resultados originales de la tesis. Cada uno de los análisis presentados cumple uno de los tres objetivos que se propusieron al comienzo de esta tesis.

El **análisis de conexión directa** del Capítulo 4 responde al objetivo de desarrollar análisis con la intención de incrementar la aceleración de los programas Edén disminuyendo su sobrecarga. Esto se consigue mediante la reducción del número de mensajes y del número de hebras creadas cuando estas no realizan ningún trabajo útil para el programa en el sentido de que solamente se dedican a redirigir mensajes de un procesador a otro. El análisis detecta hebras que se limitan a copiar valores de un canal de entrada a un canal de salida. En tal caso el proceso que contiene a la hebra es un mero intermediario entre el productor y el consumidor del valor. La información producida por el análisis consiste en anotaciones de las abstracciones de procesos y las ligaduras donde se lanzan los procesos. Con esta información se modifica el protocolo de comunicación entre los procesos, de forma que no solamente se evita la creación de las hebras inútiles sino que también se conecta de forma directa al productor y consumidor del valor. De esta forma se generan otros tipos de topologías distintas de las arbóreas, como la tubería, el toroide, la malla, el anillo, etc.

La definición del análisis está basada en un simple criterio sintáctico: se buscan canales utilizados exactamente una vez como entrada y exactamente una vez como salida, y no utilizados como variables libres en ninguna expresión. Aunque hemos utilizado un dominio de valores de conexión directa con el objetivo de representar el incremento en el número de usos de los canales, este análisis no se puede considerar basado en interpretación abstracta.

Existe un caso especial de conexión directa que hemos llamado conexión directa entre ancestros. El análisis incluye este caso, pero no así el protocolo aquí presentado. El protocolo finalmente implementado por Ulrike Klusik es una mejora de este que sí incluye este caso especial.

El nuevo protocolo no implica sobrecargas adicionales en el caso de que no exista ningún caso de conexión directa, por lo que no supone en ningún caso un perjuicio para la eficiencia de los programas. Además, en tal caso la topología se mantiene. El ahorro obtenido por la optimización proviene del hecho de que el número de mensajes de datos que se transmiten entre los procesos es habitualmente mucho mayor que los mensajes del protocolo, ya que muchos de los canales son listas cuya transmisión implica tantos mensajes de datos como elementos hay en la lista. Así, hemos visto que para una cadena de reenvío de longitud n, donde $n \geq 2$, se ahorra un total de $n-1$ mensajes por cada mensaje de datos que se envíe a través de la conexión directa. Por otra parte, el número de mensajes adicionales del nuevo protocolo con respecto al anterior es como mucho tres, por lo que incluso aunque se transmita un único dato se obtienen beneficios para cadenas de reenvío de longitud mayor que cuatro.

Para poder tener en la práctica los beneficios de este análisis es necesario implementarlo en el compilador de Edén. Esto implica desarrollar todo el sistema transformacional descrito: definición de la sintaxis abstracta de CoreEdén, traducción de Core a CoreEdén, análisis de conexión directa y posterior traducción de CoreEdén a Core. Este marco es además el adecuado para incorporar nuevos análisis y transformaciones que permitan incrementar la eficiencia de los programas Edén. Al igual que Core es una simplificación de Haskell en la que es sencillo llevar a cabo análisis y transformaciones, CoreEdén es su contrapartida paralela, y sobre él se podrán desarrollar todos los análisis y transformaciones relacionados con el paralelismo.

El análisis aquí presentado tiene ciertas limitaciones. No es capaz de manejar topologías complejas como el toroide donde los canales no son explícitos sino que aparecen dentro de listas manejadas por funciones de orden superior. Para tratar estos casos, en [PRS00] se propone una metodología para transformar topologías jerárquicas en topologías con conexión directa o no jerárquicas. Una combinación de ambas técnicas sería lo adecuado para tratar todos los casos.

El **análisis de no determinismo** del Capítulo 5 responde al segundo de los objetivos propuestos. Puesto que Edén está implementado reutilizando el compilador GHC, se ha estudiado cómo interfieren las transformaciones realizadas por este compilador con las construcciones introducidas por Edén. Se ha visto que dicha interacción a veces influye en la eficiencia y otras veces sobre la semántica. La influencia sobre la semántica se debe a la presencia de no determinismo. Por otra parte, el no determinismo también afecta a la definitud de las variables. En un lenguaje funcional perezoso, cuando una variable se evalúa, su clausura se actualiza con el nuevo valor, de forma que todas sus apariciones de uso lo comparten. Sin embargo, en Edén puede suceder que una variable no determinista se copie en otro procesador antes de ser evaluada, por ejemplo en $(f\ x)\ \#\ x$ donde x es no determinista. Al ser reevaluadas en procesadores diferentes se pueden producir valores distintos. Esto provoca que distintas apariciones de una misma variable en un programa puedan tener valores distintos, con la consiguiente pérdida del razonamiento ecuacional. Con el doble objetivo de desactivar selectivamente las transformaciones peligrosas y de acotar aquellas partes del programa Edén donde se pueda razonar ecuacionalmente, se han desarrollado tres análisis de no determinismo y se ha implementado uno de ellos.

Los tres análisis se han definido utilizando interpretación abstracta. En el primero de ellos no existen dominios funcionales, por lo que se trata de un análisis eficiente. Sin embargo, pierde demasiada información en las funciones, ya que no se pueden expresar las dependencias de la salida de una función con respecto a la entrada. El segundo análisis trata de paliar estas deficiencias utilizando dominios abstractos funcionales, con todos los problemas de eficiencia que esto conlleva. El análisis pasa a ser exponencial y por tanto impracticable. Finalmente se define un tercer análisis, como resultado de un compromiso entre la eficiencia y la potencia. En el desarrollo de este análisis se han utilizado técnicas conocidas de aproximación de puntos fijos y de representación eficiente de funciones. Solamente pierde información en los puntos fijos y resulta ser cúbico en las definiciones recursivas y aproximadamente lineal en el resto de definiciones, por lo que se ha considerado que es el candidato adecuado para ser implementado. Se ha implementado un prototipo de este análisis en Haskell y se ha ejecutado sobre varios ejemplos de programas Edén obteniendo un coste añadido a la compilación de aproximadamente el 1%. El algoritmo implementado no solamente obtiene los valores abstractos de las expresiones sino que también anota las mismas con información de no determinismo. Su implementación en un lenguaje funcional perezoso ha proporcionado enormes ventajas ya que se puede utilizar la evaluación perezosa como mecanismo de cómputo de los valores abstractos: una función abstracta se puede representar como una suspensión y su aplicación a un argumento se traduce en la continuación de la evaluación de su cuerpo. Esto nos ha evitado tener que construir un intérprete completo, lo cual hubiera sido necesario en caso de haber utilizado un lenguaje impaciente o uno imperativo.

En una primera aproximación al problema, el primer análisis se expresó utilizando un sistema de tipos anotados, equivalente a la interpretación abstracta aquí presentada. Sin embargo, vistos los problemas de imprecisión (*envenenamiento*) que sufrían los análisis de uso basados en tipos [LGH+92, TWM95, WJ99], descritos en el Capítulo 2 (Sección 2.3.2), se decidió abandonar esta técnica en favor de la interpretación abstracta. A pesar de ello, las mejoras introducidas recientemente en [GS01] nos animan a intentar redefinir el segundo análisis en términos de un sistema de tipos que permita mantener la máxima potencia a un coste razonable (polinómico).

Aunque se han proporcionado pruebas de corrección entre los tres análisis definidos, queda pendiente una definición de la corrección de los mismos con respecto a la semántica de Edén. El desarrollo de una semántica denotacional de Edén es sin embargo el objetivo de otra tesis en curso [Hid00], por lo que de momento se contempla como trabajo futuro.

Finalmente, el **análisis de terminación y productividad** desarrollado en el Capítulo 6 responde al tercer objetivo, el estudio de propiedades de los programas. En Edén se pueden definir de forma sencilla topologías genéricas llamadas esqueletos. El programador puede introducir en ellos inadvertidamente bucles infinitos o bloqueos. Puesto que los esqueletos son topologías reutilizadas muchas veces en distintas aplicaciones, es deseable disponer de demostraciones de ausencia de estos efectos negativos. La teoría de tipos con tamaño [HPS96, Par00], resumida en el Capítulo 2 (Sección 2.3.3), proporciona un marco en el que realizar análisis automático de las propiedades de terminación y productividad de los programas. Esta teoría permite definir un sistema de tipos para una versión simplificada de Haskell de manera que está garantizado que un programa bien tipado está libre de bucles infinitos y/o bloqueos. Para demostrar las propiedades deseadas para las funciones recursivas, en este sistema se distingue entre los tipos finitos (inductivos) y los infinitos (coinductivos), y se utiliza un principio de inducción sobre el número de constructores (tamaño). El análisis aquí descrito se ha definido como una extensión de dicho sistema de tipos. Puesto que Edén extiende el lenguaje Haskell modificando su semántica perezosa en ciertos puntos, es necesario modificar el sistema de tipos para que tenga en cuenta dichas características. Las principales extensiones son: (1) un sistema de clases de tipos para controlar la concreción de las variables polimórficas que representan canales, ya que por estos solamente se pueden transmitir valores finitos y (2) la inclusión de tipos estrictos para poder hacer demostraciones por inducción para procesos que reciben y/o producen tuplas o listas de canales. Este nuevo sistema se ha aplicado para demostrar la terminación y/o productividad de algunos esqueletos Edén. En esta aplicación se han puesto de manifiesto algunas limitaciones del sistema de tipos y han surgido nuevos problemas como la necesidad de anotaciones de tamaño no lineales (en los ejemplos estudiados bastaba con tener expresiones cuadráticas) y la necesidad de hacer inducción sobre expresiones generales de tamaño (por ejemplo, suma de variables), en lugar de solamente sobre variables.

Esta tesis se ha planteado, por un lado, como respuesta a determinadas necesidades surgidas en el entorno de desarrollo del lenguaje Edén y, por otro, como una aportación al mundo de los análisis en lenguajes funcionales paralelos. Las técnicas utilizadas en los análisis de lenguajes funcionales se han revelado igualmente útiles en los lenguajes funcionales paralelos. De entre ellos parece que los sistemas de tipos anotados y los sistemas de efectos son especialmente útiles, en particular para recopilar información sobre los efectos laterales derivados del paralelismo.

Con respecto al **trabajo futuro**, quedan algunos problemas abiertos en los análisis presentados.

En cuanto al análisis de conexión directa, el propuesto aquí no resuelve todas las situaciones, y la metodología propuesta en [PRS00] exige el uso de construcciones de bajo nivel de Edén no mencionadas en esta tesis. Una solución más definitiva seguiría la línea de apoyar el análisis llevado a cabo actualmente, con anotaciones introducidas por el programador para los casos más difíciles. A partir de aquí, el código generado y el protocolo descrito en la Sección 4.4.7 serían suficientes para establecer la topología deseada.

En el caso del análisis de no determinismo, ya se ha mencionado la necesidad de demostrar la corrección de los análisis propuestos con respecto a la semántica denotacional de Edén (para ello bastaría con demostrar la corrección del segundo análisis, pues los demás son correctos respecto a él), cuando esta se haya completado.

En cuanto al análisis de productividad y terminación, por una parte, deseamos continuar tipando otros esqueletos más complejos como el anillo y el toroide. Por otra, es necesario formalizar las nuevas características del sistema y demostrar su corrección con respecto a la semántica de Edén. También hace falta estudiar cómo afecta al algoritmo de comprobación de tipos la introducción de expresiones de tamaño no lineales en general o cuadráticas en particular. Y finalmente es necesario también fundamentar e introducir en el sistema la inducción sobre expresiones de tamaño.

La experiencia alcanzada en el desarrollo de los análisis aquí presentados, será sin duda útil para abordar nuevos análisis en Edén o en otros lenguajes. En este sentido, se ha establecido una colaboración con los desarrolladores de GpH en Edimburgo y St. Andrews (Escocia) para estudiar problemas específicos de dicho lenguaje, así como de otro lenguaje funcional de esos mismos grupos, Hume [Mic01], actualmente en definición, y pensado para aplicaciones críticas. Hume ofrece un amplio campo para la experimentación de nuevos análisis, ya que se pretenden establecer en tiempo de compilación cotas superiores al tiempo y memoria necesarios para la evaluación de cada expresión.

Apéndice A

Conceptos básicos

En este apéndice se presentan algunos conceptos básicos utilizados en la tesis. Con ello se pretende que ésta sea lo más autocontenida posible. Se presentan algunos conceptos de teoría de dominios [Rea89, AJ94] y de teoría de categorías [Pie91, Mac71, Bar93].

A.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.

A.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 A.1). En ellos 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. Gracias a la transitividad no es necesario pintar todas las líneas.

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

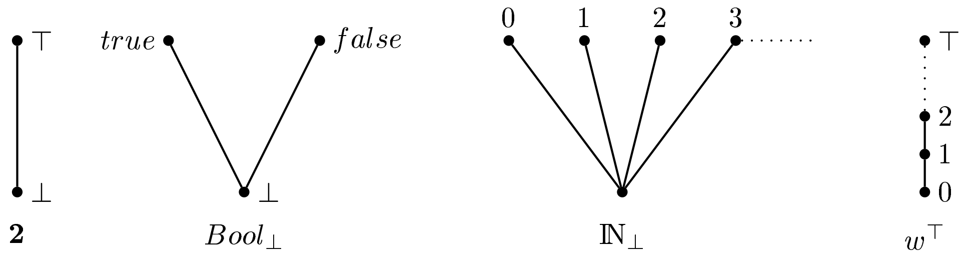


Figura A.1: Algunos ejemplos de posets que también son cpos

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 A.1.
- El cpo de los booleanos (ver Figura A.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 A.1, donde

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

- Los ordinales w no constituyen un cpo, ver Figura A.3. Para que formen un cpo es necesario añadir un elemento supremo $\top$, para obtener $w^\top$ (ver Figura A.1).
- Dado un conjunto S, el conjunto $P(S)$ de todos los subconjuntos de S ordenados por inclusión forman un cpo. Como ejemplo, en la Figura A.2 se muestra el cpo $P(\{0, 1, 2\})$.

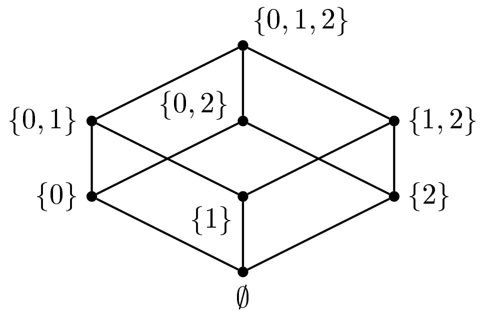


Figura A.2: El cpo de las partes de $S = \{0, 1, 2\}$

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

  $$s_1 \sqsubseteq s_2 \text{ si y sólo si } s_1 = \perp \text{ o } s_1 = s_2.$$

  Este tipo de cpos apuntados reciben el nombre de **planos** puesto que todos los elementos distintos de $\perp$ se encuentran al mismo nivel. Ejemplos de cpos planos son los cpos de los booleanos y de los naturales de la Figura A.1.

Figura A.3: Un ejemplo de poset no completo: w

A.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 A.4, formado por cuatro elementos:

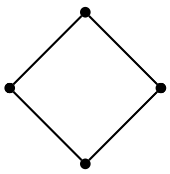


Figura A.4: Un retículo completo de cuatro puntos

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.

A.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,

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

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 49 *(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 (propiedad de cadena ascendente). 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 $f(\perp_D) = \perp_E$ y se dice que **refleja el ínfimo** si $f(d) = \perp_E \Rightarrow d = \perp_D$.

El conjunto de las funciones continuas $[D \to E]$ entre dos cpos D y E es otro cpo, en el que el orden $f \sqsubseteq g$ 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, pues $\forall d \in D. \perp_{[D \to E]}(d) = \perp_E$.

Teorema 50 *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.*

La cadena $\{f^n(\perp)\}_{n \in \mathbb{N}}$ recibe el nombre de cadena ascendente de Kleene.

A.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.

Un 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.

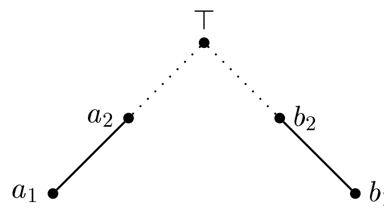


Figura A.5: Ejemplo de cpo no algebraico

Un ejemplo de cpo no algebraico es el de la Figura A.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.

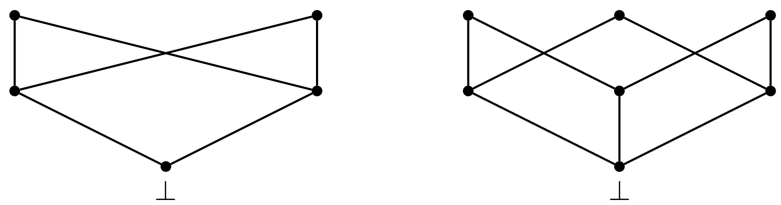
Existen cpos algebraicos D que cumplen que $[D \to D]$ no es algebraico. Para lograr que $[D \to D]$ pertenezca a la misma categoría de objetos, es necesario imponer algunas condiciones adicionales.

Un conjunto $X \subseteq D$ es **consistente** si tiene cota superior. Luego, todo conjunto dirigido en un cpo 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 A.6 se muestran ejemplos de cpos algebraicos, uno de los cuales es coherente y el otro no lo es.



cpo algebraico no coherente   cpo algebraico y coherente

Figura A.6: Ejemplos de cpos algebraicos coherentes y no coherentes

Un **dominio de Scott** es un cpo algebraico consistentemente completo. Para 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:

$$\begin{array}{rcl} M & = & \{x \mid \exists y \in E.(x, y) \in L\}, \\ N & = & \{y \mid \exists x \in D.(x, y) \in L\}. \end{array}$$

Si ambos son dominios, entonces el producto es dominio en el que

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

Al definir la semántica estricta de lenguajes 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 con

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

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, en el que los elementos $\perp_{D_1}$ y $\perp_{D_2}$ se identifican en un único elemento $\perp_{D_1 \oplus D_2}$ (ver Figura A.7). En $D_1 \oplus D_2$ tendremos $d \sqsubseteq d'$ si y sólo si:

- $d = \perp_{D_1 \oplus D_2}$, 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 A.7.

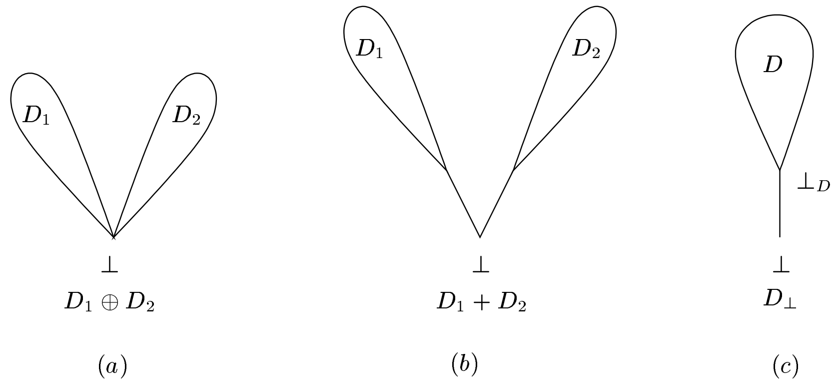


Figura A.7: (a) Suma coalescente de D_1 y D_2; (b) suma separada de D_1 y D_2; (c) elevación de D

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 elemento 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 evalúa a $C\ E$ donde E es una expresión indefinida. Es decir, queremos un nuevo elemento distinto de $C\perp$ que represente la indefinición total, el cual se consigue elevando el dominio con un $\perp$ nuevo por debajo de $C\ \perp$.

Dado un cpo D, llamamos $D_\perp$ al cpo elevado, cuyos elementos son los de D junto con un ínfimo añadido $\perp$ por debajo de $\perp_D$, ver Figura A.7.

A veces se representan los elementos de la siguiente forma:

- $\perp \in D_\perp$,
- $\langle 0, d\rangle \in D_\perp$ si $d \in D$.

Si D es un dominio, entonces $D_\perp$ es un dominio, en el que

$$K(D_\perp) = \{\perp\} \cup \{\langle 0, d\rangle \mid d \in K(D)\}.$$

Se cumple que:

$$D + E = D_\perp \oplus E_\perp.$$

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

$$f_\perp(\perp) = \perp,\ f_\perp(\langle 0, d\rangle) = \langle 0, f(d)\rangle.$$

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

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

Ecuaciones recursivas entre dominios

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 A.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 2.2.3).

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

A.2 Definición de categorías

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

(a) Una colección de objetos $Obj(K)$.

(b) Para cada par de objetos A y B en $Obj(K)$, existe un conjunto denotado por $Morf_K(A, B)$ llamado el conjunto de los morfismos de A a B.

(c) Dados $f \in Morf_K(A, B)$ y $g \in Morf_K(B, C)$, se puede formar la composición $g \circ f \in Morf_K(A, C)$.

(d) 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 $id_B \circ f = f$.

(e) La composición definida en el tercer apartado es asociativa.

Algunos ejemplos de categorías son los siguientes:

(a) 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.

(b) Los conjuntos parcialmente ordenados forman una categoría, donde los objetos son conjuntos parcialmente ordenados y los morfismos funciones monótonas (el resto igual que en el punto anterior).

(c) 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**. Sin embargo, decimos que F es **contravariante** si 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 $F(f) \in Morf_{\mathcal{D}}(F(B), F(A))$ satisfaciendo las siguientes condiciones:

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

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:

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

Apéndice B

Código Haskell del análisis de no determinismo

A continuación se muestra el código Haskell del algoritmo de análisis de no determinismo mostrado en la Sección 5.8. Se trata de código Haskell literario, donde las partes de código Haskell aparecen delimitadas por el texto `\begin{code}` y `\end{code}` y el resto son comentarios. Estos aparecen en inglés con el objetivo de que el módulo sea comprensible para los no hispanohablantes.

Se utiliza el módulo `NewPP.hs` definido por Hughes [Hug95] para mostrar de forma legible (*pretty printing*) la salida producida por el análisis. Puesto que el análisis aún no está incluido en el compilador de Edén, el prototipo incluye también un analizador léxico y uno sintáctico, los cuales son generados respectivamente por las herramientas Alex y Happy. El módulo `Gram.hs` contiene el analizador sintáctico generado por Happy, donde se encuentran los tipos que definen la sintaxis del lenguaje descrito en la Sección 5.3, por lo que aquí aparecen comentados.

Cuando el análisis sea incorporado al compilador de Edén, la información de tipos estará disponible en las expresiones. Por el momento proporcionamos solamente los tipos de las variables ligadas, los operadores primitivos y del proceso predefinido `merge`, y el resto de los tipos es reconstruido a partir de estos cuando sea necesario mediante la función `ofType`. De esta forma los programas a ser analizados no se ven oscurecidos por una excesiva información de tipos.

Este código incluye una mónada de estados `ST` para contar el número de iteraciones realizado para calcular los puntos fijos que aparecen en el programa. Esta información ha sido útil para obervar que en la mayor parte de las ocasiones no se alcanza el peor de los casos, en el que el número de iteraciones es igual a la profundidad del dominio de signaturas.

```
\begin{code}
module NonDet5 where
import NewPP
import Gram
import GlaExts(unsafePerformIO)
\end{code}




*******************************************************************
****                        Identifiers                        ****
*******************************************************************

- We distinguish between binding occurrences and applied occurrences,
  although they are type synonyms.

- We need type information in very concrete places:
    - binders: lambda abstraction, process abstraction, case patterns
               and let bindings (for the fixpoint calculation)
    - primitive operators
    - merge is considered an Id

-The types of the expressions can be initially pasted to the
expressions, but by now we calculate them when needed


\begin{code}

--newtype Id a = MkId (String, Type, a) -- binding occurrence
              -- deriving Show

--instance Eq (Id a) where
--  MkId (s1,_,_) == MkId (s2, _, _) = s1 == s2

--type OId a = Id a            -- applied occurrence


gettype :: Id a -> Type
gettype (MkId (s,t,x)) = t

getname :: Id a -> String
getname (MkId (s,t,x)) = s

getAbsVal :: Id AbsVal -> AbsVal
getAbsVal (MkId (s,t,a)) = a


\end{code}
```

```
*******************************************************************
****      Constructors, literals, primitive operators          ****
*******************************************************************

- In this way, we don't have to distinguish between primitive and
  algebraic case alternatives


\begin{code}

--data Con a =  DataCon DataCon     -- constructor
--            | Literal Literal     -- literal (integers, chars etc)
--            | PrimOp  (PrimOp a)  -- primitive operators
--type DataCon = (String,Type)
--type Literal = (String, Type)
--type PrimOp a = Id a     -- we need their types


--Tuples are different from the rest of algebraic types
isCTup :: DataCon -> Bool
isCTup ("Tup",_) = True
isCTup _ = False

--This function transforms a Constructor in an annotated constructor
annotCons cs = map annotCon cs
annotCon :: Con a -> Con AbsVal
annotCon (PrimOp op) = error "Primitive operator in case alternative"
annotCon (DataCon s) = DataCon s
annotCon (Literal k) = Literal k
\end{code}

********************************************************************
****                  Arguments                                 ****
********************************************************************

- In fact they are just variables or literals, but we want recursion

\begin{code}
--type Arg a = Expr a

getanarg = getanexpr
\end{code}

********************************************************************
****            Bindings: recursive and non recursive           ****
********************************************************************

\begin{code}
--data Bind a =  NonRec (Id a) (Expr a)
--             | Rec [(Id a, Expr a)]

\end{code}
```

```
*********************************************************************
****                        Expressions                          ****
*********************************************************************

- We distinguish between type lambdas and value lambdas, and
  correspondingly between normal applications and type applications.

\begin{code}

{-data Expr a =  Var (OId a)
             | Con (Con a) [Arg a] a
             | App (Expr a) (Arg a) a
             | Lam (Id a) (Expr a) a
             | TApp (Expr a) Type  a
             | TLam TId (Expr a) a
             | Case (Expr a) [Alt a] a
             | Let (Bind a) (Expr a) a
             | PInst (OId a) (Arg a) a
             | PAbs (Id a) (Expr a) a



type Alt a = (Con a, [Id a], Expr a) -- [Id] is [] if Con is a Literal
-}

--To get the annotation of an annotated expression/alternative
getanexpr::Expr AbsVal -> AbsVal
getanexpr (Var id) = getAbsVal id
getanexpr (Con _ _ an) = an
getanexpr (App _ _ an) = an
getanexpr (Lam _ _ an) = an
getanexpr (TApp _ _ an) = an
getanexpr (TLam _ _ an) = an
getanexpr (Case _ _ an) = an
getanexpr (Let _ _ an) = an
getanexpr (PInst _ _ an) = an
getanexpr (PAbs _ _ an) = an

getanalt (AlgAlt (c,vs,e)) = getanexpr e
getanelt (Default (v,e)) = getanexpr e
\end{code}

*********************************************************************
****                          Program                            ****
*********************************************************************

\begin{code}

--type Program a = [Bind a]

\end{code}
```

```
******************************************************************
****                           Types                          ****
******************************************************************

- As we have also used Id for type variables, we give then an EmptyType

- As merge is an Id we have to give its type:

  ForAll alpha (TyProc (TyConApp List (TyConApp List (TyVar alpha)))
                       (TyConApp List (Var alpha))
               )
  where alpha = ("alpha", EmptyType)


\begin{code}

--newtype TId = MkTId String
--           deriving Eq


{-data Type =  EmptyType
           | TyVar TId               -- type variable
           | FunTy Type Type        -- functional type
           | ForAllTy TId Type       -- polymorphic type
           | TyConApp TyCon [Type]  -- type constructor application
           | TyProc Type Type       -- process type
           deriving Eq           -}

{-type TyCon = String --We are interested in tuple type constructor:
  "TConTup"
-}


--merge's type is the following
{-mergeType = ForAllTy mergeTyVar
    (TyProc (TyConApp "List" [TyConApp "List" [TyVar mergeTyVar]])
            (TyConApp "List" [TyVar mergeTyVar]))
mergeTyVar = MkTId "mergeTyVar"
mergeVar = MkId ("merge", mergeType, ())-}


--Obtains the result type of a function
resultType (FunTy t1 t2) = resultType t2
resultType (TyProc t1 t2) = resultType t2
resultType (ForAllTy beta t) = resultType t
resultType t = t


--The abstract domain corresponding to a type is Basic if it is a
--type variable or a non-tuple algebraic type
basic::Type -> Bool
basic (TyConApp tc _ ) = tc /= "TTup"
     --basic types are included here also
basic (TyVar _) = True
```

```
basic _ = False

--Given a type, it returns true if it is a tuple type
isTup::Type -> Bool
isTup (TyConApp "TTup" _) = True
isTup _ = False

--Given a type, it returns true if it is not functional
nonfun::Type -> Bool
nonfun (FunTy _ _) = False
nonfun (TyProc _ _) = False
nonfun (ForAllTy _ t) = nonfun t
nonfun _ = True

--Given a functional type, it returns the types of the argument
--and the result
argrestType ::Type -> (Type,Type)
argrestType (FunTy t1 t2) = (t1,t2)
argrestType (TyProc t1 t2) = (t1,t2)
argrestType (ForAllTy _ t) = argrestType t
argrestType t | nonfun t = error "This is not a functional type"

--Given a type, it returns the number of arguments
numArgs::Type -> Int
numArgs t | nonfun t = 0
numArgs (FunTy t1 t2) = 1 + numArgs t2
numArgs (TyProc t1 t2) =  1 + numArgs t2
numArgs (ForAllTy _ t) = numArgs t

\end{code}

*****************************************************************
****            Type reconstruction function                 ****
*****************************************************************


\begin{code}
ofType::Expr a -> Type
ofType (Var id) = gettype id
ofType (Con (Literal (k,t)) [] _) = t

--two special cases: tuples and lists. They just have an abbreviated
--syntax in source programs
ofType (Con (DataCon ("Tup",_)) xs _) = TyConApp "TTup" (map ofType xs)
ofType (Con (DataCon ("Cons",_)) (x:xs) _) = TyConApp "List" [ofType x]

ofType (Con (DataCon (c,t)) _ _) = resultType t
ofType (Con (PrimOp op) _ _) = resultType (gettype op)
ofType (Lam id e _) = FunTy (gettype id) (ofType e)
ofType (PAbs id e _) = TyProc (gettype id) (ofType e)
ofType (TLam alpha e _) = ForAllTy alpha (ofType e)
ofType (App e a _) = t2
  where
     FunTy t1 t2 = ofType e
```

```
ofType (PInst v a _) = t2
  where
   TyProc t1 t2 = gettype v
ofType (TApp e t _) = sust t' alpha t
  where
    ForAllTy alpha t' = ofType e
ofType (Let bind e _) = ofType e
ofType (Case e alts _) = ofTypeAlt (head alts)

ofTypeAlt (AlgAlt (c,vs,e)) = ofType e
ofTypeAlt (Default (v,e)) = ofType e


--The replacement of a type variable by a type inside a type expression
sust (TyVar beta) alpha t
 | beta == alpha = t
 | otherwise = TyVar beta
sust (TyConApp c ts) alpha t = TyConApp c (map f ts)
  where f t' = sust t' alpha t
sust (FunTy t1 t2) alpha t = FunTy (sust t1 alpha t) (sust t2 alpha t)
sust (TyProc t1 t2) alpha t = TyProc (sust t1 alpha t) (sust t2 alpha t)
sust (ForAllTy beta t1) alpha t
  | alpha == beta = ForAllTy beta t1
  | otherwise = ForAllTy beta (sust t1 alpha t)

\end{code}



***********************************************************************
****                   Abstract Values Definition                 ****
***********************************************************************

- Signatures and abstract values are mixed.

\begin{code}
data EnumAnnot = Det | NonDet deriving (Eq,Ord)

data AbsVal  = Basic EnumAnnot       --an abstract value and a signature
             | Tup [AbsVal]         --tuple of abstract values
             | Fun  (Id ()) (Expr ()) Env  --lambda suspension
             | AprFun Int Type [AbsVal]    --a functional signature
             | AllDet Int Type AbsVal      --special functional signature
             | FLub [AbsVal]               --a lub suspension
             | Gamma Type TId Type AbsVal  --polymorphism suspensions:
             | Alpha Type TId Type AbsVal  --t',beta,t

--Given an abstract value, tells if it is a signature
issignature::AbsVal -> Bool
issignature (Basic _) = True
issignature (Tup as) = and (map issignature as)
issignature (AprFun _ _ _) = True
issignature (AllDet _ _ _) = True
issignature _ = False
```

```
instance Eq AbsVal where
 (==) = equals --only defined for signatures

--Signatures equality: used in the fixpoint calculation
equals (Basic b) (Basic b') = (b==b')
equals (Tup as) (Tup as')   = and (zipWith equals as as')
equals (AprFun m t as) (AprFun m' t' as')
 | (m==m') && (t==t') = and (zipWith equals as as')
 | otherwise = error "Eq comparison error"
equals (AllDet m t s) (AllDet m' t' s')
 | (m==m') && (t==t') = equals s s'
 | otherwise = error "Eq comparison error"
equals (AprFun m t as) (AllDet m' t' s')
 | (m==m') && (t==t') = (equals a s') && and (map (equals (signondet tr)) as')
 | otherwise = error "Eq comparison error"
  where
    (as',a)=sepsig as
    tr = resultType t
equals (AllDet m t s) (AprFun m' t' as') =
    equals (AprFun m' t' as') (AllDet m t s)
equals _ _ = error "This equality comparison is unexpected"


sepsig xs = (init xs, last xs)

allequals [s1,s2] = s1==s2
allequals (s1:s2:ss) = (s1 == s2) && allequals (s2:ss)

instance Ord AbsVal where
 (<=) = lesseq --only defined for signatures

--Order between signatures: used in application of signatures
lesseq (Basic b) (Basic b') = (b<=b')
lesseq (Tup as) (Tup as') = and (zipWith lesseq as as')
lesseq (AprFun m t as) (AprFun m' t' as')
 | (m==m') && (t==t') = and (zipWith lesseq as as')
 | otherwise = error "Leq comparison error"
lesseq (AllDet m t s) (AllDet m' t' s')
 | (m==m') && (t==t') = lesseq s s'
 | otherwise = error "Leq comparison error"
lesseq (AllDet m t s) (AprFun m' t' as)
 | (m==m') && (t==t') = (lesseq s a) && and(map (equals (signondet tr)) as')
 | otherwise = error "Leq comparison error"
  where
     (as',a) = sepsig as
     tr = resultType t
lesseq (AprFun m t as) (AllDet m' t' s')
  | (m==m') && (t==t') = lesseq (last as) s'
  | otherwise = error "Leq comparison error"
lesseq _ _ = error "This inequality comparison is unexpected"

\end{code}
```

```
***********************************************************************
****                        Environments                           ****
***********************************************************************

\begin{code}

--This environment can be a three valued environment to contain: the
--abstract value, the signature and the basic abstract value; but here
--we use annotated identifiers. However this can be changed easily

type Env = [Id AbsVal]

(!-) :: Env -> Id a -> AbsVal
  --extracts an Id annotation from the environment

(id:rho) !- id'
        | eqname id id' = getAbsVal id
        | otherwise = rho !- id'
[] !- id' = error ("not in scope variable"++(show (getname id')))

--Looks in the environment for an identifier
inenv id [] = False
inenv id (id':rho)
  | eqname id id' = True
  | otherwise = inenv id rho

--Identifier's name equality
eqname (MkId (s,_,_)) (MkId (s',_,_)) = s==s'

--Adds an identifier to the environment
addEnv :: Env -> (Id AbsVal) -> Env
addEnv rho id = id:rho

--Adds an annotation to an identifier
addAbsVal :: AbsVal -> Id a -> Id AbsVal
addAbsVal  a (MkId (s,t,x)) = MkId (s,t,a)

--Extends an environment
extendEnv :: Env -> Env -> Env
extendEnv = (++)
-- this can be improved by checking that identifiers are not repeated

addAbsVals = zipWith addAbsVal
\end{code}

*******************************************************************
****                Least Upper Bound function                 ****
*******************************************************************
\begin{code}
lub (Basic NonDet) (Basic b) = Basic NonDet
lub (Basic Det) (Basic b) = Basic b

lub (Tup as) (Tup as') = Tup (zipWith lub as as')
```

```
lub (AprFun m t as) (AprFun m' t' as')
 |(m==m') && (t==t') = AprFun m t (zipWith lub as as')

lub (AllDet m t s) (AllDet m' t' s')
 |(m==m') && (t==t') = AllDet m t (lub s s')

lub (AprFun m t as) (AllDet m' t' s')
 |(m==m') && (t==t') = AllDet m t (lub (last as) s')

lub (AllDet m t s) (AprFun m' t' as')
 |(m==m') && (t==t') = AllDet m t (lub s (last as'))


--we assume the types of the functions in the list are the same
lub (FLub fs) f = FLub (f:fs)
lub f (FLub fs) = FLub (f:fs)
lub f g@(Fun x e rho) = FLub [f,g] --f is not a FLub
lub f@(Fun x e rho) g = FLub [f,g] --g is not a FLub

--for polymorphic suspensions

lub f@(Gamma t' beta t a) g = FLub [f,g]
lub f g@(Gamma t' beta t a) = FLub [f,g]
lub f@(Alpha t' beta t a) g = FLub [f,g]
lub f g@(Alpha t' beta t a) = FLub [f,g]

lub _ _ = error  "Type conflict in lub operation"


supN as = foldr1 lub as

\end{code}

*****************************************************************
****            Abstraction and concretisation functions     ****
*****************************************************************

-The concretisation function is given as a signature

\begin{code}
sigdet::Type -> AbsVal --It is gamma t d
sigdet t | basic t = Basic Det
sigdet (TyConApp "TTup" ts) = Tup (map sigdet ts)
sigdet t@(FunTy t1 t2) = AprFun m t (s1:ss)
  where
    s1 = auxnondet t2
    rs = sigdet t2
    ss = extract rs
    m = numArgs t
sigdet (TyProc t1 t2) = sigdet (FunTy t1 t2)
sigdet (ForAllTy _ t) = sigdet t
```

```
extract (AprFun m t as) = as
extract s = [s]

signondet::Type -> AbsVal
signondet t | basic t = Basic NonDet
signondet (TyConApp "TTup" ts) = Tup (map signondet ts)
signondet t@(FunTy t1 t2) = AprFun m t (s1:ss)
  where
    s1 = auxnondet t2
    rs = signondet t2
    ss = extract rs
    m = numArgs t
signondet (TyProc t1 t2) = signondet (FunTy t1 t2)
signondet (ForAllTy _ t) = signondet t


auxnondet::Type -> AbsVal
auxnondet t | nonfun t = signondet t
auxnondet (FunTy t1 t2) = auxnondet t2
auxnondet (TyProc t1 t2) = auxnondet t2
auxnondet (ForAllTy _ t) = auxnondet t

gamma::Type -> AbsVal -> ST State AbsVal --the second one is a Basic
gamma t (Basic Det)    = return (sigdet t)
gamma t (Basic NonDet) = return (signondet t)
gamma _ _ = error "Gamma must be applied to a basic annotation"

flip2 f b c a = f a b c

alpha::Type -> AbsVal -> ST State AbsVal
alpha t a | basic t = return a
alpha (TyConApp "TTup" ts) (Tup as) =
     do
       parc <- zipWithM alpha ts as
       return (supN parc)
alpha t@(FunTy t1 t2) a =
    do
      parc <- absApply a (sigdet t1)
      alpha t2 parc
alpha t@(TyProc t1 t2) a =
    do
      parc <- absApply a (sigdet t1)
      alpha t2 parc
alpha (ForAllTy _ t) a = alpha t a


\end{code}
```

```
*******************************************************************
****           Aplication of an abstract function             ****
*******************************************************************

\begin{code}

absApply::AbsVal  -> AbsVal -> ST State AbsVal --The first arg is a function
absApply (Fun id e rho) arg =
  do
   let rho' = addEnv rho (addAbsVal arg id)
   (a,_) <- evalExpr e rho'
   return a


absApply (AprFun 1 t [s1,sd]) arg =
   do
     let (ta,tr) = argrestType t
     esmenor <-  lessThanDet ta arg
     let out = if esmenor then sd else s1
     return out

absApply (AprFun m t (s1:ss)) arg =
   do
    let (ta,t') = argrestType t
    esmenor <- lessThanDet ta arg
    let out = if esmenor then (AprFun (m-1) t' ss) else (AllDet (m-1) t' s1)
    return out


absApply (AllDet 1 t s) arg =
  do
   let (ta,tr) = argrestType t
   esmenor <- lessThanDet ta arg
   let out = if esmenor then s else (signondet tr)
   return out

absApply (AllDet m t s) arg =
  do
    let (ta,t')=argrestType t
    esmenor <- lessThanDet ta arg
    let out = if esmenor then (AllDet (m-1) t' s) else (signondet t')
    return out

absApply (FLub fs) arg =
    do
     ls <- mapM (flip absApply arg) fs
     return (supN ls)


--for the polymorphic suspensions
absApply (Gamma t' beta t a) arg =
 do
   let (t1,t2) = argrestType t'
   a1 <- alphap t1 beta t arg
```

```
   a2 <- absApply a a1
   a' <- gammap t2 beta t a2
   return a'

absApply (Alpha t' beta t a) arg =
 do
  let (t1,t2) = argrestType t'
  a1 <- gammap t1 beta t arg
  a2 <- absApply a a1
  a' <- alphap t2 beta t a2
  return a'


absApply _ arg
   = error ">que pasa con la funcion?"

lessThanDet targ arg =
     do
       warg <- signature targ arg
       return (lesseq warg (sigdet targ))

absApplys :: AbsVal -> [AbsVal] -> ST State AbsVal
absApplys f  [a] = absApply f a
absApplys f  (a:as) = do
                        g <- absApply f a
                        a <- absApplys g as
                        return a
\end{code}

**********************************************************************
****                    Probing function                          ****
**********************************************************************

\begin{code}
zipWithM::Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM f [] [] = return []
zipWithM f (x:xs) (a:as) =
                   do
                     h <- f x a
                     hs <- zipWithM f xs as
                     return (h:hs)

signature::Type -> AbsVal -> ST State AbsVal --The result is a signature
signature t a | issignature a = return a
signature t a | basic t = return a
signature (TyConApp "TTup" ts) (Tup as) =
    do
      zw <- zipWithM signature ts as
      return (Tup zw)
signature t@(FunTy t1 t2) a =
  do
    ap1 <- absApply a (signondet t1)
    s1 <- auxp t2 ap1
    ap2 <- absApply a (sigdet t1)
```

```
    rs <- signature t2 ap2
    let
      ss = extract rs
      m = numArgs t
    return (AprFun m t (s1:ss))

signature (TyProc t1 t2) a = signature (FunTy t1 t2) a
signature (ForAllTy _ t) a = signature t a

auxp::Type -> AbsVal -> ST State AbsVal
auxp t a | nonfun t = signature t a
auxp t@(FunTy t1 t2) a =
    do
      parc <- absApply a (sigdet t1)
      auxp t2 parc
auxp t@(TyProc t1 t2) a =
    do
     parc <- absApply a (sigdet t1)
     auxp t2 parc
auxp t@(ForAllTy _ t1) a = auxp t1 a
\end{code}

**********************************************************************
****           Abstract interpretation function                  ****
**********************************************************************

Abstract interpretation for a program
\begin{code}

--To repeat n times the evaluation of a program: just to obtain
--bigger programs we add an integer parameter n and force the
--evaluation of the bindings several times

evalProg :: Int -> Program () -> ST State (Program AbsVal)
evalProg n binds
   | n == 1    =   evalbinds binds []
   | otherwise =   do  bs <- evalbinds binds []
                       seq (rnfBinds bs) (evalProg (n-1) binds)
pon f t c = seq (unsafePerformIO (writeFile f t)) c

--Reduction to normal form

rnfBinds []     = ()
rnfBinds (b:bb) = rnfBind b `seq` rnfBinds bb

rnfBind (NonRec id e)     = rnfId id `seq` rnfExp e `seq` ()
rnfBind (Rec [])          = ()
rnfBind (Rec ((id,e):bs)) = rnfId id `seq` rnfExp e `seq` rnfBind (Rec bs)

rnfId (MkId (s,t,a)) = s `seq` rnfType t `seq` a `seq` ()

rnfType  EmptyType       = ()
rnfType  (TyVar id)      = id `seq` ()
rnfType  (FunTy t1 t2)   = rnfType  t1 `seq` rnfType  t2
```

```
rnfType  (ForAllTy tid t)= tid `seq` rnfType  t
rnfType  (TyConApp c ts) = c `seq` rnfTlist ts
rnfType  (TyProc t1 t2)  = rnfType  t1 `seq` rnfType  t2

rnfTlist []     = ()
rnfTlist (t:ts) = rnfType  t `seq` rnfTlist ts

rnfExp (Var x)        = x `seq` ()
rnfExp (Con c xs a)   = c `seq` xs `seq` a `seq` ()
rnfExp (App e x a)    = rnfExp e `seq` x `seq` a `seq` ()
rnfExp (Lam id e a)   = rnfId id `seq` rnfExp e `seq` a `seq` ()
rnfExp (TApp e t a)   = rnfExp e `seq` rnfType t `seq` a `seq` ()
rnfExp (TLam tid e a) = tid `seq` rnfExp e `seq` a `seq` ()
rnfExp (Case e alts a)= rnfExp e `seq` rnfAlts alts `seq` a `seq` ()
rnfExp (Let b e a)    = rnfBind b `seq` rnfExp e `seq` a `seq` ()
rnfExp (PInst oid x a)= oid `seq` x `seq` a `seq` ()
rnfExp (PAbs id e a)  = rnfId id `seq` rnfExp e `seq` a `seq` ()

rnfAlts []     = ()
rnfAlts (a:as) = rnfAlt a `seq` rnfAlts as

rnfAlt (AlgAlt (c,ids,e)) = c `seq` rnfIds ids `seq` rnfExp e `seq` ()
rnfAlt (Default (id,e))   = rnfId id `seq` rnfExp e `seq` ()

rnfIds []       = ()
rnfIds (id:ids) = rnfId id `seq` rnfIds ids

----------------------------------------------------------------------

--Annotation of a list of bindings
evalbinds :: [Bind ()] -> Env -> ST State [Bind AbsVal]
evalbinds [] _ = return []
evalbinds (bind:binds) rho =
  do
   (rho',bind') <- evalBind bind rho
   binds' <- evalbinds binds rho'
   return (bind':binds')

--Annotation of an expression
evalExpr :: Expr () -> Env -> ST State (AbsVal, Expr AbsVal)

evalExpr (Var (MkId ("merge",t,_))) rho =
 return (a,Var (MkId ("merge",t,wa)))
  where
   a = signondet t
   wa = a

evalExpr (Var id) rho
 = do
     let a = rho !- id
         tid = gettype id
     wa <- signature tid a
     let id' = addAbsVal wa id
     return (a, Var id')
```

```
evalExpr (Con (Literal k) [] _) rho =
  return (Basic Det, Con (Literal k) [] (Basic Det))

evalExpr ne@(Con (DataCon c) xs _) rho
  | isCTup c =
     do
        ps <- mapM (flip evalExpr rho) xs
        let
         (as,xs') = unzip ps
         was = map getanexpr xs'
         a = Tup as
         wa = Tup was   --the same as signature (ofType ne) a
        return (a, Con (DataCon c) xs' wa)
  | otherwise =
     do
        ps <- mapM (flip evalExpr rho) xs
        let (as,xs') = unzip ps
            txs = map ofType xs
        basics <- zipWithM alpha txs as
        let a = if (null basics) then (Basic Det) else (supN basics)
            wa = a                      --the same as signature (oftype ne) a
        return (a, Con (DataCon c) xs' wa)

evalExpr ne@(Con (PrimOp op) xs _) rho =
   do
     ps <- mapM (flip evalExpr rho) xs
     let
       (as,xs') = unzip ps
       aop = sigdet top
       top = gettype op
     a <- absApplys aop as
     let op' = addAbsVal aop op  --as waop = aop
     wa <- signature (resultType top) a
     return (a, Con (PrimOp op') xs' wa)

evalExpr ne@(Lam v e _) rho =
  do
   let
     a = Fun v e rho
     v' = addAbsVal wv v
     wv = signondet tv
     tv = gettype v
     rho' = addEnv rho v'
   wa <- signature (ofType ne) a
   (_,e') <- evalExpr e rho'
   return (a, Lam v' e' wa)

evalExpr ne@(PAbs v e _) rho =
 do
  let
   a = Fun v e rho
   v' = addAbsVal wv v
   wv = signondet tv
```

```
   tv = gettype v
   rho' = addEnv rho v'
  wa <- signature (ofType ne) a
  (_,e') <- evalExpr e rho'
  return (a, PAbs v' e' wa)

evalExpr ne@(App e arg _) rho =
 do
  (ve,e') <- evalExpr e rho
  (va,arg') <- evalExpr arg rho
  a <- absApply ve va --if ve is Unknown, absApply deals with it
  wa <- signature (ofType ne) a
  return  (a, App e' arg' wa)

evalExpr ne@(PInst v arg _) rho =
 do
  p <- evalExpr (Var v) rho --a trick
  let (vv,Var v') = p
  p' <- evalExpr arg rho
  let (va,arg') = p'
  a <- absApply vv va
  wa <- signature (ofType ne) a
  return (a, PInst v' arg' wa)

evalExpr ne@(Case e [AlgAlt (DataCon c, vs, e')] _) rho
 | isCTup c =
   do
     (ae, e1) <- evalExpr e rho
     let
       Tup as = ae
       wae = getanexpr e1
       Tup was = wae
       rho' =  extendEnv rho (addAbsVals as vs)
     (a,e1') <- evalExpr e' rho'
     let vs' = addAbsVals was vs
         wa = getanexpr e1'   --more efficient than signature (ofType ne) a
     return (a, Case e1 [AlgAlt (DataCon c, vs', e1')] wa)

evalExpr ne@(Case e alts _) rho =
 do
  (ae,e') <- evalExpr e rho
  palts <- mapM (flip (evalAlt ae) rho) alts
  let
    (valts,alts') = unzip palts
    tne = ofType ne
    walts = map getanalt alts'
    a = case ae of
          Basic NonDet -> signondet tne
          Basic Det -> supN valts
  wa <- signature tne a
  return (a, Case e' alts' wa)
```

```
evalExpr ne@(Let bind e _) rho =
 do
   (rho',bind') <- evalBind bind rho
   (a,e') <- evalExpr e rho'
   let wa = getanexpr e' --more efficient than signature (ofType ne) a
   return (a, Let bind' e' wa)


--Polymorphism

evalExpr (TLam beta e _) rho =
 do
   (a,e') <- evalExpr e rho
   let wa = getanexpr e'
   return (a,TLam beta e' wa)

evalExpr ne@(TApp e t _) rho =
 do
  let tne = ofType ne
      ForAllTy beta t' = ofType e
  (a',e') <- evalExpr e rho
  a <- gammap t' beta t a'
  wa <- signature tne a
  return (a,TApp e' t wa)


--Annotation of alternatives
evalAlt ::AbsVal -> Alt () -> Env -> ST State (AbsVal, Alt AbsVal)
evalAlt ae (AlgAlt (c,vs,e)) rho =
 do
  let
    rho' = extendEnv rho vs'
    as = map (discr ae) ts
    discr a = case a of
              Basic Det -> sigdet
              Basic NonDet -> signondet
    ts = map gettype vs
    vs' = addAbsVals as vs
         --in this case was=as as they are already signatures
    c' = annotCon c
  (a,e') <- evalExpr e rho'
  return (a,AlgAlt (c',vs',e'))

evalAlt ae (Default (v,e)) rho =
 do
  let
     v' = addAbsVal ae v --as ae is basic abstract value wae = ae
     rho' = addEnv rho v'
  (a,e') <- evalExpr e rho'
  return (a,Default (v',e'))
```

```
--Annotation of recursive and non-recursive bindings
evalBind (NonRec v e) rho =
 do
   (a,e') <- evalExpr e rho
   let
     wa = getanexpr e'   --more efficient than signature (ofType e') a
     v' = addAbsVal wa v
     rho' = addEnv rho (addAbsVal a v)
   return (rho', NonRec v' e')

evalBind (Rec bs) rho =
 do
   nuevoCont
   let
      (ids,es)=unzip bs
  --we don't modify the bindings until the end
  --In the fixpoint calculation we only look for the environment
      f = \(rho',exit)->
               do
                pes <- mapM (flip evalExpr rho') es
                let
                  (_,es') = unzip pes
                  waes = map getanexpr es' --the widening
                  rho2 = addAbsVals waes ids
                  (nrho,nexit) = modify rho' rho2
                return (nrho,nexit)
      init = extendEnv (addAbsVals dets ids) rho
      dets = map sigdet tids
      tids = map gettype ids
   rho' <- fix f (init,False)
      --one more time to get the definitive rhss
   pes <- mapM (flip evalExpr rho') es
   let
      (_,es') = unzip pes
      waes =map getanexpr es'
      ids' = addAbsVals waes ids     --the definitive ids
      bs'= zip ids' es'
   return  (rho', Rec bs')

modify rho [] = (rho,True)
modify (id:rhos) (id':rhos')
  |  id == id'= (id':nrhos,(a==a') && exitrest) --a and a' are signatures
   where
    a = getAbsVal id
    a' = getAbsVal id'
    (nrhos,exitrest) = modify rhos rhos'
--fix::((Env,Bool) -> ST State (Env,Bool)) -> (Env,Bool) -> ST State Env
fix f (x,exit) --the counter is increased each time we apply f
   | exit = return x
   | otherwise = do
                  ef <- f (x,exit)
                  inc
                  ff <- fix f ef
                  return ff
```

```
\end{code}


**********************************************************************
****              Pretty-printing using Hughes library             ****
**********************************************************************

\begin{code}

banToD::EnumAnnot -> Doc
banToD (Det)    = text "d"
banToD (NonDet) = text "n"

anToD::AbsVal -> Doc
anToD (Basic b)     = banToD b
anToD (Tup as) = rodear d "(" ")"
 where
   d = sep (g as)
   g [a] = [anToD a]
   g (a1:a2:ss) = (anToD a1 <> text ","): (g (a2:ss))

anToD (AprFun m t as) =  rodear d "{" "}"
 where
  d = sep (h as)
  h [a] = [text "+" <> anToD a] --the last value
  h (a1:a2:as) = (anToD a1):(h (a2:as))

anToD (AllDet m t s) = rodear (anToD s) "{[" "]}"
--we save some space, we could also reconstruct the AprFun equivalent version
--anToD (AprFun m t (as'++[s])) where as'= rep m (signondet (resultType t))

anToD (FLub _) = error "trying to show a lub value"
anToD _ =error "Trying to show a non-signature functional abstract value"

tnil = text ""
idToD::Id AbsVal -> Doc
idToD (MkId(s,t,x)) = (text(s++"::"))<>(anToD x)

conToD::Con AbsVal -> Doc
conToD (DataCon (s,t)) = text (s++" ")
conToD (Literal (l,t)) = text (l)
conToD (PrimOp x)  = idToD x<>text " "

progToD::Program AbsVal -> Doc
progToD bs = foldr (($$).bindToD) tnil bs

bindToD :: Bind AbsVal -> Doc
bindToD(NonRec x e) = onebToD (x,e)
bindToD(Rec bs)     = foldr (($$) . onebToD) tnil bs

onebToD:: (Id AbsVal, Expr AbsVal) -> Doc
onebToD (x,e) = sep[idToD x <> text " = ",nest 4 (exprToD e)]

rodear :: Doc -> String -> String -> Doc
```

```
rodear d s s' = text s <> d <> text s'

rodexpr :: Expr AbsVal -> String -> String -> Doc
rodexpr e = rodear (exprToD e)
rodearid id = rodear (idToD id)

rodepare e = rodexpr e "(" ")"
rodeparid id = rodearid id "(" ")"
rodepart t = rodear (typeToD t) "(" ")"
rodecore e = rodexpr e "[" "]"
anConDP :: AbsVal -> Doc
anConDP a = text "::" <> anToD a

exprToD :: Expr AbsVal -> Doc
exprToD (Var id) = idToD id
exprToD (Con c xs a) =
  sep [d,anConDP a]
   where
     d = if (null xs) then (conToD c)
         else ((conToD c) <> (h xs))
     h [x] = rodepare x
     h (x:y:ys)= sep (map rodepare (x:y:ys))
exprToD (App e x a) = sep[rodepare e,rodepare x,anConDP a]
exprToD (Lam id e a) =
 sep[(text "\\ ") <> (idToD id)<> (text "."), nest 4 (exprToD e)]
 $$ anConDP a
exprToD (TApp e t a) = sep[(rodepare e),(rodepart t), anConDP a]
exprToD (TLam tid e a) =
  sep[(text "/\\ ")<> tidToD tid <> text ".",nest 4(exprToD e)]
  $$ anConDP a
exprToD (PInst id e a) = rodeparid id <> text " # "<> rodepare e
                         <>  anConDP a
exprToD (PAbs  id e a) =
 sep[(text "process ") <> (idToD id)<>(text " ->"),nest 4 (exprToD e)]
 $$ anConDP a
exprToD (Let b e a) =
 (text "let ")$$nest 4 (bindToD b)$$text "in "$$nest 3 (exprToD e)
 $$ anConDP a
exprToD (Case e alts a) = (text "case "<> rodepare e <> text " of ") $$
                          nest 4 (foldr (($$).altToD) tnil alts)$$anConDP a


altToD :: Alt AbsVal -> Doc
altToD (AlgAlt (c,ids,e)) =
 sep[(conToD c) <> opc <> text " ->",nest 6 (exprToD e)]
  where
   opc = if (null ids) then tnil else  sep (map k ids)
   k id = rodeparid id <> text" "
altToD (Default (id,e)) =
 sep[idToD id <> text "->",nest 6 (exprToD e)]

tidToD :: TId -> Doc
tidToD (MkTId s) = text s
```

```
typeToD :: Type -> Doc
typeToD (EmptyType) = text "( )"
typeToD (TyVar tid) = tidToD tid
typeToD (FunTy t1 t2) = sep[typeToD t1<> text " ->",typeToD t2]
typeToD (ForAllTy tid t) = text "\\/"<> tidToD tid <> text"." <> typeToD t
typeToD (TyConApp tcon ts) = text (tcon) <> (d ts)
 where
     d [] = tnil
     d (t:ts) = text " " <> sep (map rodepart (t:ts))
typeToD (TyProc t1 t2) = sep[text "Process "<> typeToD t1, nest 7 (typeToD t2)]

\end{code}


*****************************************************************
****                Monad to count iterations              ****
*****************************************************************

\begin{code}
newtype ST s a = ST (s -> (s,a))
instance Monad (ST s) where
 return x = ST (\ s -> (s,x))
 m >>= f = ST (\s -> let
                      ST fm = m
                      (s',b) = fm s
                      ST m' = f b
                     in
                      m' s')
 m >> m' = ST (\s -> let
                      ST fm = m
                      (s',_) = fm s
                      ST fm' = m'
                     in
                      fm' s')

type State = [Int]

nuevoCont::ST State ()
nuevoCont = ST (\ s -> ((0::Int):s,()))

inc::ST State ()
inc = ST (\s -> (add1 s,()))

add1 (x:xs) = (x+1):xs

runST :: Int -> Program () -> (State, Program AbsVal)
runST n p = let
              ST fm = evalProg n p
            in
              fm []
\end{code}
```

```
****************************************************************
****                     Polymorphism                       ****
****************************************************************

\begin{code}
--Polymorphic abstraction and concretisation functions
gammap::Type -> TId -> Type -> AbsVal -> ST State AbsVal
gammap (TyConApp tc ts) beta t a
 | tc /= "TTup" = return a
 | otherwise =
     do
      let Tup as = a
          f t1 a1 = gammap t1 beta t a1
      ps <- zipWithM f ts as
      return (Tup ps)
gammap (TyVar delta) beta t a
 | delta /= beta  = return a
 | otherwise = gamma t a

gammap t'@(FunTy t1 t2) beta t a = return (Gamma t' beta t a)
gammap t'@(TyProc t1 t2) beta t a = return (Gamma t' beta t a)
gammap (ForAllTy delta t1) beta t a
 | delta /= beta = gammap t1 beta t a
 | otherwise = error "Hay captura de variables de tipo"


alphap::Type -> TId -> Type -> AbsVal -> ST State AbsVal
alphap (TyConApp tc ts) beta t a
 | tc /= "TTup" = return a
 | otherwise =
     do
      let Tup as = a
          g t1 a1 = alphap t1 beta t a1
      ps <- zipWithM g ts as
      return (Tup ps)
alphap (TyVar delta) beta t a
 | delta /= beta  = return a
 | otherwise = alpha t a

alphap t'@(FunTy t1 t2) beta t a = return (Alpha t' beta t a)
alphap t'@(TyProc t1 t2) beta t a = return (Alpha t' beta t a)
alphap (ForAllTy delta t1) beta t a
 | delta /= beta = alphap t1 beta t a
 | otherwise = error "Hay captura de variables de tipo"

\end{code}
```

```
--El modulo principal es el siguiente
module Main(main) where
import Tokens
import Gram
import NewPP
import NonDet5
main = do
        s    <- readFile "reps.txt"
        let n = read s :: Int -- numero de repeticiones
        print n
        prog <- readFile "ejemplos.txt"
        let (its,prog’) = runST n (parser prog)
        writeFile "results.txt" (pretty 200 150 (progToD prog’))
```

Apéndice C

Algunos ejemplos para el análisis de no determinismo

En las figuras C.1 y C.2 mostramos algunos ejemplos de programas escritos en el lenguaje descrito en el Capítulo 5. No se trata del código fuente proporcionado al analizador sintáctico, sino de una versión (azucarada) más legible. En las figuras C.3 y C.4 se muestra la salida producida por el análisis para estos ejemplos, tal y como lo hace el *pretty printing*. Si una expresión `e` ha sido anotada con una signatura `aw`, se muestra como `e::aw`. Las signaturas funcionales se escriben entre llaves. Las tuplas y las listas se muestran, respectivamente, con los constructores `Tup` y `Cons`.

En primer lugar (Figura C.1) se muestran algunos ejemplos sencillos. Son funciones pequeñas que pretender cubrir el uso de tuplas, el orden superior y la recursión. A continuación (Figura C.2) se muestra el contraejemplo del polimorfismo de la Sección 5.8.3. Aquí f es representada por `counter` y q corresponde a `pair`. La variable `cinstap` corresponde to $(\gamma_{t'\,tinst}(f)\ q)\ d$, por lo que debe tener una anotación d. En la Figura C.4 podemos ver que `countinst` recibe la signatura $nn{+}d$, lo que corresponde a $\mathcal{W}_{tinst}(\gamma_{t'\,tinst}(f))$. Sin embargo se usa el valor abstracto $\gamma_{t'\,tinst}(f)$ para obtener el valor abstracto de `cinstap`, que, como cabía esperar, es d.

Finalmente se presenta una versión simplificada de la topología de trabajadores replicados correspondiente a $n = 2$.

```
--Algunas listas
zero::[Int] = 0:[]
one::[Int] = 1:[]
zdnil::[[Int]] = zero:[]
zoxss::[[Int]] = one:zdnil

--Un entero no determinista
mergeint::Process [[Int]] [Int] = merge Int
xs::[Int] = mergeint # coxss
nondet::Int = head xs

--Algunas funciones con tuplas como resultado
pf1::Int -> (Int,Int) = \pf01::Int.(pf01,2)
pf3::Int -> (Int,Int) = \pf03::Int.(pf03,pf03)
pf4::Int -> (Int,Int) = \pf04::Int.(pf04,nondet)

--Algunas funciones con tuplas como argumentos
at1::(Int,Int) -> Int = \pat1::(Int,Int).
                          case pat1 of (pat11::Int,pat12::Int) -> pat11
at3::(Int,Int) -> Int = \pat3::(Int,Int).
                          case pat3 of (pat31::Int,pat32::Int) -> pat31 + pat32

--Algunas funciones de orden superior
high1::(Int -> Int) -> Int = \f1::Int -> Int.f1 1
high2::(Int -> Int) -> Int -> Int = \f2::Int -> Int.\arg::Int.f2 arg
--La funcion suma
sum::[Int] -> Int = \l1::[Int].case l1 of
                                 [] -> 0
                                 x1:xs1 -> let rest1::Int = sum xs1
                                           in  x1 + rest1
```

Figura C.1: Ejemplo de programa

```
--El contraejemplo del polimorfismo
id::Int -> Int = \v::Int.v
fn::Int -> Int = \z::Int.nondet
counter::\/ delta.(delta,delta) -> delta =
   /\delta.\cp::(delta,delta). case cp of (cp1::delta,cp2::delta) -> cp1
countinst::(Int -> Int, Int -> Int) -> Int -> Int = counter (Int -> Int)
pair::(Int -> Int,Int -> Int) = (id,fn)
cinstap::Int = (countinst pair) 6

--La topologia de trabajadores replicados
replicated ::\/ alpha.\/ beta. Process ([alpha],[alpha]) (beta,beta,beta) ->
             Process beta [alpha] -> Process beta [alpha] -> [alpha] -> beta =
/\alpha./\beta.\m::Process ([alpha],[alpha]) (beta,beta,beta).
       \w1::Process beta [alpha].
       \w2::Process beta [alpha].
       \ts::[alpha]
           let rec
            t::([alpha],[alpha]) = (ts,is)
            om::(beta,beta,beta) = m # t
            px1::beta = case om of (u1::beta,u2::beta,u3::beta) -> u1
            px2::beta = case om of (v1::beta,v2::beta,v3::beta) -> v2
            px3::beta = case om of (q1::beta,q2::beta,q3::beta) -> q3
            o1::[alpha] = w1 # px1
            o2::[alpha] = w2 # px2
            lo1::[[alpha]] = o1:[]
            o1o2::[[alpha]] = o2:lo1
            mergealpha::Process [[alpha]] [alpha] = merge alpha
            is::[alpha] = mergealpha # o1o2
           in
             px3
```

Figura C.2: Ejemplo de programa

```
zero::d =  Cons (0 ::d) ([]::d) ::d
one::d =  Cons (1 ::d) ([]::d) ::d
zdnil::d =  Cons (zero::d) ([]::d) ::d
zoxss::d =  Cons (one::d) (zdnil::d) ::d
xs::n =  (mergeint::{n +n}) # (zoxss::d)::n
nondet::n =  head::{n +d} (xs::n) ::n
pf1::{(n, d) +(d, d)} =
    \ pf01::n. Tup (pf01::n) (2 ::d) ::(n, d)
    ::{(n, d) +(d, d)}
pf3::{(n, n) +(d, d)} =
    \ pf03::n. Tup (pf03::n) (pf03::n) ::(n, n)
    ::{(n, n) +(d, d)}
pf4::{(n, n) +(d, n)} =
    \ pf04::n. Tup (pf04::n) (nondet::n) ::(n, n)
    ::{(n, n) +(d, n)}
at1::{n +d} =
    \ pat1::(n, n).
        case (pat1::(n, n)) of
            Tup (pat11::n)  (pat12::n)  -> pat11::n
        ::n
    ::{n +d}
at3::{n +d} =
    \ pat3::(n, n).
       case (pat3::(n, n)) of
           Tup (pat31::n) (pat32::n) -> Plus::{n n +d} (pat31::n) (pat32::n) ::n
        ::n
    ::{n +d}
high1::{n +d} =
    \ f1::{n +n}. (f1::{n +n}) (3 ::d) ::n
    ::{n +d}
high2::{n n +d} =
    \ f2::{n +n}.
        \ arg::n. (f2::{n +n}) (arg::n) ::n
        ::{n +n}
    ::{n n +d}
sum::{n +d} =
    \ l1::n.
        case (l1::n) of
            Nil  -> 0 ::d
            Cons (x1::n)  (xs1::n)  ->
                  let
                      rest1::n =  (sum::{n +d}) (xs1::n) ::n
                  in
                     Plus::{n n +d} (x1::n) (rest1::n) ::n
                  ::n
        ::n
    ::{n +d}
```

Figura C.3: La salida para los ejemplos de la Figura C.1

```
id::{n +d} =
    \ v::n. v::n
    ::{n +d}
fn::{n +n} =
    \ z::n. nondet::n
    ::{n +n}
counter::{n +d} =
    /\ delta.
        \ cp::(n, n).
            case (cp::(n, n)) of
                Tup (cp1::n)  (cp2::n)  -> cp1::n
            ::n
        ::{n +d}
    ::{n +d}
countinst::{n n +d} =  (counter::{n +d}) (Int -> Int) ::{n n +d}
pair::({n +d}, {n +n}) =  Tup (if::{n +d}) (fn::{n +n}) ::({n +d}, {n +n})
cinstap::d =  ((countinst::{n n +d}) (pair::({n +d}, {n +n})) ::{n +d})
              (6 ::d) ::d
replicated::{n n n n +n} =
 /\ alpha.
   /\ beta.
     \ m::{(n, n, n) +(n, n, n)}.
       \ w1::{n +n}.
        \ w2::{n +n}.
         \ ts::n.
            let
             t::(n, n) = Tup (ts::n) (is::n) ::(n,n)
             om::(n, n, n) =  (m::{(n, n, n) +(n, n, n)}) # (t::(n, n))::(n, n, n)
             px1::n =
                case (om::(n, n, n)) of
                    Tup (u1::n)  (u2::n)  (u3::n)  -> u1::n
                ::n
             px2::n =
                case (om::(n, n, n)) of
                    Tup (v1::n)  (v2::n)  (v3::n)  -> v2::n
                ::n
             px3::n =
                case (om::(n, n, n)) of
                    Tup (q1::n)  (q2::n)  (q3::n)  -> q3::n
                ::n
             o1::n =  (w1::{n +n}) # (px1::n)::n
             o2::n =  (w2::{n +n}) # (px2::n)::n
             lo1::n =  Cons (o1::n) ([]::d) ::n
             o1o2::n =  Cons (o2::n) (lo1::n) ::n
             mergealpha::{n +n} =  (merge::{n +n}) (alpha) ::{n +n}
             is::n =  (mergealpha::{n +n}) # (o1o2::n)::n
            in
             px3::n
            ::n
         ::{n +n}
        ::{n n +n}
       ::{n n n +n}
     ::{n n n n +n}
  ::{n n n n +n}
 ::{n n n n +n}
```

Figura C.4: La salida para los ejemplos de la Figura C.2

Bibliografía

[Abr86] S. Abramsky. Strictness Analysis and Polymorphic Invariance. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark, 1985*, volume 217 of *LNCS*, 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.

[AEL00] M. Alpuente, S. Escobar, and S. Lucas. Redundancy Analyses in Term Rewriting (extended abstract). In *Proccedings of the Ninth International Workshop on Functional and Logic Programming, WFLP'00*, pages 309–323. Technical Report 2000/2039, Dep. Sistemas Informáticos y Computación, Univ. Politécnica de Valencia, 2000.

[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. In Anil Nerode and Mikhail Taitslin, editors, *Proceedings of Logical Foundations of Computer Science (Tver '92)*, volume 620 of *LNCS*, pages 33–44, Berlin, Germany, July 1992. Springer-Verlag.

[Ben93] P. N. Benton. Strictness Properties of Lazy Algebraic Datatypes. In *Proceedings of the Third International Workshop, WSA'93, Padova, Italy, September 22-24, 1993*, volume 724 of *LNCS*, pages 206–217. Springer-Verlag, 1993.

[BGP00] C. Baker-Finch, K. Glynn, and S. L. Peyton Jones. Constructed Product Result Analysis for Haskell. Technical Report TR2000/13, Department of Computer Science of The University of Melbourne, 2000.

[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 *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.

[BLOMP98] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language Definition and Operational Semantics. Technical Report, Bericht 96-10. Revised version 1.998, Philipps-Universität Marburg, Germany, 1998.

[Bon90] A. Bondorf. Self-Applicable Partial Evaluation. Ph.D. Thesis 90/17, DIKU, Univ. of Copenhagen, Denmark, 1990.

[BS96] E. Barendsen and S. Smetsers. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, December 1996.

[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.

[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.

[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*, pages 238–252. ACM Press, 1977.

[CC79] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Conference Record of the 6th Annual ACM Symposium on Principles on Programming Languages*, pages 269–282. ACM Press, 1979.

[CC92a] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2-3):103–180, July 1992.

[CC92b] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[CD85] J. H. Chang and A. M. Despain. Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis. In *Proceedings of the International Symposium on Logic Programming*, pages 10–21. IEEE Computer Society, Technical Committee on Computer Languages, The Computer Society Press, July 1985.

[CG83] K. L. Clark and S. Gregory. PARLOG: a Parallel Logic Programming Language. Technical Report 83/5, Imperial College, London, 1983.

[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*, volume 201 of *LNCS*, pages 35–49. Springer-Verlag, 1985.

[Deb86] S. K. Debray. Dataflow analysis of logic programs. Technical Report, Stony Brook, New York, 1986.

[DW90] K. Davis and P. L. Wadler. Backwards Strictness Analysis: Proved and Improved. In *Proceedings of the 2nd Glasgow Workshop on Functional Programming, Workshops in Computing*, pages 12–30. Springer-Verlag, August 1990.

[Dyb87] P. Dybjer. Inverse Image Analysis. In T. Ottmann, editor, *Automata, Languages and Programming, 14th International Colloquium*, volume 267 of *LNCS*, pages 21–30, Karlsruhe, Germany, 13–17 July 1987. Springer-Verlag.

[FS91] G. Filè and P. Sottero. Abstract Interpretation for Type Checking. In J. Małuszyński and M. Wirsing, editors, *Proceedings of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, PLILP91, Passau, Germany*, volume 528 of *LNCS*, pages 311–322. Springer-Verlag, August 1991.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. FACILE: A Symmetric Integration of Concurrent and Functional Programming. *Journal of Parallel Programming*, 18(2), 1989.

[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.

[GS01] J. Gustavsson and J. Sveningsson. A Usage Analysis with Bounded Usage Polymorphism and Subtyping. In *Selected Papers of the 12th International Workshop on Implementation of Functional Languages, IFL'00*, volume 2011 of *LNCS*, pages 140–157. Springer-Verlag, 2001.

[Han94a] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[Han94b] M. Hanus. Towards the Global Optimization of Functional Logic Programs. In *Proceedings of the 5th International Conference on Compiler Construction*, volume 786 of *LNCS*, pages 68–82. Springer, 1994.

[Han95] M. Hanus. Analysis of Residuating Logic Programs. *Journal of Logic Programming*, 24(3):219–245, 1995.

[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 Press, January 1985.

[Hen82] P. Henderson. Purely Functional Operating Systems. In *Functional Programming and its Applications: An Advanced Course*, pages 177–191. Cambridge University Press, 1982.

[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 *LNCS*, pages 219–232. Springer, Berlin, 1992.

[Hid00] M. Hidalgo. Semánticas formales para un lenguaje funcional paralelo y reactivo. Tesis matriculada en el Dpto. de Sistemas Informáticos y Programación (UCM); Directora: Yolanda Ortega, 2000.

[HL90] R. J. M. Hughes and J. Launchbury. Projections for Polymorphic Strictness Analysis. Research Report CSC/90/R33, Department of Computer Science, University of Glasgow, Glasgow, UK, 1990.

[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. A Type-based Framework for Program Analysis. In *Proceedings of the First International Static Analysis Symposium, SAS'94*, volume 864 of *LNCS*, pages 380–394. Springer-Verlag, 1994.

[HL94b] C. Hankin and D. Le Metayer. Lazy Type Inference for the Strictness Analysis of Lists. In *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming*, volume 788 of *LNCS*, pages 257–271. Springer-Verlag, 1994.

[HL99] M. Hanus and S. Lucas. A Semantics for Program Analysis in Narrowing-Based Functional Logic Languages. In *Proc. of the 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, volume 1722 of *LNCS*, pages 353–368. Springer, 1999.

[HLA94] L. Huelsbergen, J. R. Larus, and A. Aiken. Using the Runtime Sizes of Data Structures to Guide Parallel-Thread Creation. In *Proceedings of the Conference on Lisp and Functional Programming*, pages 79–90. ACM Press, June 1994.

[HLP95] K. Hammond, H. W. Loidl, and A. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *HPFC'95, High Performance Functional Computing, A. P. W. Bohm, J. T. Feo (eds.)*, pages 208–221, 1995.

[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.

[HM99] K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming.* Springer, 1999.

[HO90] R. J. M. Hughes and J. O'Donnell. Expressing and Reasoning About Non-Deterministic Functional Programs. In *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, pages 308–328. Springer-Verlag, 1990.

[HP99] R. J. M. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space; Towards Embedded ML Programming. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, ACM Sigplan Notices, pages 70–81, Paris, France, September 1999. ACM Press.

[HPS96] R. J. M. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT*, pages 410–423, 1996.

[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*, ACM SIGPLAN NOTICES, pages 154–165. ACM Press, September 1991.

[HS00] M. Hanus and F. Steiner. Type-based Nondeterminism Checking in Functional Logic Programs. In *Proc. of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'2000)*, pages 202–213. ACM Press, 2000.

[Hud86a] P. Hudak. Para-Functional Programming. *IEEE Computer*, 19(8):60–69, Aug. 1986.

[Hud86b] 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 Press, 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 University, 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.

[Hug95] R. J. M. Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 53–96. Springer Verlag, 1995.

[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.

[HW88] P. Hudak and P. Wadler. Report on the Functional Programming Language Haskell. Technical Report RR-656, Yale University. Dep. of Computer Science, 1988.

[HY85] P. Hudak and J. Young. A Set-theoretic Characterisation of Function Strictness in the Lambda Calculus. Technical Report YALEU/DCS/RR-391, Yale University, 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 Press, New York, NY, 1986.

[HZ94] M. Hanus and F. Zartmann. Mode Analysis of Functional Logic Programs. In *Proc. 1st International Static Analysis Symposium*, volume 864 of *LNCS*, pages 26–42. Springer, 1994.

[Jen91] T. P. Jensen. Strictness Analysis in Logical Form. In R. J. M. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 352–366. Springer-Verlag, New York, NY, 1991.

[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.

[JLAV93] M. C. Williams J. L. Armstrong and S. R. Virding. *Concurrent Programming in Erlang.* Prentice Hall, 1993.

[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*, volume 432 of *LNCS*, pages 227–239. Springer-Verlag, 1990.

[JMMMN93] J. A. Jiménez-Martin, J. Mariño, and J. J. Moreno-Navarro. Some Techniques for the Efficient Compilation of Lazy Narrowing into Prolog. In *Proceedings of LOPSTR'92*, Workshops in Computer Science. Springer-Verlag, 1993.

[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.

[Jon96] S. L. Peyton Jones. Compiling Haskell by Program Transformations: A Report from the Trenches. In *Proceedings of the 6th European Symposium on Programming, Linköping, Sweden, April 22-24, 1996, ESOP'96*, volume 1058 of *LNCS*, pages 18–44. Springer-Verlag, 1996.

[JS87] N. D. Jones and H. Søndergaard. A Semantics-Based Framework for the Abstract Interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood, Chichester, England, 1987.

[Jun98] S. B. Junaidu. *A Parallel Functional Language Compiler for Message-Passing Multicomputers.* Phd. thesis, School of Mathematical and Computational Sciences. University of St. Andrews, March 1998.

[Kam92] S. Kamin. Head-strictness is not a Monotonic Abstract Property. *Information Processing Letters*, 41(4):195–198, March 1992.

[Kel89] Paul Kelly. *Functional Programming for Loosely Coupled Multiprocessors.* Pitman, 1989.

[Klu98] U. Klusik. Implementors Handbook for the Eden compiler 0.04. Available from Author, December 1998.

[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.

[KOMP99] U. Klusik, Y. Ortega-Mallén, and R. Peña. Implementing Eden - or: Dreams Become Reality. In *Implementation of Functional Languages, IFL'98, London, Sept. 1998. Selected Papers*, volume 1595 of *LNCS*, pages 103–119. Springer-Verlag, 1999.

[KPR01] U. Klusik, R. Peña, and F. Rubio. Replicated Workers in Eden. In *2nd International Workshop on Constructive Methods for Parallel Programming (CMPP 2000).* To be published by Nova Science, 2001.

[KPS00] U. Klusik, R. Peña, and C. Segura. Bypassing of Channels in Eden. In *Trends in Functional Programming. Selected Pa-*

pers of the 1st Scottish Functional Programming Workshop, SFP'99, pages 2–10. Intellect, 2000.

[Lau93] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proceeding of the 20th Conference on Principles of Programming Languages, POPL'93*, pages 144–154. ACM, 1993.

[LGH+92] J. Launchbury, A. Gill, R. J. M Hughes, S. Marlow, S. L. Peyton Jones, and P. L. Wadler. *Avoiding Unnecessary Updates.* Springer-Verlag, New York, NY, 1992. Springer-Verlag Workshops in Computing.

[LMT+97] H. W. Loidl, R. Morgan, P.W. Trinder, S. Poria, C. Cooper, S.L. Peyton Jones, and R. Garigliano. Parallelising a Large Functional Program; Or: Keeping LOLITA Busy. In *Implementation of Functional Languages, 9th International Workshop, IFL'97, St. Andrews, Scotland, UK, September 10-12, 1997*, volume 1467 of *LNCS*, pages 198–213, 1997.

[Loi96] H. W. Loidl. *GranSim User's Guide*, 1996. Department of Computing Science. University of Glasgow http://www.dcs.glasgow.ac.uk/fp/software/gransim/.

[Loi98] H. W. Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow, March 1998.

[Mac71] S. MacLane. *Categories for Working Mathematicians.* Springer-Verlag, New York, 1971.

[Mar93] S. Marlow. Update Avoidance Analysis by Abstract Interpretation. In J. T. O'Donnell, editor, *Proceedings of the Glasgow Workshop on Functional Programming*, Workshops in Computer Science. Springer-Verlag, 5th–7th July 1993.

[McC63] J. McCarthy. Towards a Mathematical Theory of Computation. In *Proc. IFIP Congress 62*, pages 21–28, Amsterdam, 1963. North-Holland.

[Mel86] C. S. Mellish. Abstract Interpretation on Prolog Programs. In E Shapiro, editor, *International Conference on Logic Programming, London, United Kingdom*, volume 225 of *LNCS*, pages 107–116. Springer-Verlag, July 1986.

[MH87] J. C. Martin and C. Hankin. Finding Fixed Points in Finite Lattices. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 426–445. Springer-Verlag, Berlin, DE, 1987.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(1, 2, 3 and 4):315–347, 1992.

[Mic01] G. Michaelson. The Dynamic Properties of Hume: A Functionally-Based Concurrent Language with Bounded Time and Space Behaviour. In *Selected Papers of the 12th International Workshop on Implementation of Functional Languages, IFL'00*, volume 2011 of *LNCS*, pages 122–139. Springer-Verlag, 2001.

[MMN98] J. Mariño and J.J. Moreno-Navarro. Recovering Sequentiality in Functional-Logic Programs. In Freire, Falaschi, and Vilares, editors, *APPIA-GULP-PRODE'98*, pages 137–150. Universidad A Coruña, 1998.

[MNKM+93] J. J. Moreno-Navarro, H. Kuchen, J. Mariño, W. Hans, and S. Winkler. Efficient Lazy Narrowing using Demandedness Information. In *Proceedings of the fifth International Symposium on Programming Language Implementation and Logic Programming}*,, volume 714 of *LNCS*, pages 167–183. Springer-Verlag, 1993.

[Mo84] A. Mycroft and R. A. o'keefe. A Polymorphic Type System for PROLOG. *Artificial Intelligence*, 23(3):295–307, August 1984.

[Mog89] T. Mogensen. Binding Time Analysis for Polymorphically Typed Higher Order Languages. In J Diaz and F Orejas, editors, *TAPSOFT'89, Barcelona, Spain*, volume 352 of *LNCS*, pages 298–312. Springer-Verlag, March 1989.

[Mos94] C. Mossin. Polymorphic Binding-Time Analysis. Master's thesis, DIKU, University of Copenhagen, Denmark, July 1994.

[MS89] Kim Marriott and Harald Søndergaard. Semantics-based Dataflow Analysis of Logic Programs. In *IFIP'89*, pages 610–606. North-Holland, 1989.

[Myc81] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* Phd. thesis, technical report cst-15-81, Dept Computer Science, University of Edinburgh, December 1981.

[NN94] H. R. Nielson and F. Nielson. Higher-Order Concurrent Programs with Finite Communications Topology. In *Conference*

Record of POPL '94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, pages 84–97, New York, NY, January 1994. ACM Press.

[NN96] F. Nielson and H. R. Nielson. Operational Semantics of Termination Types. *Nordic Journal of Computing*, 3(2):144–187, Summer 1996.

[NN97] F. Nielson and H. R. Nielson. Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 332–345, Paris, France, 15–17 January 1997.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[Par97] L. Pareto. Sized Types. Licenciate Dissertation, Chalmers University of Technology, Göteborg, Sweden, 1997.

[Par00] L. Pareto. *Types for Crash Prevention*. PhD thesis, Chalmers University of Technology and Göteborg University, Sweden, 2000.

[PC87] 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.

[Pe97] J. Peterson and K. Hammond (eds.). Report on the Programming Language Haskell. version 1.4. Technical Report, Yale University, April 1997.

[PGF96] S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'96*, pages 295–308. ACM Press, 1996.

[PHH$^+$93] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Inf. Technology, Keele, DTI/SERC*, pages 249–257, 1993.

[Pie91] B. C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, Cambridge, Mass., 1991.

[Pla84] D. A. Plaisted. The Occur-check Problem in Prolog. In *1984 International Symposium on Logic Programming*. IEEE, New York, USA, 1984.

[PP93] S. L. Peyton Jones and W. Partain. Measuring the Effectiveness of a Simple Strictness Analyser. In *Glasgow Workshop on Functional Programming 1993*, Workshops in Computing, pages 201–220. Springer-Verlag, 1993.

[PPRS00a] C. Pareja, R. Peña, F. Rubio, and C. Segura. Optimizing Eden by Transformation. In *Trends in Functional Programming (Volume 2). Selected Papers of 2nd Scottish Functional Programming Workshop, SFP'00*, pages 13–26. Intellect, 2000.

[PPRS00b] C. Pareja, R. Peña, F. Rubio, and C. Segura. Optimizing Eden by Transformation. In *Proceedings of the Ninth International Workshop on Functional and Logic Programming, WFLP'00*, pages 89–103. Technical Report 2000/2039, Dep. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2000.

[PPS96] S. L. Peyton Jones, W. Partain, and A. L. M. Santos. Let-floating: Moving Bindings to give Faster Programs. *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP'96*, pages 1–12, 24-26 May 1996.

[PR01] R. Peña and F. Rubio. Parallel Functional Programming at Two Levels of Abstraction. In *Principles and Practice of Declarative Programming, PPDP'01*, pages 187–198. ACM Press, 2001.

[PRS00] R. Peña, F. Rubio, and C. Segura. Deriving Non-hierarchical Process Topologies. In *Proceedings of 3rd Scottish Functional Programming Workshop, SFP'01*, pages 157–174, 2000. Submitted to referees.

[PS98a] R. Peña and C. Segura. Bypassing of Channels in Eden at Process Creation Time. Technical Report 87/98. Dep. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain, 1998. (http://dalila.sip.ucm.es/miembros/clara/publications.html).

[PS98b] S. L. Peyton Jones and A. L. M. Santos. A Transformation-based Optimiser for Haskell. *Science of Computer Programming 32(1-3):3-47*, September 1998.

[PS00a] C. Pareja and C. Segura. Efecto de las Transformaciones de GHC sobre Edén. Technical Report 101-00. Dep. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain, 2000.

[PS00b] R. Peña and C. Segura. Two Non-determinism Analyses in Eden. Technical Report 108-00. Dep. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain, 2000. (http://dalila.sip.ucm.es/miembros/clara/publications.html).

[PS01a] R. Peña and C. Segura. A Comparison between three Non-determinism Analyses in a Parallel-Functional Language. In *Primeras Jornadas sobre Programación y Lenguajes, PROLE'01*, pages 263–277, 2001.

[PS01b] R. Peña and C. Segura. Three Non-determinism Analyses in a Parallel-Functional Language. Technical Report 117-01, Dep. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain, 2001. (http://dalila.sip.ucm.es/miembros/clara/publications.html).

[PS01c] R. Peña and C. Segura. A Polynomial Cost Non-Determinism Analysis. In *Proceedings of the 13th International Workshop on Implementation of Functional Languages, IFL'01*, pages 303–318, 2001. Submitted to referees for publication in LNCS.

[PS01d] R. Peña and C. Segura. Non-Determinism Analysis in a Parallel-Functional Language. In *Selected Papers of the 12th International Workshop on Implementation of Functional Languages, IFL'00*, volume 2011 of *LNCS*, pages 1–18. Springer-Verlag, 2001.

[PS01e] R. Peña and C. Segura. Sized Types for Typing Eden Skeletons. In *Proceedings of the 13th International Workshop on Implementation of Functional Languages, IFL'01*, pages 33–51, 2001. Submitted to referees for publication in LNCS.

[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.

[Rep91] J. H. Reppy. CML: A Higher-order Concurrent Language. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 293–305. ACM Press, 1991.

[RG94] B. Reistad and D. K. Gifford. Static Dependent Costs for Estimating Program Execution Time. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 65–78, 1994.

[Rub01] F. Rubio. *Programación funcional paralela eficiente en Edén*. PhD thesis, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2001.

[SA89] P. Sestoft and G. Argo. Detecting Unshared Expressions in the Improved Three Instructions Machine. 12 pages. DIKU, University of Copenhagen, Denmark. Unpublished, November 1989.

[San95] A. L. M. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Glasgow University, Dept. 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.

[Seg99] C. Segura. Técnicas de análisis de programas en lenguajes funcionales. Trabajo de Tercer Ciclo. Dep. Sistemas Informáticos y Programación. Universidad Complutense de Madrid, 1999.

[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.

[Sha83] E. Shapiro. A Subset of Concurrent Prolog and its Interpreter. Technical Report TR-003, ICOT Institute for New Generation Computer Technology, Tokyo, 1983.

[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.

[Sij89] B. A. Sijtsma. On the Productivity of Recursive List Definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, October 1989.

[SNN97] K. L. Solberg Gasser, F. Nielson, and H. R. Nielson. Systematic Realisation of Control Flow Analyses for CML. In

Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming, pages 38–51, Amsterdam, The Netherlands, 9–11 June 1997.

[Søn86] H. Søndergaard. An Application of Abstract Interpretation of Logic Programs: Occur-check Reduction. In *ESOP'86, Saarbrucken, Germany*, volume 213 of *LNCS*, pages 327–338. Springer-Verlag, 1986.

[SS90] H. Søndergaard and P. Sestoft. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, 27(6):505–517, May 1990.

[SS92] H. Søndergaard and P. Sestoft. Non-Determinism in Functional Languages. *Computer Journal*, 35(5):514–523, October 1992.

[THM+96] P. W. Trinder, K. Hammond, S. J. Mattson, Jr., A. S. Partridge, and S. L. Peyton Jones. GUM : A Portable Parallel Implementation of Haskell. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemantation*, pages 79–88, New York, May 21–24 1996. ACM Press.

[TWM95] D. N. Turner, P. L. Wadler, and C. Mossin. Once Upon a Type. In *7'th International Conference on Functional Programming and Computer Architecture*, pages 1–11, La Jolla, California, June 1995. ACM Press.

[UAH74] J. D. Ullman, A. V. Aho, and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, 1974.

[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.

[WH87] P. L. Wadler and R. J. M. Hughes. Projections for Strictness Analysis. In G. Kahn, editor, *Proceedings of Conference Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 385–407. Springer-Verlag, Berlin, DE, 1987.

[WJ99] K. Wansbrough and S. L. Peyton Jones. Once Upon a Polymorphic Type. In *Conference Record of POPL '99: The 26th*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 15–28, San Antonio, Texas, January 1999. ACM Press.

[WJ00] K. Wansbrough and S. L. Peyton Jones. Simple Usage Polymorphism. In *The Third ACM SIGPLAN Workshop on Types in Compilation*, Montreal, Canada., September 2000.

[Wra85] S. C. Wray. A New Strictness Detection Algorithm. In *Aspenäs Workshop on Implementation of Functional Languages, Göteborg*, 1985.

[Zar97] F. Zartmann. Denotational Abstract Interpretation of Functional Logic Programs. In *Proceedings of the 4th International Symposium, SAS '97, Paris, France, September 8-10, 1997*, volume 1302 of *LNCS*, pages 141–159. Springer-Verlag, 1997.