Tesis de Licenciatura

Inferencia de Tipos Concretos en Squeak

Director: Dr. Leandro Caniglia
Co-Director: Dr. Marcelo Frias
Alumno: Francisco Garau

Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

caniglia@dm.uba.ar — fg0u@dc.uba.ar — mfrias@dc.uba.ar
fgarau@softhome.net

A mi esposa Alejandra,

sin su ayuda y aliento, no lo hubiera logrado.

Indice

Figuras

Agradecimientos

En primer lugar quiero agradecer a mi director Leandro Caniglia. Su espíritu inquieto y entusiasmo contagioso fueron los principales motivos me impulsaron a encarar la inferencia de tipos. El me acompañó a lo largo de este trabajo, con numerosos consejos, charlas y reuniones. A su lado aprendí lo que significa investigar y luego comprobé lo gratificante que ello es.

En segundo lugar quiero agradecer a Valeria Murgia, Luciano Notarfrancesco y Torge Husfeldt. No solamente aportaron ideas sino también aportaron código. Valeria hizo la primera implementación de los estados en las variables. Luciano implementó por primera vez el mecanismo *lazy* para obtener el tipo de un objeto. Torge hizo diversas extensiones a las primitivas, e implementó el análisis de la expresión de cascada[1].

En tercer lugar, quiero agradecer a todos aquellos que se interesaron en este trabajo y participaron en las reuniones organizadas por Leandro. Las reuniones de los *MathMorphs* podían ser tanto en la Facultad como en Ezeiza. Además de Leandro, Luciano y Valeria, solían estar Gerardo Richarte, Pablo Malavolta y Alejandro Weil.

En este trabajo utilice extensivamente los *MorphicWrappers*[2]. Gracias a ellos, la experiencia de interacción directa con los objetos es más real que nunca. Quiero agradecer entonces, a todos aquellos que participan en el proyecto.

Las ideas de este trabajo serán presentadas en el seminario ESUG 2001, organizado por el grupo de usuarios europeos de Smalltalk. Allí también será expuesto en el Simposio Doctoral con el objetivo de conocer la opinión de expertos sobre cómo continuar esta investigación.

[1] Torge es alemán y se enteró de este proyecto a través de nuestro *swiki* (`http://typeinference.swiki.net`). Sin ninguna documentación pudo mirar el código, entenderlo y extenderlo.

[2] `http://mathmorphs.swiki.net`

Abstract

Squeak is a language without type declarations; this simplifies and allows exploratory programming. However, concrete type information is a valuable aid to program understanding, application extraction or code optimization. We present the design and implementation in Squeak of a *type inference engine* that can infer concrete types for some expressions of the language. Our engine can analyze code that includes assignments, inheritance, message sends, primitives, blocks and even #perform:. We extend the definition of concrete type to include the types of instance variables. We add a notion of state to variables and message arguments, that allows us to distinguish the types of their different accesses. We infer types by doing an abstract interpretation over the domain of concrete types, emphasizing the relationship that exists between run-time and inference-time.

Resumen

Squeak desciende directamente de Smalltalk y como tal no requiere declaraciones de tipos. Esto permite la programación exploratoria y facilita la escritura de código genérico y reusable. Pero cierta información de tipos es necesaria para realizar diversos análisis estáticos. Distinguimos entre *tipo concreto* y *tipo abstracto* porque consideramos que el código fuente tiene dos lectores de naturaleza distinta: *el compilador* y *el ser humano*. El compilador está interesado en aquellos tipos que describen la estructura interna de los objetos porque los necesita para realizar optimizaciones de código: estos son los *tipos concretos*. El ser humano está interesado en aquellos tipos que describen las propiedades de los objetos porque desea saber cómo utilizar una porción de código: estos son los *tipos abstractos*. Insertar declaraciones de tipos concretos produce código especializado y poco reusable. Sin embargo, la información de tipos concretos es importante para optimizar código, extraer aplicaciones, garantizar que no ocurran ciertos errores o facilitar la comprensión de un programa. Nuestro objetivo es *inferir* los tipos concretos presentes en un programa Squeak.

Presentamos el diseño e implementación en Squeak de un *motor de inferencia* capaz de inferir tipos concretos para ciertas expresiones del lenguaje. Nuestro motor puede analizar código con asignaciones, herencia, envíos de mensajes, primitivas, bloques e incluso #perform:. Ampliamos la definición de tipo concreto para considerar también el tipo de las variables de instancia. Incorporamos a las variables y argumentos de mensajes, una noción de estado que nos permite distinguir los tipos de sus diferentes accesos. Nuestra inferencia de tipos se basa en la analogía que existe entre el tiempo de ejecución y el tiempo de análisis. Inferimos tipos realizando una interpretación abstracta sobre el dominio de los tipos concretos.

1. Introducción

Motivación

En Smalltalk —y por ende en Squeak— no hay declaraciones de tipos; esto facilita la programación exploratoria pues se puede comenzar a programar teniendo una idea vaga o incompleta del problema. A medida que se programa el diseño se adapta, las vaguedades se eliminan y las ambigüedades se aclaran; la programación es una herramienta fundamental del aprendizaje. Las declaraciones estáticas de tipos dificultan la programación exploratoria pues requieren un diseño previo.

En otros lenguajes de programación, las declaraciones estáticas de tipos tienen el doble propósito de ayudar al compilador y de servir como documentación a otras personas. Estos objetivos tienen un conflicto intrínseco porque la información que necesita el compilador es radicalmente diferente a la que necesitan las personas. El compilador necesita información precisa y específica para realizar optimizaciones de código. Por el contrario, el programador quiere documentar su código fuente en forma general y abstracta, porque quiere reutilizarlo en la mayor cantidad de situaciones posibles. Necesitamos distinguir entre la noción de *tipo concreto* y *tipo abstracto*.

Los tipos concretos sirven para describir la implementación y estructura interna de los objetos—información típicamente requerida por el compilador y otras herramientas. El *tipo concreto* de un objeto es la clase a la cual pertenece. No tiene sentido hablar del tipo concreto de una expresión en forma aislada; es necesario contar con una expresión inicial o *programa*. Veamos el método Object class | id para apreciar como varía el tipo del argumento x cuando cambia la expresión inicial. Si partimos de la expresión Object id: 3 el tipo concreto de x es ⟨SmallInteger⟩, y si partimos de Object id: 'hola' es ⟨String⟩.

```
Object class | id: x
    ↑ x
```

Los tipos abstractos sirven para describir las propiedades que los objetos deben cumplir—información útil para facilitar la comprensión del código

fuente.[1] Los tipos abstractos no se definen para los objetos sino para las expresiones. El *tipo abstracto* de una expresión es el conjunto de mensajes que debe entender. En el ejemplo de Object class | id:, el tipo abstracto de x es el conjunto vacío pero si consideramos el método Object | printString, el tipo abstracto de self es el conjunto {#printOn:}. Un objeto puede satisfacer muchos tipos abstractos al mismo tiempo (tantos como subconjuntos de mensajes entienda) pero tiene un solo tipo concreto.

```
Object | printString
    | stream |
    stream ← WriteStream on: String new.
    self  printOn: stream.
    ↑ stream contents
```

El interés principal de este trabajo es obtener la información de tipos concretos presentes en un programa Smalltalk. Los tipos concretos son útiles porque permiten reducir las posibilidades de la vinculación tardía (*late-binding*). En Smalltalk, mirar una expresión de envío de mensaje no alcanza para determinar cuál método será invocado; esto depende de la clase del objeto receptor, información ausente en el código fuente. Los tipos concretos permiten conocer las clases que pueden ocurrir en cada expresión y, en consecuencia, los métodos que pueden ser invocados. Conocer los métodos que pueden ser invocados desde cada expresión, facilita la construcción del grafo de llamadas (*call graph*). El grafo de llamadas es el punto de partida para realizar diversos análisis estáticos. Luego se lo puede utilizar para optimizar código, extraer aplicaciones, garantizar que no ocurran ciertos errores o facilitar la comprensión de un programa [PS91, PC94, GPMB95, Gro98].

Nuestro objetivo es obtener la información de tipos concretos sin obligar al programador a ingresar declaraciones estáticas. Escribir declaraciones estáticas de tipos concretos genera código demasiado especializado y poco reusable, dificultando además, la programación exploratoria. Obtenemos información de tipos concretos, mediante el análisis del código fuente. Como resultado de este análisis, *inferimos* un conjunto de tipos concretos para cada expresión del programa. Este programa, en realidad, es una *expresión inicial*, y es el único requisito que tenemos para comenzar el análisis.

Resultados

En este trabajo, presentamos el diseño e implementación en Squeak de un *motor de inferencia* capaz de deducir tipos concretos, en forma precisa, para

[1] En Java los *tipos abstractos* se llaman *Interfaces*.

ciertas expresiones del lenguaje. Realizamos la inferencia, mediante la interpretación abstracta de un programa sobre el dominio de los tipos concretos. Esta idea fue implementada originalmente por Ole Agesen en el lenguaje *Self* , dónde presenta en forma intuitiva la inferencia de tipos como un problema de flujo [Age94, Age96]. Inferimos tipos, basándonos en la analogía que Agesen presenta entre tiempo de *ejecución* y tiempo de *inferencia*. La guía de este trabajo ha sido: *emular en tiempo de inferencia lo que sucede en tiempo de ejecución*. No solamente usamos esta idea para razonar el problema, sino que también la reflejamos en el diseño e implementación del motor.

Nuestro trabajo no solamente resuelve el paso de Self a Squeak, sino que también aporta soluciones teóricas a cuestiones que quedaron pendientes en la tesis de Agesen. Nuestro motor puede inferir con precisión el tipo de las variables de instancia pues ampliamos la definición de tipo concreto para contemplarlas. Incorporamos a las variables una noción de estado que nos permite distinguir los tipos de sus diferentes accesos (algo solamente insinuado por Agesen) y aplicamos esta noción a los argumentos de los mensajes, para contemplar así, los efectos colaterales. Nuestro motor de inferencia puede analizar código con asignaciones, herencia, envíos de mensajes, primitivas, bloques y #perform:.

Contribuciones

- Una implementación en Squeak, donde queda reflejada la correspondencia entre tiempo de ejecución y tiempo de análisis (en nuestro diseño existen clases como TiInterpreter, TiCompiler, TiPrimitive, TiClass, etc).[2]

- Una noción de múltiples estados para las variables y argumentos de mensajes. Esto permite analizar con precisión las variables locales (que a diferencia de Self, nacen inicializadas con nil) y contemplar los efectos colaterales que sufren los los argumentos de los mensajes. Realizamos de esta forma, un análisis *sensible al flujo* [Age96, §4.5].

- Una definición de tipo concreto que incluye las variables de instancia y las variables indexadas. En nuestro trabajo, el tipo del punto 1.0@1.0 es ⟨Point x: ⟨Float⟩ y: ⟨Float⟩⟩ y el tipo de #(5.0 'dos') es ⟨Array (⟨Float⟩ ⟨String⟩)⟩. Agesen, en cambio, diferencia el tipo de los objetos según el lugar dónde fueron clonados (*allocation points*) [Age96, §4.3.1].

- Una adaptación de la noción de estado para las variables de instancia. Los cambios de estado en las variables de instancia producen cambios

[2] Ti son las iniciales de Type inference que utilizamos para distinguir nuestras clases

de tipo. Si analizamos la expresión (1@1) x: 'hello' obtenemos el nuevo tipo ⟨Point x: ⟨String⟩ y: ⟨SmallInteger⟩⟩, lo cual solamente es insinuado por Agesen [Age96, §4.5].

- Una implementación *lazy* para averiguar el tipo de las variables de instancia, donde sólo se expanden los tipos cuando es necesario. Por ejemplo, el tipo de la variable aPoint es ⟨Point ?⟩ y el de la variable global Smalltalk es ⟨SystemDictionary ?⟩. Esto nos permite realizar el análisis en forma eficiente.

- Una implementación capaz de analizar código con envíos indirectos de mensajes (#perform: y sus variantes). Para resolver esto, en el caso de los símbolos incluimos al propio objeto en la definición de su tipo concreto. Así, tenemos tipos como ⟨Symbol #hello⟩ o ⟨Symbol #goodbye⟩.

- Una implementación capaz de detectar métodos recursivos y mensajes que no serán entendidos. Un método es recursivo si durante su análisis es necesario volver a analizarlo con el mismo juego de parámetros. Un mensaje no será entendido si alguno de los tipos de la expresión receptora no tiene un método para responderlo.

Limitaciones

El conjunto de expresiones que actualmente podemos tipar es limitado (los principales ejemplos están en las clases TObject, TPoint y T3DPoint). Nuestro trabajo es un primer paso para lograr el objetivo más ambicioso de inferir tipos en programas reales. Confiamos que otros miembros de la comunidad Squeak podrán continuarlo, extenderlo o utilizarlo para comprender los problemas fundamentales de la inferencia de tipos. También somos conscientes de los límites intrínsecos de la inferencia de tipos. Hay expresiones donde nunca será posible inferir tipos concretos en forma precisa. Por ejemplo, aquellas expresiones que dependen de las acciones del usuario y no se conocen hasta el momento de la ejecución.

```
string ← FillInTheBlank request: 'Envie un mensaje a Smalltalk '.
↑ Smalltalk perform: string asSymbol

string ← FillInTheBlank request: 'Evalue una expresion '.
↑ Compiler evaluate: string
```

La inferencia de tipos concretos requiere fijar el conjunto de clases y métodos antes de comenzar el análisis. No es posible analizar métodos que todavía no existen o inferir tipos correspondientes a clases aún no creadas. Los tipos concretos imponen una visión de *mundo cerrado* [Age96, p.8].

2. Elección de Squeak

Squeak es un descendiente directo de Smalltalk.[1] Se diferencia de las demás implementaciones comerciales por su espíritu innovador, abierto y colaborativo. Squeak es actualmente uno de los proyectos más activos y atractivos que existen en el mundo, con participación de especialistas de todo el planeta. Lo elegimos como ambiente de desarrollo porque es un excelente ambiente para investigar, experimentar y compartir nuevas ideas.

Squeak y Smalltalk son prácticamente iguales desde el punto de vista del lenguaje, y en consecuencia, de la inferencia de tipos. Esto permite que nuestro trabajo sea fácilmente adaptable a cualquier versión de Smalltalk. Debido a esta similitud, nos referimos en forma indistinta a cualquiera de ellos.

El Proyecto Smalltalk

Smalltalk es un proyecto que nació hace más de 30 años. El proyecto tuvo origen en el laboratorio creado por la empresa Xerox para investigar en Ciencias Físicas y de la Computación. El laboratorio de Palo Alto Research Center (PARC) fue creado en 1970 y estuvo entre los mejores de su tiempo. Numerosas invenciones que disfrutamos hoy en día, tuvieron allí su origen: la impresora láser, la red Ethernet, el disco óptico, la arquitectura cliente/servidor, la interface gráfica de usuario y la Programación Orientada a Objetos. Uno de los grupos dentro del laboratorio, el Learning Research Group, estuvo dedicado a la investigación del aprendizaje. Allí surgió la necesidad de construir un lenguaje de programación más cercano al pensamiento humano (luego bautizado *Smalltalk*). El grupo liderado por Alan Kay e integrado por Dan Ingalls, Ted Kaehler y Adele Goldberg, perseguía el objetivo de construir un sistema que pudiera ser comprendido por una sola persona —y especialmente por los niños—. Ellos querían *"proveer soporte computacional para el espíritu creativo que hay en todo ser humano"* [Ing81]. Esto imprimió a Smalltalk un carácter especial que lo diferencia de la mayoría de los lenguajes de programación.[2]

[1] La página oficial de Squeak es `http://www.squeak.org`

[2] Logo es una excepción pues Papert lo pensó a partir de las necesidades de los niños

En el proyecto Smalltalk ocurrió una idea que sobresale por su sencillez y extrema adaptabilidad. Como la rueda o el arco arquitectónico, fue un descubrimiento que inicialmente no es intuitivo, resulta obvio una vez conocido y puede ser adaptado a una gran variedad de casos. Los investigadores del proyecto descubrieron que sólo dos nociones bastan para describir un sistema de programación: *objeto* y *mensaje*. Hasta ese momento existía la discusión sobre cuántas y cuáles debían ser las características de un lenguaje de programación para que fuera lo suficientemente expresivo. La tendencia era agregar a cada lenguaje, nuevas reglas, conceptos y elaboraciones, con la idea subyacente de hacerlo más poderoso. Alan Kay comprendió que esta tendencia era incorrecta y tomó el camino inverso. En lugar de agregar nuevos conceptos, eliminó todo lo innecesario del lenguaje dejando solamente lo esencial: *objeto* y *mensaje*.[3] En Smalltalk todo es un objeto y el procesamiento ocurre únicamente mediante el envío de mensajes. Esto permite que el sistema permanezca pequeño, sencillo y homogéneo.

En Smalltalk, los objetos son entidades activas siempre dispuestas a recibir mensajes —cualquier mensaje—. El único error posible, surge cuando un objeto no puede entender uno de estos mensajes; pero en ese momento otro objeto se encarga de informar el error. Un objeto no entiende un mensaje cuando no posee un método apropiado para responderlo (es decir, un método con el mismo nombre). Por ejemplo, al evaluar la expresión 'abc' cosine, la máquina virtual de Smalltalk detecta que el objeto 'abc' (instancia de la clase String) no posee un método con dicho nombre y por lo tanto no puede responder al mensaje; en ese momento se crea una instancia de la clase MessageNotUnderstood responsable de señalar dicho error.[4] De esta manera, los errores nunca pasan inadvertidos para manifestarse tiempo después causando comportamientos inexplicables. Por el contrario, los errores son inmediatamente capturados e informados. Esta definición —enunciada por Luca Cardelli— categoriza a Smalltalk como un lenguaje *seguro* [Car97].

En Smalltalk, la programación consiste únicamente en crear, inspeccionar o modificar objetos, enviando los mensajes apropiados. Incluso al escribir un método, se crea un objeto de la clase CompiledMethod. La programación en Smalltalk es algo sumamente especial y divertido. Compartimos plenamente la opinión que Dan Ingalls expresó hace 23 años [Ing78]:

> "... además de esto, debo agregar que programar en Smalltalk es divertido. Por un lado, la acción de ensamblar expresiones en sentencias y luego en métodos, no es muy diferente a la programación convencional. Por otro lado, la experiencia es totalmente diferente, ya que los objetos que pueblan y atraviesan el código

[3] La idea de simplificar al máximo el lenguaje tuvo sus raíces en Lisp [Kay93].
[4] 'abc' cosine corresponde al coseno de 'abc' y obviamente carece de sentido.

> son entidades activas, y la sensación al escribir las expresiones, es más parecida a organizar animales entrenados que a mover cajas de un lado a otro".

Los investigadores del proyecto Smalltalk vieron que la computadora podía ser usada como un nuevo medio para la comunicación del pensamiento. Estaban convencidos que la máquina debía ser adaptada a las necesidades del ser humano y no a la inversa. Una idea novedosa para la época pues las computadoras valían fortunas, ocupaban habitaciones enteras y eran consideradas por la mayoría de la gente como poderosas herramientas de cálculo. Pocos preveían que el mismo poder de cómputo de esos inmensos aparatos podría ser llevado 20 años más tarde en el bolsillo de una camisa.

Smalltalk no fue el primer proyecto en variar el enfoque sobre la computadora. Desde 1962, el ingeniero Douglas Engelbart investigaba la forma de aumentar y modificar el pensamiento humano mediante la interacción con la máquina. Su proyecto de "Augmentation of Human Intellect" fue llevado a cabo en universidad de Stanford y se hizo ampliamente conocido en 1968. Ese año presentó el primer sistema interactivo (NLS) en una conferencia de ingenieros. Allí mostró la operación de un sistema de hiper-texto y video-conferencia, mientras manejaba una pequeña caja con ruedas llamada *mouse*.

Muchos de los integrantes del equipo de investigación de Engelbart fueron contratados por Xerox PARC para trabajar en el proyecto POLOS (PARC On-Line Office System). Existían dos diferencias fundamentales entre este proyecto y el liderado por Alan Kay. La primera es que seguían considerando las computadoras como bienes tan excesivamente caros que debían ser compartidos —todo lo contrario a la idea de la computación personal—. La segunda diferencia involucraba a los niños. Al sostener que la computadora servía para aumentar el intelecto humano, elaboraron una intrincada y compleja interface de usuario que requería unos seis meses de entrenamiento. Por el contrario, Alan Kay sostenía que la interface debía ser amigable e intuitiva para cualquier persona.

El proyecto Smalltalk encontró que la noción de mensaje constituía una poderosa forma para describir el procesamiento (la noción de objeto, en cambio, fue tomada de Simula). Otros aportes importantes ocurrieron en el campo de las interfaces de usuario. Hasta ese momento las interfaces estaban pensadas por y para técnicos ya que las personas comunes no tenían acceso a las computadoras. Pero algún día esto cambiaría y las computadoras serían populares. Con esta visión de futuro se idearon las ventanas solapables y los menues desplegables, para permitir que cualquier persona pudiera realizar varias tareas al mismo tiempo —tal como sucede en los escritorios donde se apilan los papeles—. En Smalltalk se implementó por primera vez la

metáfora de escritorio como interface de usuario.

A pesar de todos los avances que se realizaron en el proyecto Smalltalk, la empresa Xerox dejó de financiarlo en 1981. Diversas son las razones que motivaron esta desacertada decisión.[5] Afortunadamente, mucha gente pudo comprender que Smalltalk brindaba un ambiente de inigualable productividad para la construcción de software y decidió adoptarlo. Lamentablemente en el mundo de los negocios el espíritu innovador de Smalltalk se fue diluyendo.

El Proyecto Squeak

El espíritu de Smalltalk volvió a encarnarse —16 años más tarde— en el proyecto Squeak. A fines de 1995, Alan Kay y otros integrantes del equipo original de Xerox, fueron reunidos por Apple para formar un nuevo grupo de investigación. Tenían como objetivo experimentar en software educativo, nuevas interfaces de usuario y darle una oportunidad a la construcción del *Dynabook* (la visión de Alan Kay desde 1968).

El grupo necesitaba un ambiente de desarrollo para construir software que pudiera ser usado —y hasta programado— por niños y personas sin conocimientos técnicos. El software debería ser accesible en medios masivos (como organizadores personales, consolas de juego, Internet, etc.) donde es esencial que los sistemas sean compactos debido a la velocidad de transmisión, la escasa capacidad de almacenamiento, la diversidad del hardware y la variedad de sistemas operativos. El sistema que pensaron como ideal debería estar formado por un núcleo pequeño y portable, que tuviera al mismo tiempo un diseño simple y uniforme; esto les permitiría adaptarlo rápidamente a los nuevos medios de distribución.

En su búsqueda, consideraron utilizar el lenguaje Java, pero fue descartado por su estado de inmadurez y falta de mecanismos para lograr los cambios dinámicos que preveían. Smalltalk cumplía con los requisitos técnicos, pero ninguna de las implementaciones disponibles les permitía tener el control deseado sobre gráficos, sonido y máquina virtual; tampoco tendrían libertad para distribuir gratuitamente el resultado de su trabajo por Internet. Sintieron que no eran los únicos con la necesidad de un ambiente de programación efectivo, abierto, portable y maleable; había también una comunidad de investigadores sedienta de un ambiente de estas características. El camino era construir un nuevo Smalltalk diseñado desde el principio con todos estos objetivos en mente para compartirlo con todo aquel que quisiera.

Semejante emprendimiento no es algo sencillo de lograr pero ellos hicieron

[5] Hiltzik relata la increíble historia de Xerox en [Hil99]

fácil lo que es difícil. El equipo formado por integrantes de primer nivel —Alan Kay, Dan Ingalls, Ted Kaehler, Scott Wallace y John Maloney— logró construir en tiempo record la primera versión. Tomaron como base la imagen de Smalltalk-80 para Apple y en tan sólo dos meses obtuvieron un Smalltalk fácilmente portable a diversas plataformas. Tuvieron la brillante idea de escribir la máquina virtual en un subconjunto de Smalltalk que pudiera ser traducido a C. Esto les permitió pensar, programar y simular la máquina virtual en el propio Smalltalk, al mismo tiempo que otros miembros del equipo se dedicaban a programar el traductor. Sin salir nunca de la comodidad y facilidades que brinda Smalltalk, siguieron trabajando en otros aspectos del sistema: punteros directos a memoria, nuevas versiones del garbage-collector (más sencillas y eficientes), extensiones a BitBlt para manejar color, y soporte para la síntesis en tiempo real de música y sonido. Diez meses después de comenzar el proyecto, lanzaron en Internet la primera versión de Squeak para Apple Macintosh. Cinco semanas más tarde estaba portado, en forma independiente, a Unix, Windows 95 y NT. En 1997, el proyecto Squeak es presentado en la conferencia de objetos más importante del mundo [IKM+97].

Squeak atrajo desde el primer momento a una interesante comunidad de programadores, académicos y personas independientes de las Ciencias de la Computación. Llamamos *independientes* a aquellas personas más interesadas en alguna materia particular —como la música, la matemática, los gráficos o la biología— que en un determinado lenguaje o sistema operativo. Publicado con todo el código fuente, completamente gratuito y disponible por Internet, Squeak es algo difícil de resistir. Su naturaleza abierta permite que todo pueda ser visto, comprendido, modificado o extendido —al punto de incluir la capacidad de regenerar el código fuente de la propia máquina virtual—. Squeak rápidamente se convirtió en uno de los proyectos colaborativos más activos y atractivos del mundo.

La comunidad que se ha formado en torno al proyecto Squeak difícilmente tenga parangón. Su equipo central (SqC) está formado por investigadores altamente competentes que atrajeron a una multitud de entusiastas. Podríamos caracterizar a esta comunidad bajo sus dos principales orientaciones: *investigación* y *desarrollo*. Aquellos interesados en la investigación, ven en Squeak un vehículo donde pueden realizar sus experimentos. Los interesados en el desarrollo, ven un Smalltalk gratis que corre en las principales plataformas. Ambos grupos tienen ciertas características en común: disfrutan el arte de programar, quieren mejorar una herramienta gratuita y están empeñados en lograr los mejores resultados posibles.

3. Programación Orientada a Objetos

En este capítulo introducimos las nociones fundamentales de la programación orientada a objetos. Lo hacemos desde una perspectiva muy particular, la perspectiva de Smalltalk. Hay una serie de ideas detrás de este paradigma que es necesario tener en cuenta. Estas ideas las explicamos en el contexto histórico que se originaron. Luego explicamos los cinco conceptos básicos: objeto, mensaje, clase, método e instancia.

Reseña Histórica

El primer lenguaje en incorporar formalmente las nociones de objeto y de clase fue Simula, especialmente en su versión de 1967. Sin embargo, el germen de las ideas que dieron origen a lo que más tarde se llamó *Programación Orientada a Objetos* había cobrado vida en 1962, cuando Ivan Sutherland concibió su notable Sketchpad, el primer editor de gráficos orientado a objetos de la historia. En 1966, Alan Kay uno de los integrantes del equipo ARPA Net de la Universidad de Utah, recibe el encargo de poner en funcionamiento una nueva implementación de Algol a partir de unas cintas llegadas de Noruega. Los comentarios en escandinavo insertados en el interminable listado que Kay estudiaba sobre el piso, poco ayudaban a desentrañar aquel extraño lenguaje cuyo pretendido parentesco con Algol se desdibujaba a cada instante. De pronto, en medio del desconcierto, Kay pudo distinguir que el enfoque de los escandinavos coincidía con el de Sutherland. Las mismas ideas que Sketchpad había aplicado al diseño gráfico aparecían ahora formando parte de un lenguaje de programación procedural de propósito general. No se trataba de Algol, sino de Simula.

Mientras Sketchpad permitía definir un *master drawing* del cual se podían derivar *instance drawings*, Simula permitía definir *activities* de las cuales se podían crear un número de *processes*. En Simula cada proceso era un objeto y cada simulación tenía su propia clase. Aunque la terminología era diferente de la actual, cada objeto, es decir, cada proceso, tenía sus propios datos y su propio comportamiento y no podía intervenir en los datos del los demás objetos sin el consentimiento de estos. Así, Simula implementaba lo que más tarde se daría en llamar encapsulamiento.

Además de eso, cada objeto Simula podía actuar al mismo tiempo que otros objetos, lo cual era de vital importancia, ya que un sistema capaz de realizar simulaciones debe proveer la ilusión de la ejecución múltiple.

Como Kay también había estudiado Biología, le resultó natural hacer un paralelismo entre los objetos y las células vivas. Vio que cada objeto se podía pensar como una entidad indivisible e independiente, y con capacidad para intercambiar estímulos e información con otros objetos, del mismo modo que lo hacen las células a través de su membrana externa. Y puesto que los seres vivos están formados por células individuales, era concebible pensar en sistemas informáticos enteramente construidos a partir de objetos. Lo que Kay buscaba era alcanzar la construcción de sistemas que fueran robustos y veía que eso se podía lograr imitando a la naturaleza, en donde un organismo sigue viviendo aún cuando unas cuantas células puedan fallar o morir.

Del lado de su formación matemática Kay encontraba una gran afinidad con la programación funcional, en esos años encarnada en el lenguaje Lisp. Kay conocía muy bien ese lenguaje y deseaba crear otro que fuera *"el Lisp de la programación orientada a objetos"*. La visión de Kay iba, sin embargo, más allá de los lenguajes de programación. Comprendía la creación de un sistema de Computación Personal. Pero en una época en donde el costo del hardware convertía en ridícula semejante pretensión, todos los esfuerzos estaban destinados a la creación de sistemas de tiempo compartido. Estos sistemas eran la antítesis exacta de la computación personal. Sin embargo, Kay estaba convencido de que el destino de la computación debía ser el de prestar asistencia al espíritu creativo de cada individuo.

Con estas ideas en mente, Kay dedicó sus esfuerzos a la construcción del sistema FLEX. Esta máquina era una computadora personal completamente programable y basada en objetos. Aunque FLEX no era la primera computadora personal, en 1969 encarnaba un ideal que iba decididamente en contra de la corriente. En esta máquina confluían las ideas de Sketchpad y Simula y estaba proyectado incluir una tableta para dibujo a mano alzada. En el horizonte de Kay también estaban las ideas que Douglas Engelbart había desarrollado en su sistema NLS. El fundamento de ese sistema era una teoría del propio Engelbart que proponía el uso de computadoras para aumentar la inteligencia humana. Aunque Engelbart había publicado su teoría en 1962, fue en 1968 cuando deslumbró a la audiencia en una conferencia memorable donde dio a conocer por primera vez sus invenciones: el mouse, paneles de display múltiples e integración con video captado en vivo.

Así y todo, el sistema FLEX no llegaba a satisfacer por completo las expectativas puestas en él. En parte por sus limitaciones arquitectónicas (sólo contaba con 16 Kbytes de RAM), pero también porque su uso había resultado mucho más complicado de lo conveniente. Fue entonces cuando Kay conoció a Seymour Papert, el creador del original sistema Logo. La impre-

sión que el proyecto de Papert produjo en Kay fue decisiva. Comprendió que un sistema que fuera incapaz de ser usado por chicos estaba destinado al fracaso. Su visión de la computación personal dio un paso adelante. El lo explicaba argumentando que si los chicos de habla inglesa aprendían a hablar en la lengua de Shakespeare, la computación personal debía proporcionar un mismo lenguaje a personas de todas la edades. Sólo un sistema que pudiera ser usado tanto por un niño de 5 años como por un adulto iba a servir para usar a las computadoras como espacios donde volcar ideas.

La pasión de Kay por la música y la literatura contribuyó a concebir su ideal de computadora personal como si se tratase de un libro dinámico, el Dynabook. Un sistema de computación personal para chicos de todas las edades, que pudiera reunir bajo una metáfora uniforme textos, gráficos y dibujos, música y sonidos y que fuese simple y poderoso, maleable y totalmente programable. Sólo un sistema que reuniera estas características podría servir al desenvolvimiento del espíritu creativo que hay en las personas.

En 1970, durante una conversación informal con Ted Kaheler y Daniel Ingalls, Kay lanzó la temeraria afirmación de que el lenguaje de programación más poderoso se podía especificar en una sola página. Kaheler e Ingalls lo desafiaron a demostrar su aseveración y Kay, pasó las siguientes semanas, escribiendo la primera versión de su sistema Smalltalk. Al poco tiempo Ingalls, entusiasmado por el nuevo lenguaje, escribió el prototipo en Basic. Era tremendamente lento, pero funcionaba.

El sistema Smalltalk fue desarrollado a lo largo de la década del 70 en los laboratorios de Xerox en Palo Alto Research Center (PARC). El principal implementador del sistema fue Dan Ingalls. A Ted Kaheler se debe la construcción del sistema de música de Smalltalk 72, la tortuga de Smalltalk y la estructura de la memoria de objetos. Durante esos años Ingalls tuvo que resolver incontables problemas, entre ellos el de presentación de texto en áreas no rectangulares, el cual encaró de manera brillante con su inédito BitBlt. A Adele Goldberg se debe la completa serie de experiencias con escolares y estudiantes de niveles medios. El grupo, conocido bajo el nombre de Learning Research Group, culminó su proyecto con la publicación de una serie de cuatro libros que exponían los diferentes aspectos del Sistema Smalltalk. Entre ellos, *Smalltalk–80: The language and its implementation* de Goldberg y Robson, también llamado el *Blue Book*, constituye una obra fundamental en la literatura de la programación orientada a objetos [GR83].

Para poner a prueba la portabilidad del sistema, la especificación de Smalltalk–80 fue enviada a unas cuantas compañías. En esa época no era fácil encontrar computadoras dotadas de display gráfico y suficiente memoria (300 Kbytes de RAM y 1 Mbyte de disco rígido). Las empresas elegidas fueron Hewlett-Packard, DEC, Apple, TI y Tektronix. Cada laboratorio dedicó un equipo a portar Smalltalk–80 partiendo de la especificación escrita de su

Máquina Virtual. Todas estas experiencias fueron recogidas en otro libro de la serie *Smalltalk–80: Bits of history, words of advice*, también conocido como el *Green Book*.

Los sistemas Smalltalk fueron los primeros en tener display gráfico, ventanas superpuestas, menús, iconos, sonido y mouse. Los sistemas operativos Microsoft Windows, X-Windows en Unix y Macintosh tienen, todos ellos, su origen en Smalltalk.

El sistema Smalltalk ha demostrado de que todo lo que hay en computación puede hacerse utilizando únicamente las nociones de objeto y mensaje que toda otra construcción provista por un lenguaje de programación es superflua y sólo agrega complejidad. La influencia de Smalltalk ha sido profunda y trascendente. En la actualidad los principales lenguajes de programación incorporan nociones de orientación a objetos. Sin embargo, la mayoría de ellos no ha podido alcanzar la pureza y la potencia de Smalltalk puesto que han mantenido elementos que no son objetos, sino tipos básicos y estructuras.

En 1983 el Learning Research Group se disolvió y Xerox formó la compañía ParcPlace, dedicada a vender el primer dialecto comercial de Smalltalk, años más tarde devenido en VisualWorks. Más tarde otras compañías como DigiTalk, ObjectShare, Object-Arts, IBM y Cincom, produjeron dialectos de Smalltalk de uso intensivo en todos los campos de la programación. Entre ellas, GemStone, ocupa un lugar importante por haber realizado la única versión comercial de un Smalltalk persistente, equiparable a los motores de bases de datos relacionales más difundidos en el mercado.

En los años siguientes Smalltalk se difundió, se desarrolló, tuvo una enorme influencia en la manera de entender la computación personal, pero prcticamente a partir de la culminación del proyecto de PARC no produjo ninguna novedad digna de su ilustre pasado. Smalltalk dejó de evolucionar cuando se hizo comercial.

En diciembre de 1995, Alan Kay, Dan Ingalls y Ted Kaheler volvieron a reunirse contratados por Apple. Los años habían pasado, pero ellos seguían pensando que el sueño del Dynabook aún no se había alcanzado. El movimiento Open Source y la Internet habían enriquecido el campo de acción de la computación personal, le habían dado mayores posibilidades. Y si bien era cierto que los fabricantes de hardware seguían desconociendo la existencia de la programación orientada a objetos, estaban produciendo computadoras cada vez más veloces; el software podía reparar los errores de diseño del hardware.

Decidido a continuar el proyecto iniciado veintiséis años antes, el grupo se abocó a la tarea de construir un nuevo sistema. La premisa fundamental era que el sistema debía poder construirse enteramente en sí mismo. Partiendo

de una implementación de Smalltalk–80 que era propiedad de Apple, el equipo copió la especificación de la Máquina Virtual escrita en Smalltalk en el Blue Book de Goldberg y Robson y la adaptó a las nuevas necesidades. Después escribió un traductor de un subconjunto del lenguaje Smalltalk a C y compiló la primera máquina virtual del nuevo sistema. Así nació Squeak.

El haber decidido que el Smalltalk de Apple hiciera de Máquina Virtual de Squeak permitió que el nuevo sistema se pudiera debuggear completamente en Smalltalk. Y lo que es más importante, Squeak heredó esa propiedad. En Squeak es posible debuggear todo, incluso el Debugger. También los métodos primitivos se pueden escribir en Squeak y luego generar el código C que, al compilarlo, permitirá modificar la máquina virtual. La metáfora uniforme de objetos y mensajes y su capacidad de auto-generación, hicieron que Squeak fuera rápidamente portado a las principales plataformas. En pocos meses se convirtió en el sistema que corría en la mayor cantidad de plataformas del mundo. Incluso existen versiones de Squeak que funcionan sin sistema operativo [RNCM01].

El carácter abierto de Squeak junto con su licencia fuertemente liberal, creó las condiciones para que alrededor del proyecto se formara una comunidad internacional que comenzó a aportar ideas, código y hasta subsistemas completos. El alto grado de sinergía entre el equipo central de Squeak y los usuarios es otra muestra más del potencial del movimiento Open Source y una fuente inagotable de motivación para quienes participamos activamente en él.

Conceptos Básicos

El vocabulario de Smalltalk consta solamente de cinco palabras:

- objeto
- mensaje
- método
- clase
- instancia

Objetos

En Smalltalk todo es un objeto. Esta frase la vamos a repetir muchas veces. Parece inocente, pero no lo es.

En los lenguajes convencionales, hay tipos básicos de datos como caracteres, números enteros, números de punto flotante, booleanos, registros o arrays.

A partir de los tipos básicos el programador puede crear tipos compuestos. Los tipos compuestos son estructuras de datos construidas a partir de tipos básicos o de estructuras mas sencillas. En los últimos años todos los lenguajes han incluído alguna forma de la noción de *objeto*. Sin embargo esos *objetos* solamente son estructuras de datos más sofisticadas que las anteriores y no objetos de primera clase, como en el caso de Smalltalk.

En Smalltalk no existe distinción entre tipos básicos y tipos compuestos. Siempre se trata de objetos. A primera vista esto podría parecer sólo una cuestión de nombres. Sin embargo, esa diferencia es profunda y otorga a Smalltalk poder y belleza.

En Smalltalk todo es un objeto. Un número es un objeto, una letra es un objeto y un array es un objeto. Pero también es un objeto el panel de texto sobre el que escribo, el font que elegí para dibujarlo, el puntero del mouse que me permite seleccionar texto. Incluso la selección misma es un objeto.

¿Qué diferencia a un objeto Smalltalk de una estructura de datos? Las estructuras de datos son construcciones que capturan la forma en que la información está organizada. Los objetos también capturan esa organización, que podríamos llamar física. La diferencia es que los objetos, además, capturan el comportamiento que queremos darle a esa organización.

Carl Sagan decía que los humanos somos *"el cosmos que ha tomado conciencia de sí mismo"*. En el plano virtual lo mismo puede decirse de los objetos. Son estructuras de datos que han cobrado conciencia de sus significado semántico. Por ejemplo, en cualquier lenguaje podríamos crear la estructura de datos Point. Físicamente una estructura así tendría dos partes constitutivas (o campos): una para x y la otra para y. En pseudo-código tendríamos algo así:

```
Point {
    x: Integer;
    y: Integer;
}
```

Esta estructura le serviría al programador para almacenar puntos en la memoria y para tener variables que representen puntos. Sin embargo, en la programación estructurada, lo que el programador vaya a hacer con un Point estará desparramado a lo largo de todas las rutinas (funciones o procedimientos) que accedan a points. Al mismo tiempo, los points no serán conscientes de lo que esas rutinas puedan hacer con ellos. Más aún, cuándo una rutina lea o cambie, digamos, el campo x, el point no se dará por enterado. No podrá intervenir en ese acceso, no podrá prohibirlo ni tomar nota de él.

En Smalltalk, un punto como 2@3 tiene, además de sus coordenadas, una clara conciencia de lo que representa. El punto se entera cuando sus coor-

denadas son accedidas y, de hecho, podría prohibir que le fueran cambiadas. La conciencia de este objeto hace que el mismo esté convencido de ser un punto. Por eso se comportará como un punto y ofrecerá servicios para realizar operaciones aritméticas, de cambio de coordenadas, de transformaciones geométricas tales como una rotación, etc. El comportamiento de un Point no estará desparramado en todo el sistema, sino que será una característica más del objeto mismo.

Un objeto, por el sólo hecho de ser objeto, ya tiene un comportamiento mínimo que le permite mostrarse en una forma significativa. Los objetos saben mostrarse en forma de texto. Al hacer esto, no se muestran completamente, sino parcialmente. Aún así, esta forma de mostrarse que tienen los objetos sirve para darnos una idea rápida de su estado.

Los objetos también se dejan inspeccionar. Si queremos internarnos en la estructura interna de un objeto, podemos realizar una inspección de las partes que lo componen. Si inspeccionamos el objeto 2 / 3 veremos el interior de la fracción. Este objeto consta de dos partes: numerador y denominador.

Mensajes

Un mensaje es un estímulo dirigido a un objeto. El objeto que recibe el mensaje se llama receptor. Para enviar un mensaje lo que hacemos es escribirlo como una expresión Smalltalk y luego evaluarla. Por ejemplo, la siguiente expresión representa el mensaje #reverse a punto de ser enviado al objeto 'Smalltalk'.

```
'Smalltalk' reverse
```

El texto que nombra al mensaje se llama selector. En el ejemplo de arriba el selector es #reverse. Los mensajes más simples constan de un receptor y un selector. Estos mensajes se llaman unarios porque no llevan argumentos.

Los selectores son símbolos, es decir, palabras de una clase especial que representan símbolos. En Smalltalk cuando queremos que una palabra se la interprete como un símbolo, la precedemos con el cardinal '#'. Por ello, escribimos el selector del mensaje anterior como #reverse.

Un mensaje también puede tener argumentos como en la siguiente expresión.

```
'Smalltalk' at: 6
```

Aquí, 'Smalltalk' es el receptor, #at: es el selector y 6 es el argumento. Cuando enviamos el mensaje, el receptor nos responde un resultado.

Antes digimos que en Smalltalk todo es un objeto. Pero ahora estamos diciendo que, además de objetos, también hay mensajes. ¿No es esto una

contradicción? ¿No deberíamos haber dicho que en Smalltalk hay objetos y mensajes?

La respuesta es no, no hay ninguna contradicción. Ocurre que en Smalltalk los mensajes también son objetos. Así como los números, las letras, los arrays y los puntos son objetos, los mensajes también son objetos. Es por eso que no hay ninguna contradicción cuando afirmamos que todo es un objeto y que en Smalltalk hay mensajes.

Métodos

¿Cómo hace un objeto para responder a un mensaje? Muy simple, toma el selector del mensaje y con él busca en un diccionario el significado de ese mensaje. Así, cuando el objeto 'Smalltalk' recibe el mensaje cuyo selector es #reverse, el objeto busca en el diccionario el significado de #reverse. La explicación que ahí encuentra se denomina método. El diccionario se llama diccionario de métodos o MethodDictionary. Un método contiene las intrucciones que le indican al objeto cómo proceder frente a un determinado mensaje.

Observe que existe una distinción entre mensaje, selector y método. Un mensaje es una expresión *ejecutable*. Un selector es un símbolo con el cual se da nombre al mensaje. Un método indica qué debe hacer el receptor para responder al mensaje. Los métodos están en el diccionario de métodos del receptor. El receptor recurre a ese diccionario para buscar el significado de cada mensaje que recibe. La búsqueda la efectúa entrando al diccionario por el selector del mensaje.

Polimorfismo: En los lenguajes procedurales, cada función o procedimiento debe tener un nombre distinto. En ellos, la vinculación entre la llamada y la función que lo implementa es estática; es decir, que se resuelve en tiempo de compilación. En Smalltalk, pueden existir distintos métodos con el mismo nombre (su selector). La vinculación entre el envío de mensaje y el método que lo implementa es dinámica; es decir, que se resuelve en tiempo de ejecución. De esta forma podemos tener *expresiones polimórficas*. Por ejemplo, evaluando la expresión anObject draw, podríamos dibujar un círculo, un rectángulo o una ventana, dependiendo del valor de la variable anObject. El polimorfismo, en Smalltalk, es consecuencia de la vinculación dinámica (o tardía). Según Alan Kay, la vinculación tardía es una idea recurrente que ha permitido el avance de la Ciencia de la Computación en diferentes momentos históricos [Guz00, p.3].

Por último, con los métodos vale la misma aclaración que hicimos con los mensajes: en Smalltalk los métodos son objetos.

Clases

En Smalltalk los objetos están clasificados. El tipo de un objeto se denomina clase. Así el objeto 3 está clasificado como SmallInteger, mientras que el objeto 'Smalltalk' es un String.

En Smalltalk el programador no declara tipos. Al contrario, son los objetos los que saben de qué clase son. El mensaje unario #class, cuando se lo enviamos a un objeto, responde la clase de ese objeto.

Todos los objetos son de alguna clase. Las clases no representan conjuntos de objetos, sino que tienen otras responsabilidades más interesantes, incluyendo las siguientes:

- Las clases son las encargadas de guardar el diccionario de métodos que se usa en el envío de un mensaje.
- Las clases conocen la estructura interna de los objetos (Por ejemplo la clase Point sabe que sus objetos tienen dos coordenadas, x e y).
- Las clases tienen la capacidad de crear nuevos objetos, son *fábricas* de objetos.

La clase de un objeto, también es un objeto. El hecho de que en Smalltalk las clases también sean objetos es, por un lado, algo trivial puesto que en Smalltalk todo es un objeto y por otro, algo profundo que merece ser destacado especialmente. Las consecuencias de tener a las clases en el mismo plano conceptual de las entidades que clasifican son determinantes. En primer lugar, Smalltalk es un sistema reflexivo, es decir, definido y construído en sí mismo. Esta propiedad hace que Smalltalk se modifique a sí mismo mientras corre. Es decir, mientras el sistema está en ejecución podemos crear nuevas clases, nuevos objetos y nuevos mensajes. Todo lo que hagamos será el producto de haberle enviado el mensaje apropiado al objeto apropiado. Como resultado el sistema cambiará, adaptándose a las nuevas condiciones.

Todo en Smalltalk se implementa como un objeto. Los mensajes son objetos, los métodos son objetos, y existen objetos (las clases) que sirven para clasificar, para crear nuevos objetos, para determinar la estructura interna de los mismos y para mantener el conocimiento de cómo se resuelve el envío de mensajes. Por otro lado, los objetos nacen con una importante caudal de conocimiento que los hace aptos para vivir en el ambiente y para relacionarse con otros objetos. Los objetos tienen conciencia de sí mismos, tienen conciencia de su propia identidad, saben mostrarse de forma significativa como un texto, saben cómo hacer para dejarse inspeccionar, pueden recibir mensajes y devolver una respuesta, saben enviar mensajes, saben acceder a su estructura interna.

Usando todo ese conocimiento, el programador puede conferirle a los objetos un comportamiento especializado que responda a lo que los objetos representan como modelos de la realidad o la fantasía.

Dijimos que todo lo que un objeto puede hacer, lo hará al recibir los diferentes mensajes que forman su protocolo. Y como las clases son objetos, también cumplirán sus funciones específicas como resultado del envío de mensajes. En los lenguajes convencionales, los tipos son declaraciones destinadas al compilador. Están fuera del modelo que el programador está construyendo. En Smalltalk, en cambio, las clases están modeladas como objetos cuyo funcionamiento responde a las mismas leyes que gobiernan a todos lo objetos, es decir, a la recepción y envío de mensajes.

Herencia: Normalmente se dice que la herencia es una característica fundamental de los lenguajes orientados a objetos, pero eso no es cierto. En Smalltalk, la herencia es un mecanismo para compartir código, para *heredar* implementación. Toda clase es subclase de alguna otra y así forman una estructura de árbol con Object en la raíz. En esa clase se definen los métodos básicos que deben responder todos los objetos. En las subclases se pueden redefinir cualquiera de estos métodos y agregar otros nuevos.

Instancias

Una traducción apropiada del término inglés *instance* sería *ejemplar*. Lamentablemente el término *instancia* se ha apoderado de esa noción. Decimos que un objeto obj es una instancia de la clase C, simplemente para indicar que C es la clase de obj.

Una instancia es, entonces, un ejemplar de una clase. Es decir, al referirnos a un objeto como una *instancia*, de alguna manera estamos poniendo el énfasis en la relación que éste tiene con su clase.

Esta noción puede parecer superflua. No lo es porque como en Smalltalk las clases también son objetos, hay veces que es necesario decir si uno está pensando en una clase en tanto clase, o en tanto instancia.

Dado que todo objeto es de alguna clase, y dado que las clases son objetos, tenemos que las clases también son instancias de sus propias clases.

La clase de una clase se llama metaclase. Las metaclases, a su vez, son instancias de la clase Metaclass. Es interesante ver cómo Smalltalk es coherente con las nociones que define. Esa coherencia asegura que todas las nociones se puedan definir y construir en términos de la noción de objeto.

4. Inferencia de Tipos

Most ideas come from previous ideas

— Alan Kay

En este capítulo presentamos nuestra distinción entre tipos concretos y tipos abstractos. Damos una definición de cada uno de ellos y algunos ejemplos. Explicamos cuáles son las dificultades para extraer la información de tipos concretos y enumeramos algunas aplicaciones. Presentamos un ejemplo para ilustrar la interpretación abstracta en funcionamiento y finalizamos con una revisión de trabajos relacionados.

El interés principal de este trabajo es obtener la información de tipos concretos presentes en un programa Smalltalk. Queremos *inferir* los tipos concretos porque las declaraciones estáticas generan código especializado y poco reusable. Nuestro objetivo es averiguar el tipo concreto de cada variable y expresión que participa en la ejecución de un programa.

4.1 Información de Tipos

Los sistemas de tipos se idearon para brindar información auxiliar sobre un programa. Esta información es utilizada esencialmente para:

- Generar código más eficiente
- Facilitar la comprensión de un programa
- Realizar chequeos estáticos

Si observamos cada uno de estos objetivos, veremos que la información necesaria para lograrlos es muy diferente. Generar código más eficiente requiere información radicalmente distinta a la necesaria para facilitar la comprensión de un programa. Esto se debe a que el código fuente es leído por dos entidades de distinta naturaleza: *el compilador* y *el ser humano*. Cada uno de ellos tiene objetivos distintos, y por lo tanto, distintas necesidades de información. El compilador necesita información lo más precisa y de bajo

nivel posible, pues su objetivo es realizar optimizaciones de código (ie. *tipos concretos*). El ser humano quiere información lo más abstracta y genérica posible, pues está interesado en documentar y facilitar la comprensión de programas a otras personas (ie. *tipos abstractos*).

4.2 Tipos Concretos

Los tipos concretos son útiles para describir la implementación y estructura interna de los objetos. Como primera aproximación:

> *El tipo concreto de un objeto es su clase exacta.*

Enfatizamos *exacta* para no confundir con la noción de *clase y subclases* utilizada en C^{++} y otros lenguajes. Los tipos concretos indican la clase exacta y por esta razón, no pueden existir tipos concretos de clases sin instancias (Magnitude, Number, Boolean, etc.). En la siguiente figura mostramos ejemplos de tipo concreto, según esta definición (fig. 4.1).

1	→	⟨SmallInteger⟩
1.0	→	⟨Float⟩
'hola'	→	⟨String⟩
1@1	→	⟨Point⟩

Figura 4.1: Objetos y su tipo concreto

La definición de tipo concreto, recién dada, se aplica a los objetos y no a las expresiones. No tiene sentido hablar acerca del tipo concreto de una expresión en forma aislada. Por ejemplo, el tipo concreto de la variable x depende de lo que tenga asignado, y esto a su vez, depende de lo previamente ejecutado.

Los tipos concretos de las expresiones dependen de un contexto de ejecución que llamamos *programa* o *expresión inicial*. Por ejemplo, si el programa es x ← 1, el tipo concreto de x es ⟨SmallInteger⟩ y si el programa es x ← 'hola', el tipo concreto de x es ⟨String⟩.

El código fuente está compuesto por diversas expresiones (expresión de asignación, mensaje, retorno, variable, literal, etc). En tiempo de ejecución, las expresiones se evalúan y retornan objetos de distintas clases. Dependiendo de la evaluación, una expresión puede retornar diferentes tipos concretos. Esto quiere decir, que una expresión tiene un *conjunto* de tipos concretos.

Por abuso de notación, nos referimos a este conjunto como el tipo concreto de la expresión. Por ejemplo, en la siguiente figura mostramos algunas expresiones y su tipo concreto (fig. 4.2).

x = y	→	{⟨True⟩ ⟨False⟩}
x isNil ifTrue:[1] ifFalse: [1.0]	→	{⟨SmallInteger⟩ ⟨Float⟩}
x	→	depende de la expresión inicial o *programa*

Figura 4.2: Expresiones y su tipo concreto

¿Cómo sabemos que el tipo de la expresión x = y es {⟨True⟩ ⟨False⟩}? En realidad no lo sabemos: lo inferimos. Además de la expresión inicial, la inferencia depende también del conjunto de objetos que componen la imagen Smalltalk. Podría suceder que un programador (con fuertes reminiscencias en C) redefina la igualdad en alguna clase para que retorne 0 o 1. Si x fuera instancia de esa clase, el tipo de x = y sería {⟨SmallInteger⟩}.

Es importante definir un criterio de precisión para evaluar el comportamiento de la inferencia. El resultado es más preciso cuanto más pequeño es el conjunto de tipos concretos. El conjunto vacío es muy preciso pero obviamente es incorrecto; el conjunto con todos los tipos posibles es correcto pero obviamente es impreciso (e inútil). Nuestro objetivo es inferir el subconjunto más chico de todos los correctos.

4.2.1 Envíos de Mensajes

En Smalltalk, no se invocan funciones sino que se envían mensajes; incluso las estructuras de control se implementan con mensajes enviados a bloques de código (que también son objetos). El compilador no puede vincular el envío de mensaje con un método determinado. La vinculación se resuelve a último momento en base al objeto receptor. Cuando el objeto recibe el mensaje se busca el método en su clase, y una vez encontrado, se lo ejecuta. La vinculación entre mensaje y método es dinámica y tardía (*dynamic-binding*, *late-binding*).

El envío de mensaje puede ser visto como una combinación de un condicional con una llamada a un procedimiento (*if-then + procedure-call*). El condicional es sobre la clase del objeto receptor y el procedimiento a invocar depende de dicha clase. Esto pone un nivel de indirección entre el código que envía un mensaje y el código que lo implementa. Así se facilita la creación de código genérico, flexible y extensible, pero al mismo tiempo, se dificultan las optimizaciones que abarcan varios métodos (*interprocedural optimizations*) [Gro98, §2.4] [Höl95].

En los lenguajes con vinculación estática es posible deducir la función que será invocada con sólo examinar el texto de la llamada. En Smalltalk, por el contrario, examinar el texto de una expresión de mensaje no alcanza para determinar cuál método será invocado; eso depende de la clase del objeto receptor. Esta razón hace que el código Smalltalk sea difícil de entender en forma estática (ie. sin ejecución) y difícil de analizar en forma automática.

Es necesario distinguir entre *expresión de mensaje* y envío de mensaje. La expresión define el envío, pero no lo realiza; el envío se produce cuando se ejecuta la expresión. La diferencia es similar a la existente entre variable y objeto. Variables y expresiones son entidades del código fuente. Objetos y envíos de mensaje son entidades de la ejecución. El nombre de una variable no alcanza para determinar cuál objeto puede apuntar. El texto de una expresión de mensaje no alcanza para determinar cuál método puede invocar.

Frecuentemente se habla de *mensajes polimórficos* pero en realidad, eso no existe. Los mensajes son siempre monomórficos. Es así, porque en el momento de la evaluación cada argumento es un único objeto. Lo que existe en realidad son *expresiones polimórficas*. Por ejemplo, las variables son expresiones intrínsecamente polimórficas porque pueden apuntar a cualquier objeto.

El envío de mensajes es la principal dificultad para inferir tipos concretos en Smalltalk, porque para inferir tipos se deben resolver los envíos de mensaje; pero para resolver los envíos de mensaje, se necesita información de tipos. La inferencia de tipos es difícil porque se debe calcular simultáneamente el flujo de datos (*data-flow*) y el flujo de control (*control-flow*), pues ambos se influyen mutuamente [Age96, pp 17] [HA96, §2.4] [Gro98, §2.1] [BG93, §3.3] [Bar92].

4.2.2 Aplicaciones

Los tipos concretos nos permiten resolver en forma estática, los envíos de mensajes. Es cierto que algunos envíos no se pueden resolver completamente hasta el momento de la ejecución, pero pueden ser acotados considerablemente. La información de tipos concretos nos permite conocer las clases que pueden ocurrir en cada expresión y en consecuencia, los métodos que pueden ser invocados. Esta información es útil para:

Optimizar los envíos de mensajes:

Conociendo los tipos del receptor de un mensaje, se puede realizar la búsqueda del método (*method lookup*) por adelantado e identificar aquellos que se pueden invocar. Si se identifica un solo método, el

envío de mensaje puede reemplazarse por una llamada normal a procedimiento (*procedure call*). Si el código del método no es muy extenso, puede evitarse la llamada dejándolo embebido (*in-lined*).

Garantizar que todo mensaje será entendido:

Si se realiza la búsqueda de método por adelantado, se pueden identificar los casos donde el receptor tenga un tipo cuya clase no sepa responder el mensaje. En esta situación, se puede advertir al programador para que tome las acciones correctivas.

Eliminar chequeo de tipos:

Sabiendo que las primitivas se invocarán con los tipos adecuados, no es necesario que se compruebe el tipo de sus argumentos. Esto mejoraría la eficiencia en tiempo de ejecución pues las primitivas se usan en forma intensiva.

Construir el grafo de llamadas:

Realizando por adelantado la búsqueda de métodos, se puede identificar aquellos métodos que se invocarían desde cada expresión de envío de mensaje. Con esta información puede construirse el *grafo de llamadas* (*call graph*). En este grafo dirigido, los nodos representan a los métodos y los arcos representan a los mensajes. El método a se conecta hacia el método b, cuando a tiene una expresión de envío de mensaje que puede ser respondida por el método b; esto implica que la expresión de mensaje y el método que lo responde, tienen el mismo selector.

Extraer aplicaciones:

Recorriendo el grafo de llamadas se pueden obtener todos los métodos que intervienen en la ejecución de un programa. De esta forma es posible construir una imagen únicamente con dichos métodos.

Traducir código fuente:

Teniendo disponible el grafo de llamadas se facilita la traducción de código fuente a otro idioma. Al traducir un nodo, se traducirían también los arcos y de esta forma los métodos que serían invocados. Esto también puede servir para traducir a una representación más compacta que los símbolos (por ejemplo, a números enteros) y compactar así el tamaño de la imagen.

4.3 Tipos Abstractos

Los tipos abstractos son conocidos también como *interfaces*, *protocolos*, o *tipos de especificación*. Estos tipos se definen en forma independiente a los

objetos y pueden ser vistos como predicados. Estos predicados son conjuntos de mensajes y los objetos que entienden todos esos mensajes, *cumplen* con el tipo abstracto [PS92, §2.1]. En otras palabras:

> *Un tipo abstracto T es un conjunto de mensajes. Un objeto x tiene el tipo abstracto T, si x entiende todos los mensajes de T.*

Según la definición anterior, un objeto puede tener muchos tipos abstractos, pero también pueden existir tipos abstractos que no sean satisfechos por ningún objeto (por ejemplo, el tipo con los mensajes {#foo #bar:}). Los tipos abstractos, generalmente reciben un nombre, pues están destinados al ser humano. Por ejemplo, podríamos llamar Streamable al tipo abstracto que contiene los mensajes {#atEnd #next}, y tanto las instancias de Stream como las de SequenceableCollection (con sus correspondientes subclases) lo satisfacen.

Los tipos abstractos son útiles para comunicar intenciones de diseño e incluso como documentación. Tomemos el ejemplo de una aplicación real para buscar términos dentro de un conjunto de documentos (un *search engine*). En esta aplicación cada documento está formado por una serie de términos y cada término puede ser una palabra, una fecha, un número, una dirección de email, etc. Los términos pertenecen a clases de jerarquías disjuntas, y en principio no tiene por qué existir una clase Term de la cual hereden todos ellos. Sin embargo, los términos deben entender una serie de mensajes en común; por ejemplo para averiguar las posiciones o intervalos donde aparecen dentro del documento. Podríamos crear el tipo abstracto Term formado por los mensajes {#positionsInDocument: #intervalsInDocument:}. Así, el concepto de término se haría tangible y quedaría documentado el conjunto de mensajes que futuros términos deberían implementar.

SmallInterfaces implementa la idea de tipos abstractos en Smalltalk, y recientemente está disponible para Squeak [Sad00]. Finalizamos esta sección, presentando un resumen de las diferencias entre tipo concreto y tipo abstracto (fig. 4.3).

4.3.1 Declaraciones Estáticas

Es interesante conocer el origen de la confusión entre tipo concreto y tipo abstracto. Los creadores del lenguaje Self, Randall Smith y David Ungar, nos cuentan esta historia [SU95].[1]

[1] Otra versión del origen de los tipos y su utilidad puede encontrarse en [Ler98].

	Tipo Concreto	**Tipo Abstracto**
interesado	compilador	ser humano
definición	clase exacta	conjunto de mensajes
definido para	objeto	expresión
describe	estructura interna de los objetos	mensajes que los objetos deben entender
información	específica	general
utilidad	construcción del grafo de llamadas	comprensión de programas

Figura 4.3: Resumen de diferencias entre tipo concreto y tipo abstracto

En el principio estaban FORTRAN, ALGOL y Lisp. En todos ellos el programador sólo debía decir lo que es necesario para ejecutar los programas. Como Lisp era interpretado, no se brindaba ninguna información de tipo. Como ALGOL y FORTRAN eran compilados, era necesario que los programadores especificaran información primitiva de tipos, como ser si una variable contenía un integer o un float, para que el compilador generase la instrucciones correctas. A medida que los lenguajes evolucionaron, se descubrió que agregando más declaraciones estáticas de tipos, el compilador podría llegar a crear código más eficiente. Por ejemplo, los procedimientos de PL/I tenían que ser explícitamente declarados como recursivos, así el compilador podía usar una llamada más rápida para los no-recursivos.

Los programadores notaron que estas declaraciones estáticas podrían ser de gran valor en hacer más comprensible un programa. Hasta ese momento, el principal beneficiario de esas declaraciones era el compilador, pero con SIMULA y PASCAL nació un movimiento; usar declaraciones para el beneficio tanto de humanos como de compiladores.

Esta tendencia ha sido poco feliz. El problema es que la información que un humano necesita para entender un programa, o para razonar acerca de su correctitud, no es necesariamente la misma información que un compilador necesita para generar un programa más eficiente. La mayoría de los lenguajes con declaraciones de tipos confunden estas dos dimensiones, ya sea limitando la eficiencia que podría ganarse con las declaraciones o, más frecuentemente, limitando el reuso de código a tal extremo que es necesario duplicar algoritmos o engañar al sistema de tipos.

Las declaraciones estáticas facilitan al compilador vincular estáticamente la llamada a un procedimiento con el código que lo implementa (*static* o *early-binding*). Para sumar dos variables numéricas, por ejemplo, el compilador debe conocer la función de suma que deberá invocar (suma entera o suma en punto flotante) y esto depende del tipo de los operandos. La vinculación estática fija en tiempo de compilación cuál es el procedimiento que se utilizará y con esta información, el compilador puede realizar optimizaciones globales (ie. que abarquen más de un procedimiento). Se gana en eficiencia pero se pierde flexibilidad.

En Smalltalk no hay declaraciones estáticas y por eso es difícil extraer información de tipos; pero las declaraciones estáticas de tipos concretos traerían más inconvenientes que beneficios. Difícilmente puedan servir para inferir tipos concretos en forma *precisa*; esto requeriría que las declaraciones tuvieran el máximo nivel de detalle, y esto sería algo sumamente engorroso para cualquier persona. Además, se generaría una proliferación de código muy similar con la única diferencia en dichas declaraciones. Podemos imaginar múltiples versiones del método Object class | id: (presentado en la introducción) dónde x estuviera declarado con tipos ⟨SmallInteger⟩, ⟨String⟩ o ⟨Float⟩. Cada clase de objeto que utilizara este método, debería tener una versión con las declaraciones apropiadas.

4.4 Interpretación Abstracta

Nuestra forma de inferir tipos es mediante la interpretación abstracta sobre el dominio de los tipos concretos. La interpretación abstracta es una simulación de la ejecución y en nuestro caso utilizamos *tipos concretos* en lugar de objetos.

La interpretación abstracta es una forma sencilla e intuitiva de realizar la inferencia, alejada de formalismos teóricos y más cercana a la visión que poseen los programadores. El análisis o inferencia comienza a partir de una expresión inicial y continúa con la simulación de su evaluación en el dominio de los tipos concretos. Nuestra simulación sigue todos los posibles caminos de ejecución; ante un condicional, por ejemplo, simulamos la ejecución de las diferentes ramas y combinamos los resultados.

Presentamos un programa sencillo que utilizaremos luego para mostrar la inferencia de tipos en funcionamiento (fig. 4.4). En la izquierda de la figura vemos el código de TObject class | simpleProgram (que simplemente responde el carácter $A), a su derecha observamos el programa abstracto y abajo vemos las expresiones que lo componen (E_1–E_9). Por comodidad, las expresiones correspondientes a las variables x y p, las escribimos en el programa abstracto como x y p (e ignoramos, por el momento, los subíndices).

```
| x p |                              | x0 p0 |
x ← 'hello'.                         x1 ← ⟨String⟩.
p ← 1@2.                             p1 ← ⟨SmallInteger⟩@⟨SmallInteger⟩.
p isPoint                            p1 isPoint
    ifTrue:                              ifTrue:
        [x ← p x].                       ⟨BlockContext[x2 ← p1 x]⟩.
↑ (x + 64) asCharacter               ↑ (x2 + ⟨SmallInteger⟩) asCharacter
```

$E_1 = x_1 \leftarrow \langle \text{String} \rangle$ $E_4 = p_1$ isPoint $E_7 = p_1$ x
$E_2 = p_1 \leftarrow \langle \text{SmallInt} \rangle @ \langle \text{SmallInt} \rangle$ $E_5 = \langle \text{BlockContext}[E_6] \rangle$ $E_8 = x_2 + \langle \text{SmallInt} \rangle$
$E_3 = E_4$ ifTrue: E_5 $E_6 = x_2 \leftarrow E_7$ $E_9 = E_8$ asCharacter

Figura 4.4: Código fuente de TObject class | simpleProgram (izquierda), su correspondiente programa abstracto (derecha) y las expresiones que lo componen (abajo).

4.4.1 Paralelo entre Análisis y Ejecución

En el diseño e implementación de nuestro *motor de inferencia*, reflejamos la correspondencia que existe entre tiempo de *ejecución* y tiempo de *análisis*. Cada concepto que intervienen en la ejecución de un programa (clase, objeto, mensaje, método) tiene un concepto similar en el análisis. Los métodos compilados, por ejemplo, tienen su correspondiente método abstracto. Cada expresión en el método abstracto corresponde a una expresión en el método compilado. Las expresiones del método compilado están en el dominio de los objetos; las expresiones del método abstracto están en el dominio de los tipos concretos.

Veamos el paralelo entre ejecución y análisis en el programa recién presentado (fig. 4.4). Allí, la expresión literal 'hello' se corresponde con la expresión literal ⟨String⟩, la expresión de envío de mensaje 1@2 encuentra su correspondiente en la expresión ⟨SmallInteger⟩@⟨SmallInteger⟩, la variable p se corresponde con la variable de tipo p, y así sucesivamente. La evaluación del mensaje 1@2 retorna un nuevo punto, mientras que su análisis retorna el *tipo* de un punto con coordenadas enteras. El tipo puede ser nuevo, o puede no serlo; eso depende de lo previamente analizado. En Smalltalk hay una memoria de objetos (conocida como *imagen*); en nuestro análisis hay una *memoria de tipos*. Cada tipo concreto es un objeto de la jerarquía TiType y a medida que avanza el análisis, se los instala en la memoria de tipos. Futuras referencias al mismo tipo concreto lo obtienen de dicha memoria.

Durante la ejecución el entero 1 recibe el mensaje #@, cuando esto sucede, el intérprete de Smalltalk busca en la clase SmallInteger un método con el mismo nombre, lo encuentra y lo ejecuta. Durante el análisis el tipo

⟨SmallInteger⟩ recibe el mensaje #@, cuando esto sucede, nuestro intérprete (instancia de TiInterpreter) busca el método abstracto con el mismo nombre, lo encuentra y lo analiza. La búsqueda sucede en la clase del tipo concreto (instancia de TiClass). Así como las clases tienen un diccionario de métodos compilados, nuestras clases tienen un diccionario de métodos abstractos (instancias de TiAnalyzedMethod). Puede suceder que la clase abstracta no tenga un determinado método; en ese caso, utilizamos el compilador (instancia de TiCompiler) para construir el método abstracto a partir del método compilado. De esta forma, los métodos abstractos se van creando a medida que avanza el análisis.

En Smalltalk, las variables nacen inicializadas con nil y la asignación cambia el objeto al que apuntan. En nuestro análisis, las variables nacen inicializadas con ⟨UndefinedObject⟩ y la asignación cambia el tipo que contienen. ¿Qué sucede con el tipo que había antes? ¿Lo descartamos? La respuesta es *no*—porque introducimos en las variables la noción de *estado*. Esta noción nos permite distinguir el tipo de las diferentes ocurrencias de una variable y para ello, utilizamos un subíndice (x_0 x_1 x_2 en la figura anterior). Las variables cambian de estado cuando están en lado izquierdo de una asignación.[2] La noción de estado nos permite mejorar la precisión de la inferencia porque reduce el tamaño del conjunto de tipos. Sin estados, por ejemplo, x_2 tendría los tipos {⟨UndefinedObject⟩ ⟨String⟩ ⟨SmallInteger⟩}, pero utilizando estados el conjunto se reduce a {⟨SmallInteger⟩}.

4.4.2 Ejemplo

Utilizamos un ejemplo para mostrar la interpretación abstracta en funcionamiento. Allí, podemos apreciar los diferentes estados que toma el intérprete abstracto en el programa anterior (fig. 4.5).

La primera columna de la figura indica la expresión que se interpreta; las demás, muestran el tipo de diferentes expresiones. La primera fila muestra el estado inicial del programa; por esta razón, *x* y *p* tienen el tipo ⟨UndefinedObject⟩. La segunda fila muestra la interpretación abstracta de E_1. En esta expresión *x* cambia su estado de ⟨UndefinedObject⟩ a ⟨String⟩. Después de las asignaciones iniciales, se interpreta E_4 y luego E_5 pues son los argumentos del mensaje #ifTrue:. Cuando se interpreta la expresión E_4, se analiza el método Point | isPoint y se obtiene ⟨True⟩. Si E_4 fuera p class == Point, se obtendría el conjunto más impreciso {⟨True⟩ ⟨False⟩} porque es el resultado de analizar la primitiva de equivalencia.

Cuando se interpreta E_5 se crea el tipo especial ⟨BlockContext[E_6]⟩ correspondiente al bloque de código; es especial, porque cada bloque tiene un tipo

[2] Los cambios de estado de las variables se explican con más detalle en §5.3 (p.43)

	x	p	E_4	E_7	E_8	E_9
	⟨Undef⟩	⟨Undef⟩				
E_1	⟨String⟩	"				
E_2	"	⟨IntPoint⟩				
E_4	"	"	⟨True⟩			
E_5	"	"				
E_3	"	"				
E_7	"	"		⟨SmallInt⟩		
E_6	⟨SmallInt⟩	"				
E_8	"	"			⟨SmallInt⟩	
E_9	"	"				⟨Char ?⟩

⟨Undef⟩ = ⟨UndefinedObject⟩ ⟨IntPoint⟩ = ⟨Point x:⟨SmallInteger⟩ y:⟨SmallInteger⟩⟩
⟨SmallInt⟩ = ⟨SmallInteger⟩ ⟨Char ?⟩ = ⟨Character ?⟩

Figura 4.5: Interpretación abstracta del programa anterior. La primera columna indica la expresión que se interpreta; las demás, muestran el tipo de diferentes expresiones.

distinto que incluye su definición (en este caso E_6). Sin esta distinción, todos los bloques compartirían el mismo tipo y no sería posible analizar el código de nuestro ejemplo. Luego se interpreta el mensaje #ifTrue: (E_3) y como el receptor es ⟨True⟩, se evalúa el bloque. La evaluación del bloque consiste en interpretar su definición; como la definición del bloque es una asignación, primero se interpreta su lado derecho (E_7).

Finalmente, la expresión E_9 retorna ⟨Character ?⟩. El signo de pregunta indica que el tipo no se conoce completamente. En este caso sabemos que es un carácter, pero no sabemos cómo es su estructura interna (cuántas variables de instancia tiene, cuáles son los tipos de sus variables de instancia, etc.). Si a ⟨Character ?⟩ le enviamos el mensaje #asInteger, el tipo *muta* a ⟨Character value: ⟨SmallInteger⟩⟩; esta es la implementación *lazy* para averiguar el tipo de las variables de instancia. Sin ella, el análisis de Point new implicaría recorrer gran parte de la imagen, pues el tipo de la clase Point tiene toda su jerarquía de superclases y cada una de ellas, todas sus subclases. En nuestra implementación, el tipo de Point es ⟨Point class ?⟩.

Limitaciones

La inferencia de tipos concretos necesita fijar el conjunto de clases y métodos antes de comenzar. No es posible analizar métodos que todavía no existen

o inferir tipos correspondientes a clases aún no creadas. El cambio de un solo método incluso, puede invalidar gran parte de lo inferido. Cuando eso sucede, descartamos todo lo inferido y volvemos a comenzar el análisis.

4.5 Trabajo Relacionado

Debemos aclarar que nuestra inferencia es muy distinta a la realizada en lenguajes funcionales; allí, el interés principal es calcular los tipos más generales que sirven para describir usos legales de una función [PC94]. Nosotros, en cambio, estamos interesados en los tipos concretos que tienen información específica y detallada.

En los lenguajes orientados a objetos, la construcción del grafo de llamadas y la inferencia de tipos concretos deben resolver el mismo problema: la vinculación dinámica. David Grove trabajó en la construcción del grafo de llamadas con el objetivo de realizar optimizaciones inter-procedurales [GDDC97, Gro98]. En su trabajo observa que los lenguajes funcionales, también presentan una forma de vinculación dinámica. En ellos, las funciones son valores y su evaluación no puede resolverse hasta el momento de la ejecución. Grove resuelve el problema de la vinculación dinámica de la misma forma que lo hacemos nosotros: determinando en cada sitio de invocación (*call site*) el conjunto de clases de la expresión receptora.

Una técnica para obtener información de tipos en forma dinámica es *type feedback* [HU94, Höl95]. Esta técnica, creada por Urs Hölzle, primero realiza la ejecución y luego obtiene los tipos. Así se obtiene el tipo de *una* ejecución. Nuestro objetivo, en cambio, es averiguar el tipo de todas las posibles ejecuciones. En realidad, se realizan varias ejecuciones, registrando en cada una de ellas los tipos que ocurren. De todas formas, no se garantiza que la información de tipo finalmente obtenida sea completa y, por ejemplo, no se la puede utilizar para extraer aplicaciones.

Debemos expresar profunda gratitud al proyecto Self porque allí se originaron los trabajos más importantes relacionados con la inferencia de tipos. Urs Hölzle y Ole Agesen trabajaron en el equipo de investigación de este proyecto, enfocando el problema de la obtención de tipos desde distintas perspectivas. Hacia el final, realizaron conjuntamente comparaciones entre sus técnicas [AH95, HA96]. Craig Chambers, director de tesis de David Grove, también fue integrante del proyecto Self.

Diversos investigadores intentaron agregarle un sistema de tipos a Smalltalk [Suz81, BI82, Gra89, BG93]. La mayoría de ellos con un objetivo diferente (permitir optimizaciones de código, realizar chequeos estáticos, facilitar la comprensión de los programas), con una concepción distinta de tipo (una

clase, un conjunto de clases, un conjunto de selectores) y sin realizar la distinción entre *tipo abstracto* y *tipo concreto*. Varios de estos trabajos, requerían que el programador ingrese declaraciones estáticas y algunos combinaban estas declaraciones con inferencia de tipos.

5. Análisis de Tipos

El análisis de tipos que realizamos, consiste en la interpretación abstracta a partir de una expresión inicial. En este capítulo presentamos los elementos más importantes que componen el análisis. Comenzamos con una nueva definición de tipo concreto y presentamos los detalles de su implementación. Continuamos con el mecanismo básico de nuestra inferencia: slots y conexiones. Elevamos el nivel de abstracción, introduciendo el concepto de template y método analizado. Finalizamos con una idea que nos permite mejorar la precisión del análisis. La idea de distinguir estados en las variables.

5.1 Tipos Concretos

Explicamos anteriormente el concepto de tipo concreto y por qué nos interesan. En adelante y por comodidad, nos referimos a ellos utilizando solamente el término *tipo*.

5.1.1 Definición

En el capítulo anterior, presentamos la primera definición de tipo concreto. Allí afirmamos que el tipo de un objeto es su clase y mostramos varios ejemplos. Según esa definición, el punto `1@1` de coordenadas enteras tiene el mismo tipo que el punto `1.0@1.0` de coordenadas reales. En ambos casos ⟨Point⟩. Inmediatamente surgen ciertas preguntas: ¿cuál es el tipo de la expresión `1@1 x` ? ¿y el tipo de `1.0@1.0 x` ? En el primer caso queremos que sea ⟨SmallInteger⟩ y en el segundo caso, ⟨Float⟩. Necesitamos una definición que tome en cuenta las variables de instancia:

Definición 1: *El tipo de un objeto esta compuesto por su clase y por los tipos de sus variables de instancia.*

Esta nueva definición nos permite conocer completamente la estructura interna de los objetos, y cumplir así el objetivo primordial de los tipos concretos. Presentamos algunos ejemplos de acuerdo a esta definición (fig. 5.1).

1@1	→	⟨Point x: ⟨SmallInteger⟩ y: ⟨SmallInteger⟩⟩
1.0@1.0	→	⟨Point x: ⟨Float⟩ y: ⟨Float⟩⟩
Point new	→	⟨Point x: ⟨UndefinedObject⟩ y: ⟨UndefinedObject⟩⟩
0@0 corner: 1@1	→	⟨Rectangle origin: ⟨Point x: ⟨SmallInt⟩ y: ⟨SmallInt⟩⟩ corner: ⟨Point x: ⟨SmallInt⟩ y: ⟨SmallInt⟩⟩⟩

Figura 5.1: Tipos concretos según la primera definición

La definición recién enunciada, sin embargo, necesita ser extendida para tomar en cuenta a las variables indexadas. Estas variables no se acceden por nombre sino por índice y además, distintos objetos de la misma clase pueden tener distinta cantidad de ellas. Por ejemplo, Array new: 2 crea un objeto con dos variables indexadas, pero también podría haberlo hecho con cinco, cincuenta o cinco mil. Una alternativa es considerar a las variables indexadas como variables de instancia cuyo nombre es el índice. Así, el tipo de #(1 'dos') sería ⟨Array 1: ⟨SmallInteger⟩ 2: ⟨String⟩⟩ que sería distinto del tipo de #('tres' 4). Esto es coherente con nuestra idea de diferenciar los objetos según su estructura interna, pero existe un problema.

Las variables indexadas se acceden enviando el mensaje #at:, utilizando un índice como argumento. Durante el análisis no tenemos el valor del índice sino que tenemos su *tipo*, y ahí radica el problema. El tipo del índice (probablemente ⟨SmallInteger⟩) no permite distinguir cuál de todas las variables indexadas se quiere acceder y entonces devolvemos el tipo de todas ellas. Pero si hacemos eso, ya no tiene sentido distinguir cada una de las variables según su posición. Nuestra alternativa es mantener en un mismo conjunto los tipos de todas las variables indexadas. Completamos la definición anterior de la siguiente manera:

Definición 2: *El tipo de un objeto con variables indexadas, está compuesto por su clase y por la unión de los tipos de sus variables indexadas.*

Presentamos nuevos ejemplos según esta última definición (fig. 5.2). Debemos aclarar, sin embargo, que existe una disyuntiva entre precisión y correctitud. En nuestra implementación, por ejemplo, la expresión #(1 'dos') at: anIndex tiene el tipo {⟨SmallInteger⟩ ⟨String⟩}. Este resultado es incorrecto pues la evaluación de esa expresión puede provocar un error de *acceso fuera de rango*, dependiendo del valor de la variable anIndex. Esto puede solucionarse, agregando al resultado anterior un tipo especial que represente al error. Sin embargo se estaría generando un resultado impreciso para aquellos casos dónde previamente se chequea el rango de acceso (ej: Collection | do:).

#(1 'dos')	→	⟨Array (⟨SmallInteger⟩ ⟨String⟩)⟩
#('tres' 4 5)	→	⟨Array (⟨SmallInteger⟩ ⟨String⟩)⟩
#(1 'dos' 3.0)	→	⟨Array (⟨SmallInteger⟩ ⟨String⟩ ⟨Float⟩)⟩
Array new: 50	→	⟨Array (⟨UndefinedObject⟩)⟩

Figura 5.2: Tipos concretos de acuerdo a la segunda definición

5.1.2 Implementación

Las nuevas definiciones de tipo son de naturaleza recursiva. Eso impacta directamente en la implementación pues es normal que aparezcan en el código objetos complejos, como son las clases. Una implementación ingenua puede hacer que el análisis prosiga indefinidamente (si por ejemplo el objeto tiene referencias circulares).

Utilizamos una implementación *lazy* donde los tipos de las variables de instancia se van calculando a medida que se necesitan. De esta forma, el tipo de la clase Point es ⟨Point class ?⟩ y el de #(1 'dos') es ⟨Array ?⟩. El signo de pregunta indica que el tipo no está completamente resuelto y tiene una referencia al objeto original. Cuando sea necesario conocer el tipo de alguna variable, se elimina el objeto original y se realiza la expansión del tipo. Esta expansión es una *mutación* pues se elimina la referencia al objeto original y se agregan los tipos de las variables de instancia.

En nuestro análisis los tipos son objetos (instancias de la jerarquía TiType). Un solo tipo representa a *todos* los objetos con la misma estructura interna. Así, el tipo ⟨SmallInteger⟩ es un objeto que representa a todos los enteros pequeños y el tipo ⟨String⟩ es otro objeto que representa todas las posibles cadenas de caracteres.[1] Para lograr esto, utilizamos una *memoria de tipos* que va recordando todos aquellos que aparecen durante el análisis. Esto garantiza la unicidad de los tipos porque antes de crear uno nuevo, se consulta en la memoria si el mismo está presente.

Veamos un nuevo ejemplo (fig. 5.3). Luego de analizar el código de la izquierda, el tipo de las variables aPoint, intPoint y floatPoint es ⟨Point ?⟩. A pesar que se imprimen igual, son tres tipos distintos dónde el signo de pregunta representa a diferentes objetos. Si a continuación, analizamos el código de la izquierda, el tipo de aPoint se expande a ⟨Point x: ⟨SmallInteger⟩ y: ⟨SmallInteger⟩⟩. Cuando se expanda el tipo de intPoint, se encontrará uno equivalente en la memoria de tipos y se producirá una mutación (más precisamente se hace un #become:). Luego, el tipo de floatPoint se expandirá a ⟨Point x: ⟨Float⟩ y: ⟨Float⟩⟩.

Los objetos son entidades que mantienen su identidad. Un objeto puede

[1] Los enteros pequeños en Squeak, pertenecen al intervalo $[-2^{30}, 2^{30})$.

```
aPoint ← 1@1.
intPoint ← 2@2.
floatPoint ← 1.0@1.0
```

```
aPoint x + intPoint y.
floatPoint x.
```

Figura 5.3: Mutación del tipo ⟨Point ?⟩

cambiar internamente pero sigue siendo el mismo. Por ejemplo, si evaluamos aPoint x: 'hello', el punto cambia internamente a 'hello'@1. Los cambios internos deben reflejarse con cambios de tipo; así, el tipo de aPoint cambia a ⟨Point x: ⟨String⟩ y: ⟨SmallInteger⟩⟩. Pero los tipos son entidades inmutables que *no* pueden cambiar internamente. Cuando sucede un cambio de tipo, en realidad creamos uno nuevo y en la memoria de tipos quedan ambos instalados. Resumimos en una tabla, el paralelo entre tipo y objeto (fig. 5.4).

Tiempo de Ejecución	**Tiempo de Análisis**
objeto	tipo
imagen	memoria de tipos
objetos con igual estructura interna	sólo un tipo los representa
cambio interno de un objeto	cambio de tipo

Figura 5.4: Paralelo entre tipo y objeto

5.2 Slots y Conexiones

Los slots y sus conexiones constituyen el mecanismo básico que se utiliza para inferir tipos. En esta sección, explicamos detalladamente cómo funcionan, trazando el paralelo con lo que ocurre en tiempo de ejecución.

El slot es un objeto (instancia de TiSlot) dónde se acumulan los tipos de una expresión. Asignamos a cada expresión, un slot distinto. Las instancias de TiSlot, acumulan tipos en su variable typeSet y mantienen las conexiones en su variable connections. Inicialmente cada slot está vacío, y a medida que avanza el análisis, se le agregan nuevos tipos. Nunca se eliminan tipos de un slot y de esta forma, el conjunto de tipos inferidos crece en forma monótona.[2] Cuando finalmente concluye el análisis, el tipo de una expresión es el conjunto de tipos que tiene su slot (fig. 5.5).

[2] La propiedad de monotonicidad puede utilizarse para razonar acerca de la complejidad de la inferencia [Age96, p.38]. Recomendamos consultar *"The Complexity of Type Analysis of Object Oriented Programs"* para un estudio en profundidad [GI98].

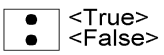
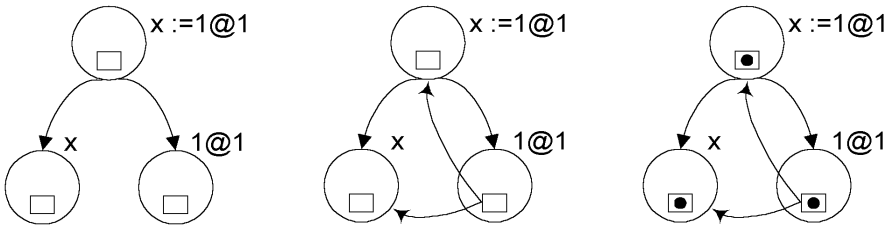


Figura 5.5: Slot inseminado con tipos booleanos

El slot, además de acumular tipos, participa activamente en el análisis. Un slot se conecta con otros slots y por estas conexiones fluyen tipos. Las conexiones tienen un sentido de circulación que refleja el flujo de datos. Por ejemplo, si el slot x está conectado al slot y , entonces x typeSet $\subseteq$ y typeSet ; es decir, los tipos de x fluyen a y. Las conexiones deben realizarse con sumo cuidado, pues una conexión mal definida puede provocar resultados incorrectos. Las conexiones deben reflejar el flujo de datos pues definen el flujo de tipos.

Veamos el paralelo entre análisis y ejecución. En tiempo de ejecución se interpretan los métodos compilados; en tiempo de análisis se interpretan los métodos abstractos. Los métodos compilados están compuestos por bytecodes; los métodos abstractos están compuestos por slots y conexiones. Los slots y sus conexiones son el paralelo de los bytecodes.

Veamos el ejemplo de una asignación. La evaluación de x $\leftarrow$ 1@1 provoca un flujo de datos porque el objeto 1@1 pasa a la variable x. En tiempo de análisis imitamos ese flujo, pasando el tipo del punto 1@1 hacia la variable x. Así obtenemos en el slot de la variable, el tipo del punto con coordenadas enteras. Este flujo es posible debido a la forma que realizamos las conexiones. Conectamos la expresión de asignación, realizando una conexión desde la expresión derecha hacia la expresión izquierda (fig. 5.6).



(a) Expresión de asignación

(b) Expresión conectada

(c) Tipos propagados

Los círculos son las expresiones, los rectángulos son los slots, los tipos son los puntos. Las flechas de punta recta denotan la estructura del árbol, y las de punta curva denotan las conexiones.

Figura 5.6: Conexión de una expresión de asignación

Los slots y las conexiones forman la infraestructura que permite realizar la inferencia de tipos. A diferencia de los bytecodes, que simplemente son

códigos para el intérprete, los slots son objetos con responsabilidades; esto último simplifica la tarea del intérprete abstracto. Las responsabilidades básicas del slot son: 1) acumular tipos y 2) propagarlos.

El slot propaga los tipos en dos oportunidades: 1) cuando se agrega una nueva conexión y 2) cuando llega un nuevo tipo. Si se agrega una conexión, el slot propaga todos sus tipos por dicha conexión. Si llega un nuevo tipo, el slot propaga dicho tipo por todas sus conexiones.

Afirmamos que las conexiones se realizan entre slots, pero en realidad somos más flexibles. El origen de una conexión siempre es un slot, pero el destino puede ser cualquier objeto. El único requisito es que el destino entienda el mensaje #seed: ; debe hacerlo porque el slot propaga sus tipos enviando este mensaje por sus conexiones. En el ejemplo de la asignación podemos observar que el slot de la expresión izquierda se conecta directamente a la expresión derecha. Cuando la expresión derecha recibe el mensaje #seed: , se lo delega a su slot, quien acumula el tipo recibido (fig. 5.6).

5.2.1 Templates

Los métodos compilados elevan el nivel de abstracción de los bytecodes. De la misma forma, elevamos el nivel de abstracción de los slots introduciendo el concepto de *template*.[3] Este concepto nos permite analizar métodos y, por ejemplo, afirmar que Object | returnArg: retorna el tipo ⟨String⟩ cuando su argumento también es ⟨String⟩.

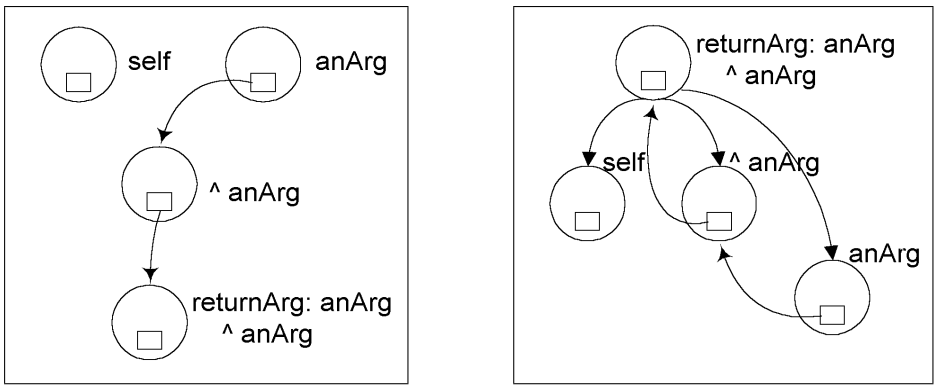
```
Object | returnArg: anArg
    ↑ anArg
```

Analizamos cada método con un template distinto. El template es construido cuando analizamos el método por primera vez y la construcción consiste en conectar las expresiones del método, respetando el flujo de datos. El template recién construido está *virgen* porque todavía no lo inseminamos. La inseminación del template se produce cuando inseminamos sus argumentos, y la llamamos *inseminación inicial*.

Un template virgen tiene los caminos por donde fluirán los tipos pero no hay tipos que fluyan, porque faltan los tipos de los argumentos. El único flujo en un template virgen es el proveniente de las expresiones literales (enteros, floats, strings, símbolos, arrays). Un template virgen es el equivalente a un método sin ejecutar. Los métodos tienen las expresiones que definen la ejecución y los templates tienen las conexiones que definen la inferencia; en ambos casos faltan los argumentos.

[3] Utilizamos el término *template* por coherencia con el trabajo de Agesen [Age96, p.42].

Dibujamos, de dos formas distintas, el template virgen de Object | returnArg: (fig. 5.7). En la izquierda lo hacemos de la misma forma que Agesen y en la derecha presentamos nuestra forma de hacerlo. El template está virgen y por ello sus slots están vacíos. La inseminación del template se realiza por los slots de los argumentos, también llamados *slots iniciales*. El diagrama de Agesen induce una visión gravitatoria pues los tipos *caen* desde los slots iniciales hacia el slot final. Los slots iniciales están en la parte superior y el *slot final* en la parte inferior. En este ejemplo, observamos que el slot de self no tiene conexiones pues no participa en el método.



(a) Diagrama de Agesen

(b) Nuestro Diagrama

Las conexiones entre slots se dibujan con flechas de punta curva. El árbol sintáctico se dibuja con flechas de punta recta.

Figura 5.7: Template virgen del método Object | returnArg:

Nuestro diagrama muestra las conexiones y la estructura de árbol que forman las expresiones del método. En la parte superior está la raíz del árbol; expresión que representa al método en su conjunto. La raíz del árbol tiene el slot final, que corresponde al valor de retorno del método. En nuestro diagrama es difícil ubicar los slots iniciales, pero también es más fácil de dibujar. Primero dibujamos el árbol y luego conectamos sus expresiones. Algo similar sucede en la implementación: primero replicamos el árbol sintáctico y luego conectamos sus expresiones.[4] Al árbol conectado lo llamamos *árbol de inferencia*. En realidad, no es un árbol sino un grafo, porque los nodos de las variables pueden estar en varias ramas y así se pueden formar ciclos.

[4] El árbol sintáctico también se conoce como *árbol de parseo* [ASU86].

5.2.2 Métodos Analizados

En el capítulo anterior, mostramos un método sencillo y su correspondiente método abstracto. En realidad, los métodos abstractos no existen en nuestro diseño; el equivalente a los métodos compilados son los métodos analizados. Los métodos analizados son instancias de la clase TiAnalyzedMethod. Estos objetos tienen un template virgen en su variable definition y un diccionario de resultados en su variable invocations (fig. 5.8).

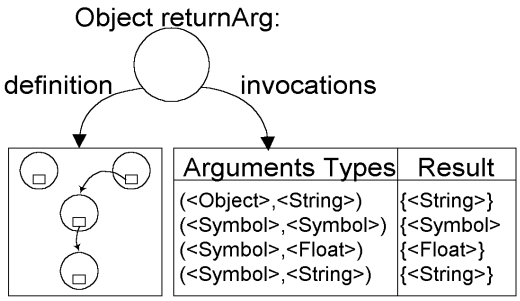


Figura 5.8: Método Analizado de Object | returnArg:

El método compilado recién se ejecuta cuando están disponibles sus argumentos. Para ejecutarlo, el intérprete crea un registro de activación utilizando un sólo objeto por argumento. Nosotros inseminamos el template, utilizando un sólo tipo por argumento; esto se llama *inseminación monomórfica* [Age96, p.57]. El template inseminado es equivalente al registro de activación ejecutado.

El template virgen define cómo se realiza el análisis, pero nunca se lo insemina directamente. Las inseminaciones se realizan sobre copias del template virgen. En el diccionario de invocaciones se guardan los resultados de los diferentes análisis; esto lo hacemos para no repetir dos veces el mismo análisis. Las claves del diccionario son los tipos (monomórficos) de los argumentos, y los valores del diccionario son los resultado de cada inseminación. Podemos descartar el template inseminado porque guardamos el *resultado de la inseminación*. Este resultado, es el conjunto de tipos presentes en el slot final (que corresponde al valor que retorna el método).

El paralelo entre método compilado y método analizado es múltiple. El método compilado se ejecuta según sus bytecodes; el método analizado se analiza según su template. El método compilado se ejecuta con objetos; el método analizado se insemina con tipos. La ejecución se realiza con un sólo objeto por argumento; la inseminación se realiza con un sólo tipo por argumento.

Sin embargo, también existen diferencias entre el método compilado y el método analizado. La ejecución de un método puede devolver distintos re-

sultados dependiendo de factores externos y ajenos a los argumentos; el análisis de un método devuelve un resultado que sólo depende de los argumentos. El método analizado puede guardar un registro de sus análisis; el método compilado no puede guardar un registro con sus ejecuciones, porque es impraticable (requiere memoria infinita).

5.3 Estados de las Variables

En Smalltalk nada impide que en una misma variable primero se guarde un número, luego un punto, un string, o cualquier otro objeto (fig. 5.9). Ese código es perfectamente legal y debemos analizarlo en forma precisa. Para lograr esto, introducimos la noción de *estado* en las variables. Luego de una asignación, por ejemplo, se genera un cambio de estado en la variable asignada y los siguientes accesos utilizan el estado recién creado. El cambio de estado se implementa agregando un nuevo slot a la variable. Cada slot representa un estado distinto y lo notamos con un subíndice. En nuestro diseño, las variables son las únicas expresiones que pueden tener más de un slot.

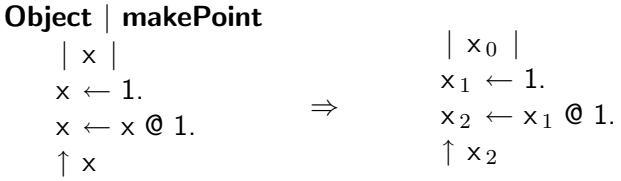


Figura 5.9: Múltiples estados de la variable x, indicados con un subíndice

En este ejemplo, la noción de múltiples estados nos permite inferir nicamente el tipo de un punto con coordenadas enteras. Si no utilizáramos la noción de estados, obtendríamos además el tipo de un número entero. Inferir un conjunto más grande implica una pérdida de precisión, y esta pérdida, se propaga a medida que avanza el análisis. Por ejemplo, si luego analizamos anObject makePoint degrees detectaríamos un mensaje que no podría ser entendido (pues SmallInteger no entiende el mensaje #degrees). Pero en realidad esto no puede suceder, pues #makePoint nunca devuelve un SmallInteger. La pérdida inicial de precisión se incrementa en forma no-lineal. La vinculación dinámica de mensajes provoca que los tipos superfluos disparen el análisis de nuevos métodos. Los nuevos métodos generan nuevos tipos, que a su vez, disparan el análisis de otros métodos y así sucesivamente. Es conveniente inferir la menor cantidad de tipos posibles. A menor cantidad de tipos, menor cantidad de métodos para analizar y más preciso es el resultado.

Las variables tienen un slot por estado. Esto nos permite ser precisos y registrar, al mismo tiempo, todos los tipos que toma la variable. Esta información es importante si luego queremos realizar extracción de aplicaciones. En el ejemplo anterior, la variable x fue durante un breve lapso, un SmallInteger; esta clase tiene que estar en el conjunto de las clases a extraer.

5.3.1 Variables de Instancia

Las variables de instancia de un objeto sólo pueden cambiarse enviándole un mensaje a dicho objeto. Este cambio en la variable de instancia también altera al receptor del mensaje. El receptor sigue siendo el mismo objeto pero su estructura interna está alterada (sufrió un efecto colateral). La nueva estructura interna debe reflejarse con un nuevo tipo.

Los cambios de estado en las variables locales son sencillos de comprender e implementar. Los cambios de estado en las variables de instancia son más complicados. Es así, porque hay que sincronizar el estado de la variable cambiada con el estado del receptor del mensaje. En el siguiente ejemplo podemos ver que: 1) las variables de instancia toman su valor inicial del receptor (denotado con self_0) y 2) el cambio de estado en la variable de instancia debe seguirse de un cambio de estado en el receptor y posterior sincronización de sus tipos. Por ejemplo, si el tipo del receptor es ⟨Point x: ⟨Float⟩ y: ⟨Float⟩⟩ el análisis de #changeX cambia su tipo a ⟨Point x: ⟨String⟩ y: ⟨Float⟩⟩.

$$\begin{array}{l}\textbf{Point} \mid \textbf{changeX} \\ \quad \mathsf{x} \leftarrow \text{'hello'}\end{array} \quad \Rightarrow \quad \begin{array}{l}\mathsf{x}_0 \leftarrow \mathsf{self}_0.\mathsf{x} \\ \mathsf{x}_1 \leftarrow \text{'hello'} \\ \mathsf{self}_1 \leftarrow \mathsf{merge}(\mathsf{self}_0\ ,\ \mathsf{x}_1)\end{array}$$

Figura 5.10: Cambio de estado en una variable de instancia

El análisis de un mensaje puede cambiar el tipo del receptor, y en consecuencia debemos generarle un cambio de estado. No solamente hay que hacerlo con el receptor del mensaje, sino también con cada uno de los argumentos. Esto es así, porque cualquiera de ellos puede sufrir un efecto colateral debido al envío de otro mensaje. Por ejemplo, el objeto 1@1 sufre un efecto colateral al evaluar la expresión anObject changePoint: 1@1 y queda transformado en 'hello'@1 (la definición de Object | changePoint está en la próxima figura).

La asignación, entonces, no es la única expresión que introduce un cambio de estado; la expresión de envío de mensaje también lo hace con cada uno de sus argumentos. Implementamos el cambio de estado en los mensajes, agregando un slot a cada argumento. En realidad, sólo podemos agregar slots a aquellos argumentos que son variables; esto incluye a los argumentos

formales del método. Los argumentos formales son vistos como variables que no pueden ser asignadas. En la siguiente figura, vemos el nuevo estado que le agregamos al argumento (aPoint_1) luego de recibir el mensaje #changeX. El nuevo estado se crea a partir del resultado del envío de mensaje (sendResult).

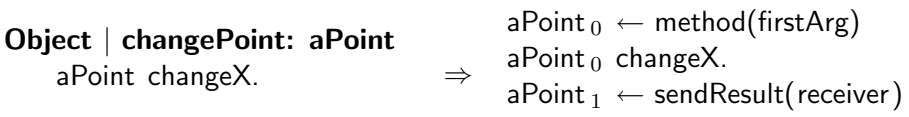


Figura 5.11: Cambios de estado provocados por un mensaje

En las últimas figuras, utilizamos una notación especial en el lado derecho (fig. 5.10, 5.11). Esta notación sólo cumple fines didácticos y no guarda relación con la implementación. El objetivo es mostrar los cambios de estado que se producen en las variables. Para ello, cada estado se muestra con un subíndice distinto, que se incrementa con cada cambio. Los envíos de mensajes introducen nuevos estados en los argumentos que son inseminados con el resultado del análisis del mensaje (sendResult). En la próxima figura, mostramos la forma genérica de anotar los cambios de estados en los argumentos de un mensaje.

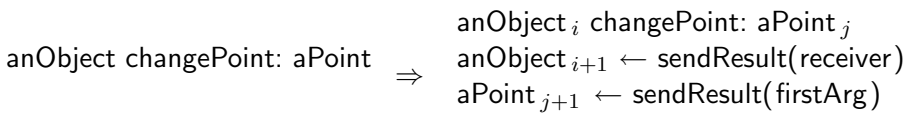


Figura 5.12: Cambios de estado en los argumentos de un mensaje

Análisis Sensible al Flujo

Además de la pérdida de precisión, otra de las desventajas de no distinguir entre estados es que posiblemente, haya que analizar la misma expresión más de una vez. En el siguiente ejemplo, analizaríamos el mensaje x odd en dos oportunidades. La primera vez, x sería un SmallInteger y entonces habría que analizar el método Number odd. La segunda vez, x sería un String y entonces habría que analizar el método String odd.

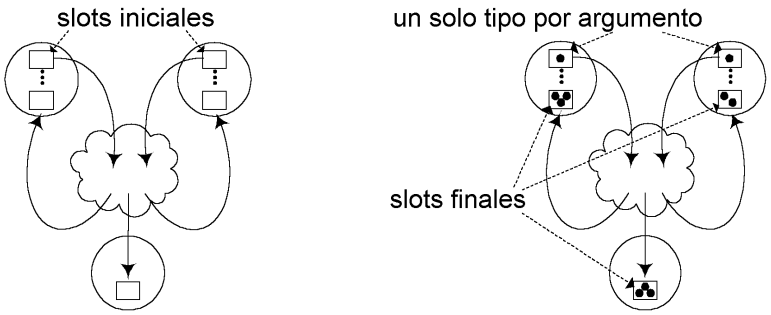
```
x ← 1.
x odd.
x ← 'bye'.
```

```
String | odd
    ↑ self size odd
```

Esto no sucede si utilizamos más de un slot. Luego de la segunda asignación, se agrega un slot a la variable x. Este nuevo slot es el que se insemina con ⟨String⟩. El mensaje x odd no se entera de esta inseminación pues está viendo el slot anterior. De esta forma, realizamos un análisis sensible al flujo [Age96, §4.5] [Gro98, §2.4].

5.3.2 Resultado de una Inseminación

Antes explicamos que el resultado de inseminar un template es el conjunto de tipos presentes en el slot final, y que dicho slot, corresponde al valor que devuelve el método. Vimos después, que el resultado de evaluar un mensaje no es solamente el valor que retorna sino también los efectos colaterales sobre sus argumentos. Incluimos, entonces, al último slot de cada argumento en el resultado de la inseminación (fig. 5.13).



(a) Template virgen (b) Template inseminado

Los slots iniciales se inseminan con un sólo tipo; los slots finales pueden terminar con varios tipos. Los slots finales corresponden a los argumentos del método y su valor de retorno. Estos slots forman el resultado de la inseminación.

Figura 5.13: Diagrama conceptual de un template.

Los argumentos formales de un método son vistos por el compilador como variables normales.[5] La única restricción sobre ellos, es que no pueden ser reasignados (es decir, participar en el lado izquierdo de una asignación). Sin embargo, pueden participar en expresiones de envío de mensaje en el papel de argumentos actuales. Cuando un argumento participa en un envío mensaje, se le genera un cambio de estado. El último de estos estados, es un slot donde se acumula el tipo final del argumento. Los tipos de este slot final deben utilizarse para inseminar el argumento que invocó el mensaje y

[5] El *argumento formal* es el argumento que figura en la definición del método; el *argumento actual* es el argumento del mensaje (e incluye al receptor).

así emular los efectos colaterales. Por esta razón, agregamos al resultado de la inseminación, el contenido de los últimos slots de cada argumento.

TiResult (return arguments)
TiBlockResult (nonLocalReturn)
TiMessageResult (receiver)
TiPrimitiveResult (successFlag)

Figura 5.14: Jerarquía de TiResult

El resultado de una inseminación es una instancia de TiMessageResult. Su responsabilidad es guardar el contenido de los slots finales, dónde cada contenido, es un conjunto de tipos. Estas instancias tienen el resultado de lo que devuelve el método en la variable return, y el resultado de cada argumento en arguments.

6. Dinámica del Análisis

En este capítulo presentamos los objetos que intervienen en el análisis de los métodos y su dinámica. Comenzamos explicando los objetos que componen el motor de inferencia, continuamos con la conexión de las distintas expresiones y finalizamos con la expresiones de mensajes que permiten avanzar el análisis hacia nuevos métodos.

6.1 El Motor de Inferencia

El *motor de inferencia* es un conjunto de objetos que interactúan entre sí y hacen posible la inferencia tipos. Ellos permiten que los tipos fluyan *entre* los métodos. No podemos definir el motor de inferencia en forma precisa, porque no es un objeto del diseño; es un término que utilizamos para referirnos a varios objetos. Estos objetos son instancias de las clases que explicamos a continuación (fig. 6.1).

TiSystem (typeMemory classes interpreter compiler)
TiClass (class methodDict)
TiAnalyzedMethod (class selector primitive definition invocations)
TiInterpreter (stack)
TiCompiler (sharedVariables literals variables analyzedMethod)

Figura 6.1: Principales clases del motor de inferencia

Aclaración: El análisis comienza por un método. En varias oportunidades aseguramos que el análisis comienza a partir de una expresión inicial o programa, pero no es así. Comienza por un método porque en Smalltalk no existen las expresiones aisladas (aunque tengamos esa sensación al trabajar en un *workspace*). Las expresiones ocurren dentro de los métodos y sólo ellos se ejecutan. Cuando se evalúa una expresión, se crea un método cuyo código es el texto seleccionado, se lo ejecuta y se lo borra. Adoptamos el mismo esquema para el análisis de expresiones. Conservamos esta sensación de estar analizando expresiones aisladas, pero en realidad el punto de partida de cada análisis es un método.

6.1.1 El Sistema

El principal objeto del motor de inferencia, es instancia de TiSystem. Su responsabilidad fundamental es mantener el estado global del análisis. Existe una única instancia de esta clase; representa al sistema Smalltalk y por eso la llamamos *sistema*. El sistema mantiene la memoria de tipos en su variable typeMemory, las clases analizadas en classes, un intérprete abstracto en interpreter y un compilador abstracto en compiler.

La memoria de tipos es el paralelo a la imagen Smalltalk, y es un conjunto dónde se acumulan los tipos inferidos (instancias de TiType o subclases). El propósito de esta memoria es garantizar la unicidad de los tipos. Gracias a ella, podemos asegurar, por ejemplo, que existe una única instancia del tipo ⟨SmallInteger⟩ para todos los números enteros (en el rango adecuado). Es decir, la expresión 1 type == 2 type, siempre es verdadera.

El sistema mantiene un diccionario con las clases que han sido analizadas. Las clases analizadas son instancia de TiClass, y estas a su vez, mantienen un diccionario con los métodos analizados en methodDict. Las clases y los métodos analizados, se crean a medida que el intérprete lo necesita. Los métodos analizados son instancias de TiAnalyzedMethod; estas instancias tienen un template virgen en su variable definition y guardan los resultados de los distintos análisis en su variable invocations. El resultado de un análisis es instancia de TiMessageResult y está asociado a un juego de argumentos monomórficos.

6.1.2 El Intérprete

El intérprete abstracto es instancia de TiInterpreter. Su responsabilidad es analizar los envíos de mensajes y detectar la recursividad en el análisis. Para detectar la recursividad utiliza su variable stack; allí apila cada método antes de interpretarlo y lo remueve cuando termina. Detecta la recursión cuando apila un método con idéntico juego de argumentos a otro presente en su stack. Esto permite detectar la recursión directa e indirecta, y evitar así, que el análisis continue indefinidamente.

El intérprete analiza los envíos de mensajes mediante su simulación. Para simular un envío, recibe el selector y los argumentos del mensaje. Los argumentos son tipos monomórficos y el primero de ellos es el receptor. El intérprete realiza la búsqueda del método utilizando el selector y la clase del receptor (presente en el tipo). La búsqueda procede a través de la jerarquía y cuando se encuentra el método analizado, se lo insemina. Puede suceder que la búsqueda falle, lo cual significa que en tiempo de ejecución el receptor no entenderá el mensaje; si esto sucede, el intérprete informa la situación y

detiene el análisis. En realidad, la búsqueda no devuelve un método analizado, sino la clase y el selector del mismo. El método analizado lo crea nuestro compilador a partir del método compilado, y luego lo instala en la clase analizada.

Incluimos una interesante mejora para el intérprete, como trabajo futuro. En lugar de detener el análisis cuando falla la búsqueda del método, el intérprete debería enviar el mensaje #doesNotUnderstand:. Así se podría analizar las clases que redefinen este especial método (por ejemplo, los proxies); y las clases que no lo redefinen, terminarían analizando Object | doesNotUnderstand:. En realidad, este método en Object no se analizaría, sino que el intérprete crearía un objeto especial. En este objeto especial, quedarían registradas las condiciones del error (sugerimos el nombre TiWillNotUnderstand para su clase). El análisis continuaría y estos objetos podrían ser luego inspeccionados.

6.1.3 El Compilador

El compilador es instancia de TiCompiler. Su principal responsabilidad es construir el método analizado a partir del método compilado. Para ello, decompila al método y obtiene el árbol sintáctico; luego lo replica y se obtiene el *árbol de inferencia*. El árbol de inferencia conserva la estructura del árbol sintáctico, pero tiene otra clase de nodos. Los nodos del árbol sintáctico son instancias de la jerarquía ParseNode; los nodos del árbol de inferencia son instancias de la jerarquía TiParseNode. Los nodos del árbol sintáctico están especializados en la generación de bytecodes; los nodos del árbol de inferencia están especializados en la inferencia de tipos.

El compilador construye el árbol de inferencia, replicando el árbol sintáctico. Para realizar la réplica, utilizamos la técnica de *double dispatch* entre los nodos y el compilador [Bec97]. El compilador envía a cada nodo el mensaje #mirrorIn: pasándose a sí mismo como parámetro; cada nodo, a su vez, envía al compilador el mensaje que lo replica (por ejemplo, ReturnNode envía #buildReturn:, LiteralNode envía #buildLiteral:, etc.). De esta forma sólo es necesario agregar un método sencillo en la jerarquía de ParseNode y todo el conocimiento de cómo construir los nodos del árbol de inferencia queda concentrado la clase TiCompiler.

Los nodos correspondientes a las variables son hojas en el árbol sintáctico. Las variables pueden repetirse en varias expresiones, y entonces, sus nodos pueden estar en varias ramas del árbol. Así, los nodos pueden forman un grafo y no, un árbol. Replicar un grafo es más difícil que replicar un árbol, pero el compilador sabe hacerlo correctamente. Cuando aparece una variable por primera vez, la replica y la guarda en un diccionario. La segunda vez,

utiliza la réplica previamente creada. El compilador guarda el diccionario de variables locales en variables, y el de variables globales en sharedVariables.

6.2 Conexión de Expresiones

Los nodos del árbol de inferencia son instancias de la jerarquía TiParseNode (fig. 6.2). Estas instancias tienen el comportamiento especializado para la inferencia y representan distintas expresiones. Las expresiones representadas tienen una variable heredada slot, que se utiliza para acumular y propagar los tipos; así logramos fácilmente, que cada expresión tenga un slot distinto. Luego conectamos estas expresiones y queda definido el template virgen. El template virgen es el árbol de inferencia conectado.

TiParseNode (slot)
TiLiteralNode (value)
TiAssignmentNode (variable expression)
TiReturnNode (expression)
TiMethodNode (receiver arguments temporaries statements)
TiCascadeNode (temporaryVariable receiver messages)
TiMessageNode (receiver selector arguments
cartesianProduct results)

Figura 6.2: Jerarquía parcial de TiParseNode

El árbol de inferencia se conecta enviando un mensaje especial a su raíz (#makeConnections). Cada nodo delega este mensaje a sus hijos (subexpresiones) y luego les pide que se conecten a él (#connectTo:). Así se desencadena la conexión recursiva del árbol.

6.2.1 Expresiones Básicas

Ahora explicaremos la forma de conectar las distintas expresiones que pueden aparecer en los métodos, comenzando por las más sencillas. Hacia el final de esta sección, incluimos las figuras de cada expresión conectada (fig. 6.3).

Expresión Literal

Las expresiones literales son instancias de TiLiteralNode. Estas expresiones corresponden a números, strings, o arrays, que aparecen como literales en el código fuente. El tipo de una expresión literal es el tipo del objeto literal, presente en la variable value. La conexión de esta expresión consiste, simplemente, en inseminarla con el tipo del literal (fig. 6.3(a)). La conexión de

una expresión literal tiene efectos de ejecución. Esto es así, porque luego de inseminar un slot, sus tipos comienzan a fluir; y el flujo de tipos es el equivalente a la ejecución.

Expresión de Asignación

Las expresiones de asignación son instancias de TiAssignmentNode. Al igual que las asignaciones, estas instancias tienen: la variable que será asignada, en el lado izquierdo (variable) y la expresión a asignar, en el lado derecho (expression). Una asignación, introduce un nuevo estado en la variable izquierda, quién a partir de ese momento, tiene el resultado de la expresión derecha. La conexión de una asignación consiste en crear un nuevo slot en la variable izquierda y conectar hacia él, la expresión derecha (fig. 6.3(b)).

Expresión de Retorno

Las expresiones de retorno son instancias de TiReturnNode. Estas expresiones aparecen al final de cada método y dentro de los bloques. La expresión de retorno tiene el tipo de la expresión que contiene (expression) y su conexión es trivial (fig. 6.3(c)).

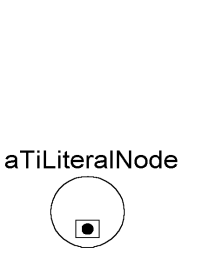
Expresión de Método

Las expresiones de método son instancias de TiMethodNode. Cada una de estas instancias corresponde a la raíz del árbol de inferencia de un método. En su variable statements tienen las sentencias que componen el método. La última de estas sentencias, es una expresión de retorno (insertada por el compilador si es necesario). El tipo del método, es el tipo de esta última expresión, y así lo conectamos (fig. 6.3(d)).

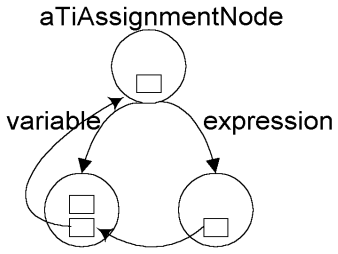
6.2.2 Expresión de Mensaje

Las expresiones de mensaje son instancias de TiMessageNode.[1] Estas instancias tienen el receptor, selector y argumentos del mensaje, en las variables receiver, selector y arguments, respectivamente. La conexión de estas expresiones tiene una dificultad particular: no se puede realizar. Las demás expresiones se conectan dentro de un método, pero las expresiones de mensaje deberían conectarse con *otros* métodos. Resolvemos este problema,

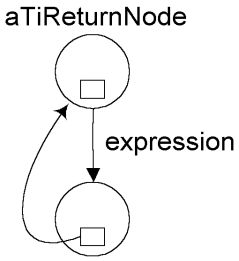
[1] Utilizamos los términos *expresión de mensaje*, *expresión de envío*, o *expresión de envío de mensaje*, en forma indistinta.



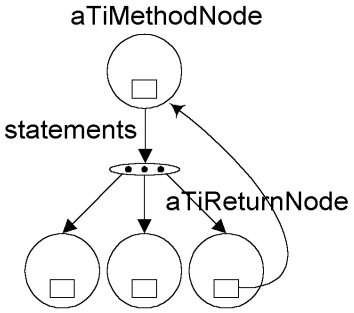
(a) Expresión Literal



(b) Expresión de Asignación



(c) Expresión de Retorno



(d) Expresión de Método

Figura 6.3: Conexión de Expresiones

utilizando la idea propuesta por Ole Agesen en el algoritmo del producto cartesiano [Age95, Age96].

Veamos por qué no podemos conectar la expresión de mensaje. Así como el resultado de evaluar un mensaje depende del objeto receptor, el resultado de analizar un mensaje depende del *tipo receptor*. Las conexiones se realizan durante la construcción de template virgen, y en ese momento, no hay un tipo receptor, hay una *expresión receptora*. La expresión receptora puede tener un conjunto de tipos, pero en el momento de realizar las conexiones, no tiene ninguno. El template no está totalmente construido y entonces no hay inseminación que haya disparado el flujo de tipos. Se debe terminar de construir el template para poder inseminarlo y que los tipos comienzen a fluir. Pero justamente, no podemos terminar de construir el template porque no sabemos cómo conectar la expresión de mensaje. Resumiendo, para conectar la expresión de mensaje necesitamos tipos en la expresión receptora, pero para tener tipos en la expresión receptora, necesitamos conectar la expresión de mensaje. El problema parece ser circular.

El problema anterior nos demuestra la principal dificultad para analizar código Smalltalk, dónde el flujo de datos y el flujo de control se influyen mutuamente (*data-flow* y *control-flow*). Esto se debe a la resolución dinámica de mensajes o vinculación tardía (*dynamic-dispatch* o *late-binding*). Frente a una expresión de mensaje, no podemos saber cuál es el método que se terminará invocado hasta conocer el valor del receptor. Por ejemplo, la expresión de mensaje anObject position podría invocar cualquiera de los diecisiete métodos que lo implementan, y en consecuencia, no podemos saber si obtendremos una posición en una, dos, o tres dimensiones.[2] Por otro lado, se necesita el grafo de control o grafo de llamadas para analizar el flujo de datos (donde este grafo se construye a partir del flujo de control). En Smalltalk, no es trivial construir el grafo de llamadas, y para hacerlo hay que calcular el flujo de datos. No queda otra alternativa que calcular el flujo de datos y el flujo de control en forma simultanea, porque no es posible analizarlos en forma separada [Age96, p.17] [HA96, §2.4] [Gro98, §2.1] [BG93, §3.3] [Bar92].

Conexión Demorada

La expresión de mensaje no se puede conectar hasta conocer los tipos de la expresión receptora. Pero en tiempo de ejecución, sucede algo similar: no se puede vincular un mensaje con un método, hasta conocer el objeto receptor. Esto nos sugiere la solución: imitar lo que sucede en tiempo de ejecución.

[2] Las implementaciones del método #position devuelven un número entero en la jerarquía Stream, un punto en la jerarquía Morph, y un vector de tres coordenadas en la categoría Ballon-3D.

Imitamos la *vinculación demorada* entre mensaje y método, realizando una *conexión demorada* entre ellos. La conexión está demorada hasta que lleguen los tipos a la expresión receptora. Así podemos romper la circularidad del problema.

En el template virgen las expresiones de mensaje quedan sin conectar; recién podemos conectar estas expresiones, luego de inseminar el template. Una vez realizada la inseminación, los tipos comienzan a fluir y llegan a las distintas expresiones. Cuando los tipos llegan a una expresión receptora, es posible analizar el mensaje. Para analizar el mensaje se buscan los métodos, se los insemina y se obtienen los resultados. El intérprete es el responsable de realizar la búsqueda y la inseminación, y esto lo hace, como parte de la simulación del mensaje.

La simulación de un mensaje requiere un solo tipo por argumento, pero las expresiones tienen un *conjuntos de tipos*. Para simular cada mensaje, es necesario obtener diferentes juegos monomórficos de argumentos. Utilizamos el producto cartesiano para calcular todas las posibles combinaciones, y analizar cada juego en forma independiente. Realizamos el producto cartesiano entre los conjuntos de las expresiones y obtenemos los juegos de argumentos monomórficos. Cada juego de argumentos es utilizado por el intérprete para simular un mensaje distinto.

Veamos un ejemplo. Supongamos que analizamos un método que contiene la expresión aString returnArg: aNumber. El método es inseminado y los tipos comienzan a fluir. Fluyen por las conexiones hasta llegar a la expresión de mensaje y ahí se detienen. Supongamos ahora, que el tipo de la expresión aString es {⟨String⟩ ⟨Symbol⟩}, y el tipo de la expresión aNumber es {⟨Float⟩ ⟨SmallInteger⟩}. Estamos listos para simular los envíos de mensajes, pero el intérprete necesita juegos de argumentos monomórficos. Obtenemos estos juegos, mediante el producto cartesiano de los argumentos (fig. 6.4).

aString		**aNumber**		**Argumentos Monomórficos**
				(⟨String⟩ , ⟨Float⟩)
⟨String⟩	×	⟨Float⟩	→	(⟨String⟩ , ⟨SmallInteger⟩)
⟨Symbol⟩		⟨SmallInteger⟩		(⟨Symbol⟩, ⟨Float⟩)
				(⟨Symbol⟩, ⟨SmallInteger⟩)

Figura 6.4: Producto cartesiano de los argumentos

Las expresiones de mensaje son responsables de calcular el producto cartesiano. Las instancias de TiMessageNode mantienen el producto cartesiano en la variable cartesianProduct. Los elementos del producto cartesiano son juegos de argumentos monomórficos, que luego el intérprete utiliza para simular los mensajes. Para cada juego de argumentos, el intérprete busca el

método a analizar, lo insemina y toma los resultados. En la siguiente figura, mostramos esto último como conexiones (fig. 6.5).

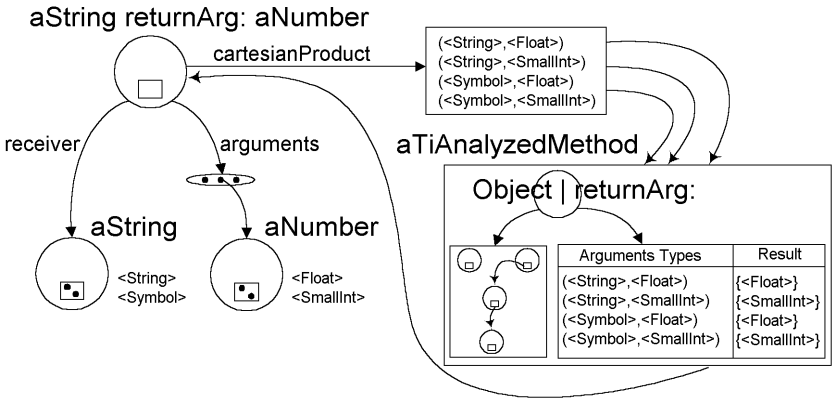


Figura 6.5: Expresión de mensaje aString returnArg: aNumber

.

La idea fundamental detrás del algoritmo del producto cartesiano es particionar el análisis de los mensajes, en un análisis de casos [Age96, p.57]. Los casos son los juegos de argumentos monomórficos, que se utilizan para simular los mensajes. De esta forma, obtenemos una técnica precisa y eficiente, donde cada mensaje se simula en forma independiente. Es precisa, porque los métodos se analizan con argumentos monomórficos y no se mezclan tipos provenientes de distintos mensajes. Es eficiente, porque permite compartir el resultado del análisis de un método, desde distintas expresiones de mensajes.

Implementación

La expresión de mensaje tiene la responsabilidad de calcular el producto cartesiano. Esta es una tarea difícil porque los tipos llegan a las expresiones en distintos momentos. Contrariamente a lo que sucede con los bytecodes, que se ejecutan en un orden preciso, el flujo de tipos es asincrónico. Primero podrían llegar los tipos al argumento de un mensaje, y luego llegar otros tipos a la expresión receptora. Podría suceder también, que luego de calcular el producto cartesiano, arriven nuevos tipos a cualquiera de los argumentos. Por esta razón, el producto cartesiano debe extenderse en forma incremental.

Para extender el producto cartesiano en forma incremental, utilizamos un objeto que actúa de intermediario entre las expresiones de los argumentos y la expresión de mensaje. Estos objetos son instancias de TiArgumentNode y su responsabilidad es informar cuál de los argumentos ha recibido un nuevo tipo. De esta forma, la expresión de mensaje puede extender orde-

nadamente el producto cartesiano. Las instancias de esta clase, tienen la expresión de mensaje en la variable message, la expresión del argumento en actualArgument, la posición del argumento en position, y el slot donde tienen que inseminar los resultados en resultSlot (fig. 6.6).

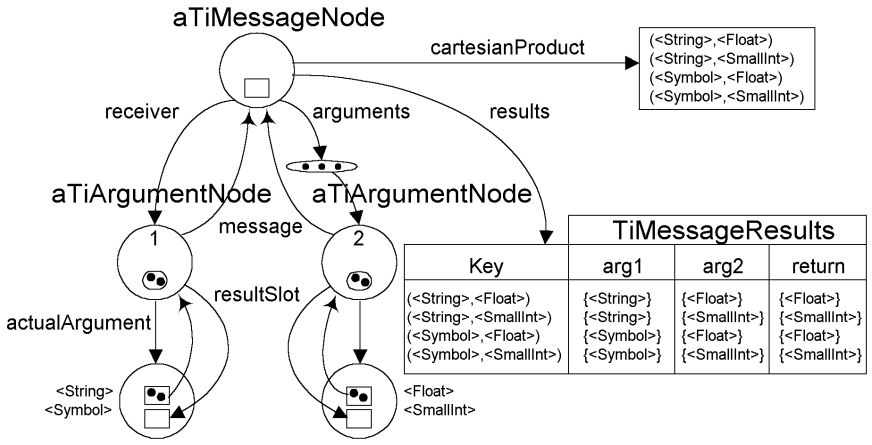


Figura 6.6: Expresión de mensaje conectada

El intérprete toma cada juego de argumentos del producto cartesiano y simula sucesivos mensajes. Luego hay que propagar a la expresión de mensaje, el resultado de cada simulación. El resultado no se propaga con una conexión sino enviando el mensaje #seedResult:. Este mensaje se envía a la expresión de mensaje (un TiMessageNode), pasando como argumento el resultado de la inseminación (un TiMessageResult). En el resultado, está el valor de retorno y los efectos colaterales sobre los argumentos. En la última figura vemos que no hay conexión entre el producto cartesiano y los métodos analizados, ni tampoco entre los resultados y la expresión de mensaje. Estas conexiones no son necesarias porque el resultado de una inseminación no puede variar. En su lugar utilizamos un diccionario donde las claves son las tuplas del producto cartesiano y los valores son los resultados de la inseminación.

Anteriormente dijimos que existía una *conexión demorada* entre la expresión de envío de mensaje y el método. En realidad, la conexión no existe, pero la idea sigue siendo válida. En las demás expresiones los tipos viajan solamente por conexiones y dichas conexiones se pueden trazar de antemano. En el caso de un envío de mensaje tenemos que esperar que lleguen tipos al receptor para saber por dónde continuará el flujo. A medida que hay tráfico se van abriendo nuevos caminos. La apertura de nuevos caminos es encontrar el método que responderá el mensaje. El intérprete se encarga de hacer esta búsqueda (*method lookup*) en base a cada tupla del producto cartesiano.

7. Desafíos del Análisis

Hay ciertas características del lenguaje Smalltalk que dificultan notablemente el análisis. Las primitivas y los bloques de código son los principales ejemplos. Otra dificultad, es mantener los tipos de las variables de instancia sincronizados con los tipos del receptor del mensaje. No podemos ignorar estos casos porque están presentes en gran parte del código. Un análisis que no los contemple, difícilmente pueda ser de utilidad.

7.1 Primitivas

Los métodos compilados pueden responderse en forma primitiva. Pero, ¿qué es una primitiva? Veamos la explicación tomada del *Blue Book*[1] [GR83]:

> Cuando se envía un mensaje, el intérprete Smalltalk normalmente responde ejecutando un método compilado (CompiledMethod). Esto provoca la creación de un nuevo MethodContext para el método compilado y la ejecución de sus bytecodes hasta encontrar uno de retorno. Sin embargo, algunos mensajes pueden ser respondidos en forma primitiva. Las respuestas primitivas son ejecutadas directamente por el intérprete sin la necesidad de crear un nuevo contexto o ejecutar otros bytecodes. Cada respuesta primitiva que el intérprete puede hacer, se describe en una rutina primitiva. La rutina primitiva remueve del stack, al receptor y los argumentos del mensaje, y los reemplaza con el resultado apropiado. Algunas rutinas primitivas tienen otros efectos sobre la memoria o algún dispositivo de hardware. Luego de completar la respuesta primitiva, el intérprete prosigue con la interpretación del bytecode siguiente al bytecode que envió el mensaje y provocó la ejecución de la primitiva.
>
> En cualquier momento de su ejecución, una rutina primitiva podría determinar que no se puede generar una respuesta. Esto podría ser, por ejemplo, debido a que el argumento de un mensaje

[1] El *Blue Book* es el libro por excelencia de Smalltalk. Fue escrito por integrantes del equipo original de Xerox y describe, en lenguaje Smalltalk, cómo implementar la máquina virtual.

> es de clase incorrecta. Esto se llama *falla de primitiva*. Cuando falla una primitiva, el método Smalltalk asociado con el selector y la clase del receptor, se ejecuta como si la primitiva no existiera.

Los métodos compilados pueden responderse en forma primitiva; los métodos analizados pueden analizarse en forma primitiva. Los métodos compilados tienen un encabezado donde se indica el número de primitiva; los métodos analizados tienen una variable primitive, donde se guarda el objeto encargado de analizar la primitiva. Los objetos encargados de analizar las primitivas son instancias de TiPrimitive. Por ejemplo, el método analizado de SmallInteger | = incluye una de estas instancias, para analizar la primitiva de igualdad entre enteros (fig. 7.1). Cuando se analiza un método que tiene una primitiva, primero se intenta su análisis. Si el análisis de la primitiva falla, se continúa con el análisis normal del método.

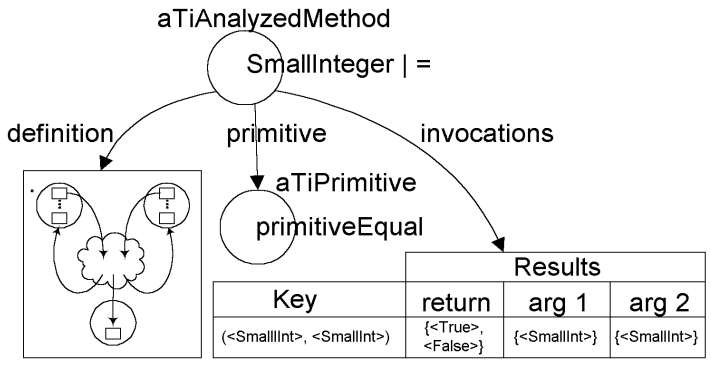


Figura 7.1: Método analizado que incluye una primitiva

La ejecución de una primitiva se realiza sin ejecutar ningún bytecode; el análisis de una primitiva se realiza sin utilizar la propagación de tipos por los slots. Cada instancia de TiPrimitive es responsable de analizar la primitiva indicada en su variable selector. En esa variable se guarda el nombre del método que se utiliza para responder el tipo de la primitiva. Para cada primitiva, entonces, es necesario programar un método que responda su tipo en base a los argumentos de entrada. Estos argumentos se guardan en las variables de instancia receiver y arguments, y son inseminados en forma monomórfica. Así como el programador de la máquina virtual debe programar cada rutina primitiva, nosotros debemos programar cada método para responder el tipo de las primitivas.

Veamos la implementación de la primitiva para verificar la igualdad entre enteros pequeños (primitiveEqual). Esta primitiva devuelve un valor booleano si ambos argumentos pertenecen a la clase SmallInteger. Pero si alguno de los argumentos no es un entero pequeño, la primitiva falla. El análisis de esta primitiva responde el tipo $\{\langle \mathsf{True} \rangle \langle \mathsf{False} \rangle\}$ cuando ambos argumentos

son de la clase SmallInteger, y falla en caso contrario. Esto lo vemos reflejado en nuestra implementación (fig. 7.2).

```
TiPrimitive | primitiveEqual
    ↑ ( self matchReceiver: SmallInteger arguments: {SmallInteger})
        ifTrue: [ TiPrimitiveResult new
                    setReceiver: self receiverTypeSet
                    arguments: self argumentTypeSets;
                    addReturn: True createType;
                    addReturn: False createType]
        ifFalse: [ TiPrimitiveResult failed ]
```

Figura 7.2: Análisis de la primitiva de igualdad entre enteros

7.1.1 Estado de Implementación

En Squeak existen 235 primitivas distintas. Hay primitivas para los números enteros y de punto flotante, para el acceso a variables indexadas y streams, para el manejo de memoria, para la evaluación de bloques y envío indirecto de mensajes, para manejo de archivos, gráficos y sonido, y para el soporte de red. Nosotros implementamos 43 primitivas; faltan implementar otras 192 primitivas más.

Implementamos las primitivas bajo demanda. A medida que intentamos el análisis de nuevas expresiones, nos encontramos con nuevas primitivas y las implementamos. Las primitivas implementadas están repartidas en varias categorías: aritmética entera y de punto flotante, creación de nuevos objetos, acceso a variables indexadas, evaluación de bloques y envío indirecto de mensajes (primitivePerform). Sin embargo, no implementamos las primitivas para el manejo de archivos, ni para el soporte de gráficos, sonidos, o red. Sabemos cómo implementar algunas de estas, pero existen otras, que todavía falta pensar en profundidad.

En Squeak, la implementación de cada primitiva es un trabajo arduo. Hay que comprender la implementación de la primitiva en el intérprete de la máquina virtual (instancia de Interpreter), y luego expresar esa misma lógica, en el dominio de los tipos concretos. Comprender el funcionamiento de una primitiva no es algo inmediato, pues es necesario conocer los detalles del intérprete. Por ejemplo, la primitiva de igualdad que presentamos anteriormente, se implementa de forma totalmente diferente en el intérprete Smalltalk (fig. 7.3). Sin embargo, también es verdad que dentro de una categoría, muchas primitivas se parecen entre sí. Por ejemplo, no hay demasiada diferencia entre sumar, restar o multiplicar números enteros.

```
Interpreter | primitiveEqual
    | integerReceiver integerArgument result |
    integerArgument ← self popStack.
    integerReceiver ← self popStack.
    result ← self compare31or32Bits: integerReceiver
                  equal: integerArgument.
    self checkBooleanResult: result
```

Figura 7.3: Implementación de la primitiva de igualdad entre enteros

7.1.2 Características Generales

Hay ciertas características que comparten todas las primitivas y conviene destacar. En tiempo de ejecución una primitiva puede fallar por diversas causas. Algunas causas son inherentes al ambiente de ejecución (se acabó la memoria, error al escribir un archivo, etc.) y no tiene sentido tratar de analizar los errores correspondientes. Simplemente asumimos que corremos en un ambiente ideal con recursos infalibles e infinitos. Esto debemos tenerlo en cuenta, si luego queremos realizar extracción de aplicaciones, ya que no debemos olvidar de extraer aquellas clases que se utilizan para informar los errores.

Algunas primitivas están diseñadas para trabajar sólo con ciertas clases de argumentos. Si le pasamos un argumento de una clase no esperada, simplemente falla (la suma de enteros, trabaja sólo con enteros). En tiempo de análisis tenemos la información necesaria para hacer esta clase de chequeo. Solamente hace falta verificar que los tipos de los argumentos sean los esperados por la primitiva.

Aún cuando los argumentos sean de la clase esperada, la primitiva puede fallar debido a los valores de dichos argumentos. Por ejemplo, la primitiva de división entera falla si el divisor es cero. En tiempo de análisis los argumentos para evaluar una primitiva son tipos. Los tipos representan muchos objetos, con lo cual no sabemos el valor que podrá tomar en tiempo de ejecución. En el caso de la división podemos saber que el tipo es ⟨SmallInteger⟩, pero no podemos saber si su valor será cero. En estos caso tomamos una actitud conservadora: suponemos que la primitiva tendrá éxito y combinamos el resultado con la suposición de que fallará.

Hay ciertas primitivas que pueden trabajar con cualquier clase de argumento y nunca fallan. La primitiva para verificar la equivalencia de objetos, o la primitiva para averiguar la clase, son algunos ejemplos (primitiveEquivalent, primitiveClass). En estos casos, no es necesario analizar el método asociado y entonces no construimos su template virgen (porque nunca se utilizará).

7.1.3 Casos Especiales

Las primitivas son una fuente de casos especiales, y muchas de ellas plantean importantes desafíos. Cuando agregamos las primitivas de creación de objetos, fue necesario pensar cómo crear los nuevos tipos. La primitiva de acceso a variables indexadas, nos obligó a pensar en los errores de acceso fuera de rango. Para implementar la primitiva de envío indirecto de mensajes, tuvimos que modificar la definición de tipo para el caso de los símbolos. Y así, con muchos otros casos. Consideramos necesario un rediseño de las primitivas para simplificar la implementación de cada una de ellas, sin favorecer alguna categoría en especial. Dejamos esto como trabajo futuro.

Creación de Tipos

Un método puede hacer referencia a objetos presentes en la imagen, mediante el uso de variables globales o expresiones literales. Cuando el análisis encuentra alguna de estas expresiones, se las insemina con el tipo del objeto. El tipo del objeto se obtiene, enviándole el mensaje #type. Por ejemplo, la expresión $a type devuelve el tipo del carácter 'a' y Smalltalk type devuelve el tipo de la variable global Smalltalk.

Pero normalmente, un método hace referencia a objetos que todavía no existen en la imagen. Esos objetos son creados *durante* la ejecución y recién existirán en el futuro. La creación de un nuevo objeto involucra la evaluación de alguna primitiva (primitiveNew, primitiveClone, primitiveMakePoint, etc.). De la misma forma, la creación de un nuevo tipo involucra el análisis de una primitiva.

La primitiva donde se crean la mayor cantidad de objetos, y en consecuencia, la mayor cantidad de tipos, es primitiveNew. Así como los objetos se crean con las variables de instancia en nil, los tipos se crean con las variables de instancia en ⟨UndefinedObject⟩. Para crear el tipo de una determinada clase, le enviamos el mensaje #createType. Por ejemplo, Character createType devuelve el tipo ⟨Character value: ⟨UndefinedObject⟩⟩. Esto mismo lo podemos observar en el código de la primitiva de igualdad, donde se crean los tipos booleanos (fig. 7.2).

Envío Indirecto de Mensajes

En Smalltalk es posible enviar un mensaje en forma indirecta utilizando el mensaje #perform:, o alguna de sus variantes. Por ejemplo, la evaluación de la expresión 1 asFloat es totalmente equivalente a la evaluación de la expresión 1 perform: #asFloat. Sin embargo, no podemos analizar esta última expresión sin cambiar la definición de tipo que venimos utilizando.

El problema es que en tiempo de análisis no está disponible el valor del selector, y entonces, no es posible saber cuál es el método por donde continuar el análisis. Al simular el mensaje ⟨SmallInteger⟩ perform: ⟨Symbol⟩, el intérprete abstracto no puede continuar, porque no tiene el valor del selector para realizar la búsqueda del método. El símbolo utilizado como selector podría #asFloat, o #asCharacter, o cualquier otro símbolo.

Squeak utiliza intensivamente el #perform: en tres lugares distintos: 1) en el mecanismo de dependencias, 2) en los objetos *pluggables*, y 3) en el mecanismo de coerción aritmética. Este último mecanismo, nos obligó a encontrar una solución, pues de no hacerlo, no sería posible analizar expresiones tan sencillas como 3.0 + 4.

Solucionamos este problema, cambiando la definición de tipo para el caso de los símbolos. Con la nueva definición, el propio símbolo es parte del tipo; por ejemplo, el tipo de #asFloat es ⟨Symbol #asFloat⟩ y el tipo de #asCharacter es ⟨Symbol #asCharacter⟩. Esto nos permite analizar la primitiva primitivePerform en aquellos casos que el símbolo se encuentre en el literal de algún método. Si por el contrario, el símbolo se lee de un archivo o se le pregunta al usuario, no vemos solución posible. La implementación de primitivePerform es uno de los casos que deberían simplificarse con un rediseño (fig. 7.4).

```
TiPrimitive | primitivePerform
    | selectorType messageResult |
    selectorType ← arguments first.
    (selectorType exactClass == Symbol)
        ifFalse: [↑ TiPrimitiveResult failed].
    messageResult ← TiSystem current interpreter
                        sendMessage: selectorType symbol
                        withArgs: self messageArguments.
    ↑ TiPrimitiveResult new
        setReceiver: messageResult receiver
        arguments: (self argumentTypeSetsFrom: messageResult)
        return: messageResult return
```

Figura 7.4: Implementación de primitivePerform

7.2 Bloques

En Smalltalk, los bloques de código se utilizan para implementar las diferentes estructuras de control (ciclos, condicionales, manejo de excepciones, etc.). Y de esta forma, están presentes en gran parte de los métodos. Los bloques de código no son construcciones sintácticas: son objetos. Estos objetos tienen ciertas características que complican el análisis considerablemente: la evaluación en dos etapas, el acceso a variables de su entorno léxico y los retornos no locales, son las principales dificultades. El análisis de los bloques debe tener en cuenta estos aspectos y se debe realizar en forma precisa y eficiente.

7.2.1 Definición y Evaluación

Los bloques se evalúan en dos etapas. La primera etapa es la definición y la segunda etapa es la evaluación. La definición es el conjunto de sentencias que vemos encerradas entre corchetes. La evaluación se provoca, al enviarle al bloque el mensaje #value. Normalmente, un bloque se define en un método y se evalúa en otro método distinto. Sin embargo, nada impide definir y evaluar un bloque en el mismo método (por ejemplo, [5 asFloat] value).

Los bloques son objetos creados por el intérprete Smalltalk durante la ejecución de los métodos compilados. En el código fuente, tenemos la impresión de estar viendo bloques, pero en realidad vemos sus definiciones. Los bloques son instancias de BlockContext y se utilizan para guardar el estado de ejecución de las sentencias que lo componen. Estas instancias sólo existen durante la ejecución.

En el código fuente, la definición de un bloque está compuesta por las sentencias que se encuentran entre sus delimitadores (los corchetes). A partir de esas sentencias, se generan los bytecodes que forman la definición del bloque en el método compilado. Las sentencias de un bloque se encuentran en el código fuente de un método; los bytecodes del bloque se encuentran en el método compilado correspondiente. Llegado el momento de la ejecución, el BlockContext se crea con su instrucción inicial apuntando al primer bytecode de su definición (fig. 7.5).

El bloque se evalúa cuando recibe el mensaje #value. Como respuesta se ejecuta el método BlockContext | value que termina llamando a la primitiva de evaluación (primitiveValue). Esta primitiva activa el contexto de ejecución y los bytecodes del bloque comienzan a ser ejecutados. Un bloque, también puede recibir argumentos y para evaluarlo se requieren los argumentos actuales. Por ejemplo, el bloque succ ← [:x | x + 1] recibe un solo argumento y se lo puede evaluar con las expresiones succ value: 1 o succ value: 1.0 (obteniendo 2 o 2.0 respectivamente).

Las sentencias de un bloque pueden hacer referencia a variables de su entorno léxico. El entorno léxico de un bloque está compuesto por los bloques que lo encierran y el método donde se encuentra definido. Por ejemplo, en un bloque se puede hacer referencia a la pseudo-variable self, que tienen todos los método. Durante la ejecución, los bloques y los métodos, tienen un contexto donde se guarda el estado de la ejecución (instancias de BlockContext y MethodContext, respectivamente). Las instancias de BlockContext acceden a los otros contextos, utilizando la variable home (fig. 7.5).

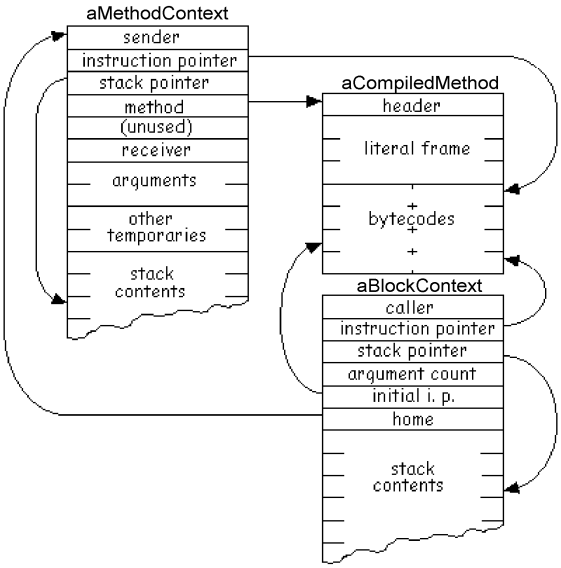


Figura 7.5: Bloque en tiempo de ejecución (figura extraída del *Blue Book*)

Una complicación adicional de los bloques son los retornos no locales (*non-local returns*). Un retorno no local, es la expresión de retorno que puede aparecer dentro de un bloque (por ejemplo, [↑ self]). Cuando se evalúa un bloque y se encuentra una expresión de retorno, el control se devuelve al método donde está definido el bloque (independientemente si está anidado dentro de otro bloque). En Smalltalk, los retornos no locales se utilizan en forma extensiva.

7.2.2 Análisis e Implementación

Implementamos el análisis de los bloques, agregando clases en varias jerarquías. Los bloques tienen un tipo especial, un resultado especial y su evaluación también es especial. Así como tenemos objetos responsables de analizar los métodos, también tenemos objetos responsables de analizar los bloques. A continuación, presentamos las diversas clases que intervienen en el análisis de un bloque (fig. 7.6).

TiType ()
    TiBlockType (blockNode)
TiParseNode (slot)
    TiBlockNode (nonLocalReturnSlot analyzedBlock)
    TiBlockDefinitionNode (arguments statements)
TiAnalyzedBlock (definition evaluations)
TiResult (return arguments)
    TiBlockResult (nonLocalReturn)

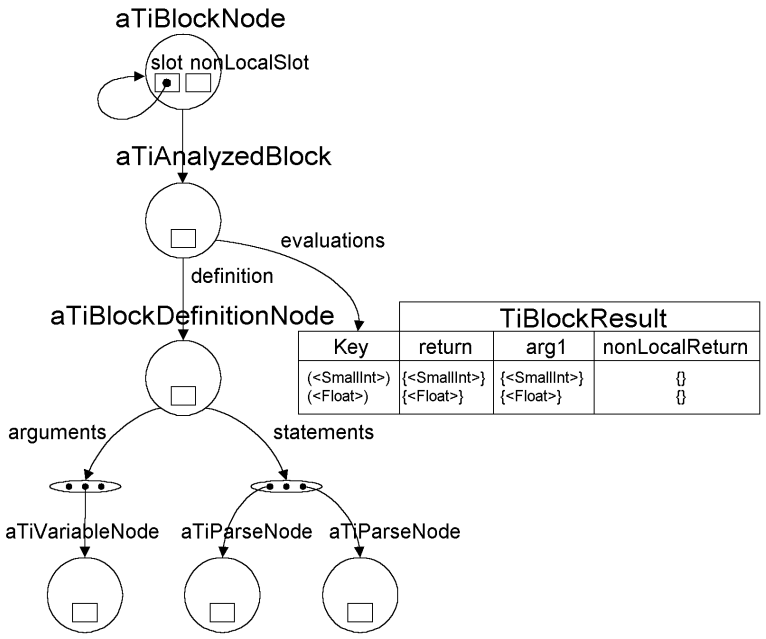
Figura 7.6: Clases que intervienen en el análisis de un bloque

Un bloque se analiza cuando recibe el mensaje de evaluación. El análisis de un bloque consiste en conectar sus sentencias, inseminar los argumentos y obtener los resultados. Pero el mensaje de evaluación, normalmente se envía en un método distinto donde se encuentra la definición del bloque (por ejemplo, los bloques que son argumentos de un condicional, se terminan evaluando en el método #ifTrue:ifFalse: de la clase True o False). Esto requiere que las inseminaciones se puedan realizar en forma *remota*; es decir, inseminar en un método, las expresiones de *otro* método.

Los bloques son objetos y como tales, tienen un tipo. En general, los objetos de la misma clase comparten el mismo tipo, pero en el caso de los bloques, debemos hacer una excepción. Asignamos a cada bloque un tipo distinto que depende de su definición (es decir, de las sentencias que lo componen). De esta forma el tipo del bloque [5 asFloat] es ⟨BlockContext [5 asFloat]⟩ y el tipo de [:x | x + 1] es ⟨BlockContext [:x | x + 1]⟩. Sin esta distinción, no tendríamos forma de analizar las sentencias del bloque.

El tipo de un bloque es instancia de TiBlockType. Estas instancias guardan en la variable blockNode una referencia a la expresión de bloque. Estas expresiones en el árbol de inferencia, no son réplicas exactas de sus correspondientes en el árbol sintáctico. Cada expresión o nodo de bloque en el árbol sintáctico se replica con tres nodos en el árbol de inferencia: un TiBlockNode, un TiAnalyzedBlock y un TiBlockDefinitionNode (fig. 7.7).

En el árbol sintáctico, las instancias de BlockNode tienen los argumentos y las sentencias del bloque. Las réplicas de estos nodos son instancias de TiBlockNode y solamente tienen la variable analyzedBlock para guardar el bloque analizado. El bloque analizado es instancia de TiAnalyzedBlock y se utiliza para guardar la definición y las evaluaciones del bloque. Cada definición es instancia de TiBlockDefinitionNode, que finalmente tiene los argumentos y las sentencias del bloque (fig. 7.7).



El tipo del bloque tiene una referencia hacia su expresión. Lo dibujamos con un punto de donde parte una flecha hacia su expresión.

Figura 7.7: Expresión de Bloque conectada

Simulación de la Evaluación

La conexión de una expresión de bloque consiste en conectar las sentencias que lo definen y luego inseminar dicha expresión con el tipo especial del bloque. El tipo del bloque es especial porque tiene una referencia hacia su expresión en el árbol de inferencia. Esta referencia del tipo hacia el nodo del bloque, nos permite realizar inseminaciones remotas y solucionar el problema de la evaluación en dos etapas. El tipo del bloque se propaga como cualquier otro tipo, y recién en el momento de analizar el mensaje #value, simulamos la evaluación del bloque.

El responsable de simular la evaluación de un bloque es instancia de TiAnalyzedMethod. Estas instancias guardan en la variable evaluations, un diccionario con los resultados de cada evaluación y la definición del bloque en la variable definition. El diccionario de evaluaciones lo mantienen para no repetir dos veces el mismo análisis. Las claves del diccionario son los argumentos que recibe el bloque y los valores son instancias de TiBlockResult. Para realizar un nuevo análisis, el bloque analizado copia la definición, insemina los argumentos y obtiene el resultado (un mecanismo muy similar al que utiliza el método analizado).

Retornos no locales

El resultado de analizar un bloque es instancia de TiBlockResult. Estas instancias pueden guardar el resultado del retorno no local en la variable nonLocalReturn. Un bloque tiene un retorno no local cuando su última sentencia es una expresión de retorno. En estos casos, conectamos la última sentencia del bloque hacia el slot de su definición. De esta forma, el slot del TiBlockDefinitionNode tiene el tipo del retorno no local.

Por otro lado, el resultado de evaluar aquellos bloques sin retornos no locales, está dado por su última sentencia; de la misma forma, el análisis de estos bloques también está dado por su última sentencia. La definición del bloque (instancia de TiBlockDefinitionNode) es responsable de crear el resultado con los tipos apropiados (ie. una instancia de TiBlockResult con el resultado de la última sentencia en return y el retorno no local en nonLocalReturn).

Acceso al Entorno Léxico

El mecanismo que utilizamos para analizar los bloques tiene una falla. Dentro de las sentencias del bloque puede haber referencias a variables de su entorno léxico (definidas en el método o algún bloque que lo encierra). Estas variables actúan como pseudo-argumentos del bloque pues influyen en su ejecución. Deberíamos identificar estas referencias como pseudo-argumentos del bloque y tenerlas en cuenta en el momento de la inseminación.

Es nuestra máxima prioridad como futuro trabajo, implementar la solución a este problema, pues actualmente no podemos analizar en forma adecuada los bloques donde se envían mensajes a self.

Primitiva de Evaluación

En la siguiente figura, presentamos el método que analiza la primitiva de evaluación y los métodos más importantes que intervienen en su análisis (fig. 7.8). Esta primitiva verifica que el receptor sea un bloque y la cantidad de argumentos sea adecuada. Luego se simula la evaluación del bloque enviando al receptor de la primitiva el mensaje #evaluateWith:. Este receptor es instancia de TiBlockType y delega el mensaje al TiBlockNode, quien a su vez lo delega al TiAnalyzedBlock. El bloque analizado, verifica si no realizó el mismo análisis en otra oportunidad, y si no lo hizo, copia la definición del bloque e insemina los argumentos. La definición del bloque insemina los argumentos iniciales y luego crea el resultado de este análisis (instancia de TiBlockResult). Este resultado se retorna a la primitiva, donde se utiliza para simular el retorno no local. Para ello, se envía al receptor de

la primitiva el mensaje #seedNonLocalResult. El receptor delega este mensaje al TiBlockNode, que finalmente insemina los tipos en su slot especial (nonLocalReturnSlot).

```
TiPrimitive | primitiveValue
    | result |
    ↑ ( receiver isBlock and: [arguments size = receiver argumentCount])
        ifTrue:
            [ result ← receiver evaluateWith: arguments.
             receiver seedNonLocalReturn: result nonLocalReturn.
             self createResultReceiver: self receiverTypeSet
                  arguments: result arguments
                  return: result return]
        ifFalse: [ self primitiveFail ]

TiAnalyzedBlock | evaluateWith: argTypesArray
    ↑ evaluations at: argTypesArray
        ifAbsentPut:
            [ definition veryDeepCopy seedArguments: argTypesArray]

TiBlockDefinitionNode | seedArguments: argTypesArray
    arguments with: argTypesArray do: [:arg :type | arg seedInitial: type].
    ↑ TiBlockResult new
        setNonLocalReturn: slot typeSet
        arguments: self argumentTypeSets
        return: statements last typeSet

TiBlockNode | seedNonLocalReturn: aTypeSet
    nonLocalReturnSlot seedAll: aTypeSet
```

Figura 7.8: Análisis de la primitiva de evaluación

7.3 Variables

En Smalltalk existen tres tipos de variables: las variables globales, las variables locales y las variables de instancia. En nuestro diseño, las variables son objetos de la jerarquía de TiVariable. Las variables globales son instancias de TiSharedVariable, las locales de TiRegularVariable y las de instancia de TiInstanceVariable, pero además, tenemos las instancias de TiReceiverVariable que representan al objeto receptor del mensaje (self). El objeto receptor conoce sus variables de instancia y las accede por medio de la variable instVars. En forma similar, las variables de instancia conocen al receptor y lo acceden a través de la variable receiverNode (fig. 7.9).

TiVariable (name)
TiRegularVariable (slots)
TiInstanceVariable (receiverNode)
TiReceiverVariable (instVars)
TiSharedVariable (slot)

Figura 7.9: Jerarquía de TiVariable

En la sección §5.3 presentamos el concepto de estado para las variables locales. Allí explicamos que una asignación introduce un nuevo estado en la variable asignada y que cada estado se representa con un slot distinto. Ahora explicamos que las variables locales y los argumentos formales se representan con instancias de TiRegularVariable, y estas instancias guardan los estados en la variable slots.

Las variables globales son argumentos implícitos de cada método. En otras palabras, cualquier método puede hacer referencia a una variable global. Las variables globales deberían pasarse como argumentos en el análisis de cada método. Decidimos no hacer esto porque consideramos que sería sumamente ineficiente. En nuestra implementación, las variables globales tienen un solo slot.

7.3.1 Variables de Instancia

Dada que nuestra implementación está basada en el trabajo de Agesen, es importante tener en cuenta las diferencias que existen entre Smalltalk y Self. Aunque en términos generales, las ideas de Agesen son aplicables en ambos lenguajes, encontramos una diferencia substancial en el tratamiento de las variables de instancia.

En Smalltalk, los objetos se crean con sus variables de instancia incializadas en nil. En Self, los objetos se crean por clonación y sus variables de instancias toman el valor del objeto clonado. Ole Agesen utiliza una definición de tipo basada en puntos de clonación (*allocation points*) y no contempla los cambios de tipo que pueden sufrir las variables de instancia [Age96, §4.3.1]. Esta idea resulta inadecuada para Smalltalk porque es frecuente el cambio de tipo en las variables de instancia (el ejemplo más evidente es el método #initialize). Si aplicáramos taxativamente la idea de Agesen, la mayoría de los tipos tendrían ⟨UndefinedObject⟩ en sus variables de instancia, pues así fueron creados. Nosotros implementamos lo propuesto por Agesen como trabajo futuro: tomar en cuenta los cambios de estado en las variables de instancia para generar nuevos tipos [Age96, §4.3.3].

Implementación

La asignación a una variable de instancia altera al objeto receptor del mensaje pues cambia su estructura interna. Anteriormente presentamos este problema y explicamos que para resolverlo se debe agregar un slot a la variable de instancia, otro slot a la variable self y agregar un mecanismo para sincronizar los tipos (§5.3.1, p.44). Pero el objeto receptor también puede alterarse cuando participa como argumento de algún mensaje (ya que puede sufrir un efecto colateral). En esos casos, se debe agregar un slot a la variable self, un slot a cada variable de instancia y utilizar un mecanismo para distribuir el tipo de self entre las variables de instancia.

Mantener la sincronización entre receptor y variables de instancia, no es tarea sencilla. Para resolver este problema, ideamos un mecanismo que utiliza slots especiales. El receptor utiliza instancias de TiReceiverSlot y la variable de instancia de TiInstanceSlot (fig. 7.10).

```
TiSlot (typeSet connections)
    TiReceiverSlot (previousSlot instanceSlot)
TiInstanceSlot (name typeSet receiverSlot variableSlot)
```

Figura 7.10: Clases para representar los slots

Los slots especiales se crean en los lugares donde se necesita una sincronización y se los conecta en forma apropiada. Veamos al ejemplo Point | changeX donde se asigna 'hello' a la variable de instancia x, para que podamos ilustrar como se conectan sus slots (fig. 7.11).[2]

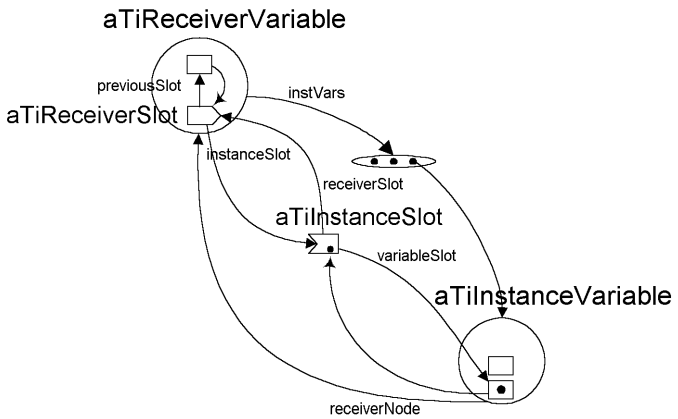


Figura 7.11: Sincronización de variables de instancia

La conexión de la expresión x ← 'hello' agrega tres slots: un slot normal en la variable x, un TiInstanceSlot y un TiReceiverSlot. El slot normal se insemina

[2] Point | changeX se define en la Figura 5.10 (p.44).

con el tipo del literal y se conecta con el TiInstanceSlot, a quien le propaga su tipo. El TiInstanceSlot se conecta con el TiReceiverSlot y también propaga su tipo (junto con el nombre de la variable de instancia). El TiReceiverSlot no acepta el tipo propagado pues no alcanza para construir el tipo de un punto (falta una coordenada). A continuación presentamos los métodos que intervienen en la creación y conexión de un nuevo slot, para el caso de una variable de instancia (fig. 7.12).

```
TiInstanceVariable | newSlot
    | newSlot |
    newSlot ← super newSlot.
    receiverNode notNil ifTrue: [receiverNode newSlotIn: self].
    ↑ newSlot

TiReceiverVariable | newSlotIn: anInstanceVariable
    ↑ slots add: ( TiReceiverSlot new
                    fromInstanceVariable: anInstanceVariable
                    previous: slots last )

TiReceiverSlot | fromInstanceVariable: aTiInstanceVar previous: aSlot
    instanceSlot ← TiInstanceSlot new
                    setName: aTiInstanceVar name
                    receiverSlot: self
                    variableSlot: aTiInstanceVar currentSlot .
    previousSlot ← aSlot.
    aSlot connectTo: self

TiInstanceSlot | setName: aSymbol receiverSlot: aReceiverSlot variableSlot: currentSlot
    name ← aSymbol.
    receiverSlot ← aReceiverSlot.
    variableSlot ← currentSlot.
    currentSlot connectTo: self.
```

Figura 7.12: Creación y conexión de los slots especiales

El slot de la instancia propaga sus tipos al slot del receptor. Si el receptor ya contiene tipos, entonces los combina generando otros nuevos. Si por el contrario, el receptor no tiene tipos, entonces quedan demorados en el slot de la instancia. Cuando llega un nuevo tipo al receptor, se los combina con todos aquellos están demorados.

Por otro lado, la inseminación inicial de un método consiste en enviar el mensaje #seedInitial: a cada uno de los argumentos. La variable del receptor es responsable de distribuir el tipo entre sus variable de instancia.

A continuación presentamos los principales métodos que intervienen en la propagación de los tipos por los slots especiales (fig. 7.13).

```
TiInstanceSlot | seed: aType
    self typeSet add: aType ifPresent: [↑ self].
    receiverSlot seed: aType fromVariable: name.

TiReceiverSlot | seed: aType fromVariable: aSymbol
    self basicSeedAll:
        (previousSlot typeSet
            collect: [:type | type atVariable: aSymbol putType: aType])

TiReceiverSlot | seed: aType
    | name |
    name ← instanceSlot name.
    self basicSeedAll:
        (instanceSlot typeSet
            collect: [:type | aType atVariable: name putType: type]).

TiObjectType | atVariable: aSymbol putType: aType
    ↑ (self instVarTypeAt: aSymbol) = aType
        ifTrue: [self]
        ifFalse: [TiSystem current
                    addTypeBecomingIntoExisting:
                        (self copy
                            instVarAt: aSymbol
                            putType: aType)]

TiReceiverVarible | seedInitial: aType
    super seedInitial: aType.
    instVars isEmptyOrNil
        ifFalse:
            [instVars do:
                [:each | each seedInitial:
                            (aType instVarTypeAt: each name)]]
```

Figura 7.13: Propagación de tipos en los slots especiales

8. Conclusiones

Es un largo anhelo de la comunidad Smalltalk, extraer información de tipos en forma automática. Esta información podría ser utilizada para diversos propósitos: extraer aplicaciones, realizar optimizaciones de código, detectar mensajes que no serán entendidos, etc. Sin embargo, la inferencia de tipos en Smalltalk es un problema de difícil resolución que aún continúa abierto. Aunque algunos investigadores afirman que esto es posible, en la práctica no hay ninguna implementación precisa y eficiente.

En este trabajo presentamos el diseño e implementación de un inferenciador de tipos concretos en Squeak. El inferenciador presenta un interesante estado de evolución que nos permite deducir tipos concretos no sólo para expresiones de juguete, sino también para algunas expresiones que podrían aparecer en la distribución estándar de Squeak. Puede, por ejemplo, analizar la expresión Display boundingBox isTall y deducir que su tipo es {⟨True⟩ ⟨False⟩} (analizando para ello, 17 métodos y 25 invocaciones). Sin embargo, aunque podemos inferir el tipo de la expresión 3.0 + 4, no podemos hacer lo mismo con la expresión 3 + 4.0 (debido al problema oportunamente explicado en el análisis de los bloques).

Inferir tipos para cualquier expresión de Smalltalk es un objetivo muy ambicioso. Si bien el lenguaje es mínimo y sencillo, las nociones de *objeto* y *mensaje* permiten lograr construcciones que en otros lenguajes son especiales. Por ejemplo, en Smalltalk no hay una sentencia como el `switch` de C, pues el mismo efecto se puede lograr utilizando una tabla de selectores y haciendo #perform: sobre ellos. Los casos especiales no están en la definición del lenguaje sino en la riqueza y variedad de su uso. El desafío es analizar el cuerpo de código existente.

Este trabajo sintetiza nuestro proceso de aprendizaje sobre la inferencia de tipos en Smalltalk. Este proceso estuvo compuesto por sucesivas iteraciones donde paulatinamente fuimos mejorando el diseño. Cada mejora nos permitió inferir tipos sobre un conjunto más grande de expresiones y al mismo tiempo, encontrar sus limitaciones. A partir de esta limitaciones, encontramos casos concretos que nos hicieron repensar parte del diseño. De esta forma, comprendimos cada vez con mayor profundidad, los problemas fundamentales de la inferencia de tipos. Prevemos varias iteraciones más en este proceso incremental, hasta lograr un inferenciador práctico y preciso.

Comenzamos este trabajo distinguiendo las nociones de *tipo concreto* y *tipo abstracto*. Esta distinción es importante porque son conceptos diametralmente opuestos, y generalmente confundidos, bajo el término de *tipo*. En Smalltalk no hay concepto ni definición de tipo, y en consecuencia, diferentes personas se refieren a diferentes conceptos, utilizando el mismo término. La inferencia de tipos concretos enfrenta el mismo problema que la construcción del grafo de llamadas: resolver en forma estática la vinculación dinámica. De esta forma, conociendo Infiriendo los tipos concretos se puede construir el grafo de llamadas y viceversa.

Este trabajo está basado en la tesis de doctorado de Ole Agesen [Age96]. Allí presenta la inferencia de tipos como un problema de análisis combinado entre el flujo de control y el flujo de datos. Allí además analiza y compara los trabajos anteriores desde una perspectiva integradora. Su aporte teórico es un algoritmo que permite analizar los envíos de mensaje en forma precisa y eficiente. Su aporte práctico es un inferenciador de tipos concretos para el lenguaje Self y una herramienta para extraer aplicaciones.

En este trabajo no solamente resolvimos el paso de Self a Squeak, sino que también solucionamos temas que quedaron abiertos en la tesis de Agesen. Las variables de instancia son consideradas como parte integral del tipo y sus asignaciones son reflejadas con *cambios de tipo*. Esta profunda diferencia con el trabajo de Agesen nos permite tratar el polimorfismo de datos con mayor precisión. Además, extendimos la noción de estado a las variables de instancia y argumentos de mensaje, simplificamos la forma de detectar la recursividad y extendimos la definición de tipo en el caso de los símbolos (para poder analizar código que utiliza el #perform:).

Realizamos la inferencia de tipos mediante la interpretación abstracta sobre el dominio de los tipos concretos: esto quiere decir que simulamos la ejecución de un programa utilizando tipos en lugar de valores. De esta forma, se manifiesta claramente el paralelismo entre ejecución y análisis. Pero no solamente utilizamos el paralelismo para razonar y explicar los problemas, sino que también lo reflejamos en el diseño e implementación del inferenciador.

La dificultad principal para analizar código Smalltalk es resolver en forma estática la vinculación dinámica. Nosotros utilizamos la idea propuesta por Agesen en el algoritmo del producto cartesiano, realizando así, un análisis de casos en cada envío de mensaje. Este algoritmo es preciso porque no mezcla resultados de diferentes análisis y es eficiente porque evita repetirlos. Pero la vinculación dinámica no es la única dificultad para inferir tipos concretos. En un ambiente real como Squeak también se deben resolver los problemas planteados por los bloques, las primitivas, los cambios de tipo y otros más específicos que surgen al construir un simulador de ejecuciones.

Este trabajo nos permite dar respuestas, pero también plantea interesantes preguntas. Cómo continuar el análisis luego de encontrar un mensaje que

no será entendido o luego de detectar un mensaje recursivo, son algunas de ellas. Aunque tenemos ideas para resolver estos problemas, es necesario validarlas en la práctica.

Trabajo Futuro

En los últimos capítulos propusimos varias ideas como trabajo futuro. De todas ellas, la más importante es implementar como pseudo-argumentos los accesos a las variables del contexto léxico de un bloque. Esto nos permitiría aumentar considerablemente el conjunto de expresiones tipables.

Sin embargo, tenemos una idea más ambiciosa: cambiar el mecanismo de propagación de tipos. Queremos eliminar los slots y las conexiones para delegar esa responsabilidad en el intérprete abstracto. Esto simplificaría el diseño general y el seguimiento de errores. Los slots y las conexiones demostraron ser un mecanismo muy poderoso, pero al propagar los tipos en forma asincrónica, dificultan su seguimiento. Utilizando la noción de estados no se necesita la propagación asincrónica porque no se vuelven a analizar expresiones anteriores, y entonces, es posible simular la ejecución con un intérprete abstracto.

Una idea interesante es extender el concepto de tipo para todos los literales (incluyendo los arrays) de forma similar a lo realizado con los símbolos. Esto combinado con el concepto de cómputo inofensivo, permitiría en muchos casos mejorar la precisión. El cómputo inofensivo es aquel que no produce efectos colaterales ni cambios de tipo; por ejemplo, las operaciones aritméticas, la concatenación de strings, el acceso a un elemento de un array, etc.

Un interrogante abierto es cómo reflejar con mayor precisión el tipo de las colecciones. Internamente las colecciones tienen arreglos inicializados en nil, y un par de índices que marcan el comienzo y fin; pero el acceso a una colección nunca retorna nil (a menos que se guarde explícitamente). Sin embargo, el tipo concreto de una colección muestra la estructura interna del objeto y el análisis de un acceso puede retornar ⟨UndefinedObject⟩. Esto es así porque durante el análisis no están los valores correspondientes a los índices de la colección; están sus tipos. Es posible que en otras clases haya casos similares, pero es particularmente grave en las colecciones por su uso intensivo.

Bibliografía

[Age94] Ole Agesen. Constraint-Based Type Inference and Parametric Polymorphism. In *SAS '94, First International Static Analysis Symposium*, pages 78–100, June 1994.

[Age95] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *ECOOP '95 Conference Proceedings*, pages 2–26, August 1995. Åarhus, Denmark.

[Age96] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Standford University, Department of Computer Science, 1996.

[AH95] Ole Agesen and Urs Hölzle. Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object Oriented Languages. In *OOPSLA '95, Object Oriented Programming Systems, Languages and Applications*, pages 91–107, Austin, TX, October 1995.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compiladores: Principios, técnicas y herramientas*. Addison-Wesley Iberoamericana, Wilmington, Delaware, 1986.

[Bar92] Andrew J. Barnard. *From Types to Dataflow: Code Analysis for an Object Oriented Language*. PhD thesis, University of Manchester Department of Computer Science, England, October 1992.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *OOPSLA '93, Object Oriented Programming Systems, Languages and Applications*, pages 215–230, Washington, DC, September 1993.

[BI82] Alan H. Borning and Daniel H. Ingalls. A Type Declaration and Inference System for Smalltalk. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 133–141. Association for Computing Machinery, 1982.

[Car97] Luca Cardelli. *Allen B. Tucker (Ed.): The Computer Science and Engineering Handbook*, chapter Type Systems, pages 2208–2236. CRC Press, 1997.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call Graph Construction in Object Oriented Languages. In *OOPSLA '97, Object Oriented Programming Systems, Languages and Applications*, Atlanta, GA, October 1997.

[GI98] J. Gil and A. Itai. The complexity of type analysis of object oriented programs. In *ECOOP '98*, pages 601–634, July 1998.

[GPMB95] Jan Gustafsson, Kjell Post, Jukka Mäki-Turja, and Ellus Brorsson. Benefits of Type Inference for an Object-Oriented Real-Time Language. In *Object-Oriented Real-Time Workshop*, San Antonio, Texas, October 1995.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[Gra89] Justin Owen Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages*. PhD thesis, University of Illinois at Urbana-Champaign, 1989.

[Gro98] David Paul Grove. *Effective Interprocedural Optimization of Object Oriented Languages*. PhD thesis, University of Washington, 1998.

[Guz00] Mark Guzdial. *Squeak: Object-Oriented Design with Multimedia Applications*. Prentice Hall, 2000.

[HA96] Urs Hölzle and Ole Agesen. Dynamic vs. Static Optimization Techniques for Object-Oriented Languages. In *TAPOS '96, Theory and Practice of Objects Systems*, 1996.

[Hil99] Michael Hiltzik. *Dealers of Lightning: Xerox PARC and the Dawn of the computer Age*. Harper, New York, 1999.

[Höl95] Urs Hölzle. *Adaptative Optimization in Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Standford University, Department of Computer Science, March 1995.

[HU94] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Call with Run-Time Type Feedback. In *SIGPLAN, Conference on Programming Language Design and Implementation*, pages 326–336, Orlando, FL, June 1994.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the Future: The Story of Squeak, A Practical Smalltalk written in itself. In *OOPSLA '97, Object Oriented Programming Systems, Languages and Applications*, October 1997.

[Ing78] Daniel Ingalls. The Smalltalk-76 Programming System. Design and Implementation. In *Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978.

[Ing81] Dan Ingalls. Design Principles Behind Smalltalk. In *Byte '81*, 1981.

[Kay93] Alan C. Kay. The Early History of Smalltalk. In *ACM SIGPLAN Notices*, volume 28, March 1993.

[Ler98] Xavier Leroy. An overview of Types in Compilation. In *TIC '98, Types in Compilation workshop*, pages 1–8, March 1998.

[NRC99] Luciano Notarfrancesco, Gerardo Richarte, and Leandro Caniglia. *Morphic Wrappers*, 1999. `http://mathmorphs.swiki.net`.

[PC94] John Plevyak and Andrew Chien. Incremental Inference of Concrete Types. In *OOPSLA '94, Object Oriented Programming Systems, Languages and Applications*, pages 324–340, Portland, Oregon, October 1994.

[PS91] Jens Palsberg and Michael Schwartzbach. Object Oriented Type Inference. In *OOPSLA '91, Object Oriented Programming Systems, Languages and Applications*, pages 146–161, Phoenix, Arizona, October 1991.

[PS92] Jens Palsberg and Michael Schwartzbach. Three Discussions on Object-Oriented Typing. In *ACM SIGPLAN OOPS Messenger*, pages 3(2):31–38, 1992.

[RNCM01] Gerardo Richarte, Luciano Notarfrancesco, Leandro Caniglia, and Valeria Murgia. *SqueakNOS: Squeak with No Operating System*, 2001. `http://squeaknos.sourceforge.net`.

[Sad00] Benny Sadeh. *SmallInterfaces for Squeak*, 2000. `http://www.mars.dti.ne.jp/smallinterfaces`.

[SU95] Randall B. Smith and David Ungar. Programming as an Experience: The Inspiration for Self. In *ECOOP '95 Conference Proceedings*, August 1995. Åarhus, Denmark.

[Suz81] Norihisa Suzuki. Inferring types in Smalltalk. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 187–199. Association for Computing Machinery, 1981.

Bibliografía Auxiliar

[Eco77] Umberto Eco. *Cómo se hace una Tesis. Técnicas y procedimientos de investigación, estudio y escritura.* Gedisa editorial, 1977.

[SW79] William Strunk and E. B. White. *The Elements of Style.* Macmillan Publishing Co., New York, 1979.