Tipos: garantías a medida

Éric Tanter

Profesor Asociado Departamento de Ciencias de la Computación, Universidad de Chile. Ph.D. Computer Science, Universidad de Nantes, Francia y Universidad de Chile (2004). Líneas de investigación: Desarrollo de Software, Lenguajes de Programación, Modularidad y Adaptabilidad del Software, Sistemas Concurrentes y Distribuidos, Ingeniería de Software.

etanter@dcc.uchile.cl

PROGRAMAS QUE AYUDAN A LOS PROGRAMADORES A PROGRAMAR

Desde que aparecieron los ensambladores, hasta hoy, el desarrollo de software se apoya de manera esencial en herramientas computacionales, cuyo objetivo es permitir al programador abstraerse de detalles de bajo nivel (como la ubicación exacta de los datos en memoria) para concentrarse en el desarrollo de soluciones expresadas en términos de conceptos inteligibles por humanos y fácilmente comunicables. El alto nivel de abstracción provisto por los lenguajes de programación, que incluye la posibilidad de construir nuevas abstracciones dedicadas (procedimientos, funciones, tipos de datos abstractos, clases, librerías, frameworks), es un habilitador crucial en el desarrollo de los sistemas altamente complejos sobre los cuales nuestra sociedad se construye.

Idealmente, uno quisiera tener, al mismo tiempo que está programando, la garantía de que el programa desarrollado es correcto. Correcto, en el sentido de que computa lo que se espera que compute, y que lo haga de la manera que uno espera (por ejemplo, respecto del uso de recursos). Sin embargo, la gran mayoría de las propiedades interesantes que uno quiere asegurar son indecidibles (partiendo con la propiedad aparentemente muy simple "¿este programa termina?"). La Figura 1 ilustra esta problemática y las distintas posibilidades a considerar.

El campo de la verificación de programas es por ende un pozo sin fin, un eterno compromiso entre la expresividad requerida para lo que se quiere verificar, y la amenaza de la indecidibilidad. En la práctica, muchas veces uno quiere la garantía absoluta que todos los programas malos son rechazados. Esto significa que los sistemas de verificación son conservadores: sólo reportan que un programa cumple una cierta propiedad cuando lo pudieron demostrar, pero inevitablemente van a rechazar algunos programas que sí cumplen la propiedad (Figura 1c).

Mientras que las técnicas más avanzadas de verificación de software, como el análisis formal y la verificación de modelos, no han tenido mucha aceptación en la industria de software –excepto en áreas críticas donde la correctitud absoluta es una necesidad vital (automóviles, cohetes, etc.)– existe una forma liviana de métodos formales que cada programador usa día a día, muchas veces sin preguntarse mucho de qué se trata: los sistemas de tipos.

¿SISTEMAS DE TIPOS?

El objetivo de un sistema de tipos es asegurar estáticamente –es decir antes de la ejecución del programa– que ciertos errores nunca ocurren en tiempo de ejecución. ¿Cuáles son estos errores? Típicamente, se trata de errores "simples" como aplicar una operación primitiva a valores inadecuados, como lo es multiplicar dos valores que no son numéricos. Para operar, un sistema de tipos clasifica las expresiones de un programa según el tipo de valores que pueden producir. Luego, verifica que solamente se usan expresiones de tipos adecuados en lugares adecuados. Por ejemplo, si `e1` y `e2` son expresiones, la expresión `e1 + e2` es válida en tipos solamente si `e1` y `e2` son expresiones válidas de tipo numérico; de ser así, la expresión `e1 + e2` misma es válida, y tiene un tipo numérico. Esta clasificación de tipos se puede basar en anotaciones de tipo que el programador escribe explícitamente en el programa (como en C y Java), o pueden ser inferidas por el mismo sistema de tipos (como en ML y Haskell).

Una característica muy importante de los sistemas de tipos, y una razón decisiva de su amplia adopción en la industria, es la componibilidad: para poder verificar un componente solamente se necesita conocer los tipos de los componentes con los cuales interactúa, y no sus implementaciones. Esto diferencia

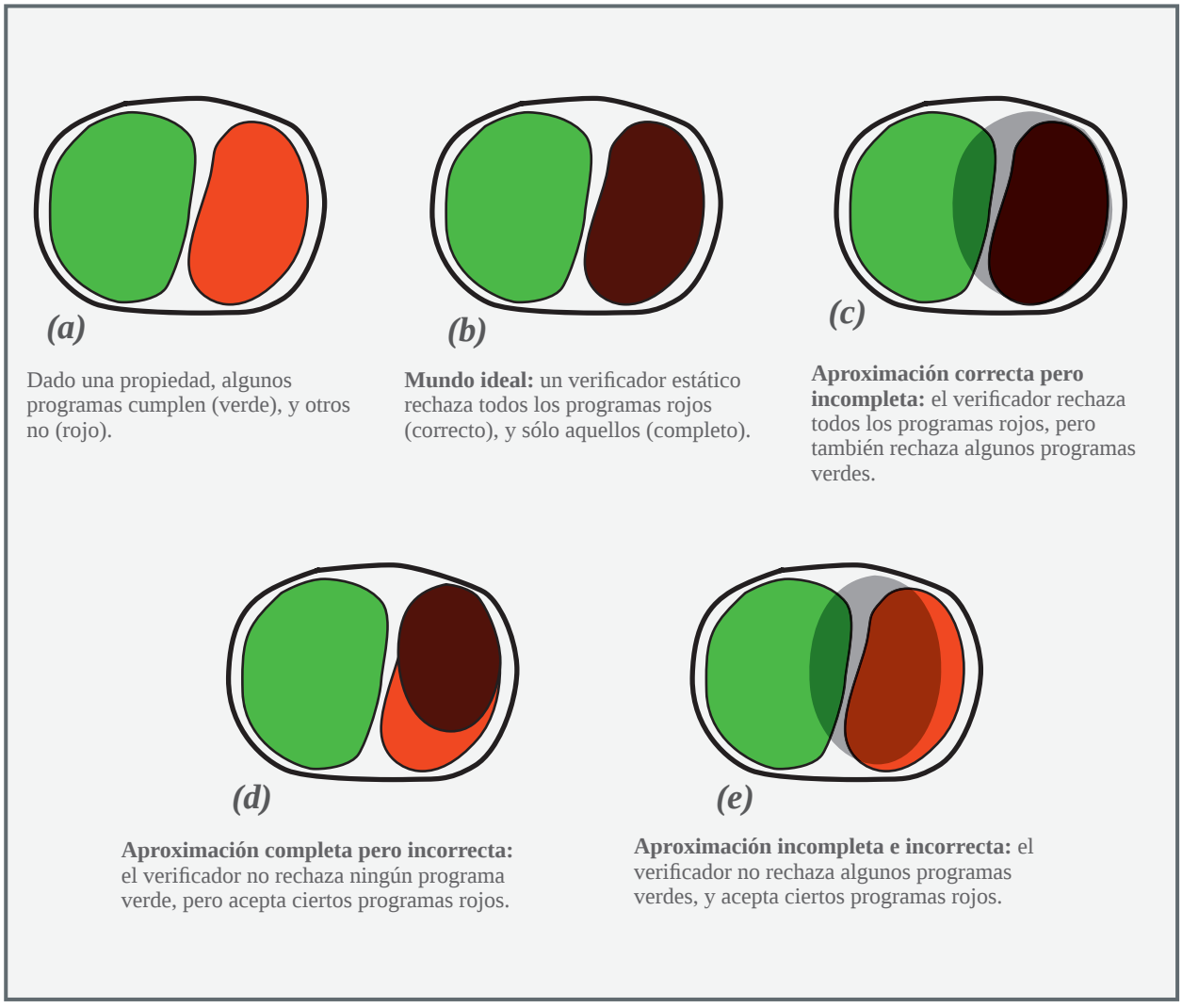

Figura 1

los sistemas de tipos de los análisis estáticos que requieren poder analizar todo el código fuente de un sistema a la vez. Por ejemplo, considerando una función, basta conocer el tipo de sus argumentos, y los tipos de las funciones que llama, para determinar si tiene un tipo de retorno bien definido y cuál es. Además, si el programador especificó el tipo de retorno esperado, se puede corroborar que la definición de la función cumple esta expectativa. En resumen, los tipos sirven como contratos que rigen las interacciones entre los componentes de un sistema, permitiendo el desarrollo independiente de los componentes y validando su integración.

¡COHERENCIA, POR FAVOR!

Una propiedad fundamental que se espera de un sistema de tipos, es que las predicciones de tipos sean coherentes con la ejecución del programa. Por ejemplo, si el sistema de tipos dice que un programa es válido y de tipo numérico, entonces ejecutarlo debería producir un valor de tipo numérico. Por más sorprendente que parezca, ciertos lenguajes carecen de esta propiedad. Por ejemplo, en C, el sistema de tipos no es coherente con la ejecución de los programas, lo que lleva a un estado extraño en el cual el sistema de tipos no asegura nada. Esto es porque el modelo de ejecución de C, de muy bajo nivel, no garantiza que se respeten las abstracciones establecidas al nivel del código fuente, y

verificadas por el sistema de tipos. La consecuencia de esto son los famosos "errores de segmentación" y otras joyas del "comportamiento no-definido" que todo programador C ha llegado a apreciar en su justo valor. Para volver a la Figura 1, el verificador de tipos de C corresponde al caso 1e: incompleto, e incorrecto.

Volviendo a lenguajes civilizados, un sistema de tipos puede ser visto como un sistema de demostración formal de que ciertos errores no ocurrirán en tiempo de ejecución. El costo a pagar por esta firme garantía es que el programador debe convencer al sistema de tipos que su programa cumple con las exigencias establecidas. Surgen nuevamente los monstruos de la indecidibilidad, y el necesario conservadurismo de la verificación de tipos. La consecuencia práctica es que en algunos casos, un programa válido será rechazado por el sistema de tipos (Figura 1c). Frente a este problema, hay dos puertas para avanzar.

PROMESAS DE TIPOS

La primera consiste en tener un mecanismo con el cual el programador pueda indicarle al sistema que una expresión es de cierto tipo, aunque no lo pueda demostrar. En buenas cuentas, el programador le dice al verificador de tipos: "Confía en mí, te juro que está bien". Este mecanismo es una coerción, o *cast*, que no se verifica estáticamente. Por ejemplo, si el sistema de tipos rechaza `e1+e2` porque no logra convencerse de que `e2` realmente sea de tipo numérico, el programador puede insertar una coerción explícita: `e1+(<Num>e2)`. Obviamente, para proteger la integridad del programa, dicha coerción se tiene que traducir en una verificación dinámica –es decir en tiempo de ejecución– de que el valor de `e2` realmente sea de tipo numérico. De no ser así, se lanza un error que señala específicamente su causa. Es importante notar que dicho error es infinitamente mejor que un *crash* del programa o que un comportamiento arbitrario indefinido. Los peores errores son los que pasan desapercibidos y los que no dejan ninguna indicación de lo que pasó; eso lo saben muy bien los hackers que explotan los hoyos de seguridad de sistemas programados en C.

MEJORES TIPOS

La segunda puerta es mejorar el sistema de tipos, haciéndolo más expresivo, para así reducir el monto de programas válidos que son rechazados. Un ejemplo clásico de esta vía es la introducción en Java 1.5 de los "generics", es decir, del polimorfismo paramétrico, tal como se encuentra en los lenguajes ML y Haskell. Anteriormente, el sistema de tipos de Java sólo razonaba en base a polimorfismo de subtipos, es decir, el hecho de que un objeto es aceptable en un lugar donde se espera que tenga algún tipo A si entiende un conjunto de mensajes más amplio que el especificado por A. Dicho sistema de tipos era sin embargo, muy limitado al momento de usar colecciones de objetos, dado que no permite especificar el tipo de los elementos contenidos dentro de una colección. Por ejemplo, una lista es de tipo `List`, y no se sabe nada del tipo de sus elementos. Esto obligaba a los programadores a hacer uso de coerciones de manera muy frecuente, lo que a su vez, disminuyó mucho la relevancia misma del sistema de tipos, al dejar la puerta abierta a muchos errores en tiempo de ejecución. La introducción de los *generics* en Java resolvió este problema, permitiendo al programador especificar el tipo de los elementos contenidos en una colección, por ejemplo, declarando que una lista sólo puede contener valores numéricos. Pero esta mejora aumentó considerablemente la complejidad del sistema de tipos, dado que la interacción entre el polimorfismo de subtipos y el polimorfismo paramétrico no es trivial.

TIPOS DINÁMICOS

Frente a esta situación existe, en realidad, una tercera opción: la de optar por un lenguaje sin verificación estática de tipos. Dichos lenguajes, llamados lenguajes dinámicos, como lo son Python, Ruby y JavaScript, son muy atractivos por su simplicidad aparente. Al no tratar de asegurar nada estáticamente, los lenguajes dinámicos ahorran un costo conceptual grande a los programadores, permitiéndoles ejecutar cualquier programa sin permiso previo. Es crucial notar que estos lenguajes son seguros, a diferencia de C, y por ende no permiten comportamiento no-definido. Más bien, corresponden a lenguajes donde todas las verificaciones de tipos se hacen dinámicamente, similar al caso de coerciones presentado anteriormente. Por ejemplo, uno siempre puede ejecutar un programa que hace `e1+e2`. El ambiente de ejecución verificará dinámicamente que `e1` y `e2` producen un valor numérico antes de ejecutar la primitiva de adición. Esta verificación tiene un costo, y además puede traducirse en un error en tiempo de ejecución si es que `e1` o `e2` resulta no producir un valor numérico. Es por eso que hablé de "simplicidad aparente": al no verificar tipos estáticamente, una parte significativa del trabajo de depuración de programas en lenguajes dinámicos tiene que ver con repetir el trabajo que un sistema de tipos hubie-

se hecho automáticamente y exhaustivamente. Tener que escribir tests para todos los posibles casos de errores de tipos es contraproducente. Es, además, limitado, porque un conjunto de tests no es nada más que un conjunto de observaciones: no provee ninguna garantía de que ciertos errores no ocurrirán nunca.

TIPOS GRADUALES

Este último punto es la razón por la cual muchos sistemas parten siendo prototipados en un lenguaje dinámico, pero a medida que el programa va creciendo, la necesidad de tener fuertes garantías estáticas se hace sentir. ¿Qué se puede hacer en estos casos? En un principio, lo único que se puede hacer es portar el sistema a otro lenguaje, que sea tipado estáticamente. Esta conversión no es sin costo ni riesgo. Reconociendo esta situación, últimamente se ha dado énfasis a los sistemas de tipos "graduales", que permiten, dentro del mismo lenguaje, que conviva código tipado con código no tipado. El sistema de tipos hace su mejor esfuerzo para detectar incoherencias estáticamente y, cuando no puede, deja pasar, delegando la responsabilidad al ambiente de ejecución del programa. En buenas cuentas, un sistema de tipos graduales inserta coerciones automáticamente en todos los lugares donde tiene una duda. Esta mezcla parece ser la panacea, combinando las ventajas de ambos mundos en un entorno común.

No debería sorprenderles entonces que todos los últimos lenguajes industriales (Dart de Google, TypeScript y C# de Microsoft, ActionScript de Adobe, entre varios otros) cuentan con una forma de tipos graduales. Las bases teóricas de los tipos graduales son más bien recientes. Permitieron establecer resultados importantes, por ejemplo que no es correcto basarse en la noción de subtipos para introducir un tipo dinámico. También ayudaron a entender bien cuál es la garantía que se obtiene de un programa donde ciertas partes son tipadas y otras no: si bien un error de tipo puede producirse al ejecutar código estáticamente tipado, la causa de dicho error siempre será una entidad no tipada, que el sistema puede identificar. A la fecha, el área sigue siendo objeto de investigación muy activa para entender mejor los distintos compromisos posibles entre garantías fuertes, costos de ejecución, y complejidad para el programador. De hecho, cada uno de los lenguajes actuales con tipos graduales lo hace a su manera, distinta de los demás. A pesar de esta cacofonía ambiente, es de apostar que de aquí en adelante, ningún nuevo lenguaje pragmático podrá ignorar el permitir combinar verificación estática de tipos con verificación dinámica.

¿QUÉ TAN ÚTILES SON ESTOS TIPOS?

Una gran crítica a los sistemas de tipos tradicionales es que solamente verifican propiedades simples y que, por lo tanto, "no valen la pena". Es cierto que es útil saber que nunca se van a sumar peras con manzanas, pero más útil aún es saber que el resultado final es correcto, no solamente algún valor del tipo esperado. En los ojos de un sistema de tipos tradicional, 1 y 1000 son lo mismo (elementos de un tipo numérico), pero claramente al usuario final no le da lo mismo tener 1 o 1000 dólares en su cuenta. De manera similar, un programa que computa el resultado deseado no es muy atractivo si es que para lograrlo, hace un uso indebido de recursos o compromete la integridad del sistema. Mucho trabajo ha sido desarrollado en el área de tipos expresivos –o verificación basada en tipos, o programación certificada– donde los tipos se usan para especificar propiedades mucho más relevantes que la simple pertenencia a un conjunto de valores. En lo que queda de este artículo, quisiera mencionar brevemente tres enfoques para análisis precisos con tipos: sistemas de tipos subestructurales, sistemas de efectos y tipos dependientes.

TIPOS AVANZADOS I: SISTEMAS SUBESTRUCTURALES

En los sistemas de tipos tradicionales, la información de tipos que utiliza el sistema para analizar un pedazo de código goza de varias características llamadas "estructurales". Por ejemplo, si se sabe que la variable `a` es de tipo `Num` y que la variable `b` es de tipo `Bool`, no importa el orden de estas premisas. Tampoco interfiere con ellas el hecho de conocer más premisas, por ejemplo que otra variable `c` también está definida. Además, se puede usar el conocimiento de que `a` es de tipo `Num` tantas veces que se requiera (por ejemplo para poder tipar la expresión `a+a`, se usa el conocimiento relativo a la variable `a` dos veces). Los sistemas de tipos llamados subestructurales son sistemas en los cuales alguna(s) de estas características no se provee; por ejemplo, donde la información relativa a una variable se tiene que usar exactamente una vez. Estos sistemas permiten verificar propiedades interesantes, fuera del alcance de los sistemas tradicionales. Un ejemplo clásico es restringir las interfaces que

proveen acceso a los distintos recursos de un sistema, como archivos, *locks*, y memoria. Está claro que cada uno de estos recursos pasa por una serie de cambios de estado a lo largo de su vida: los archivos pueden ser abiertos o cerrados; los *locks* pueden ser tomados o no; y la memoria puede ser asignada o liberada. Los sistemas de tipos subestructurales proveen mecanismos para hacer un seguimiento estático coherente de estos cambios de estado, con el fin de prevenir la realización de operaciones en objetos que están en un estado inválido, como lo es leer o escribir en un archivo cerrado. Los sistemas de tipos tradicionales no pueden verificar estas propiedades y, por ende, las operaciones asociadas a recursos siempre pueden producir errores en tiempo de ejecución.

TIPOS AVANZADOS II: EFECTOS

La segunda familia de sistemas de tipos que aumenta considerablemente el rango de problemas controlables estáticamente es la de tipos de efectos. Como hemos visto, un sistema de tipos caracteriza el rango de posibles valores que puede resultar de la evaluación de una expresión, o sea, el "qué" se computa, pero no dice nada del "cómo" se obtiene el resultado. En un sistema de efectos, el tipo de una expresión también refleja los posibles efectos computacionales (acceso y escritura en memoria, entradas y salidas, etc.) que la evaluación de la expresión puede provocar. Un ejemplo seguramente muy familiar para los que han programado en Java, es el de las excepciones verificadas. En Java, el tipo de un método no solamente refleja el tipo de los argumentos y del valor retornado, sino también la posibilidad que el método lance una excepción, la cual produce una transferencia del flujo de control desde el que lanza la excepción al que la captura. Este es un ejemplo de un efecto de control. El sistema de tipos asegura en este caso que ninguna excepción declarada puede pasar desapercibida. Un área donde se usan mucho los tipos de efectos es para evitar los infames *data races* en programación concurrente: volviendo a nuestro programa estrella `e1+e2`, el conocer estáticamente que `e1` y `e2` sólo pueden leer o escribir (efectos de memoria) en regiones de memoria disjuntas, permite garantizar que es correcto ejecutar ambas expresiones en paralelo.

En forma relacionada, el saber que una expresión no tiene efectos computacionales (es decir, que es una expresión "pura") permite varias optimizaciones derivadas del razonamiento basado en ecuaciones, que conocemos de nuestros cursos de álgebra. Si una expresión `e1` aparece múltiples veces en un programa y es pura, entonces es totalmente correcto reemplazar `e1` por su valor v, porque `e1=v`. Así evitamos calcular `e1` varias veces. Para entender porqué esto no es cierto para cualquier expresión, considere que `e1` imprime en pantalla "ok". Claramente no es lo mismo para el usuario ver una vez "ok" que verlo varias veces. Lo mismo si `e1` retira 100 dólares de su cuenta. Un caso ejemplar de estas ideas es el lenguaje de programación Haskell. Es un lenguaje funcional en el cual todas las expresiones son puras: no existe ninguna forma nativa de hacer asignaciones, por ejemplo. Como todo lenguaje práctico tiene que permitir de alguna manera tener algún efecto sobre el mundo real, los diseñadores de Haskell tuvieron que ingeniar un mecanismo para soportar efectos dentro de este marco puro. Resulta que la respuesta vino del lado de las mónadas, estructuras componibles que representan computación. Sin entrar en detalles, en Haskell una función de tipo `Int → Int` es una función pura que, dado un entero, retorna un entero. Y nada más; el compilador puede hacer todas las optimizaciones que quiera cuando ve aplicaciones de dicha función. En cambio, si la función hace uso de entradas y salidas como imprimir en pantalla, esto se ve reflejado en su tipo, que pasa a ser `Int → IO Int`, donde `IO` es la mónada de los efectos de entradas y salidas. Existen mónadas para excepciones, continuaciones, estado compartido, no-determinismo, etc. Y obviamente un programador puede definir sus propios efectos, por ejemplo si quiere razonar sobre el flujo de valores en el programa, lo cual tiene aplicaciones en seguridad. Esta disciplina, por más estricta que sea, fomenta un estilo de programación donde se separa el manejo de efectos de las partes más computacionales de un sistema, lo que mejora su modularidad al mismo tiempo que permite más razonamiento estático, incluyendo más optimizaciones.

TIPOS AVANZADOS III: TIPOS DEPENDIENTES

Finalmente, es interesante considerar el caso de los tipos dependientes, que es el más extremo –y por ende también el más prometedor y desafiante– en muchos aspectos. Un tipo dependiente es un tipo (o más bien una familia de tipos) que depende de un valor. Por ejemplo, el tipo `Array[n]` es un tipo dependiente que representa los arreglos de tamaño `n`. En su forma más expresiva, los tipos dependientes son provistos por lenguajes que consideran los tipos como cualquier otro valor. Es decir que permiten definir funciones que producen tipos. ¿Qué se gana con

esto? La posibilidad de hablar de propiedades mucho más precisas. Consideremos el tipo de una función de ordenamiento de arreglos numéricos, `sort`. En un lenguaje tradicional lo único que podemos decir es que sort tiene el tipo `Array→Array` (o, si tenemos polimorfismo paramétrico, que tiene el tipo `Array[Num]→Array[Num]`. No podemos expresar al nivel del tipo de `sort` propiedades fundamentales para la correctitud de `sort`. Por ejemplo, uno podría implementar `sort` como una función que retorna un arreglo arbitrario, y el sistema de tipos estaría satisfecho. Con un sistema de tipos dependiente, al dar a `sort` el tipo `Array[n]→Array[n]`, uno al menos garantiza que `sort` retorna un arreglo del mismo tamaño que el que recibió. Uno puede ir más lejos y especificar al nivel de tipos que el arreglo retornado tiene que ser una permutación del arreglo de entrada (o sea, que no se pueden inventar nuevos elementos), y ¡también que tiene que ser una permutación ordenada! Similarmente, el tipo de la función que concatena dos arreglos, `append`, es `Array[m]→Array[n]→Array[n+m]`, especificando que dado un arreglo de tamaño `n` y otro de tamaño `m`, el arreglo resultante es de tamaño `n+m`.

El lector atento y preocupado por los demonios de la indecidibilidad debería saltar de su silla en este momento: ¡estamos permitiendo que se haga computación (aquí `n+m`) al nivel de los tipos! Esto lleva dos preguntas no menores. Primero, ¿qué pasa con la terminación de la verificación de tipos? Ciertos lenguajes con tipos dependientes limitan la computación a nivel de tipos a dominios acotados, mientras otros permiten toda la expresividad de un lenguaje completo, abandonando la garantía de terminación de la verificación de tipos; otros lenguajes simplemente no permiten recursión generalizada en el lenguaje integrado. La segunda pregunta es cómo el sistema de tipos puede ser suficientemente poderoso para verificar estas propiedades, dado que las más interesantes son indecidibles. La respuesta es que no lo es, pero que permite al programador proveer una demostración de dicha propiedad, en conjunto con el programa. Este punto es quizás lo más fascinante, ya que explota una conexión muy profunda entre los cálculos computacionales y las lógicas formales (ver Figura 2). Un programa en un lenguaje de sistemas de tipos dependientes es entonces una mezcla de programas escritos por su contenido computacional (el trabajo que hacen) y programas escritos por su contenido "demostracional" (la demostración de que el programa computacional cumple alguna propiedad formulada como una proposición lógica al nivel de los tipos). El potencial de los tipos dependientes es el de crear programas certificados, desde su construcción. Diseñar e implementar lenguajes de programación con tipos dependientes que sean prácticos es un tema abierto, como lo es el explorar todo el espectro de aplicaciones de esta técnica.

EN CONCLUSIÓN

La verificación basada en tipos tiene un alto potencial de ser adoptada en la industria, por múltiples razones. Primero, no apunta a la verificación de propiedades globales de un sistema completo. Aún con tipos dependientes, nunca es necesario establecer un teorema sobre el sistema en su globalidad. Solamente se expresan propiedades locales. La conexión lógica entre estas propiedades locales y las propiedades globales que se desean no se verifica. La experiencia con otras técnicas de verificación formal muestra que apuntar a verificar dicho razonamiento global es un tremendo esfuerzo, fuera del alcance para la mayoría de la industria del software. Segundo, los programadores ya están acostumbrados a desarrollar en lenguajes con tipos estáticos. Si bien los sistemas de tipos que se exploraron aquí son semánticamente bastante más ricos que los tipos tradicionales, la metodología de programación es muy cercana. No es necesario aprender un nuevo lenguaje de modelamiento o una nueva herramienta de análisis. La integración fuerte entre programar, especificar y verificar, es un factor de adopción importante. Podemos prever un escenario de adopción progresiva de estas técnicas de verificación, con sistemas de tipos con una semántica paulatinamente más rica. De hecho, este escenario ya se hizo realidad cuando Java adoptó el polimorfismo paramétrico en su versión 1.5, y ahora con el lenguaje Scala –cuyo sistema de tipos es mucho más avanzado que el de Java– que está progresivamente reemplazando a Java en muchas aplicaciones (por ejemplo, Twitter y LinkedIn). Además, es posible diseñar sistemas de tipos muy expresivos que sean graduales, con tal de que no sea necesario demostrar todas las propiedades a la vez, sino que se puedan dejar varias a cargo de una verificación dinámica, e ir elaborando la verificación estática paso a paso.

Encontrar un sabio equilibrio entre la complejidad de la programación verificada y sus beneficios en términos de correctitud es un desafiante problema de Ingeniería de Software, que merece ser estudiado en profundidad. Dada la enorme dependencia de nuestra sociedad en los sistemas de software, es de esperar que la programación verificada se vuelva una práctica común, contribuyendo ampliamente a la correctitud del software que usamos día a día.

Si quiere saber más, dos excelentes libros de referencia son:

[1] Benjamin Pierce. Types and Programming Languages. MIT Press. 2002.

[2] Benjamin Pierce (editor). Advanced Topics in Types and Programming Languages. MIT Press. 2004.

Además, este artículo corto es muy claro e inspirador:

Tim Sheard, Aaron Stump, Stephanie Weirich. Language-based verification will change the world. Future of Software Engineering Research. ACM. 2010. BITS

La Correspondencia de Curry-Howard

La correspondencia de Curry-Howard establece un puente entre tipos y proposiciones lógicas, y programas y demostraciones de dichas proposiciones. Fue descubierta y refinada por los matemáticos Haskell Curry y William A. Howard.

Consideremos la proposición lógica $(A \supset B) \wedge (B \supset C) \supset (A \supset C)$, que expresa la transitividad de la implicación lógica (si A implica B y B implica C entonces A implica C). Esta proposición corresponde al tipo $(A \to B) \times (B \to C) \to (A \to C)$. Lo único que hicimos es cambiar la notación de la implicación $\supset$ con la flecha $\to$ (tipo de funciones), y la notación de la conjunción $\wedge$ con el producto cartesiano $\times$. Este tipo es el de una función que toma dos argumentos, una función de tipo $A \to B$ y una de tipo $B \to C$ y retorna un función de tipo $A \to C$. ¿Existe un programa con este tipo? Si, el que retorna la función que compone ambas funciones recibidas como argumento:

```
f (g1 : A → B, g2 : B → C) (x : A) = g1 (g2 x)
```

En cambio, el hecho de que la propiedad $(A \supset B) \supset (A \supset C)$ no es cierta se ve reflejado en que no es posible definir una función pura con el tipo $(A \to B) \to (A \to C)$. Si lo duda, ¡inténtelo!

La correspondencia de Curry-Howard no se limita a la lógica proposicional de primer orden, conectando implicancia y funciones, pares y sumas con conjunción y disyunción. También conecta las cuantificaciones universales y existenciales con tipos dependientes, la lógica clásica con el manejo de continuaciones, la lógica modal con efectos, etc. Un buen punto de partida es la entrada Wikipedia: http://en.wikipedia.org/wiki/Curry-Howard_correspondence.

Figura 2