Análisis Semántico: Sistemas de Tipos
Compiladores 2020-1, Nota de clase 9

Lourdes Del Carmen González Huesca

21 de octubre de 2019
Facultad de Ciencias UNAM

Una parte importante del análisis semántico es la verificación semántica o análisis contextual del programa en donde las reglas semánticas (estáticas) que definen al lenguaje de programación dirigen esta fase. A diferencia del uso de gramáticas con atributos para analizar el "significado" del programa, este análisis estudia cuestiones relacionadas con la correcta utilización de los tipos de valores en expresiones es decir que verifica la consistencia del uso de identificadores y expresiones para futuras fases y generación de código intermedio. Además de determinar detalles para la ejecución del programa como el espacio de almacenamiento, direcciones relativas, establecer conversiones, escoger versiones de operadores, etc.

Un tipo se puede definir como una colección de objetos o valores. Las expresiones que determinan los tipos en los lenguajes de programación son texto y las expresiones de tipo se pueden clasificar en

- tipos básicos; por ejemplo, `bool`, `char`, `integer`, etc.
- construcciones de tipos, es decir constructores que se aplican a expresiones de tipo; por ejemplo, productos, records, apuntadores, arreglos, funciones, etc.
- nombres de tipos definidos por el programador; estos tipos usualmente son construcciones nuevas.

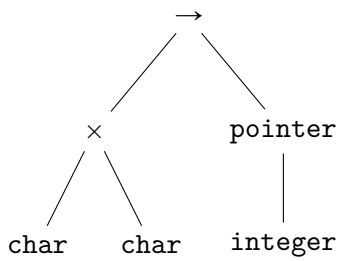
Se puede establecer la noción o el significado de un tipo de diferentes formas:

denotacional descripción a través de los valores

constructivo usando tipos primitivos y compuestos

basado en abstracción descripción de las operaciones usando una interfaz

Usualmente, los lenguajes de programación utilizan estas formas combinadas para describir los tipos disponibles y los que el programador puede crear. Algunos lenguajes de programación utilizan descriptores de tipo que son estructuras que representan las expresiones de tipo durante la compilación. Por ejemplo, el tipo de una función `char` $\times$ `char` $\rightarrow$ `pointer(integer)` se puede representar como



Sistema de Tipos

Definición 1 (Sistema de Tipos)**.** Un sistema de tipos es un método sintáctico para demostrar la ausencia de ciertos comportamientos al clasificar frases de acuerdo a los tipos de valores que calculan. Este método consiste en un conjunto de reglas para asociar tipos a construcciones o expresiones del lenguaje.

Los tipos proveen un contexto implícito para operaciones, es decir que limitan las operaciones semánticamente válidas. Por medio de las reglas, el sistema rechaza una expresión si no se le puede asociar un tipo. Las reglas se pueden especificar mediante una combinación de las siguientes:

- lenguaje natural
- restricciones en las variables de tipos
- reglas lógicas para declarar tipos, propagar o restringir tipos

Esta última forma utiliza reglas de estilo deducción natural para relacionar juicios a cerca del tipificado de expresiones bajo un contexto. Un juicio de tipo $\Gamma \vdash e : \tau$ se lee como "bajo el contexto Γ, la expresión o programa e tiene tipo τ". Una regla del sistema de tipos se ve como un conjunto de hipótesis J_i que soportan el juicio en la conclusión J:

$$\frac{J_1 \qquad J_2 \qquad \dots \qquad J_n}{J}$$

Una derivación de tipos es un árbol cuya raíz es el programa, los nodos son instancias de las reglas de tipificado y las hojas son reglas básicas o "axiomas" del sistema.

Decimos que un programa es estáticamente correcto si satisface las reglas de tipificado, es decir que existe una derivación de tipos donde el programa es la raíz de la derivación.

Dependiendo del tipo de lenguaje de programación que se utilice, los tipos o las anotaciones de tipo en los programas pueden ser explícitas o implícitas. El estilo en que aparece esta información en los programas ayuda ya sea a la verificación o la inferencia de tipos.

La Teoría de Tipos es la disciplina que estudia todo lo relacionado con tipos, en particular en el estudio de lenguajes de programación las reglas de tipificado aseguran que toda expresión debe tener un tipo conocido y fijo en tiempo de compilación. Estas reglas junto con las que definen la evaluación de expresiones ofrecen garantías a los lenguajes mediante propiedades:

Progreso Si un e programa cerrado (sin variables libres) tiene un tipo, entonces ese programa es un valor o existe otro programa o expresión e' tal que el primer programa se evalúa al segundo.

Preservación Si un programa cerrado tiene un tipo y se evalúa a otro programa o expresión entonces ese nuevo programa tiene el mismo tipo que el original.

Estas dos propiedades conforman lo que se conoce como la propiedad de seguridad, se asegura que la evaluación nunca se detendrá sin razón y que los programas bien tipificados tienen un buen comportamiento.

Verificación de Tipos

La verificación de tipos o *type checking* verifica que cada operador, función, método, etc. sea aplicado de manera correcta en número y tipo a los argumentos correspondientes. Se deben examinar los cuerpos de las funciones y usar los tipos declarados para verificar que una expresión tiene el tipo indicado. Esta verificación depende totalmente de la definición del lenguaje y se apoya en la tabla de símbolos. Al ser incluida en algunos compiladores aseguran un mejor desempeño de programas ya que previene errores además de favorecer la modularidad del compilador al ser parte de una fase.

Decimos que un compilador que realiza la verificación de tipos identifica errores potenciales y es

conservativo si algunos programas sin errores se consideran como programas que presentan errores

robusto si ningún programa con errores es considerado como correcto

estas dos características aseguran la identificación de programas con errores ya sea de manera muy estricta o justa.

La verificación de tipos también puede realizarse en tiempo de ejecución pero hará más lenta la ejecucicón ya que cada vez que se ejecuta una función, primero se verifica que los argumentos sean del tipo correcto.

La verificación toma un parse tree decorado con la información del análisis sintáctico y posiciones de la tabla de símbolos y regresa el árbol anotado con tipos para cada expresión. Veamos un algoritmo muy general para esto:

Input: un parse tree

Output: un árbol sintáctico anotado con los tipos

Asignar variables de tipos a cada uno de los nodos, se debe respetar el alcance de las variables, declaraciones, funciones, etc.

Agregar restricciones: del ambiente o contexto como por ejemplo literales, operadores, funciones conocidas, etc. y restricciones derivadas de la forma del parse tree, es decir de las contrucciones del programa como aplicaciones, abstraciones, definiciones, etc.

Resolver las restricciones usando unificación

Determinar el tipo de todas las expresiones en especial de la raíz del parse tree

Si hay alguna inconsistencia de tipos, enviar un error.

Los errores pueden ser derivados de la falta de compatibilidad o equivalencia de tipos. Estas definiciones son necesarias en la verificación de tipos y se puede realizar mediante definiciones propias del lenguaje de programación. Existen dos formas de establecer la equivalencia de tipos:

1. Equivalencia por nombres, es decir que cada nombre de tipo es un tipo diferente.
2. Equivalencia estructural, es decir dos tipos son equivalentes si y sólo si tienen la misma estructura. Esta noción es la misma que se utiliza en la unificación de expresiones:
   a) $\tau_1 \equiv \tau_2$ si y sólo si τ_1 y τ_2 son exactamente el mismo tipo básico.
   b) para cada construcción de tipo, dos tipos con el mismo constructor son equivalentes si cada una de sus partes, una a una, son equivalentes, por ejemplo $\tau_1 \times \tau_2 \equiv \rho_1 \times \rho_2$ so y sólo si $\tau_1 \equiv \rho_1$ y $\tau_2 \equiv \rho_2$.

Tipificación en la práctica: el algoritmo de verificación está dirigido por la sintaxis y es deseable que los errores que deban comunicarse incluyan mensajes precisos y la localización de los mismos, por ejemplo `Missing argument of type T in line 400`.

Veremos ahora un ejemplo de una gramática con atributos que incluye la verificación de tipos en las reglas semánticas.

Ejemplo 1. Considera la gramática para expresiones aritméticas:

$$\begin{array}{lcl} E & \rightarrow & \texttt{var}\ \text{`}=\text{`}\ E \mid E\ \texttt{aop}\ T \mid T \\ T & \rightarrow & T\ \texttt{mop}\ F \mid F \\ F & \rightarrow & (E) \mid \texttt{const} \mid \texttt{var} \end{array}$$

donde `aop` y `mop` corresponden a las operaciones aditivas y multiplicativas respectivamente. Las siguientes producciones realizan la verificación de tipos, el atributo *env* calculan el ambiente de declaraciones para variables, el atributo *ok* asegura que los tipos son compatibles y el atributo *typ* es el que almacena el tipo de la expresión.

$$\begin{array}{l} E \longrightarrow \mathsf{var}\ '='\ E \\ \quad E[1].env = E[0].env \\ \quad E[0].typ = E[0].env\ \mathsf{var}.id \\ \quad E[0].ok = \textbf{let}\ x = \mathsf{var}.id \\ \qquad \textbf{in let}\ \tau = E[0].env\ x \\ \qquad \textbf{in}\ \ (\tau \neq \mathsf{error}) \wedge (E[1].type \sqsubseteq \tau) \end{array}$$

$$\begin{array}{l} E \longrightarrow E\ \mathsf{aop}\ T \\ \quad E[1].env = E[0].env \\ \quad T.env = E[0].env \\ \quad E[0].typ = E[1].typ \sqcup T.typ \\ \quad E[0].ok = (E[1].typ \sqsubseteq \mathsf{float}) \\ \qquad \wedge (T.typ \sqsubseteq \mathsf{float}) \end{array}$$

$$\begin{array}{l} E \longrightarrow T \\ \quad T.env = E.env \\ \quad E.typ = T.typ \\ \quad E.ok = T.ok \end{array}$$

$$\begin{array}{l} T \longrightarrow T\ \mathsf{mop}\ F \\ \quad T[1].env = T[0].env \\ \quad F.env = T[0].env \\ \quad T[0].typ = T[1].typ \sqcup F.typ \\ \quad T[0].ok = (T[1].typ \sqsubseteq \mathsf{float}) \\ \qquad \wedge (F.typ \sqsubseteq \mathsf{float}) \end{array}$$

$$\begin{array}{l} T \longrightarrow F \\ \quad F.env = T.env \\ \quad T.typ = F.typ \\ \quad T.ok = F.ok \end{array}$$

$$\begin{array}{l} F \longrightarrow (E) \\ \quad E.env = F.env \\ \quad F.typ = E.typ \\ \quad F.ok = E.ok \end{array}$$

$$\begin{array}{l} F \longrightarrow \mathsf{const} \\ \quad F.typ = \mathsf{const}.typ \\ \quad F.ok = \mathsf{true} \end{array}$$

$$\begin{array}{l} F \longrightarrow \mathsf{var} \\ \quad F.typ = F.env\ \mathsf{var}.id \\ \quad F.ok = (F.env\ \mathsf{var}.id \neq \mathsf{error}) \end{array}$$

Inferencia de Tipos

Este problema consiste en que dado un término (con o sin anotaciones de tipo) se debe reconstuir un tipo válido para el término o mostrar que esto no es posible. Los beneficios de la inferencia de tipos es que se pueden manejar funciones de orden superior y estructuras de datos sin necesidad de establecer los tipos precisos o mejor aún generalizando su uso.

En la inferencia de tipos se utilizan variables de tipo para abstraer los tipos de las (sub)expresiones que darán paso a restricciones entre los tipos de ellas. Las restricciones pueden originar conflictos en las instancias de los tipos y para ello es necesario definir una relación de equivalencia entre tipos que usualmente es estructural.

El algoritmo para la inferencia de tipos es conocido como Hindley–Milner, que originalmente es un sistema de tipos para modelar el polimorfismo paramétrico que resulta en un método para obtener el tipo más general de una expresión. Otro algoritmo es el **W** que calcula en tiempo lineal el tipo más general. El que veremos en el laboratorio es el algortmo **J**. Dejaremos este algoritmo a lo que se abordará en la práctica 5 y al material que se haya cubierto en el curso correspondiente de Lenguajes de Programación.

Clasificación de lenguajes de programación

Existen muchas características que permiten clasificar a los lenguajes de programación. Tradicionalmente, los paradigmas o estilos de programación son los que definen una clasificación pero en este curso veremos que es más útil clasificarlos de acuedo a los tipos, el estilo de tipificación y otras características relacionadas con los tipos.

Fuertemente tipificados
: Estos lenguajes no permiten la aplicación de cualquier operación a cualquier objeto que no sea soportado por dicha operación. La inferencia de tipos asegura obtener el tipo más general para cualquier expresión del lenguaje.

Tipificados estáticamente
: Son lenguajes fuertemente tipificados y la verificación de tipos se realiza en tiempo de compilación. Esta forma restringe la flexibilidad del programa.
Ejemplos: Ada, Pascal, algunas implementaciones de C.

Tipificados dinámicamente
: La verificación se realiza en tiempo de ejecucíon por lo que la ejecución de los programas puede volverse lenta.
Ejemplos: Lisp, Smalltalk, Python, Ruby.

Tipificado polimórfico
: Estos lenguajes son más compactos al permitir que el mismo código sea usado para valores de diferentes tipos, sólamente en tiempo de compilación o de ejecución se verifica la compatibilidad de los valores con los tipos. Existen diferentes formas de polimorfismo:

  - paramétrico, por ejemlo el usado en OCaml
  - subtipado, característica de algunos lenguajes orientados a objetos como Java
  - polimorfismo ad-hoc, lenguajes con sobrecarga de operadores

Lenguajes seguros
: Aquellos lenguajes que cumplen con la propiedad de seguridad, por ejemplo Lisp, Haskell, Java.

Lenguajes casi seguros
: Aquellos lenguajes que cumplen con la propiedad de progreso o de preservación pero no ambas, por ejemplo Pascal o Ada.

Lenguajes inseguros
: Aquellos lenguajes que **no** cumplen con la propiedad de seguridad como por ejemplo Algol, C o C++.

Observaciones

- No es cierto que un lenguaje fuertemente tipificado requiera de anotaciones de tipo ya que los tipos se pueden inferir.

Referencias

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Pearson Education Inc., Second edition, 2007.

[2] Jean-Christophe Filliâtre. Curso Compilation (inf564) école Polytechnique, Palaiseau, Francia. `http://www.enseignement.polytechnique.fr/informatique/INF564/`, 2018. Material en francés.

[3] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Springer-Verlag, Berlin, Heidelberg, 2007.

[4] Frank Pfenning. Notas del curso (15-411) Compiler Design, 2014. `https://www.cs.cmu.edu/~fp/courses/15411-f14/`.

[5] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[6] Michael Lee Scott. *Programming Language Pragmatics*. Morgan-Kauffman Publishers, Third edition, 2009.

[7] Yunlin Su and Song Y. Yan. *Principles of Compilers, A New Approach to Compilers Including the Algebraic Method*. Springer-Verlag, Berlin Heidelberg, 2011.

[8] Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Compiler Design*. Springer-Verlag Berlin Heidelberg, 2013.

[9] Steve Zdancewic. Notas del curso (CIS 341) - Compilers, Universidad de Pennsylvania, Estados Unidos, 2018. `https://www.cis.upenn.edu/~cis341/current/`.