Sistema de Tipos

El propósito fundamental de un *sistema de tipos* es del de prevenir las ocurrencias de *errores en tiempo de ejecución* de un programa. La formalización de un sistema de tipos requiere el desarrollo de definiciones y notaciones precisas además de pruebas formales que dan la confianza a las definiciones.

Cuando el sistema de tipos se encuentra correctamente desarrollado, proveen herramientas conceptuales con las cuales se puede juzgar los aspectos de la definición de un lenguaje. Los lenguajes informales generalmente fallan en especificar una estructura de tipos para el lenguaje en detalle permitiendo la implementación de sistemas de tipos ambiguos.

Una notación muy común utilizada para describir sistemas de tipos se basa en **juicios**, que son aseveraciones formales acerca de los tipos de un programa, **reglas de inferencia** que son implicaciones entre juicios y **derivaciones o teoremas** que son deducciones basadas en las reglas de inferencia.

Lenguajes Tipificados y No-Tipificados

Lenguajes en donde las variables se les pueden asignar tipos no triviales son llamados lenguajes tipificados. Lenguajes que no restringen el rango de tipos asignados son llamados lenguajes No-tipificados, no poseen tipos o equivalentemente, poseen un tipo universal que contiene a todas las variables.

Un sistema de tipos es un componente de los Lenguajes Tipificados que lleva el con trol de los tipos de las variables y en general de los tipos de todas las expresiones en un programa. Los sistemas de tipos son utilizados para determinar si un programa está *bien comportado*.

Un lenguaje es tipificado en virtud de la existencia de un sistema de tipos para él, ya sea que aparezcan los tipos en la sintaxis del programa o no. Los lenguajes tipificados son *explícitamente tipificados* si los tipos son parte de la sintaxis e *implícitamente tipificados* si no es así.

Errores en tiempo de ejecución y seguridad

Existen dos tipos de errores en tiempo de ejecución, aquellos que causan el aborto de programa automáticamente y aquellos que no son notados hasta que causan un comportamiento extraño. Los primeros son errores *interceptados,* los segundos son errores *no-interceptados*.

Un fragmento del programa (o lenguaje de programación) se dice que es *seguro* si en este no pueden ocurrir errores *no-interceptados*. Lenguajes no tipificados implementan la seguridad por medio de evaluación en tiempos de ejecución (análisis dinámico), en cambio lenguajes tipificados usan una mezcla entre revisión estática y evaluación dinámica, rechazando

estáticamente todos aquellos programas que puedan ser eventualmente inseguros.

Un programa tiene un *buen comportamiento* si no causa ningún error *no-interceptado*. Un lenguaje en donde todos los programas legales tienen un buen comportamiento es llamado *fuertemente tipificado*. En un lenguaje *fuertemente tipificado* se tiene:

- No genera errores *no-interceptados*.
- No genera algunos errores *interceptados*.
- Para el resto de errores *interceptados*, es responsabilidad del programador evitar su ocurrencia

Lenguajes tipificados pueden forzar el *buen comportamiento* implementando una *revisión estática* (tiempo de compilación) para prevenir la ejecución de programas *inseguros*. El proceso de revisión se denomina *comprobación de tipos* y el algoritmo se llama *comprobador de tipos*.

Lenguajes no tipificados pueden forzar el *buen comportamiento* implementando una *revisión dinámica* (tiempo de ejecución). El proceso de revisión se denomina *comprobación dinámica de tipos*.

El hecho que un lenguaje sea revisado estáticamente, no significa que la ejecución del programa no genere errores. Es por ello que los *lenguajes tipificados* pueden utilizar también una *revisión dinámica*.

En la realidad, *la comprobación estática* no asegura que un programa sea seguro o no. Los lenguajes de este tipo son llamados *débilmente tipificados*.

	Tipificado	**No-tipificado**
Seguro	ML	LISP
Inseguro	C	Assembler

¿Deben los lenguajes de programación ser *seguros?*

La elección entre un lenguaje *seguro e inseguro* se relaciona al final con el balance entre tiempo de desarrollo y tiempo de ejecución.

¿Deben los lenguajes ser *tipificados?*

Desde el punto de vista de la ingeniería, se tienen los siguientes argumentos a favor de lenguajes tipificados.

- **Economía en tiempo de ejecución.** Una información de tipos correcta en tiempo de compilación lleva a la aplicación en tiempo de ejecución a operaciones apropiadas sin la necesidad de demasiadas validaciones en tiempo de ejecución.
- **Economía en desarrollo a pequeña escala.** Con un sistema de tipos bien diseñado, el comprobador de tipos puede capturar una larga

fracción de errores de programación, eliminando la necesidad de largas sesiones de revisión.

- **Economía en tiempo de compilación y desarrollo a gran escala.** La información de tipos puede ser organizada en *interfaces* para módulos de programación. Estos módulos pueden ser compilados por separado, con cada módulo dependiendo solamente en dichas interfaces. Un cambio en una interfaz no implica la re-compilación de todos los módulos. El contrato entre interfaces ayuda a que los cambios sean ordenados y sin efectos secundarios.
- **Economía en características del lenguaje.** El uso de un sistema de tipos ayuda a reducir la complejidad de los lenguajes de programación.

Propiedades de los Sistemas de Tipos

En general, características acerca de los lenguajes de programación pueden definirse de una manera informal, hasta una rigurosa definición formal sujeta a demostraciones. Los sistemas de tipos están en medio de dicho espectro. Un sistema de tipos debe de ser:

- **Verificable y decidible.** Debe de existir un algoritmo que asegure que un programa esté bien *tipificado o buen comportado*.
- **Transparente.** Un programador debe de ser capaz de predecir fácilmente cuando un programa esté bien *tipificado*, y si falla la prueba, la razón del fallo debe de ser evidente.
- **Realizable.** Un sistema de tipos debe de ser implementado, ya sea estáticamente o dinámicamente.

Formalización de Sistemas de Tipos

- Describir la sintaxis de los tipos.
- Determinar las reglas de ámbito (dinámico vs. estático-léxico)
- Definir las reglas de tipos (tiene tipo, subtipo, equidad) (M:A)
    - Ambientes de tipos estáticos ($\Gamma \vdash M:A$)
- Definir la semántica de tipos. La semántica y el sistema de tipos de un lenguaje están interconectados (operaciones, paso de parámetros, tipos de expresiones). Otra forma de verlo, es la pertenencia de elementos en clases de equivalencia.

Equivalencia de tipos

Equivalencia estructural vs. Equivalencia por nombre

El lenguaje del Sistema de Tipos

Un sistema de tipos especifica las reglas de tipos de un lenguaje de programación independientemente del algoritmo de análisis de tipos.

Juicios o proposiciones

Los juicios más importantes son los *juicios o proposiciones de tipos*, los cuales aseveran que un término M tiene un tipo A respecto a un ambiente de tipos. Tiene la forma:

$\Gamma \vdash M : A$ — M has type A in Γ

$\emptyset \vdash true : Bool$ — *true* has type *Bool*

$\emptyset, x{:}Nat \vdash x+1 : Nat$ — $x+1$ has type *Nat*, provided that x has type *Nat*

Reglas de Inferencia

Las reglas aseguran la valides de ciertos juicos basados en otros juicios que se saben de antemano que son válidos.

$$\frac{\text{(Rule name)}\quad \text{(Annotations)}}{\Gamma_1 \vdash \Im_1 \quad \ldots \quad \Gamma_n \vdash \Im_n \quad \text{(Annotations)}} \qquad \frac{}{\Gamma \vdash \Im}$$

General form of a type rule.

Cada regla de inferencia es escrita con un número finito de premisas arriba de una línea horizontal seguida de una única juicio de conclusión. Cuando todas las premisas se cumplen, la conclusión sigue.

$$\frac{\text{(Val } n\text{)}\quad (n = 0, 1, \ldots)}{\Gamma \vdash \diamond} \qquad \frac{}{\Gamma \vdash n : Nat}$$

$$\frac{\text{(Val +)}}{\Gamma \vdash M : Nat \quad \Gamma \vdash N : Nat} \qquad \frac{}{\Gamma \vdash M+N : Nat}$$

Una colección de *reglas de inferencia* es llamad un *sistema formal de tipos*. Técnicamente un sistema de tipos entra en un sistema más general denominado *sistema formal de pruebas*, una colección de reglas utilizadas para elaborar deducciones paso a paso. Las deducciones que se llevan a cabo con un sistema de tipos es la tipificación de programas.

Un juicio válido es aquel que puede obtenerse con una aplicación de reglas de inferencia válidas.

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \diamond}\ \text{by (Env } \emptyset\text{)}}{\emptyset \vdash 1 : Nat}\ \text{by (Val } n\text{)} \qquad \dfrac{\dfrac{}{\emptyset \vdash \diamond}\ \text{by (Env } \emptyset\text{)}}{\emptyset \vdash 2 : Nat}\ \text{by (Val } n\text{)}}{\emptyset \vdash 1+2 : Nat}\ \text{by (Val +)}$$

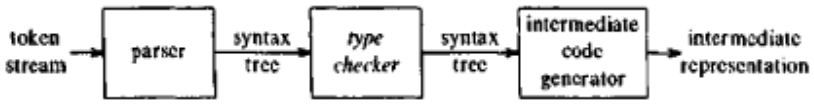
Análisis Semántico

Un compilador debe de revisar el programa fuente, de tal forma que las convecciones sintácticas y semánticas del programa sean válidas. La comprobación *estática* asegura que ciertas clases de errores de programación sean detectados y reportados. Ejemplos de estas revisiones son:

- Comprobación de tipos.
- Revisión de Flujos de Control (ej. break)
- Pruebas de unicidad (Nombres de variables)
- Revisión de nombres (Nombre de CoCoL)

Comprobador de tipos

Un comprobador de tipos verifica que el tipo de una construcción coincida con el tipo esperado en su contexto.



Sistema de Tipos

El diseño de un comprobador de tipos para un lenguaje de programación se basa en la información acerca de la sintaxis del lenguaje, la noción de tipos y las reglas de asignar tipos a las construcciones del lenguaje.

El propósito fundamental de un *sistema de tipos* es del de prevenir las ocurrencias de *errores en tiempo de ejecución* de un programa. La formalización de un sistema de tipos requiere el desarrollo de definiciones y notaciones precisas además de pruebas formales que dan la confianza a las definiciones.

Cuando el sistema de tipos se encuentra correctamente desarrollado, proveen herramientas conceptuales con las cuales se puede juzgar los aspectos de la definición de un lenguaje. Los lenguajes informales generalmente fallan en especificar una estructura de tipos para el lenguaje en detalle permitiendo la implementación de sistemas de tipos ambiguos.

Una notación muy común utilizada para describir sistemas de tipos se basa en **juicios**, que son aseveraciones formales acerca de los tipos de un programa, **reglas de inferencia** que son implicaciones entre juicios y **derivaciones o teoremas** que son deducciones basadas en las reglas de inferencia.

Expresiones de Tipos

Los tipos de las construcciones de un lenguaje se denotan por medio de *expresiones de tipos*, una *expresión de tipo* es ya sea un tipo básico o se encuentra formado por medio de aplicaciones de operadores llamados *constructores de tipos*. El conjunto de tipos básicos y de operadores depende del programa a ser revisado.

Def. Expresiones de Tipos

1. Un tipo básico es una expresión de tipo. (boolean, char, integer, real). Un tipo especial: *error*, un tipo básico *void* que permite la asignación de tipos a expresiones.
2. Nombres de expresiones de tipos. Las expresiones de tipos pueden nombrarse por medio de alias.
3. Un constructor de tipos se aplican a expresiones de tipos para producir otras expresiones de tipos.
    a. *Arreglos.* Si T ex una expresión de tipo, entonces *array(I, T)* es una expresión de tipo que denota un arreglo con elementos de tipo T y conjunto de índices I. I es generalmente un rango de números enteros.
    b. *Producto.* Si T1 y T2 son expresiones de tipos, entonces su producto cartesiano T1 x T2 es una expresión de tipo.
    c. *Records.* La diferencia entre un registro y un producto es que los campos de los registros tienen nombres. El constructor de tipos *record* se aplicará a una tupla formada por nombres y tipos de campos, *record((name1 x type1) x (name2 x type2)…)*
    d. *Pointers.* Si T es una expresión de tipo, entonces *pointer(T)* es una expresión de tipo de que denota el tipo "puntero a un objeto de tipo T".
    e. *Funciones.* Matemáticamente, una función mapea elementos de un dominio hacia otro conjunto denominado rango. En lenguajes de programación el mapeo es un dominio D hacia un rango R. El tipo de dicha función se denota D->R. (Ej. int x int -> int) ((int -> int) -> (int -> int))
4. Expresiones de tipos pueden contener variables, cuyos valores son expresiones de tipos.

Fig. 6.2. Tree and dag, respectively, for *char* × *char* → *pointer*(*integer*).

Sistema de Tipos

Un sistema de tipos es una colección de reglas para asignar *expresiones de tipos* a las distintas partes de un programa. Un comprador de tipos es la *implementación de un sistema de tipos*.

Sistema de tipos Sencillo

$P \rightarrow D\ ;\ E$
$D \rightarrow D\ ;\ D \mid \textbf{id}\ :\ T$
$T \rightarrow \textbf{char} \mid \textbf{integer} \mid \textbf{array}\ [\ \textbf{num}\ ]\ \textbf{of}\ T \mid \uparrow T$
$E \rightarrow \textbf{literal} \mid \textbf{num} \mid \textbf{id} \mid E\ \textbf{mod}\ E \mid E\ [\ E\ ] \mid E\uparrow$

Validación de Tipos para Declaración de Tipos

$P \rightarrow D\ ;\ E$	
$D \rightarrow D\ ;\ D$	
$D \rightarrow \textbf{id}\ :\ T$	{ *addtype*(**id**.*entry*, *T.type*) }
$T \rightarrow \textbf{char}$	{ *T.type* := *char* }
$T \rightarrow \textbf{integer}$	{ *T.type* := *integer* }
$T \rightarrow \uparrow T_1$	{ *T.type* := *pointer*(T_1.*type*) }
$T \rightarrow \textbf{array}\ [\ \textbf{num}\ ]\ \textbf{of}\ T_1$	{ *T.type* := *array*(1..**num**.*val*, T_1.*type*) }

Validación de Tipos para Expresiones

$E \rightarrow \textbf{literal}$	{ *E.type* := *char* }
$E \rightarrow \textbf{num}$	{ *E.type* := *integer* }
$E \rightarrow \textbf{id}$	{ *E.type* := *lookup*(**id**.*entry*) }
$E \rightarrow E_1\ \textbf{mod}\ E_2$	{ *E.type* := **if** E_1.*type* = *integer* **and** E_2.*type* = *integer* **then** *integer* **else** *type_error* }
$E \rightarrow E_1\ [\ E_2\ ]$	{ *E.type* := **if** E_2.*type* = *integer* **and** E_1.*type* = *array*(*s*, *t*) **then** *t* **else** *type_error* }
$E \rightarrow E_1\uparrow$	{ *E.type* := **if** E_1.*type* = *pointer*(*t*) **then** *t* **else** *type_error* }

Validación de Tipos para Sentencias

Las sentencias en los lenguajes de programación generalmente no tienen valores, pero se les puede asignar un tipo básico *void* a ellas. Si un error es detectado dentro de la sentencia, el tipo asignado es *error*.

$S \rightarrow \textbf{id} := E$ { $S.type :=$ **if** $\textbf{id}.type = E.type$ **then** *void* **else** *type_error* }

$S \rightarrow \textbf{if } E \textbf{ then } S_1$ { $S.type :=$ **if** $E.type = boolean$ **then** $S_1.type$ **else** *type_error* }

$S \rightarrow \textbf{while } E \textbf{ do } S_1$ { $S.type :=$ **if** $E.type = boolean$ **then** $S_1.type$ **else** *type_error* }

$S \rightarrow S_1 ; S_2$ { $S.type :=$ **if** $S_1.type = void$ **and** $S_2.type = void$ **then** *void* **else** *type_error* }

Validación de Tipos para Funciones

$E \rightarrow E_1 (E_2)$ { $E.type :=$ **if** $E_2.type = s$ **and** $E_1.type = s \rightarrow t$ **then** t **else** *type_error* }

Equivalencia de Expresiones de Tipo

Las expresiones de tipos de la última sección tienen la forma *if two type expressions are equal then retrun a certain type else return type_error*. La noción de la implementación de equivalencia de tipos puede explicarse utilizando los conceptos de equivalencia estructural o equivalencia por nombre.

Equivalencia estructural de expresiones de Tipos

Siempre y cuando las expresiones de tipo se construyen por medio de tipos básicos y constructores de tipos, dos expresiones pueden ser del mismo tipo básico o se encuentran formados por la aplicación del mismo constructor de tipo.

Dos expresiones de tipos son equivalentes estructuralmente si y solo sí estas son el mismo tipo básico o resultan de aplicar los mismos constructores de tipos a tipos equivalentes. (Ej. *pointer(integer) = pointer(integer)*)

```
(1)  function sequiv(s, t): boolean;
     begin
(2)      if s and t are the same basic type then
(3)          return true
(4)      else if s = array(s1, s2) and t = array(t1, t2) then
(5)          return sequiv(s1, t1) and sequiv(s2, t2)
(6)      else if s = s1 × s2 and t = t1 × t2 then
(7)          return sequiv(s1, t1) and sequiv(s2, t2)
(8)      else if s = pointer(s1) and t = pointer(t1) then
(9)          return sequiv(s1, t1)
(10)     else if s = s1 → s2 and t = t1 → t2 then
(11)         return sequiv(s1, t1) and sequiv(s2, t2)
         else
(12)         return false
     end
```

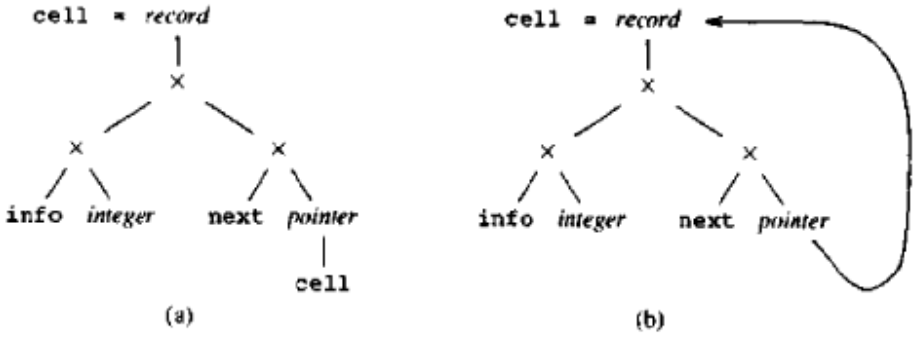
Equivalencia por nombre de Expresiones de Tipo

En algunos lenguajes de programación, se pueden generar nuevos tipos de datos a los cuales se les coloca un nombre.

```
type link = ↑ cell;
var  next : link;
     last : link;
     p    : ↑ cell;
     q, r : ↑ cell;
```

Para modelar dicha situación, permitiremos a las expresiones de tipo se les puedan asignar nombres y permitir a dichos nombres a aparecer en expresiones de tipo a donde antes solo existían tipos primitivos. Cuando estos nombres son permitidos surgen *dos nociones* de equivalencia de tipos, dependiendo de la forma en que dichos nombres son tratados. **La equivalencia por nombres** ve cada nombre de tipo como un tipo distinto, ***así que dos expresiones de tipo son equivalentes si y solo sí estos nombres son idénticos.*** Bajo la equivalencia estructural, los nombres son reemplazados por su expresión de tipo equivalente, y estas expresiones de tipo son *estructuralmente equivalentes* si estas representan dos expresiones de tipo estructuralmente equivalentes, cuando todos los nombres ya se han sustituido.

Ciclos en Representación de Tipos



Conversión de Tipos

Conversión implícita vs Conversión Explícita

La conversión de un tipo hacia otro tipo se denomina implícita si esta es echa automáticamente por el compilador. La conversión se dice que es explícita si el programador debe de escribir *algo* para causar dicha conversión. La conversión explícita es análoga a la aplicación de una función a una expresión de tipo en particular.

PRODUCTION	SEMANTIC RULE
$E \rightarrow$ **num**	$E.type :=$ *integer*
$E \rightarrow$ **num . num**	$E.type :=$ *real*
$E \rightarrow$ **id**	$E.type :=$ *lookup*(**id**.*entry*)
$E \rightarrow E_1$ **op** E_2	$E.type :=$ **if** $E_1.type =$ *integer* **and** $E_2.type =$ *integer* **then** *integer* **else if** $E_1.type =$ *integer* **and** $E_2.type =$ *real* **then** *real* **else if** $E_1.type =$ *real* **and** $E_2.type =$ *integer* **then** *real* **else if** $E_1.type =$ *real* **and** $E_2.type =$ *real* **then** *real* **else** *type_error*