Tipos

Albert Rubio

Procesadores de Lenguajes, FdI, UCM

Doble Grado Matématicas e Informática

Tipos

Contenidos

Sistema de Tipos

Un *sistema de tipos* es un conjunto de reglas asociado a un lenguaje de programación, que

- permiten asignar un tipo a las construcciones sintácticas
- comprobar si están bien formadas.

Las reglas incluyen la definición de equivalencia y de compatibilidad de tipos.

La evolución de los lenguajes de programación ha ido en el sentido de introducir sistemas de tipos cada vez más elaborados.

Tipos en los lenguajes de programación

Los tipos restringen las operaciones que se pueden aplicar a un objeto.

No toda construcción sintácticamente correcta es válida

A cambio de perder flexibilidad ofrecen las siguientes ventajas:

- **Seguridad**: Permiten la detección temprana de errores, haciendo más corta la fase de depuración y evitando errores en producción.
- **Legibilidad**: Dan a conocer mejor las intenciones del programador. Facilitan el mantenimiento del programa.
- **Abstracción**: Permiten aplazar y encapsular las decisiones de representación de tipos complejos. Aumentan la modificabilidad.
- **Reutilización**: La genericidad, el polimorfismo, la herencia y la sobrecarga permiten asignar muchos tipos distintos a un mismo fragmento de código. Aumenta la reutilización de código (que a su vez aumenta la seguridad).

Evolución de los sistemas de tipos

Los tipos ofrecidos por los lenguajes máquina no son suficientes: instrucciones, direcciones, caracteres, enteros, números en coma flotante.

A lo largo del tiempo se han ido extendiendo los sistemas de tipos:

- tipos relacionados con el hardware (1956-59).
  enteros, coma flotante y arrays
- booleanos, registros (1960).
- tipos definidos por el programador (1971).
- tipos abstractos de datos (1974).
- clases y herencia (1981).
- tipos genéricos (1983).
- sobrecarga definida por el programador (1983).
- polimorfismo paramétrico (1984).
- prototipos (1987).

Propiedades

El tipado en un lenguaje se basa en una serie de características:

- Declaración de tipos: explícita, implícita.
- Disciplina de tipos (type safety): fuerte, débil, ninguna.
- Comparación de tipos: estática, dinámica.
- Compatibilidad de tipos: por nombre (nominal), estructural.

Algunos lenguajes pueden combinar más de una opción. Por ejemplo, en declaración de tipos (Haskell) o en comprobación de tipos (atributo `dynamic` en C#).

Declaración de tipos

Esta propiedad nos indica si el lenguaje requiere que se indiquen los tipos de las variables o identificadores en general.

- Explícita. El programador indica explícitamente el tipo de cada variable. *Manifest typing*.

  Implica realizar una *comprobación de* que los *tipos* indicados son compatibles con los usos que se hagan.

  Si no es compatible se produce un error.

- Implícita. El programador **no** indica el tipo y este se deduce del uso que se realiza. *Inferred typing*.

  Implica realizar una *inferencia de los tipos* compatible con los usos que se hagan. Si no es existe solución se produce un error.

  Para simplificar esta inferencia algunos lenguajes requieren que el primer uso de un identificador determine su tipo.

Disciplina de tipos

La disciplina de tipos está relacionada con la posibilidad de que se produzcan errores relacionados con los tipo en ejecución.

- Tipado fuerte: si tiene mecanismos para forzar que a cada objeto se le apliquen exclusivamente las operaciones de su tipo.
  Por ejemplo Haskell, Java o C++.
- Tipado débil: en caso contrario. Puede aplicar conversiones implícitas de tipo o, simplemente, el tipo del resultado puede ser impredecible.
  Usual en lenguajes de scripting (por ejemplo, JavaScript).
- Seguridad de memoria (memory safety): se garantiza que no se producirán errores relacionados con accesos a memoria.
  Ejemplos de errores: desbordamientos de búfer, referencias colgantes.
  Son comunes en presencia de punteros (C, C++).

Comprobación de tipos

El momento en que se realiza la comprobación de tipos es relevante tanto para la eficiencia del lenguaje como para definir su semántica:

- Estática. Si las comprobaciones de tipos se hacen en tiempo de compilación.

  Más eficiente pero menos flexible (cualquier posible ejecución tiene que cumplir las reglas de tipado).

  La identificación de identificadores es estática.

- Dinámica. Si las comprobaciones de tipos se hacen en tiempo de ejecución.

  Se puede entender que el tipo está asociado al valor y no a la variable.

  Implica ineficiencia en tiempo y espacio. Aumenta la flexibilidad.

  La identificación de identificadores es dinámica.

Equivalencia de tipos

En los lenguajes en que se pueden definir nuevos nombres de tipo, según si se considera que son un alias o no podemos tener distintas equivalencias de tipos.

- Por nombre: Dos tipos son iguales si tienen el mismo nombre. Se aplica recursivamente.
- Estructural: Dos tipos son iguales si, haciendo abstracción de los nombres intermedios, son la misma expresión de tipos.
- En muchos lenguajes moderno se combinan ambas equivalencias.
- Duck typing: En tipado dinámico, un objeto es compatible con un tipo si tiene todos los atributos y métodos necesarios en el momento de ser usado.

Ortogonalidad

La *ortogonalidad* es una propiedad importante en el diseño de un sistema de tipos.

Las reglas del sistema deben ser lo más generales posibles
evitando las excepciones

Ejemplos de ortogonalidad:

- Todas las funciones devuelven un valor de algún tipo. Se usa el tipo void cuando no existe tal valor (C, C++, Java, etc).
- Los tipos pueden anidarse sin apenas restricciones.
- El tipo del resultado de una función puede ser cualquiera, incluso una función (Haskell).
- Permitir definir constantes literales de cualquier tipo.

Contenidos

Mecanismos de definición de tipos

Normalmente los LP disponen de

1. *Tipos primitivos*, dotados de valores y operaciones predefinidos.
2. *Plantillas genéricas* predefinidas, que se pueden componer y concretar, para formar *tipos compuestos*.
3. La posibilidad de *declarar* nuevos tipos:
   - por *renombrado* de un tipo existente.
   - por *enumeración* explícita del conjunto de valores.
   - por *concreción* de un tipo genérico.
   - por creación de un *subtipo* de otro tipo existente.

Tipos primitivos

Normalmente esto incluye los siguientes tipos:

- *Booleanos*. Algunos lenguajes usan los mismos enteros (C). Aunque pueden representarse en un byte, no siempre se hace así (alineamiento de memoria). Tiene operaciones de conjunción, disyunción, etc.
- *Carácter*. Puede ser uno o dos bytes según la codificación. Pueden soportar algunas operaciones aritméticas o relacionales.
- *Enteros*. Pueden tener un tamaño máximo (representación en bytes), en lenguajes como Java o C++, o pueden ser ilimitados como en Haskell. Pueden distinguirse según los bytes que ocupen. Por ejemplo en C++: int, long int, long long int. Soportan las operaciones aritméticas y relacionales habituales.
- Números en *coma flotante*. Normalmente se distingue entre **float** (4 bytes) o **double** (8 bytes). Algunos lenguajes admiten números racionales o complejos como un par de enteros o números en coma flotante (respectivamente).

Renombrado

Se define una equivalencia entre un tipo existente y un nuevo nombre de tipo.

El nuevo tipo hereda todas las operaciones del tipo existente.

Este renombrado puede crear:

- un tipo nuevo incompatible con el existente. Por ejemplo, `type` en Ada.
- un alias que a todos los efectos es el mismo tipo. Por ejemplo `type` en Haskell o `typedef` en C.

Enumeración

- En su forma más usual se trata de una lista de identificadores representados internamente (por defecto) como números $0, 1, 2, \ldots$.
  Esta numeración puede cambiarse en algunos lenguajes.

  ```
  enum semana {lun, mar, mie, jue, vie, sab, dom};
  ```

  Están equipados con operaciones aritméticas y, en algunos lenguajes, pueden usarse como índices de vectores o de bucles `for`.

- En lenguajes funcionales pueden usarse para definir tipos algebraicos.

  ```
  data Pila a = PVacia | Apilar a (Pila a)
  data ArBin a = AVacio | Nodo (ArBin a) a (ArBin a)
  ```

  Estos ejemplos son además tipos polimórficos: la variable de tipo `a` representa cualquier posible tipo.

  Lenguajes como Rust también permiten este tipo de enumeraciones.

El lenguaje proporciona mecanismos para definir fácilmente operaciones sobre los valores de la enumeración.

Concreción

- En la mayoría de los lenguajes imperativos es posible definir tipos a partir de plantillas como:

```
type miVector = array [T1] of T2;
type miRegistro = record
                    campoA : T3;
                    campoB : T4;
                    campoC : T5
                  end;
type miPuntero = access T6;
```

  donde `T1` es un tipo escalar y `T2`...`T6` son tipos cualesquiera.

- Son *tipos parametrizados* o *tipos genéricos*. El programador ha de *concretar* los parámetros para para obtener tipos válidos:

```
type miVector = array [1..100] of integer;
```

Definición de tipos genéricos

Los tipos genéricos también pueden ser definidos por el programador (generic en Ada, template en C++).

Admiten como parámetros

- tipos,
- operaciones sobre ellos,
- constantes,
- Otros tipos genéricos.

No pueden ejecutarse en su estado genérico. Deben ser concretadas para que el compilador genere código.

Cada concreción genera un código distinto. Puede ralentizar la compilación.

Polimorfismo paramétrico

Una alternativa a la construcciones genéricas el *polimorfismo paramétrico*. Presente en muchos lenguajes funcionales.

```
fst  :: (a,b) -> a
fst (x,_) = x
```

- No es tan potente como las construcciones genéricas: no admite parametrización con constantes y tiene una parametrización con operaciones limitada.
- El compilador genera código una sola vez para la definición polimórfica. Todas sus concreciones utilizan el mismo código.

Subtipos

Decimos que t_2 es subtipo de t_1, si toda operación de t_1 es aplicable a t_2

Llamaremos a t_1 el tipo padre, o supertipo, de t_2.

En los lenguajes orientados a objetos se crean relaciones de subtipado a través de la *herencia*.

- Si una clase B se crea como subclase de una clase A, entonces B hereda todas las operaciones de A y su representación interna.
- B puede **extender** el conjunto de métodos de A y su representación interna.
- B puede **redefinir** alguno de los métodos de A, pero sin cambiar el interfaz de A.

B es un **subtipo** de A, ya que toda operación de A puede aplicarse a B.

No es una restricción del conjunto de valores, sino una **extensión** del conjunto de operaciones.

Contenidos

Equivalencia y compatibilidad

- La equivalencia (por nombre o estructural) de tipos nos dice si un tipo es el esperado para un uso determinado
- La compatibilidad de tipos nos dice si un tipo puede ser usado en cierto contexto, a pesar de no ser equivalente.

  Normalmente está relacionado con una relación de subtipado.

  - Si se trata de subtipado por herencia no es necesario hacer cambios en la representación.

    Pero si se aplica un método a un objeto hay que tomar su definición más cercana en la relación de subtipado.

    Puede ser que el método a utilizar solo se sepa en ejecución: vinculación dinámica (dynamic binding).
  - Si se trata de subtipado por subconjunto de valores, por ejemplo de entero a real, se realiza (normalmente) una conversión de tipos.

Reglas de tipado

La *comprobación de tipos* (type checking) consiste en asegurar que todas las expresiones e instrucciones del programa son correctas desde el punto de vista de las *reglas de tipado* del lenguaje.

Un *juicio* (judgment) es una aserción de la forma $\Gamma \vdash e : t$, donde

- Γ es un **entorno de tipado** que contiene vínculos de la forma $[x_i : t_i]$ que dan el tipo t_i de cada identificador x_i,
- e es la expresión a tipar, y
- t es un tipo para e bajo el entorno Γ.

Un juicio es valido si el tipo es válido para la expresión en el entorno.
Una *regla de tipado* es una regla de la forma

$$\frac{J_1 \ldots J_n}{J}$$

Que indica que J es un juicio válido si lo son $J_1 \ldots, J_n$.

Reglas de tipado. Ejemplos

$$\frac{}{\Gamma \vdash n : int}\ LITE \qquad \frac{}{\Gamma \vdash b : bool}\ LITB \qquad \frac{x : t \in \Gamma}{\Gamma \vdash x : t}\ VAR$$

$$\frac{\Gamma \vdash e_1 : t_1 \ \cdots \ \Gamma \vdash e_n : t_n \quad \Gamma \vdash f : t_1 \times \ldots \times t_n \rightarrow t}{\Gamma \vdash f(e_1, \ldots, e_n) : t}\ APP$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : int}\ ADD \qquad \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1 \ || \ e_2 : bool}\ OR$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 < e_2 : bool}\ LESS \qquad \frac{\Gamma \vdash e_1 : \textbf{array}\ [n]\ \textbf{of}\ t \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ [e_2] : t}\ ARR$$

$$\frac{\Gamma \vdash e : \textbf{pointer to}\ t}{\Gamma \vdash e\uparrow : t}\ PTR \qquad \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : t}\ IF$$

Sistema de tipos (Haskell)

Son reglas parecidas al λ-cálculo:

Variable:
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Declaración:
$$\frac{f :: \tau \in \Gamma}{\Gamma \vdash f : \tau\xi}$$

Abstracción:
$$\frac{\Gamma \cdot \{x : \sigma\} \vdash t : \tau}{\Gamma \vdash (\backslash x.t) : \sigma \to \tau}$$

Aplicación:
$$\frac{\Gamma \vdash s : \sigma \to \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash (s\ \ t) : \tau}$$

Let:
$$\frac{\Gamma \vdash s : \sigma \quad \Gamma \cdot \{x : \sigma\} \vdash t : \tau}{\Gamma \vdash \text{let } x = s \text{ in } t : \tau}$$

Instanciación de una declaración polimórfica con contexto:
$$\frac{C[\alpha] => f :: \tau[\alpha] \in \Gamma \quad C[\sigma] \text{ se satisface en } \Gamma}{\Gamma \vdash f : \tau[\sigma]\xi}$$
donde ξ es una substitución de tipos

Sistema de tipos (Java)

Constantes:

$$\frac{}{\vdash 1 : \texttt{Int}} \quad \ldots$$

Variables:

$$\frac{x : T \in E}{E \vdash x : T}$$

Aplicación de función:

$$\frac{E \vdash e_1 : T_1 \quad \ldots \quad E \vdash e_n : T_n}{E \vdash obj.f(e_1, \ldots, e_n) : T}$$

Si $obj : Class\{\ldots T\ f(T_1, \ldots, T_n) \ldots\} \in E$

Sistema de tipos (Java)

Ejemplo de derivación de tipos

$$\frac{\dfrac{x : \mathit{Object}\{\ldots \mathit{String}\ \mathit{toString}()\ldots\} \in E}{E \vdash x.\mathit{toString}() : \mathit{String}}}{E \vdash \mathit{System.out.println}(x.\mathit{toString}()) : \mathit{void}}$$

ya que

$$\mathit{System.out} : \mathit{PrintStream}\{\ldots\ \mathit{void}\ \mathit{println}(\mathit{String})\ldots\} \in E$$

es decir, *System.out* es un objeto de la clase *PrintStream*

- Tenemos que añadir la noción de subtipo
- Extendemos las reglas y añadimos nuevas.

Sistema de tipos (Java con herencia)

Aplicación de función:

$$\frac{E \vdash e_1 : U_1 \leq T_1 \quad \ldots \quad E \vdash e_n : U_n \leq T_n \qquad E \vdash obj : O \leq H}{E \vdash obj.f(e_1, \ldots, e_n) : T}$$

$$\text{Si } H = Class\{\ldots T\ f(T_1, \ldots, T_n) \ldots\} \in E$$

Reglas para el subtipado:

Reflexividad:

$$\frac{}{\vdash T \leq T}$$

Object:

$$\frac{T \text{ no es un tipo primitivo}}{\vdash T \leq Object}$$

Implements:

$$\frac{\texttt{class}\ U\ \texttt{implements}\ T \in E}{E \vdash U \leq T}$$

En general, $U \leq T$ si aparece en E con un `implements` o un `extends`

Sistema de tipos (Java con herencia)

Más reglas para el subtipado

Arrays:

$$\frac{E \vdash U \leq T}{E \vdash U[\,] \leq T[\,]}$$

Transitividad:

$$\frac{E \vdash U \leq T \qquad E \vdash T \leq S}{E \vdash U \leq S}$$

Los métodos se heredan del supertipo más cercano posible.

Problemas con la herencia múltiple.

Sistema de tipos (Java con herencia)

Suponed que tenemos

$$D \leq B \leq A$$
$$D \leq C \leq A$$

Aquí D presenta herencia múltiple de B y C.

Si D llama a un método de A que han redefinido B y C de formas distintas.

¿De quién debe heredar el método?

Java no admite herencia múltiple.

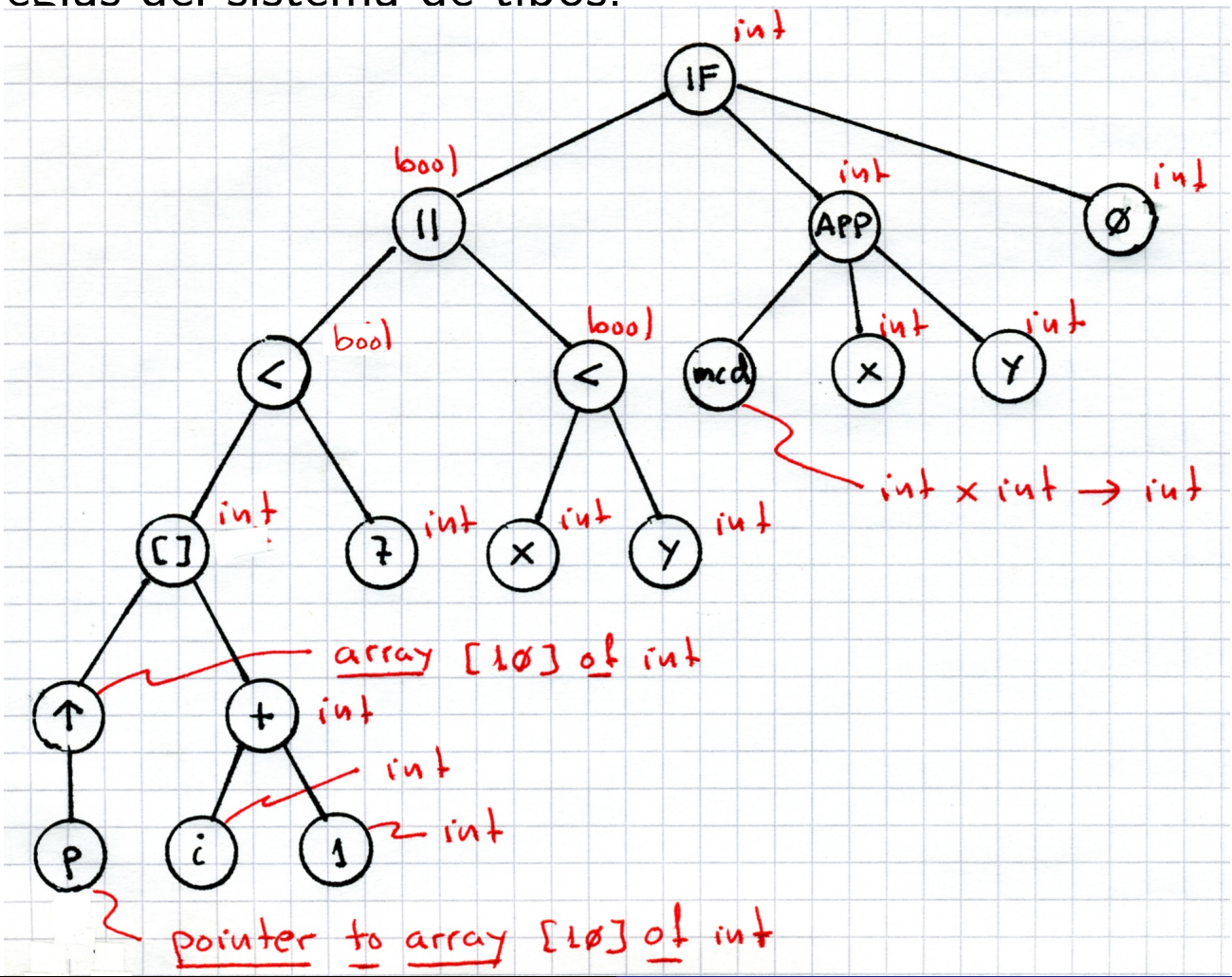
Por ejemplo, C++ y Python sí que la admiten pero de formas distintas.

Comprobación de Tipos en el AST

El proceso de comprobación recorre el árbol en sentido **ascendente**, infiriendo el tipo de un nodo a partir del tipo obtenido para los hijos de ese nodo y las reglas del sistema de tipos.

Comprobación de Tipos en el AST

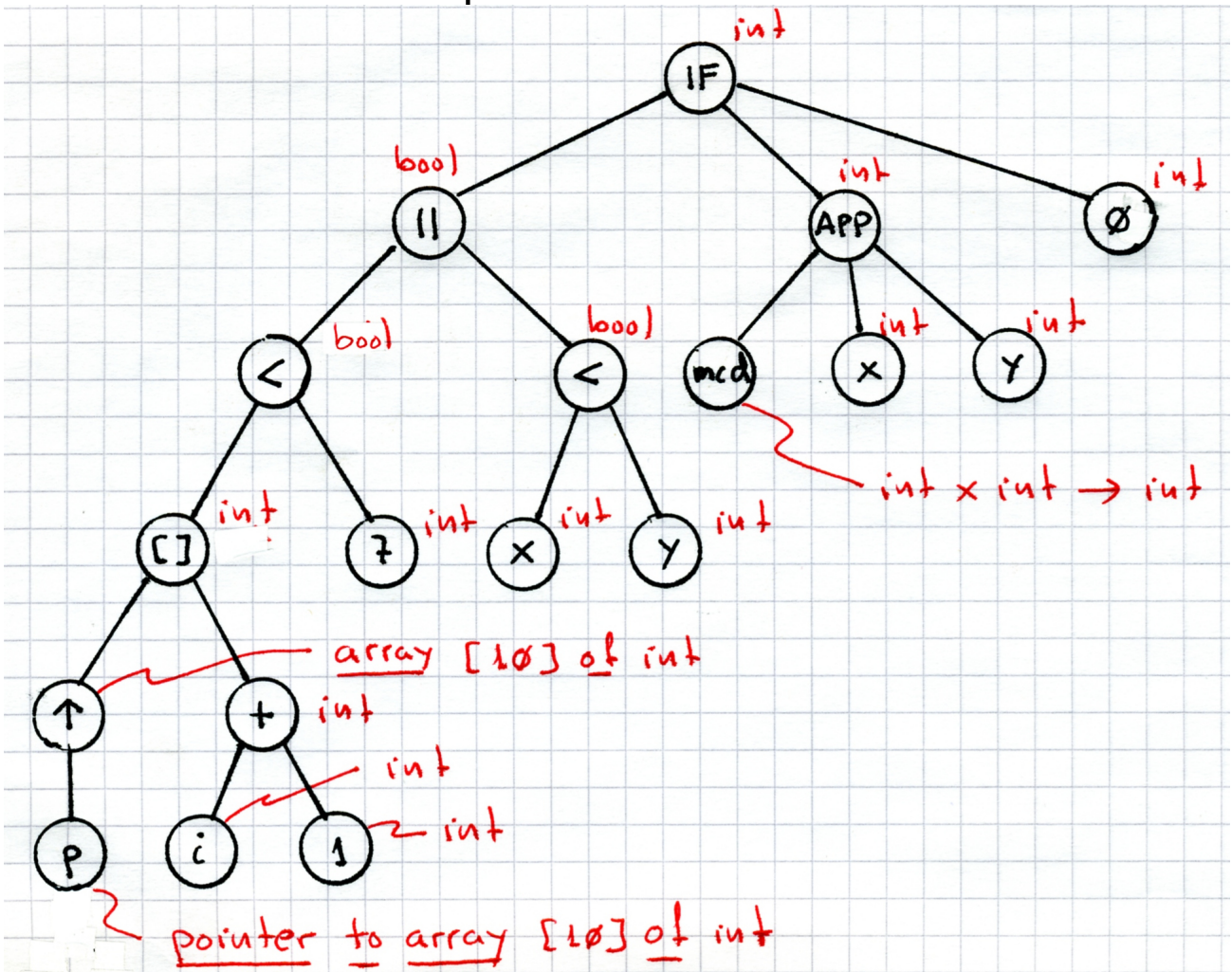
El proceso de comprobación recorre el árbol en sentido **ascendente**, infiriendo el tipo de un nodo a partir del tipo obtenido para los hijos de ese nodo y las reglas del sistema de tipos.



Comprobación de Tipos en el AST

$$e = \textbf{if } p\uparrow[i+1] < 7 \;||\; x < y \textbf{ then } mcd(x,y) \textbf{ else } 0$$

$$\Gamma = [i : int, x : int, y : int, p : \textbf{pointer to array } [10] \textbf{ of } int,]$$
$$[mcd : int \times int \to int]$$



Comprobación de Tipos en el AST

La mecánica es como sigue:

- Es conveniente aplicar primero un proceso de simplificación donde se eliminan los alias (`typedef` en C++, por ejemplo).
- Este proceso reemplaza el vínculo al identificador de tipo (que es un alias) por el vínculo de este a su definición (que debe ser un tipo). Tened en cuenta que puede haber secuencias de alias que hay que reemplazar hasta llegar al tipo (no alias) que representan.
- Una vez realizada la vinculación de identificadores y realizado el proceso de simplificación de tipos podemos hacer la comprobación de tipos recorriendo el AST con una función **chequea**.
- Si queremos hacer otras comprobaciones, también se pueden hacer en este proceso.

Comprobación de Tipos en el AST

- Sobre las declaraciones, definiciones de tipos e instrucciones, **chequea** comprueba de las restricciones de tipo y otras restricciones contextuales.
- Sobre designadores y expresiones, **chequea** fija, además, el valor del atributo **tipo** (y los necesarios para otras comprobaciones) de los nodos intermedio con un vínculo al tipo inferido para dichos nodos.
- La función **chequea** para el lenguaje ejemplo sería:

```
chequea(Ds && Is) {
  foreach D in Ds {
    chequea(D);
  }
  foreach I in Is {
    chequea(I);
  }
}
```

Comprobación de Tipos en el AST

- Definiciones:

```
chequea(var id:T) {
  chequea(T);
}
chequea(type id:T) {
  chequea(T);
}
chequea(proc Id: Proc) {
  chequeaProc(Proc);
}
chequeaProc(proc Ps Ds I) {
  foreach Modo id: T in Ps {
    chequea(T);
  }
  foreach D in Ds {
    chequea(D);
  }
  foreach I in Is {
    chequea(I);
  }
}
```

Comprobación de Tipos en el AST

- Expresiones de tipo:

```
chequea(bool) {}
chequea(int) {}
chequea( T[n] ) {
  chequea(T);
}
chequea(struct Cs) {
  foreach  id: T in Cs {
    chequea(T);
  }
}
chequea(pointer T) {
  chequea(T);
}
```

Comprobación de Tipos en el AST

- Expresiones de designadores:

```
chequea(Id/id/) {
  if (not validoComoDesignador(id.vinculo)) {
    error id debe ser una variable, parámetro o campo;
    t=null;
  } else {
    t = id.vinculo.tipo();
  }
  id.tipo = t;
}
chequea(N/D->/) {
  chequea(D);
  if D.tipo == null then t= null;
  else if D.tipo == pointer T then t=T;
  else {
    error el designador debería ser de tipo puntero;
    t=null;
  }
  N.tipo = t;
}
```

Comprobación de Tipos en el AST

- Expresiones de designadores:

```
chequea(N/D[E]/) {
  chequea(D);
  chequea(E);
  if D==null or E==null then t = null
  else if not es_array(D.tipo) then {
    error el designador debería ser de tipo array;
    t=null;
  } else if E.tipo != int then {
      error el índice debería ser de tipo entero;
      t=null;
  }
  else {
    t = D.tipo;
  }
  N.tipo = t;
}
```

Comprobación de Tipos en el AST

- Expresiones de designadores:

```
chequea(N/D.id/) {
  chequea(D);
  if (D.tipo != null) {
  if (!es_struct(D.tipo)) {
    error el designador debería ser de tipo struct;
    t=null;
  } else {
    m = obtenMiembro(D,id);
    if m == null {
      error miembro inexistente;
      t=null;
    }
    else {
      t = m.tipo;
    }
  }
  N.tipo = t;
}
```

Comprobación de Tipos en el AST

- Instrucciones:

```
chequea(D=E) {
  chequea(D);
  chequea(E);
  if (not compatibles(D.tipo,E.tipo)) error incompatibilidad;
}
chequea(write E) {
  chequea(E);
  if (not tipoPresentable(E.tipo)) error no imprimible;
}
...
chequea(if E I0 I1 ) {
  chequea(E);
  if (E.tipo != bool ) error el tipo de E debe ser booleano;
  chequea(I0);
  chequea(I1);
}
....
```

Comprobación de Tipos en el AST

- Expresiones:

```
chequea(N/true/) {N.tipo = bool; }
...
chequea(N/E0 == E1/) {
  chequea(E0);
  chequea(E1);
  if E0.tipo == null or E1.tipo == null then t=null;
  else if tiposComparables(E 0 .tipo,E 1 .tipo) then {
       t=bool;
       } else {
           error tipos no comparables;
           t = null
  }
  N.tipo = t;
}
...
```

Comprobación de Tipos con sobrecarga

La comprobación de tipos en presencia de **sobrecarga** es más complicada:

- Asociado a un nodo del árbol puede haber más de un tipo posible.
- El algoritmo realiza **varias pasadas** ascendentes y descendentes.
- En cada pasa descarta tipos imposibles.
- Si cada nodo queda con un solo tipo, la expresión es tipada con éxito.
- En otro caso o no es tipable o es ambigua.

Si solo hay sobrecarga por el tipo de los argumentos entonces se puede hacer en una sola pasada (ascendente).

Contenidos

Inferencia de tipos

Mecanismo (re)inventado por Robin Milner

(Curry y Hindley habían desarrollado ideas similares independientemente en el contexto del lambda-cálculo)

Dado un programa en un lenguaje de programación

encontrar el tipo más general para el programa (y todas sus (sub) expresiones) dentro del sistema de tipos del lenguaje.

El algoritmo es similar a la "unificación".

- Siempre presente en los lenguajes funcionales.
- Se ha extendido a otros lenguajes.
  Por ejemplo Visual Basic, C#, C++, ...

El algoritmo de Milner

Nombres:

- Hindley–Milner
- Damas–Milner
- Damas–Hindley–Milner

Propiedades:

- Completo
- Computa el tipo más general posible sin necesidad de anotaciones
- Eficiente: casi-lineal (inversa de la función de Ackermann)
  La eficiencia depende del algoritmo de unificación que se aplique.

El algoritmo de Milner

1. Se asigna un tipo a la expresión y a cada subexpresión.
   - Si el tipo es conocido se le asigna ese tipo.
   - Sino se le asigna una variable de tipo.

   Recordad que las funciones son expresiones también.
2. Se genera un conjunto de restricciones (de igualdad) a partir del árbol de la expresión.
   - Aplicación.
   - Abstracción.
   - Let
   - ...
3. Se resuelven las restricciones usando unificación.

El algoritmo de Milner

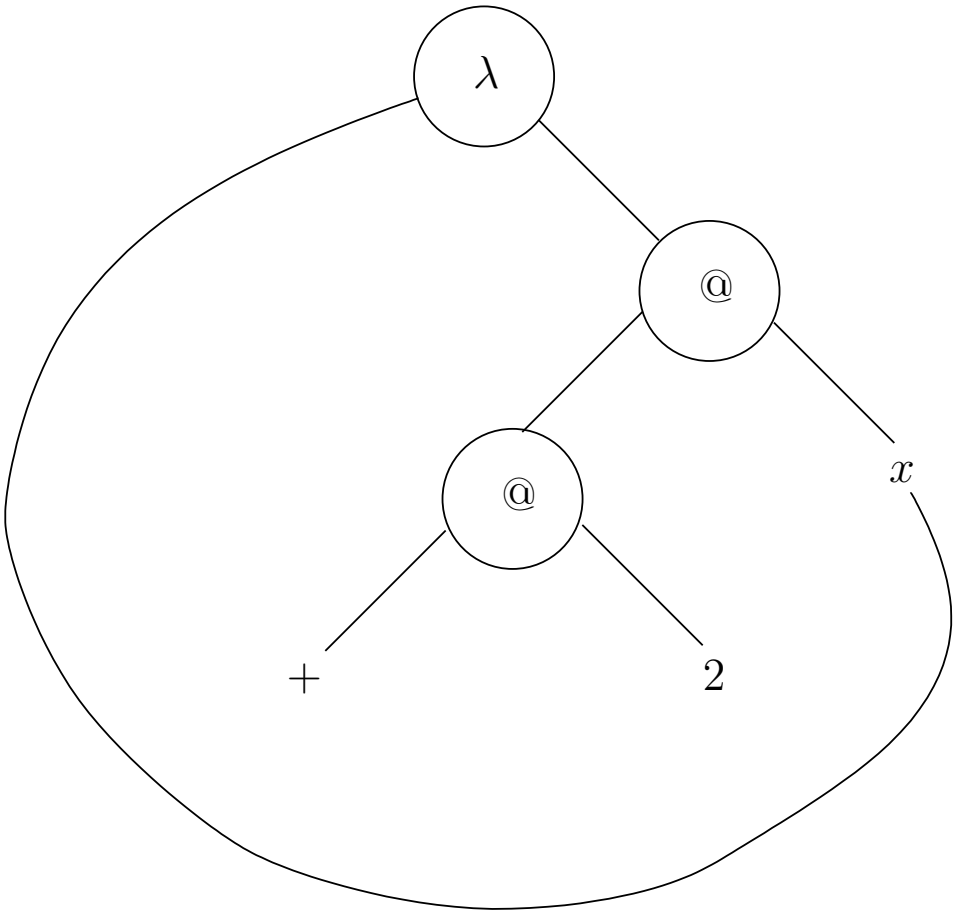
- Considerad la expresión: $(+\ 2\ x)$

El algoritmo de Milner

- Considerad la expresión: $(+\ 2\ x)$
- Vinculamos las variables libres con lambdas: $\lambda x.(+\ 2\ x)$

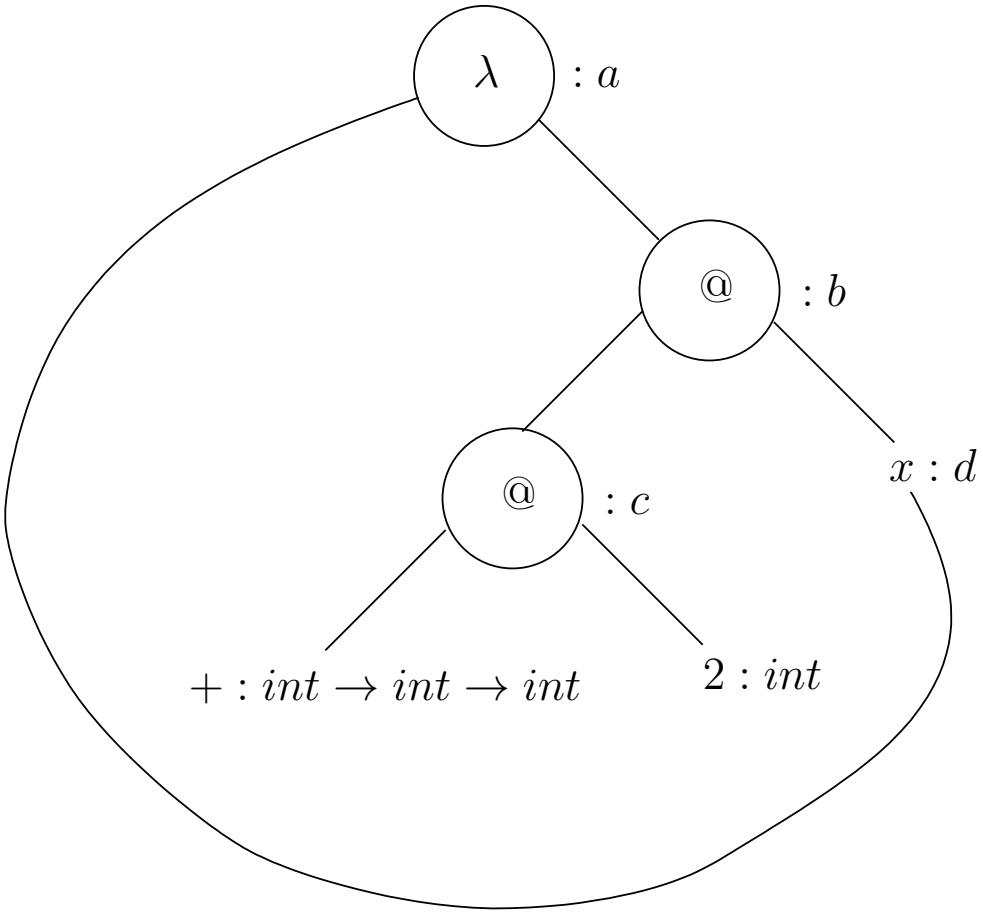
El algoritmo de Milner

- Considerad la expresión: $(+\ 2\ x)$
- Vinculamos las variables libres con lambdas: $\lambda x.(+\ 2\ x)$



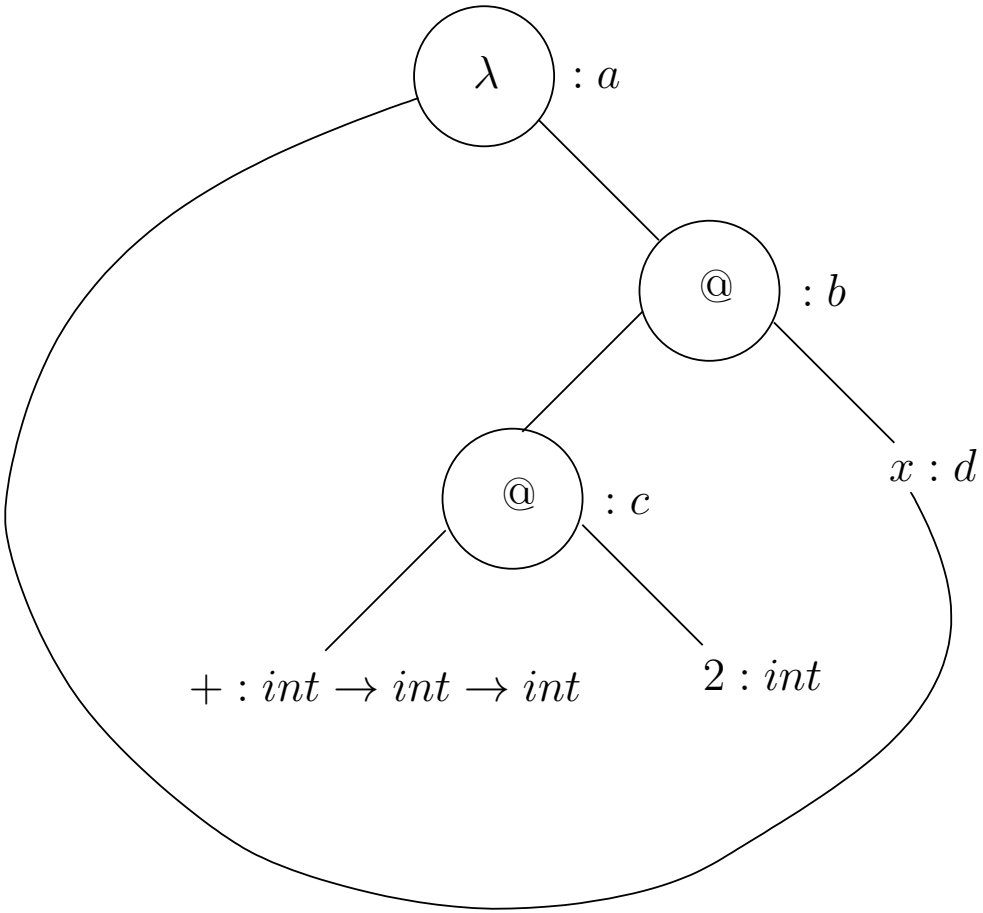
El algoritmo de Milner

1. Asignamos tipo a todas las expresiones



El algoritmo de Milner

2. Generamos las restricciones/ecuaciones



Ecuaciones:

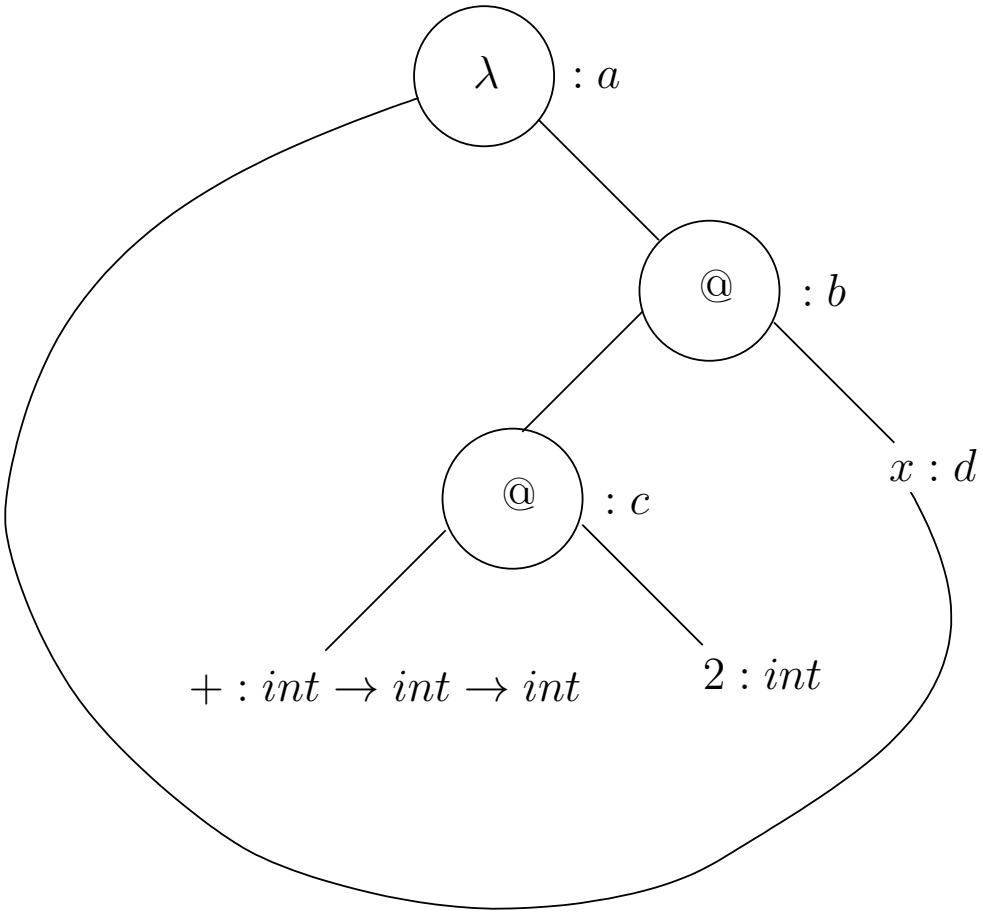
$a = d \to b$

$c = d \to b$

$int \to int \to int = int \to c$

El algoritmo de Milner

3. Solucionamos las ecuaciones con unificación



Ecuaciones:

$a = d \to b$

$c = d \to b$

$int \to int \to int = int \to c$

Solución:

$a = int \to int$

$b = int$

$c = int \to int$

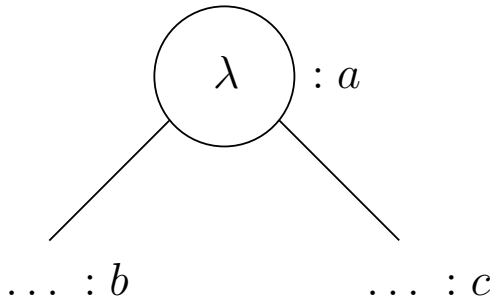
$d = int$

$\boxed{int \to int}$

El algoritmo de Milner

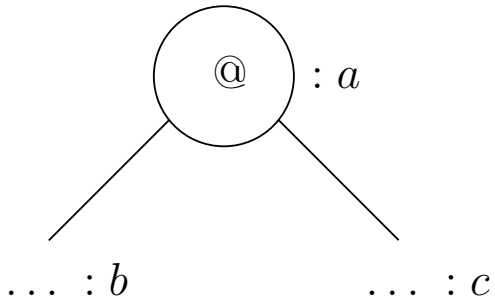
Estas son las reglas para generar las ecuaciones:

Abstracción:



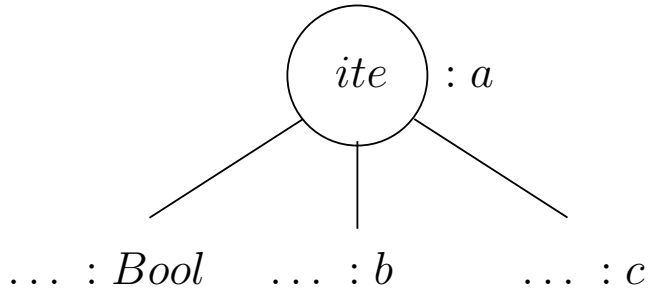
$$a = b \to c$$

Aplicación:



$$b = c \to a$$

IfThenElse:



$$a = c$$
$$a = b$$

El algoritmo de Milner

Consideramos ahora:
$map\ f\ l\ =\ if\ (null\ l)\ then\ [\,]\ else\ f\ (head\ l) : map\ f\ (tail\ l)$

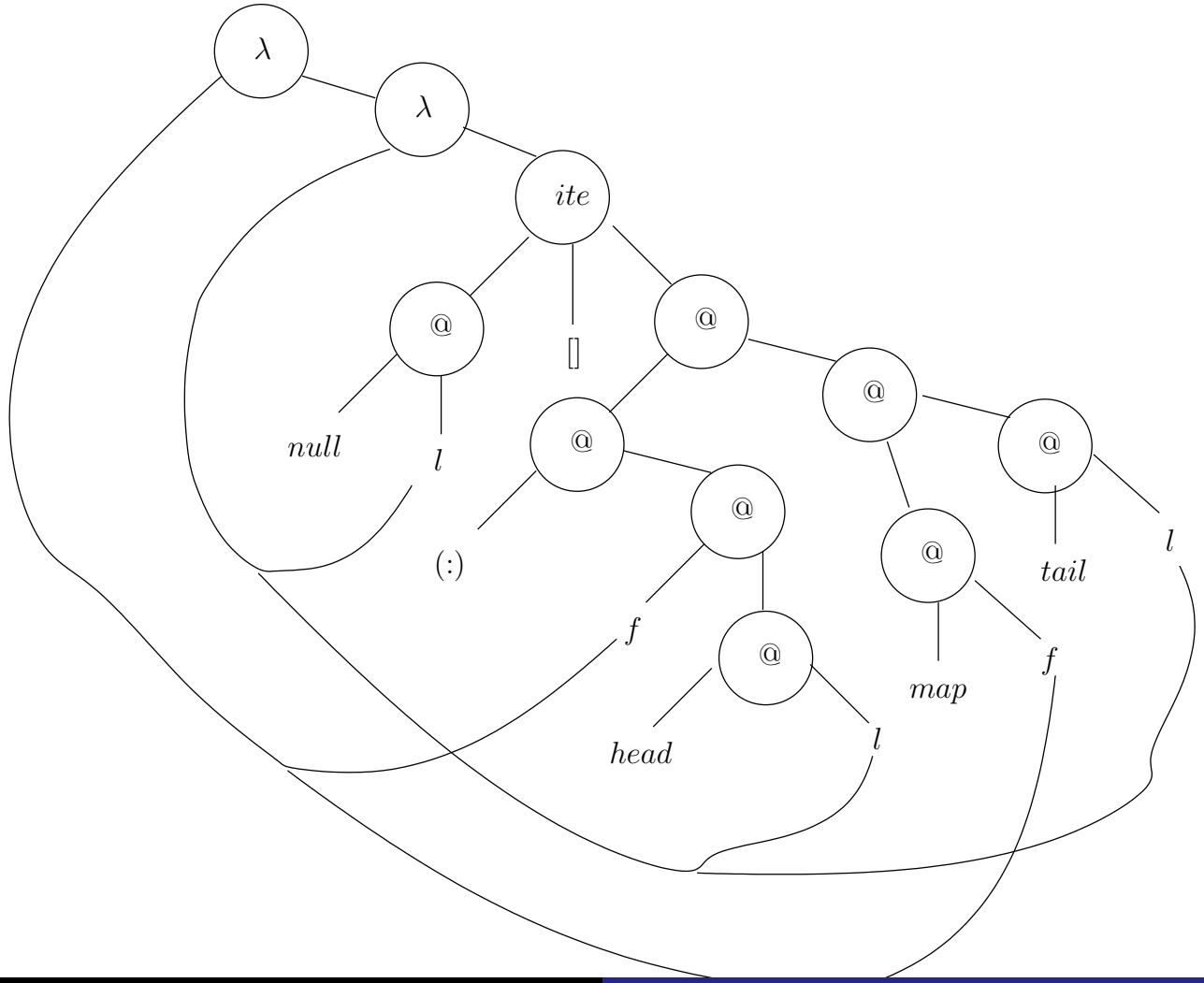
Podemos entender una definición como una función que, aplicada a los parámetros, nos devuelve la parte derecha de la definición.

$\lambda\ f.\ \lambda\ l.\ if\ (null\ l)\ then\ [\,]\ else\ f\ (head\ l) : map\ f\ (tail\ l)$

Notad que el tipo de if_then_else es:
$if\_then\_else : Bool\ \rightarrow\ a\ \rightarrow\ a\ \rightarrow\ a$

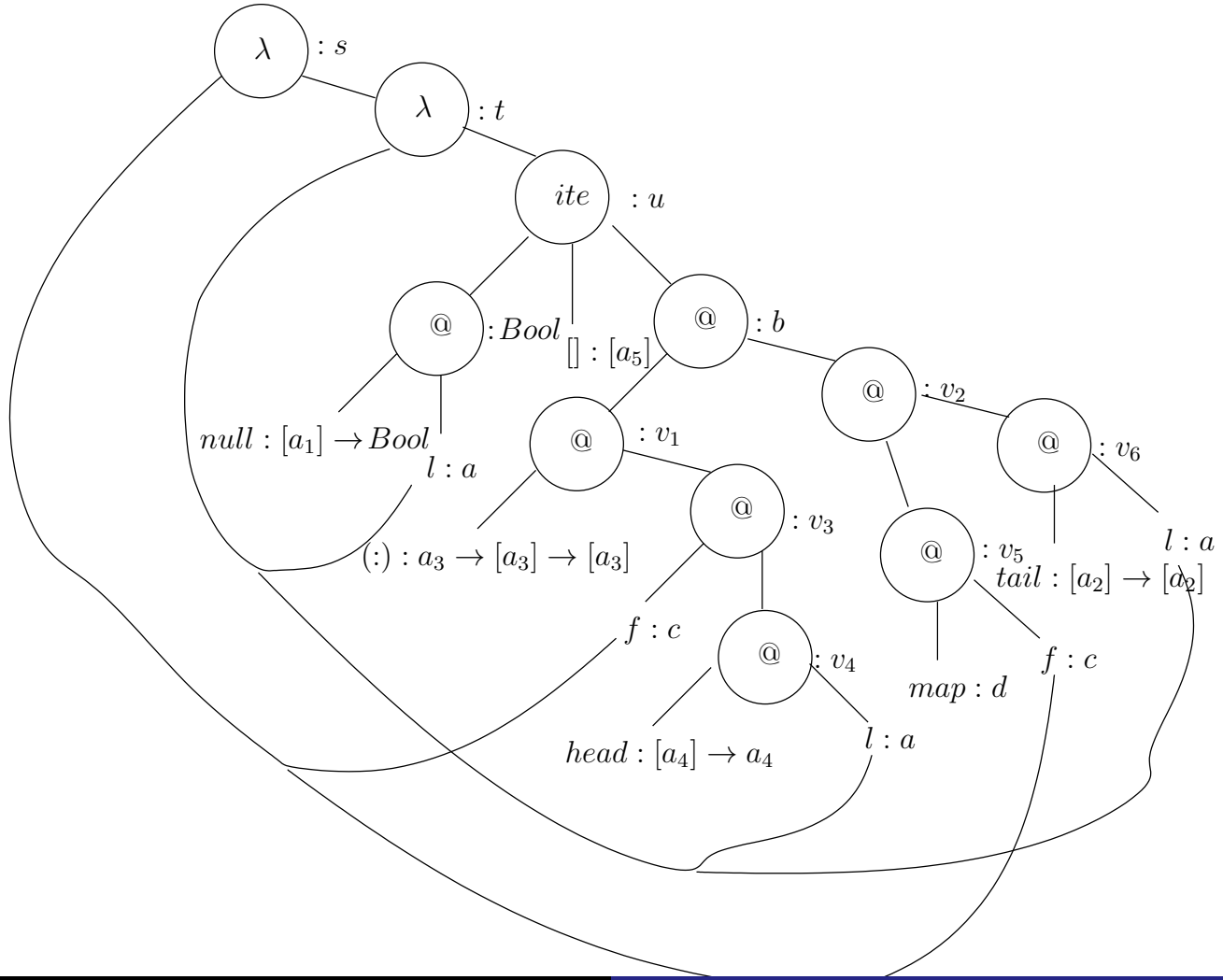
El algoritmo de Milner

$\lambda\, f.\, \lambda\, l.\, if\ (null\ l)\ then\ [\,]\ else\ f\ (head\ l) : map\ f\ (tail\ l)$



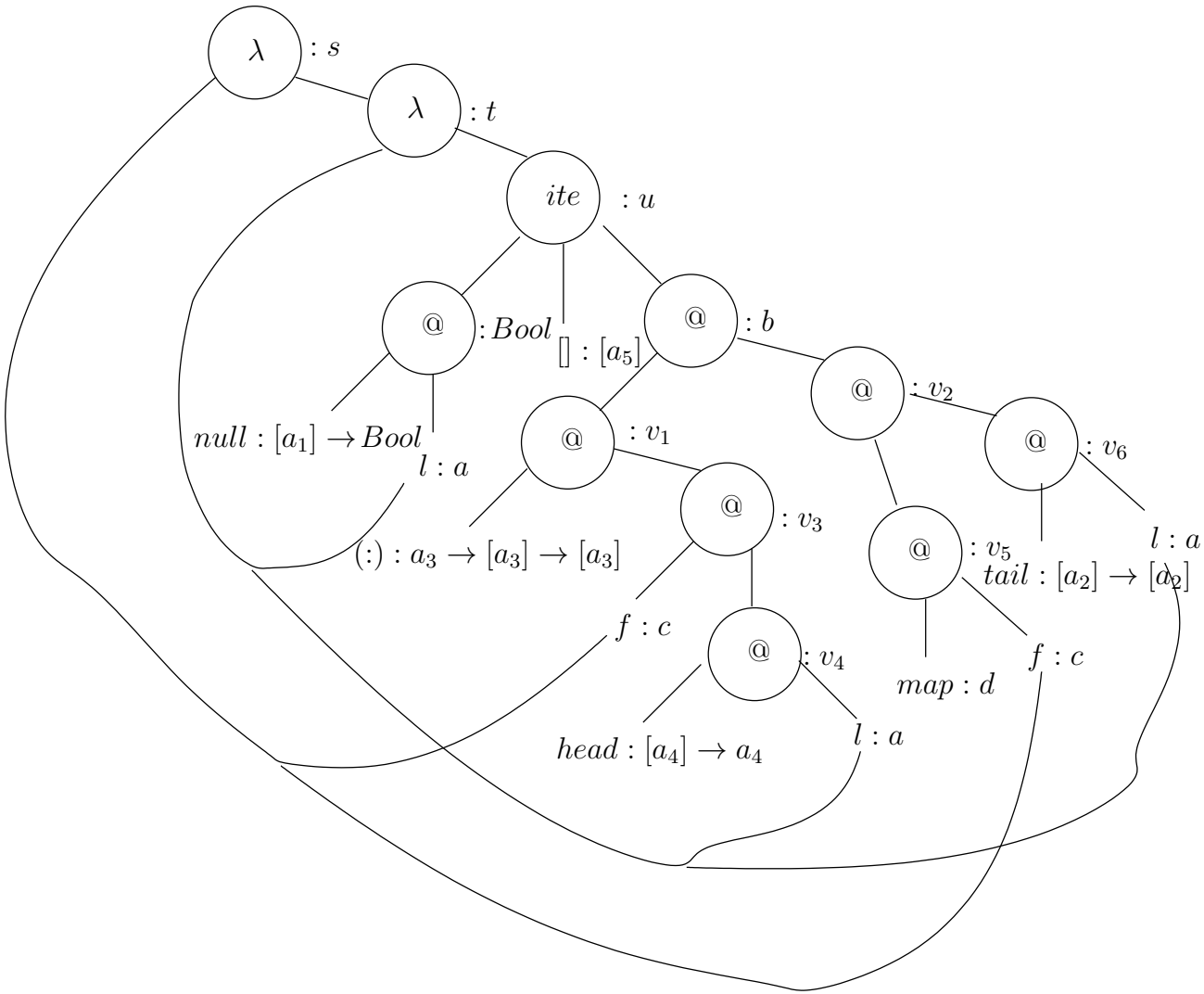
El algoritmo de Milner

$\lambda\, f.\, \lambda\, l.\, \mathit{if}\ (\mathit{null}\ l)\ \mathit{then}\ [\,]\ \mathit{else}\ f\ (\mathit{head}\ l) : \mathit{map}\ f\ (\mathit{tail}\ l)$



El algoritmo de Milner

$\lambda\, f.\, \lambda\, l.\, \mathit{if}\ (\mathit{null}\ l)\ \mathit{then}\ [\,]\ \mathit{else}\ f\ (\mathit{head}\ l) : \mathit{map}\ f\ (\mathit{tail}\ l)$



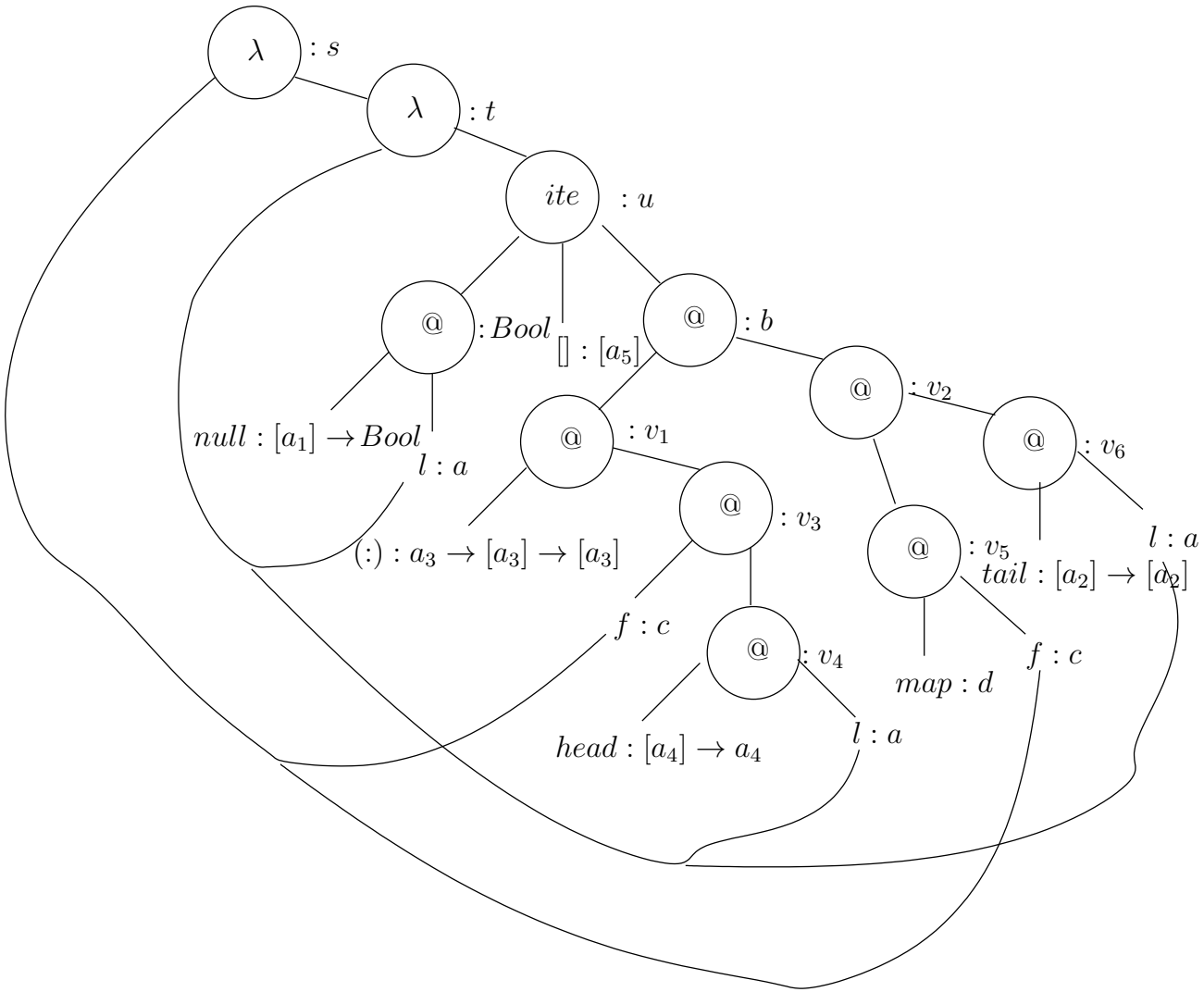
Ecuaciones:

$s = c \to t$
$t = a \to u$
$u = [a_5]$
$u = b$
$[a_1] \to \mathit{Bool} = a \to \mathit{Bool}$
$v_1 = v_2 \to b$
$a_3 \to [a_3] \to [a_3] = v_3 \to v_1$
$c = v_4 \to v_3$
$[a_4] \to a_4 = a \to v_4$
$v_5 = v_6 \to v_2$
$d = c \to v_5$
$[a_2] \to [a_2] = a \to v_6$

$s = d$

El algoritmo de Milner

$\lambda\, f.\, \lambda\, l.\, \mathit{if}\ (\mathit{null}\ l)\ \mathit{then}\ [\,]\ \mathit{else}\ f\ (\mathit{head}\ l) : \mathit{map}\ f\ (\mathit{tail}\ l)$



Solución:

a	$=$	$[a_1]$
a_2	$=$	a_1
a_4	$=$	a_1
a_5	$=$	a_3
b	$=$	$[a_3]$
c	$=$	$a_1 \to a_3$
d	$=$	$(a_1 \to a_3) \to [a_1] \to [a_3]$
s	$=$	$(a_1 \to a_3) \to [a_1] \to [a_3]$
t	$=$	$[a_1] \to [a_3]$
u	$=$	$[a_3]$
v_1	$=$	$[a_3] \to [a_3]$
v_2	$=$	$[a_3]$
v_3	$=$	a_3
v_4	$=$	a_1
v_5	$=$	$[a_1] \to [a_3]$
v_6	$=$	$[a_1]$

$\boxed{(a_1 \to a_3) \to [a_1] \to [a_3]}$