Inferencia de Tipos

Breves apuntes de la clase

1. Introducción

En lo que va de la materia hasta aquí hemos atacado los lenguajes de programación funcional. Vimos que en ellos el concepto de **tipado** es muy fuerte. Se definen las expresiones válidas a partir de reglas sintácticas (las cuales estudiamos ya en detalle) y de tipo.

Ya trabajamos en la materia con reglas de tipado: de forma explícita las conocimos al trabajar con Cálculo Lambda y de forma un tanto más implícita al trabajar con Haskell.

En este último caso más de una vez nos pasó que el intérprete nos arroje errores de tipo. Para hacer esto, Haskell necesita poder deducir el tipo de una expresión a partir de la sintaxis. Esta capacidad la conocemos como **inferencia de tipos**.

El propósito de esta clase es trabajar sobre un lenguaje muy reducido, de forma tal de dar un algoritmo de inferencia riguroso que nos permita entender cómo se podría hacer para inferir tipos en lenguajes más complejos (incluso afuera del ámbito de la programación funcional).

2. Sintaxis

Como mencionamos anteriormente, trabajaremos sobre una sintaxis muy básica, de forma tal de que la complejidad de la inferencia no sea innecesariamente alta.

Las expresiones de tipos que usaremos están definidos inductivamente por:

$tipo = \texttt{Int} \mid \texttt{Bool} \mid varDeTipo \mid tipo \rightarrow tipo$

En general usaremos letras de nuestro alfabeto como s ó t para variables de tipo.

Por ejemplo, algunas expresiones de tipo válidas son:

- `Int`
- `Bool` $\rightarrow t$
- $((a \rightarrow a) \rightarrow b) \rightarrow$ `Bool`

Usaremos la siguiente sintaxis para las expresiones del lenguaje:

$expr = cte \mid var \mid expr\ expr \mid \backslash var.expr$

Notar que es un subconjunto muy básico de Cálculo Lambda sin tipos en las abstracciones lambda y que no cuenta con recursión.

Las siguientes constantes serán usadas como predefinidas. Aquí damos su tipo:

Si n es un entero, luego `n: Int`

Además:

```
True : Bool
False : Bool
+ : Int -> Int -> Int
* : Int -> Int -> Int
/ : Int -> Int -> Int
< : Int -> Int -> Bool
&& : Bool -> Bool -> Bool
not : Bool -> Bool
id : s -> s
```

Donde las últimas tres son constantes polimórficas dado que su expresión de tipo contiene variables de tipo.

Por ejemplo, algunas expresiones del lenguaje son:

- `\x. x+3`, la cual la veremos como `\x. (+ x) 3`.
- `(3+4) * 10`, la cual la veremos como `* ((+ 3) 4) 10`.
- `(\x. x<3) 5`, la cual la veremos como `(\x. (< x) 3) 5`.

Aclaración: usaremos a lo largo del apunte λ y \ de forma indistinta, así como $\rightarrow$ y `->`.

3. Reglas de tipado

Tal como las vimos para Cálculo Lambda, podemos pensar en reglas de tipado para esta sintaxis reducida. Las mismas se dividen en axiomas y reglas de inferencia.

Como un dato curioso: visto desde un punto de vista de teorías axiomáticas, obtener el tipo para una expresión es equivalente a demostrar un teorema partiendo de estas reglas. Cuando presentemos un algoritmo formal que resuelva este problema habremos demostrado que ésta es una teoría decidible.

3.1. Axiomas

Por cada una de las constantes predefinidas c mencionadas en el punto anterior, tendremos un axioma que nos dirá su tipo.

$$\frac{}{\Gamma \triangleright c : T} \ \textsf{(cte)}$$

Notar que en este caso podríamos trabajar con un conjunto de constantes arbitrario, mientras sepamos su tipo y su nombre.

3.2. Reglas de inferencia

Aplicación

La regla app nos indica que dada una función f y una expresión e que unifica con el tipo que recibe f, el tipo de aplicar f a la expresión e es el tipo que devuelve f.

$$\frac{\Gamma \triangleright f : A \to B,\ \ \Gamma' \triangleright e : C,\ \ S = \mathit{Unify}(\{A \doteq C, A_1 \doteq B_1, \ldots, A_n \doteq B_n\})}{S(\Gamma) \cup S(\Gamma') \triangleright f\ e : S(B)} \ \textsf{(app)}$$

Donde $\{x_1, \ldots, x_n\} = \mathit{dom}(\Gamma) \cap \mathit{dom}(\Gamma')$ y $x_i : A_i \in \Gamma$ y $x_i : B_i \in \Gamma'$

Variables

Si sabemos que en nuestro entorno de tipos x tiene tipo T podemos inferir (trivialmente) que $x : T$.

$$\frac{x : T \in \Gamma}{\Gamma \triangleright x : T} \ \textsf{(var)}$$

Abstracción

Dado una variable x y una expresión e, se puede construir una expresión que tendrá el tipo de una función que va del tipo de x en el tipo de e.

$$\frac{\Gamma \cup \{x : A\} \triangleright e : B}{\Gamma \triangleright \lambda x.e : A \to B} \ \textsf{(abs)}$$

4. Intuición

Si bien no podemos pretender que un algoritmo esté basado en la intuición, sí podemos comenzar por algunos ejemplos muy sencillos para comenzar a entender cómo hacemos en nuestra cabeza para inferir el tipo de algunas expresiones.

Una vez que hayamos entendido este proceso nos será más sencillo enseñarle a una computadora a que lo haga por nosotros.

Intentemos tipar las siguientes expresiones aplicando las reglas de forma intuitiva:

1. `17 + 22`, la reescribiremos como `(+ 17) 22`

   Sabemos que la suma es una constante predefinida de tipo `Int -> Int -> Int`, o sea `Int -> (Int -> Int)`. O sea que toma un entero y devuelve una función de `Int -> Int`.

   Sabemos también que 17 es de tipo `Int` luego la aplicación es correcta.

   Finalmente, como 22 también es de tipo `Int` podemos pasarlo como argumento al resultado de la aplicación anterior.

   La expresión resulta ser de tipo `Int`.

2. `\x. 22 < x`, reescrita como `\x. (< 22) x`

   Sabemos que la comparación por menor tiene tipo `Int -> (Int -> Bool)`, la estamos aplicando sobre 22 así que nos devuelte una función de tipo `Int -> Bool`.

   A este resultado le estamos pasando x como parámetro, así que debe tener tipo `Int` y la aplicación termina teniendo tipo `Bool`.

   Luego sabemos que `22 < x : Bool` asumiendo que x es de tipo `Int`, luego la abstracción de x tiene tipo `Int -> Bool`.

Hasta aquí un primer acercamiento intuitivo sobre el problema de inferencia de tipos. Lamentablemente es un enfoque que muy rápidamente se complica, ya que no puede explicarse en castellano y con claridad cuál es la línea de pensamiento que estamos siguiendo.

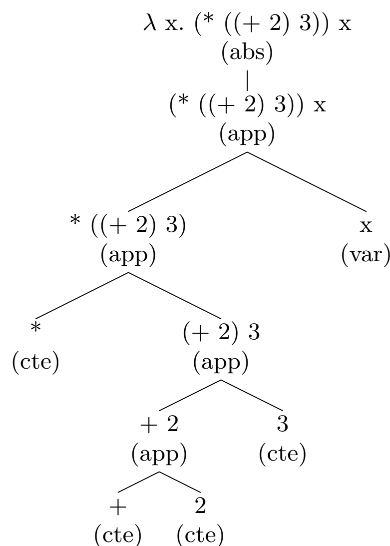
5. Método de los árboles de análisis sintáctico

Introduciremos ahora un método que nos acercará al algoritmo formal que definiremos más adelante. Este método se basa en el árbol de análisis sintáctico que genera el parser del lenguaje.

No nos interesaremos en esta materia en cómo hace el parser para generar estos árboles, sólamente trabajaremos sobre ellos para poder inferir el tipo de la expresiones que nos interesen.

Un árbol de análisis sintáctico tiene variables o constantes en las hojas y aplicaciones o abstracciones en los nodos internos. Un árbol de análisis sintáctico de la expresión e es un árbol tal que, si se reconstruye desde las hojas hacia arriba, se obtiene dicha expresión. Cada expresión tiene un único árbol de análisis sintáctico.

Por ejemplo, el siguiente es el árbol de análisis sintáctico de la expresión `\x. (2 + 3) * x` reescrita como `\x. (* ((+ 2) 3)) x`:



La idea del método de los árboles de análisis sintáctico es comenzar por tipar las hojas, aplicando las reglas de constante o variable, e ir subiendo por las ramas hasta llegar a tipar la expresión de la raíz que es la que nos interesa.

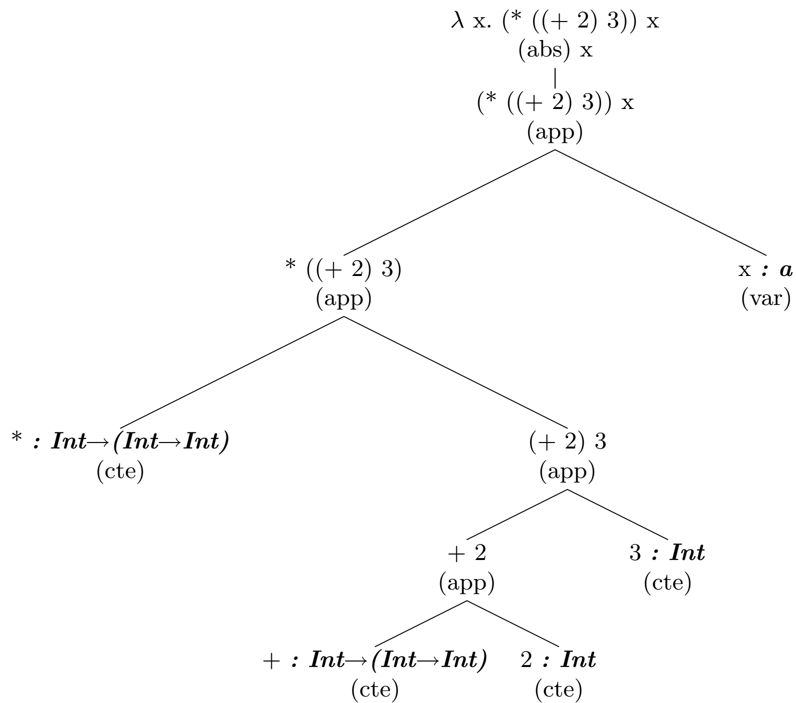
Para cada aparición de una variable utilizaremos una variable de tipo distinta, esto es porque en principio no sabemos su tipo y además por más que sea la misma variable no sabemos si se trata del mismo scope.

Una vez que tengamos resuelto el tipo de las hojas subiremos aplicando las reglas de abstracción y aplicación vistas anteriormente.

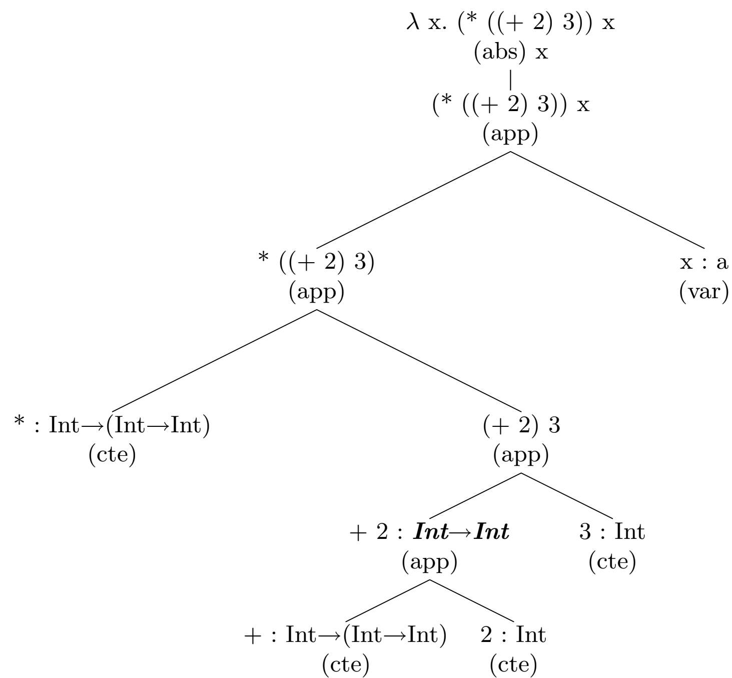
Terminaremos al llegar a la raíz, con una expresión que resultará tipada en su totalidad. Además sabremos cuál es el tipo de cada aparición de cada variable en la misma.

Veremos en la página siguiente un ejemplo completo.

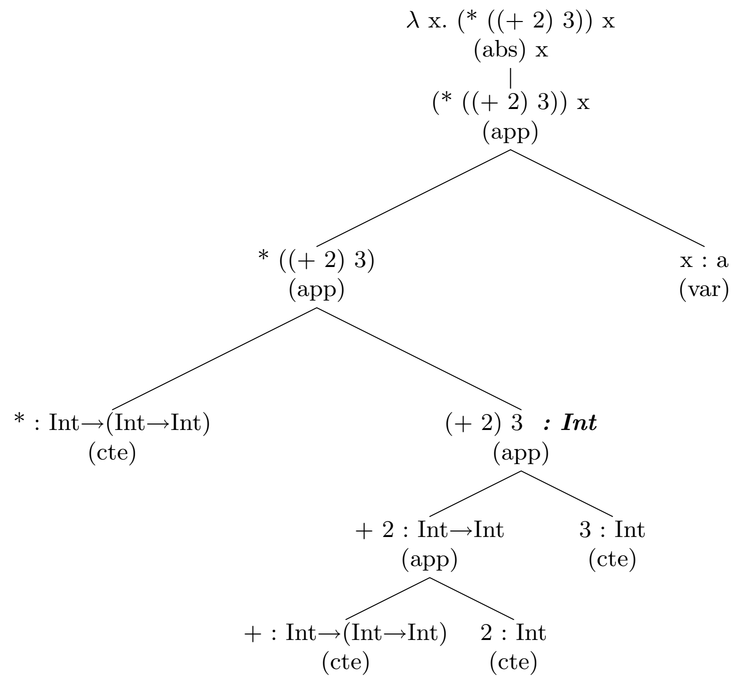
Resolvemos las hojas:



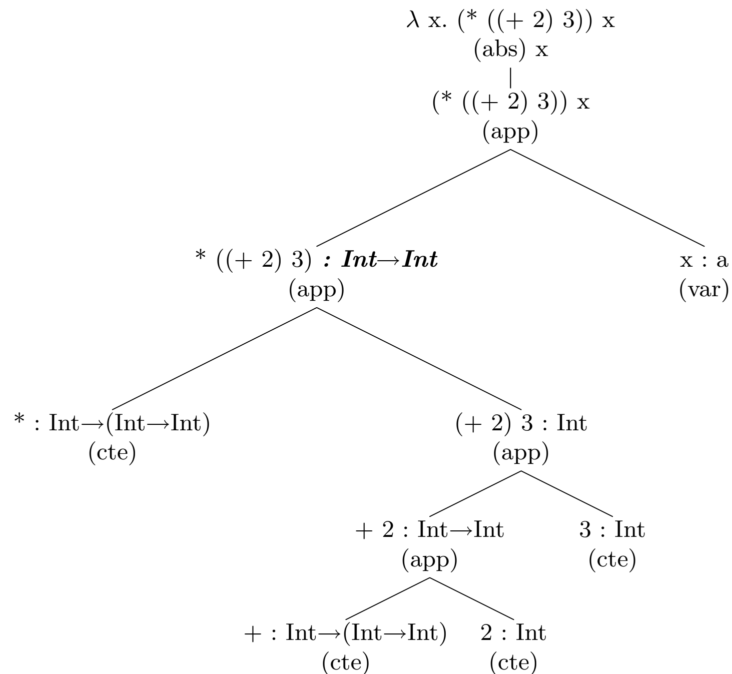
Resolvemos la suma a un término:



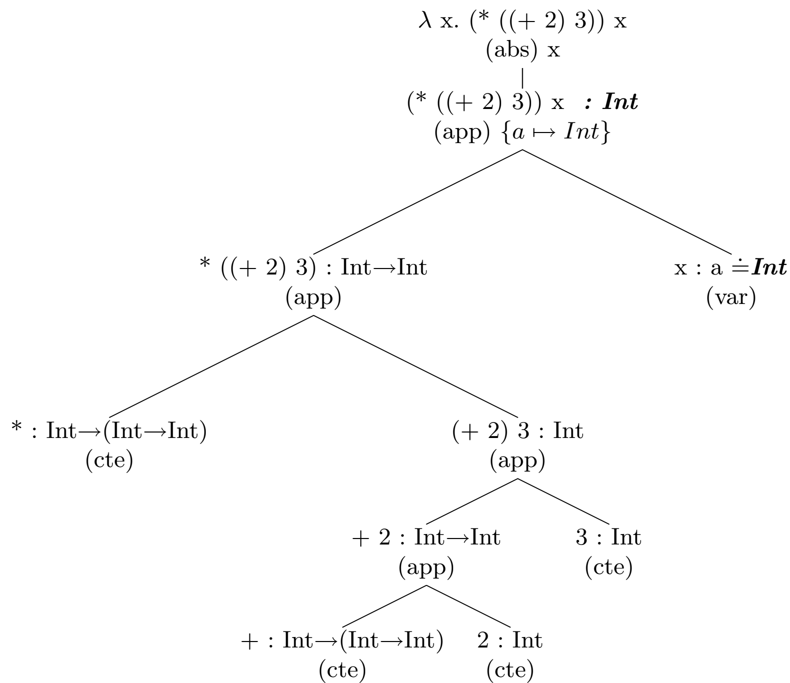
Resolvemos el tipo de la aplicación de la suma a dos términos:



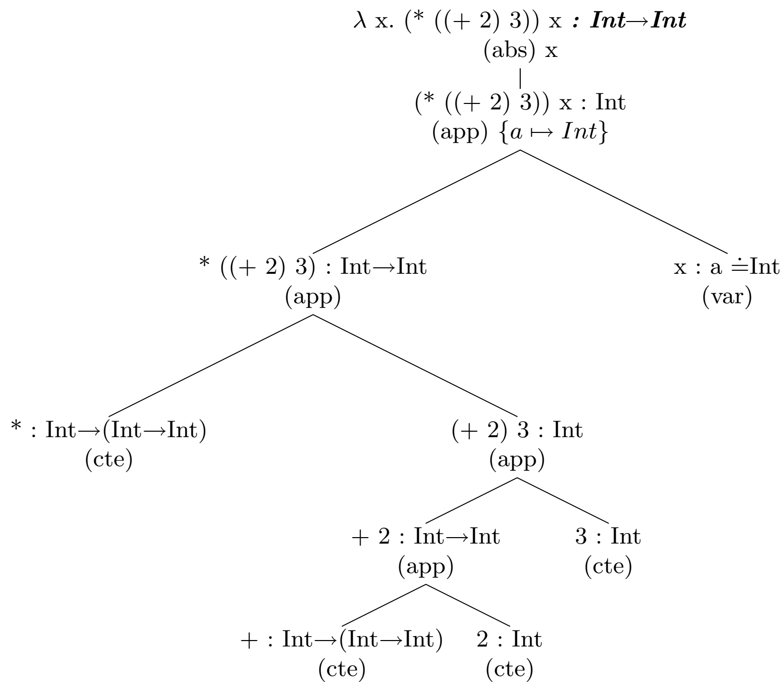
El tipo del producto aplicado a un término:



El tipo del producto aplicado a dos términos:



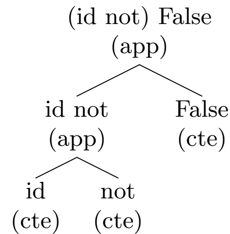
Finalmente obtenemos el tipo de la abstracción de la raíz:



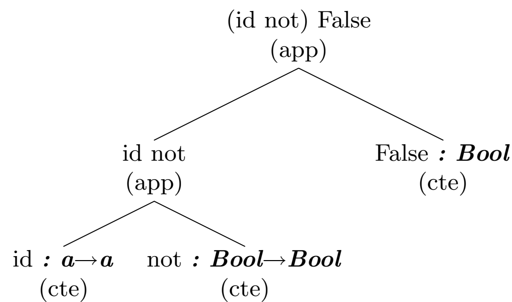
5.1. Otro ejemplo

Tipemos ahora la expresión `(id not) False`.

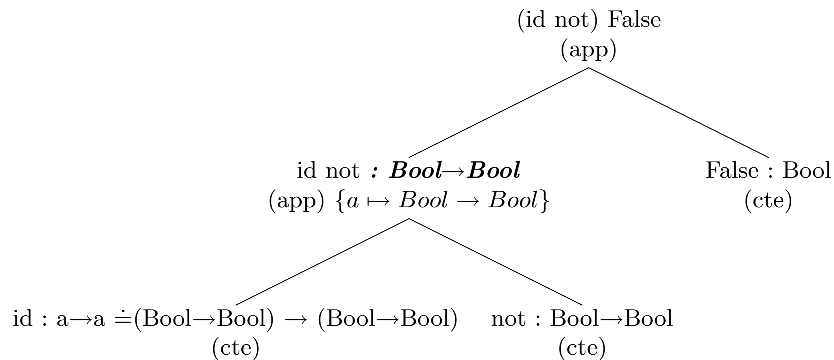
Comencemos por armar el árbol de análisis sintáctico:



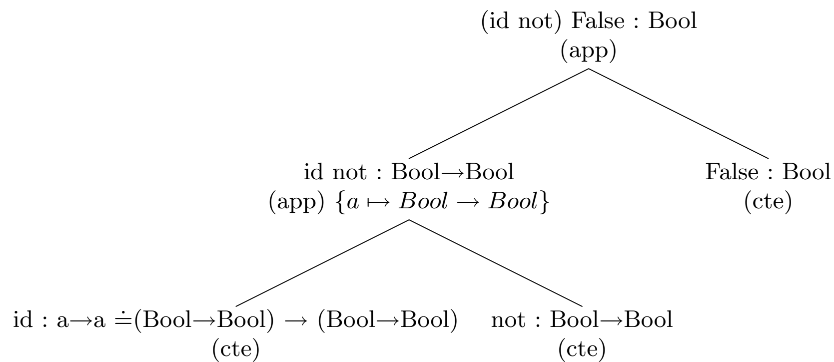
Completamos las constantes:



Resolvemos el caso de la aplicación de la constante polimórfica:



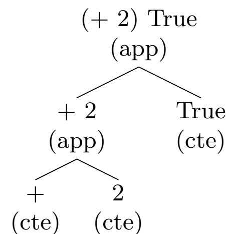
Y resolvemos el tipo final de la expresión:



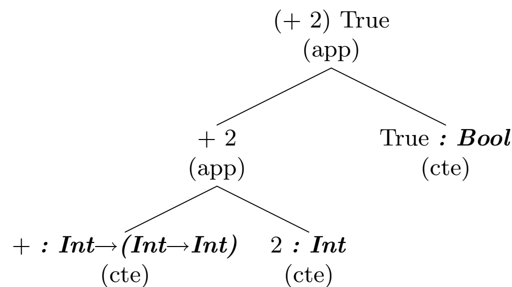
5.2. Dos casos que fallan

Hasta aquí el método funcionó porque le pasamos expresiones sin errores de tipos, veamos cómo se comporta en otros casos.

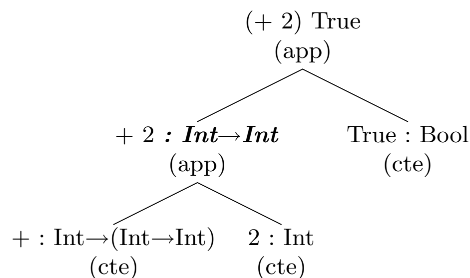
Intentemos tipar `2 + True`, reescrita como `(+ 2) True`.



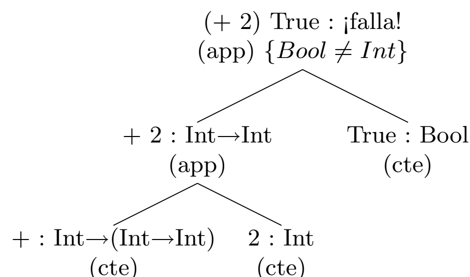
Tipamos las hojas:



Subimos con la aplicación del 2 a la suma:

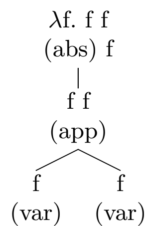


Pero al intentar aplicar el segundo argumento de la suma, nos encontramos con que $\{Bool \doteq Int\}$ no unifica:

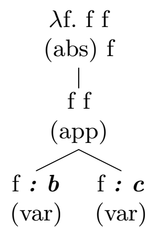


Observar que aquí la falla se da porque `Int` y `Bool` son "irreconciliables". Si se tratara de `Int` por un lado y la variable de tipo t por el otro la respuesta sería la sustitución $\{t \mapsto Int\}$.

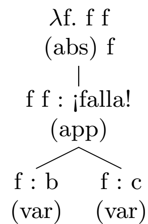
Existen otras fallas que son más sutiles, por ejemplo intentemos tipar `\f. f f`:



Tipemos las hojas (recordar que usamos siempre variables de tipos distintas):



Intentamos resolver la aplicación `f f`:



Las dos f tienen el mismo scope, luego debe valer que b unifique con c.

La f de la izquierda de la aplicación (la que tiene tipo b) juega el rol de función, por lo tanto tiene que suceder que $b \doteq s \to t$ para algún valor de s y t. Además, la f de la derecha (la que tiene tipo c) es el parámetro a la función, luego deben unificar con s.

Tenemos entonces $\{c \doteq s, b \doteq s \to t, b \doteq c\}$, lo cual se puede repensar como $\{c \doteq s, c \doteq s \to t\}$. Estas dos ecuaciones son claramente incompatibles (i.e., no unifican) así que el algoritmo falla.

6. Algoritmo PT

Si bien el método de los árboles presentado anteriormente es muy útil para trabajar de forma rápida y sencilla al tipar una expresión, a la hora hacer un programa que calcule el tipo de una expresión necesitamos una definición más formal.

El siguiente algoritmo PT[1] produce un tipado explícito de la expresión que recibe como parámetro, o bien falla si la expresión no es tipable. También se lo conoce como algoritmo W.

De hecho, PT (Principal Typing) devuelve un tipado principal de la expresión. Un tipado es principal (o el más general) si cada tipado explícito de la expresión es una instancia del tipado en cuestión.

Se utilizará Γ para contextos de tipado, τ, ρ, α ó β para expresiones de tipo, s ó t para variables de tipo, U ó V para expresiones del lenguaje, S para sustituciones, x para variables del lenguaje y c para constantes.

6.1. Definición formal

$PT(c) = \emptyset \triangleright c : \tau$
$PT(x) = \{x : t\} \triangleright x : t$
$PT(UV)$:

Si obtenemos de los pasos recursivos que:
$PT(U) = \Gamma \triangleright M : \tau$
$PT(V) = \Gamma' \triangleright N : \rho$
(Supondremos que las variables de tipo de $PT(U)$ y $PT(V)$ son disjuntas)

Construimos una sustitución que unifique los contextos de tipado Γ y Γ':
$S = \mathit{Unify}(\{\alpha \doteq \beta \mid x : \alpha \in \Gamma \wedge x : \beta \in \Gamma'\} \cup \{\tau \doteq \rho \to t\})$, donde t es una variable de tipo fresca

Entonces, devolvemos como resultado:
$PT(UV) = S\Gamma \cup S\Gamma' \triangleright S(MN) : St$

$PT(\lambda x.U)$:

Si obtenemos del paso recursivo que:
$PT(U) = \Gamma \triangleright M : \rho$

Debemos separar en casos según si había información de tipado o no para la variable que se está abstrayendo:

Si $x : \tau \in \Gamma$ para algún τ:
$PT(\lambda x.U) = \Gamma - \{x : \tau\} \triangleright \lambda x : \tau.M : \tau \to \rho$

Si no:
$PT(\lambda x.U) = \Gamma \triangleright \lambda x : s.M : s \to \rho$, donde s es una variable de tipo fresca

6.1.1. Aclaraciones útiles para entender el algoritmo

Recordar (como puede verse en la práctica de Inferencia de Tipos) que una expresión de tipo puede ser una variable de tipo, una constante de tipo o una función. Luego si ρ representa a una expresión de tipo, tiene que valer una y sólo una de las siguientes igualdades:

- $\rho = t$, con t una variable de tipo.
- $\rho = Int$

[1]Una explicación pormenorizada del algoritmo, así como las pruebas de corrección y completitud pueden encontrarse en Mitchell,J.C. *Foundations for Programming Languajes* (pp. 777-782)

- $\rho = Bool$
- $\rho = \alpha \to \beta$, con α y β expresiones de tipo.

También recordar que variable fresca quiere decir que suponemos que el entorno de ejecución donde correrá este algoritmo (o en este caso los seres humanos que lo ejecutamos a mano, paso por paso) tiene la habilidad de crear de la nada un nombre de variable que no se haya usado nunca hasta ahora en ninguna de las ramas de la recursión.

6.2. Descripción intuitiva del algoritmo

6.2.1. Caso "constante"

Si se tiene un término conocido como una constante inferir su tipo es trivial y el contexto de tipado necesario para afirmarlo es vacío.

6.2.2. Caso "variable"

Si el algoritmo se encuentra con una variable, lo máximo que puede decir sobre ella es que tiene un tipo t, donde t es una variable de tipo. Para afirmar esto requiere que el contexto de tipado diga lo mismo.

6.2.3. Caso "aplicación"

Definitivamente el más complejo de los casos de este algoritmo. Al identificar una expresión de tipo UV, el algoritmo llama recursivamente a analizar los tipos de ambas partes. Observar que el resultado del algoritmo PT sobre U y V no es sólamente un tipo, sino (por ejemplo en el caso de U) :

$PT(U) = \Gamma \triangleright M : \tau$

Se está devolviendo un contexto de tipado Γ, un tipo principal para la expresión τ y una reescritura M de la expresión U.

¿Por qué no devolver directamente $\Gamma \triangleright U : \tau$? En el caso de la abstracción se anota el código indicando (al estilo Cálculo Lambda) el tipo de la variable. Si se quitaran (con un simple algoritmo) las anotaciones de tipos de M, el resultado sería U.

Para obtener el tipo de UV es necesario construir el unificador S:

$S = \mathit{Unify}(\{\alpha \doteq \beta \mid x : \alpha \in \Gamma \land x : \beta \in \Gamma'\} \cup \{\tau \doteq \rho \to t\})$

El mismo cumple dos funciones:

1. "Compatibilizar" los contextos de tipado Γ y Γ'.

   En ambos contextos pueden figurar distintas asignaciones de tipo para la misma variable. Es por eso que se pide unificar los tipos que dan los contextos a una **misma** variable cada vez que difieran las asignaciones de tipo sobre ella. La idea intuitiva es reconciliar lo que dicen ambos contextos, para lograr uno nuevo que no sea contradictorio.

2. Dar tipo a la aplicación.

   En la segunda parte de la unificación se pide que $\tau \doteq \rho \to t$. Recordar que τ era el tipo de U (la función) y ρ era el tipo de V (el argumento). Luego es razonable pedirle al unificador que el tipo τ unifique con $\rho \to t$, para algún t que en principio no importa cuál es, por lo tanto se lo deja como una variable de tipado fresca.

Esta variable t muy probablemente sea “resuelta” y reemplazada por un tipo más concreto al finalizar la unificación.

No siempre es posible lograr esta reconciliación de expresiones de tipo. De no serlo, el algoritmo de unificación fallará y por ende el algoritmo PT lo hará también. Pero si la unificación es exitosa, el algoritmo produce una sustitución S.

El tipo devuelto para UV es:

$PT(UV) = S\Gamma \cup S\Gamma' \rhd S(MN) : St$

Recordar que el tipo de U era τ, el cual debía unificar con $\rho \rightarrow t$. El tipo del parámetro V era ρ, por lo tanto el tipo devuelto es t.

Sin embargo observar que aquí se aplica la sustitución tanto sobre ambos contextos, como sobre los términos devueltos por las llamadas recursivas M y N, y también sobre la variable fresca t. Esto es porque la unificación logró reconciliar ambas llamadas recursivas, y por ende el resultado de la misma es utilizado para que el tipo de UV no tenga contradicciones.

6.2.4. Caso “abstracción”

Dado que el tipo del llamado recursivo indica que:

$PT(U) = \Gamma \rhd M : \rho$

Luego nos encontraríamos tentados de decir que si el cuerpo U de la función tiene tipo ρ, luego la abstracción será de tipo $s \rightarrow \rho$, con s una variable de tipo fresca que es el tipo del parámetro formal x.

Sin embargo, el tipo de x puede estar restringido si x aparece presente en el cuerpo de la función. Por lo tanto en caso de que x tenga un tipo asociado lo anotaremos en su aparición al lado del λ. Además eliminaremos tal información del contexto de tipado Γ ya que la variable existe adentro de U únicamente.

En caso de que x no figure dentro de U es válido pensar que el parámetro es de tipo s, siendo ésta una variable de tipo fresca.