Exercícios sobre Verificação e Inferência de Tipos

Prof.: Carlos Camarão

27 de Abril de 2017

1 Tipos

1.1 A regra básica

A regra:

$$\frac{f : A \to B \quad x : A}{f\, x : B}$$

é a regra fundamental usada por um compilador na verificação da correção e na inferência dos tipos usados em programas.

Note que existe uma correspondência básica entre tipos e termos lógicos (termos da lógica proposicional) [4, 7]. Se consideramos tipos (A, B, $A \to B$) como proposições lógicas, e expressões (f, x, $f\,x$) como provas (demonstrações) dessas proposições, a regra acima é a regra fundamental (chamada de *modus ponens*) usada em lógica para dedução de novas formas válidas a partir de formas válidas existentes.

A correspondência entre proposições lógicas e tipos, e entre provas (demonstrações) e expressões, é conhecida como "correspondência de Curry-Howard" (também comumente chamada de "isomorfismo de Curry-Howard") [4, 7].

No caso de linguagens monomórficas, esta é a regra mais fundamental, e as demais podem ser consideradas como auxiliares. Por exemplo, para deduzir que `(`**`not`**` True)` tem tipo `Bool`, precisamos apenas considerar ou saber que **`not`** tem tipo `(Bool->Bool)` e `True` tem tipo `Bool`. Essa consideração ou conhecimento usa tipos existentes em um contexto, nesse caso tipos pré-definidos de constantes e funções pré-definidas.

1.2 Polimorfismo paramétrico

No caso de linguagens polimórficas, essa regra também deve ser aplicada, mas após *instanciação* do tipo do domínio da função para que fique igual ao tipo do argumento. No caso da inferência de tipos, essa instanciação é feita por uma função de *unificação*, que calcula a substituição mínima que se pode aplicar para que o tipo do domínio da função e o tipo do argumento se tornem iguais.

Mais detalhadamente o processo de inferência de tipos usa a seguinte regra de aplicação, que vamos chamar de regra (`APL`): para que uma função (potencialmente polimórfica) f`:` t possa ser aplicada a um argumento x`:` t', devemos *unificar* t com $t_A \to t_B$ (ou seja, t deve ter um tipo funcional) e t_A com t' (ou seja, o tipo do argumento deve ser unificado com o tipo do domínio da função); o tipo resultante é o tipo t_B resultante da unificação. Isso vai ser explicado mais precisamente na seção seguinte.

O processo de inferência de tipos de um compilador atribui a valores que ocorrem em expressões tipos que são conhecidos e, sempre que um tipo de um valor ainda não é conhecido, como por exemplo o tipo de um parâmetro formal de uma função, uma nova variável de tipo, ainda não usada, é (inicialmente) atribuída a esse valor. Depois disso, a inferência de tipos consiste em determinar os tipos resultantes das unificações especificadas na regra (`APL`), que vai ser abordada mais detalhadamente na seção seguinte.

1.2.1 Unificação e Substitituição

Dado um conjunto de variáveis V, um conjunto de constantes C e um conjunto de nomes de funções F, o conjunto de termos T pode ser definido pelo seguinte tipo algébrico recursivo:

```
data T = Con K | Var V | App T T
```

onde supomos que K é um conjunto de constantes e V de variáveis de tipo.

Um tipo como $t_1 \cdots t_n$ é visto, nessa representação, como $App\ t_1(\ldots\ App\ t_{n-1}\ t_n)$.

Uma substituição é uma função finita $S : V_0 \to T$ de variáveis em termos, onde V_0 é o conjunto finito de variáveis que ocorrem em T. A função id é a função identidade, isto é, a função definida por $id\ x = x$.

Dado um conjunto finito de equações $E = \{t_i = t'_i | i = 1, \ldots, n\}$, onde t_i, t'_i são termos (valores de T), uma substituição σ é um unificador dos termos de E se $\sigma(t_i) = \sigma(t_i)'$, para $i = 1, \ldots, n$.

Além disso, uma solução σ é mais geral que outra σ' se existe uma substituição σ_1 tal que $\sigma = \sigma_1 \circ \sigma'$. Intuitivamente, um unificador mais geral de um conjunto de equações E é o modo mais simples de tornar todas as equações do conjunto E iguais.

O seguinte algoritmo retorna *Just m*, onde m é mapeamento que representa o unificador mais geral para uma lista qualquer de pares de tipo (cada par representando uma equação), se um unificador existir, caso contrário retorna *Nothing* ($\circ$ é o operador de composição de funções):

```
unif [] = id
unif ((t, V v): eqs)             = unif ((V v, t): eqs)
unif ((V v, t): eqs)
  | t == V v                     = unif eqs
  | v `ocorreEm` t               = Nothing
  | otherwise                    = s' ○ s
  where
    s    = insert v t empty
    eqs' = ap s eqs
    s'   = unif eqs'
unif ((K k1, K k2): eqs)
  | k1 == k2                     = unif eqs
  | otherwise                    = Nothing
unif (App t1 t2, App t1' t2') = s' ○ s
  where s  = unif t1 t1'
        s' = unif (app s t2) (app s t2')
unif _                           = Nothing
```

As regras usadas na inferência de tipos de um compilador de uma linguagem com polimorfismo paramétrico são baseadas em um *contexto de tipos*, que associam tipos a variáveis (constantes podem ser consideradas como variáveis, com tipos a ela associados em um contexto global). Contextos de tipos mapeiam variáveis a tipos (é comum na literatura o uso de conjuntos de pares representados na forma *variável : tipo*). Vamos usar, como é comum, a meta-variável Γ para representar contextos de tipos.

Tipos de variáveis podem ser monomórficos, ou *simples* ou polimórficos, ou *quantificados*. Um tipo simples é formado por uma constante ou uma variável, possivelmente aplicada a um ou mais tipos simples. Ou seja, é da forma:

$$T\ ::=\ K \mid V \mid T\,\bar{T}$$

Usamos t como para denotar valores do tipo T, a ou b para denotar valores de tipo V e k para denotar valores do tipo K, podendo essas variáveis ter possivelmente subscritos e aspas simples. Usamos $\bar{X}$ para denotar sequências, possivelmente vazias, de valores X, ou conjuntos de tais valores. Por exemplo, $\bar{T}$ representa uma sequência de tipos, e um tipo polimórfico σ, do tipo de conjuntos polimórficos Σ, é da forma $\forall\bar{a}.\,t$; ou seja:

$$\Sigma\ ::=\ \forall\bar{V}.\,T$$

A notação $\Gamma, x : t$ representa o contexto Γ' tal que $\Gamma'(x') = t$ se $x' = x$, senão $\Gamma(x')$ (ou seja, Γ' é um contexto igual a Γ mas $\Gamma'(x) = t$).

A função tv retorna o conjunto de variaveis livres de um tipo. Em particular, $tv(\forall\bar{a}.t) = tv(t) - \bar{a}$.

A relação $gen(t, \sigma, \Gamma)$ é a relação de generalização de t para σ em Γ, definida de forma que $\sigma = \forall\bar{a}.\,t$, onde $\bar{a} = tv(t) - tv(\Gamma)$.

$$\frac{\Gamma(x) = \forall \bar{a}.\, t \quad \bar{b} \text{ variáveis novas}}{\Gamma \vdash x : ([\bar{a} \texttt{ := } \bar{b}]t, id)} \quad \texttt{(VAR)}$$

$$\frac{\Gamma \vdash f : (t, S) \quad S(\Gamma) \vdash e : (t', S') \quad (a \text{ variável nova})}{\Gamma \vdash f\ e : (S(a), S_u \circ S' \circ S)} \quad S_u = \mathit{unif}([(t, t' \rightarrow a)]) \quad \texttt{(APL)}$$

$$\frac{\Gamma, x : a \vdash e : (t, S) \quad a \text{ variável nova}}{\Gamma \vdash \lambda x\texttt{->}\, e : (S(a)\texttt{->}\, t, S)} \quad \texttt{(ABS)}$$

$$\frac{\Gamma \vdash e : (t, S) \quad \mathit{gen}(t, \sigma, S(\Gamma)) \quad S(\Gamma), x : \sigma \vdash e' : (t', S')}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' : (t', S' \circ S)} \quad \texttt{(LET)}$$

Figura 1: Algoritmo W

Um tipo $S(\sigma)$, ou simplesmente $S\sigma$, onde Sa é uma substituição tal que $S(a) = a$ se $a \neq a_i$, $S(a_i) = b_i q$, para i de 1 a n, representa o tipo resultante de substituir cada a_i por b_i, para i de 1 a n. O tipo $S\sigma$ pode ser denotado também por $[\bar{a} \texttt{ := } \bar{b}]\sigma$.

O algoritmo de inferência de tipos de uma linguagem como o núcleo da linguagem *ML* pode ser descrito usando as regras mostradas na Figura 1, onde $\Gamma \vdash e : (t, S)$ significa que e tem tipo t no contexto de tipos Γ. A substituição S é usada para unificação de tipos de expressões usadas em e. Esse algoritmo foi definido por Robin Milner e Luís Damas [2], sendo chamado de algoritmo W.

A regra (`VAR`) retorna o tipo da variável no contexto de tipos (supõe-se que um programa é um termo fechado, isto é, sem variáveis *livres*, no sentido de λ-cálculo).

A regra (`APL`) é a regra básica (ver seção 1.1) da inferência de tipos: usa unificação para tornar o tipo domínio da função igual ao tipo do argumento.

A regra (`ABS`) cria uma nova variável (a, na Figura 1) para o tipo do parâmetro, usando a substituição (S) obtida na inferência do tipo do corpo da λ-abstração para especialização do tipo do parâmetro. O tipo da λ-abstração é o tipo funcional $S(a) \rightarrow t$, sendo t o tipo inferido para a expressão e, corpo da λ-abstração.

A regra (`LET`) generaliza o tipo inferido para e para obter o tipo de e', que é retornado como o tipo da expressão $\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e'$. Note que a regra (`LET`) é que permite a introdução de tipos polimórficos; o uso de variáveis definidas na regra (`LET`) pode ser usada em contextos que requerem tipos distintos, como por exemplo em:

$$\texttt{let}\ f\ \texttt{=}\ \lambda x.\, x\ \texttt{in}\ (f\ \mathit{True}, f\ 1)$$

onde *True* e 1 têm tipos distintos (*Bool* e *Integer*), e o tipo de f pode ser instanciado (no algoritmo W, via unificação) para $\mathit{Bool} \rightarrow \mathit{Bool}$ e para $\mathit{Integer} \rightarrow \mathit{Integer}$.

A regra (`LET`) (e a linguagem núcleo de *ML* do algortimo W definido na Figura 1) não considera recursão: x não pode ocorrer em e na expressão $\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e'$.

No entanto, o uso do algoritmo definido na Figura 1 não é necessário para determinar o tipo de funções, na prática, uma vez que ele é baseado fundamentalmente no uso do tipo de variáveis que estão no contexto de tipos, pela introdução de novas variáveis de tipo para o tipo de variáveis introduzidas em λ-abstrações, e pelo uso de unificação em aplicações de funções a seus argumentos. Não é necessário seguir todos os passos do algoritmo. Podemos usar a seguinte técnica, que vamos chamar de técnica-informal-de-inferência-de-tipos. Considere, por exemplo, a tarefa de inferir o tipo da expressão:

$$\lambda f.\, \lambda x.\, f\ (f\ x)$$

Sabemos que o tipo dessa expressão é da forma:

$$t_f \rightarrow t_x \rightarrow t_r$$

onde: t_f (é uma variável que) representa o tipo de f,
t_x (é (uma variável que) representa o tipo de x e
t_r (é uma variável que representa) o tipo do resultado, $f\ (f\ x)$.

Introduzimos agora as informações que pudermos obter com os usos das variáveis f e x na expressão $f\ (f\ x)$, devido a unificação, que são:

- f tem tipo funcional (pois x é aplicado a f), ou seja, é da forma $t_1 \to t_2$;
- o tipo de x é o mesmo do domínio de f, ou seja, $t_1 = t_x$;
- o tipo do resultado de f (t_2) é o mesmo do argumento de f (pois o resultado, $f\,x$ é aplicado a f).

.

Note: cada ocorrência de parâmetro de função — i.e. cada variável em uma λ-abstração — tem um tipo monomórfico (essa é uma característica fundamental do sistema de tipos de *ML* e *Haskell*).

Usando essas informações, obtemos que o tipo de:

$$\lambda f.\, \lambda x.\, f\,(f\,x)$$

é igual a (chamando t_x simplesmente de a):

$$(a \to a) \to a \to a$$

Essa técnica pode ser estendida para os casos de definições recursivas, que envolvam mais de uma equação, (lembrando que parâmetros têm tipos monomórficos e) considerando que:

- uma equação $f p_1 \ldots p_n$ = e pode ser vista como tendo o tipo de uma λ-abstração: $\lambda p_1. \ldots \lambda p_n.e$, sendo esse o tipo de f.
- toda definição:

$$\begin{array}{l} f\, p_{11}\ \ldots\ p_{1n}\ =\ e_1 \\ \ldots \\ f\, p_{m1}\ \ldots\ p_{mn}\ =\ e_m \end{array}$$

  deve ser tal que os tipos de cada resultado e_i, assim como os tipos de cada parâmetro p_{ij}, para cada i, j ($i = 1, \ldots, m$ e $j = 1, \ldots, n$) são iguais: para isso os tipos de cada equação devem ser unificados (se não for possível unificar os tipos de cada equação, a definição não é bem tipada).

Por exemplo, considere a definição de *map*:

```
map f []     = []
map f (a:x)  = f a : map f x
```

Vamos calcular o tipo de *map* inicialmente na primeira equação. Poderíamos começar com $t_f \to t_l \to t_r$, sendo t_f, onde:

t_f é (uma variável que representa) o tipo de f,
t_x é (uma variável que representa) o tipo de `[]` e
t_r é (uma variável que representa) o tipo do resultado, também `[]`.

No entanto, como sabemos, pelo tipo de `[]` no contexto de tipos, que `[]` tem tipo `[`a`]`, sendo a uma variável nova, podemos considerar que o tipo de *map* na primeira equação é:

$$t_f \to [a] \to [b]$$

Note que as ocorrências de `[]` têm tipos distintos, com variáveis de tipo a e b novas, pois o tipo de `[]`, no contexto de tipos, é polimórfico.

Na segunda equação, podemos atribuir a `map` inicialmente o tipo: $t_f \to t_l \to t_r$, sendo: t_f variável que representa o tipo de f, t_l variável que representa o tipo de `a:x` e t_r variável que representa o tipo de `f a:map f x`. Temos:

- sendo t_l igual a `[`a`]` (de acordo com o tipo de `[]` usado para esse parâmetro na primeira equação), o tipo de `a` é igual a a e o tipo de `x` é igual a `[`a`]` (pois o tipo de `(:)` no contexto de tipos é igual a $a' \to$ `[`a'`]` $\to$ `[`a'`]`, para alguma variável nova a', mas a' deve ser igual a a, pois os tipos dos parâmetros nas duas equações devem ser iguais.

- t_f deve ser um tipo funcional, pois `f` é aplicado a `a`, e o tipo do resultado de `f` deve ser igual a b, igual ao tipo do elemento da lista [b]: [b] é o resultado de `map f []` na primeira e na segunda equações, e igual ao tipo do resultado de `map f x`.

Portanto, o tipo de `map`, definido pelas duas equações acima, é:

$$(a \to b) \to [a] \to [b]$$

1.2.2 Sobrecarga e Tipos restritos

Um valor sobrecarregado tem um tipo polimórfico restrito (ou constrito) — por exemplo, `(==)` tem tipo `Eq a => a -> a -> Bool`.

Quando a sobrecarga é resolvida (e apenas neste caso), devido a instanciação do tipo polimórfico (via unificação), é preciso verificar se há instâncias no contexto que satisfazem ao tipo do valor sobrecarregado que foi instanciado:

- se há apenas uma instância, a restrição é removida do tipo;
- se não ha nennhuma instância, ocorre um erro de tipo: insatisfazibilidade.
- se há duas ou mais instâncias, ocorre um erro de tipo: ambiguidade.

Por exemplo, a expressão `1 + '1'` usualmente gera um erro de tipo, porque usualmente não há instância de *Num* para o tipo *Char*.

Em Haskell, infelizmente, não há, ainda, definição de quando uma sobrecarga é resolvida, e esta condição (quando uma sobrecarga é resolvida) é confundida com ambiguidade: em Haskell um tipo é ambíguo se existe uma variável de tipo que ocorre em uma restrição e não ocorre no tipo simples. Por exemplo, o tipo da expressão `(show . read)`, que é:

```
(Read a, Show a) => String -> String
```

é considerado ambíguo, pois a variável de tipo `a` ocorre em uma restrição e não ocorre no tipo (simples) `String -> String`.

Em Haskell, não há, assim, diferença entre insatisfazibilidade e ambiguidade, nem entre sobrecarga resolvida e ambiguidade.

O teste de satisfazibilidade, que deve verificar se existe ou não uma única instância que satisfaz a uma restrição, é em geral um problema indecidível [6, 8]. Existem várias opções de compilação no GHC para restringir tipos polmórficos restritos [3]. No entanto, essas opcões poderiam ser evitadas por um mecanismo baseado em uma medida do tamanho dos tipos que formam as restrições, medida essa calculada durante o processo de testar satisfazibilidade [5, 1].

Quando há mais de uma equação em uma definição, as restrições de cada equação devem ser consideradas (isto é, devem ser unidas) para gerar a restrição final resultante da unificação de cada uma das equações. Veja por exemplo o exercício resolvido 1.

2 Exercícios Resolvidos

1. Considere a definição de `merge` a seguir:

```
merge :: ([a], [a]) -> [a]
merge ([], y) = y                         -- (1)
merge (x, []) = x                         -- (2)
merge (a:x, b:y)                          -- (3)
  | a <= b      = a:merge (x, b:y)        -- (4)
  | otherwise = b:merge (a:x, y)          -- (5)
```

Use a técnica-informal-de-inferência-de-tipos para determinar o tipo de `merge`.

Solução: O tipo de `merge` é, inicialmente, igual a $t_p \to t_r$, onde t_p é o tipo do argumento e t_r o tipo do resultado. Vamos determinar o tipo de cada equação, depois vamos considerar que o tipo de cada equação é o mesmo, e para isso vamos usar unificação, conservando (isto é, unindo) as restrições existentes nos tipos de cada equação.

O tipo de `merge` na linha `1` (primeira equação) é `(`$[a]$`,`t_y`) ->` t_y.

O tipo de `merge` na linha `2` (segunda equação) é `(`t_x`,`$[b]$`) ->` t_x.

Na linha `3` (são duas equações neste caso, uma para cada guarda), o tipo de `merge` é $Ord\ a$ `=> (`$[a]$`,`$[a]$`) ->` $[a]$; note que:

- a restrição $Ord\ a$ é inserida devido ao uso do operador sobrecarregado `<=` (na linha `4`);
- inicialmente o tipo de `merge` na linha `3` seria `(`$[a]$`,`$[b]$`) ->` t_r, mas, tanto devido à comparação `a <= b` — o tipo de `<=` é `Ord a => a -> a -> Bool`, portanto o tipo de `a` e de `b` em `a <= b` têm que ser iguais — quanto devido ao fato de que os tipos de t_r nas linhas `4` e `5` têm que ser iguais (t_r é igual $[a]$ na linha `4` e igual a $[b]$ na linha `5`, o tipo de `merge` na última "equação" é $Ord\ a$ `=> (`$[a]$`,`$[a]$`) ->` $[a]$).

Unificando os tipos das equações, obtemos o tipo de `merge`:

$$Ord\ a \Rightarrow ([a],[a]) \rightarrow [a]$$

3 Exercícios

1. Determine o tipo de $\lambda x.\ \lambda y.\ \lambda z.\ x\ z(y\ z)$, usando a técnica-informal-de-inferência-de-tipos.

2. Use a técnica-informal-de-inferência-de-tipos para determinar o tipo da função `either`, definida abaixo:

   (a) Sendo o tipo *Either* definido como:

   ```
   data Either a b = Left a | Right b
   ```

   defina como é inferido o tipo de `either`, usando a técnica-informal-de-inferência-de-tipos, definido como:

   ```
   either f g (Left x)  =  f x
   either f g (Right y) =  g y
   ```

3. Defina expressão ou função com tipo:

   (a) `(`Ord `a,` $Show$ `a) => a -> a ->` $String$

   (b) $(a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c$

   (c) $(a \rightarrow b,\ a \rightarrow c) \rightarrow a \rightarrow (b,c)$

   (d) $(a \rightarrow b,\ b \rightarrow d) \rightarrow (a,b) \rightarrow (c,d)$

   (e) $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$

   (f) $[(a,\ b)] \rightarrow ([a],\ [b])$

   (g) $(a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow [a]$

Referências

[1] Carlos Camarão, Lucília Figueiredo, and Rodrigo Ribeiro. Ambiguity and Constrained Polymorphism. *Science of Computer Programming*, 124(1):1–19, 2016.

[2] Luís Damas and Robin Milner. Principal type schemes for functional programs. In *Proc. of POPL'82*, pages 207–212, 1982.

[3] Simon Peyton Jones et al. GHC — The Glasgow Haskell Compiler. http://www.haskell.org/ghc/, 1998.

[4] P. Martin-Löf. Constructive Mathematics and Computer Programming. In *Proc. of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*, pages 167–184. Prentice-Hall, Inc., 1985.

[5] Carlos Camarão Rodrigo Ribeiro. Ambiguity and Context-Dependent Overloading. *Journal of the Brazilian Computer Society*, 19(3):313–324, 2013.

[6] Geoffrey Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping.* PhD thesis, Cornell University, 1991.

[7] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics).* Elsevier Science Inc., New York, NY, USA, 2006.

[8] Dennis Volpano and Geoffrey Smith. On the Complexity of ML Typability with Overloading. In *Proc. of the ACM Symposium on Functional Programming Computer Architecture.*, number 523 in LNCS, pages 15–28, 1991.