Análise semântica

Alexsandro Santos Soares
adaptado dos slides de Aarne Ranta

Universidade Federal de Uberlândia
Faculdade de Computação

Propósitos da verificação de tipos

- Definir o que um programa deveria fazer.
  - Ex: ler um vetor de inteiros e retornar um double.
- Garantir que o programa é significativo.
  - Ele não soma uma string a um inteiro.
  - As variáveis são declaradas antes de serem usadas.
- Documentar as intenções do programador.
  - Melhor que comentários que não são verificados pelo compilador.
- Otimizar o uso do hardware.
  - Reservar a quantidade mínima de memória e nada mais.
  - Usar as instruções de máquina mais apropriadas.

Especificação de um verificador de tipos

- Vamos especificar um verificador de tipos de forma independente da linguagem de implementação.
- Para isso usaremos um **sistema de tipos** com **regras de inferência**.

Exemplo

$$\frac{\mathrm{a} : \texttt{bool} \qquad \mathrm{b} : \texttt{bool}}{\mathrm{a}\ \texttt{\&\&}\ \mathrm{b} : \texttt{bool}}$$

Se a possui tipo `bool` e b possui tipo `bool`, então a *&&* b possui tipo `bool`

Regras de inferência

- Uma regra de inferência possui um conjunto de **premissas** $J_1, \ldots, J_n$ e uma **conclusão** J, separadas por uma linha horizontal:

$$\frac{J_1 \qquad \ldots \qquad J_n}{J}\, C$$

- Ela pode ser lida de diferentes formas:
  - Das premissas $J_1, \ldots, J_n$ pode-se concluir J, se a condição C é assegurada.
  - Dada C e se $J_1, \ldots, J_n$, então J.
  - Para verificar J, verifique $C, J_1, \ldots, J_n$.

Sentenças

- As premissas e as conclusões são denominadas **sentenças**.
- As sentenças mais comuns em sistemas de tipos possuem a forma

$$e \, : \, T$$

lida como: a expressão e possui tipo T.

De regras de tipagem a pseudocódigo

- É possível converter a regra

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

em um pseudocódigo da forma

$$\begin{array}{l} J : \\ \qquad J_1 \\ \qquad \ldots \\ \qquad J_n \end{array}$$

- Assim a conclusão torna-se um caso para casamento de padrão e as premissas tornam-se chamadas recursivas.

Verificação de tipos e inferência de tipos

- Dois tipos de programas:

  Verificação de tipos dada uma expressão e e um tipo T, decida se $e : T$.

  Inferencia de tipos dada uma expressão e, encontre um tipo T tal que $e : T$.

- Ambos os programas podem ser necessários.
- Ambos são deriváveis das regras de tipagem.

Exemplo: verificação de tipos para &&

```
verifica(a && b, bool):
   verifica(a, bool)
   verifica(b, bool)
```

Não há casamento de padrões para outros tipos além de `bool`, assim, a verificação de tipo falha para eles.

Exemplo: inferência de tipos para `&&`

```
infere(a && b):
   verifica(a, bool)
   verifica(b, bool)
   retorne bool
```

Note que a função também deve verificar que os operandos são do tipo `bool`.

Do pseudocódigo ao código

Embora tenhamos usado a sintaxe concreta na especificação de padrões para as regras

```
infere(a && b):
   verifica(a, bool)
   verifica(b, bool)
   retorne bool
```

No código real, devemos usar a sintaxe abstrata. Por exemplo, em OCaml

```
let rec infere exp = match exp with
   | ExpOp(Mais, a, b) ->
     if (verifica a TipoBool) && (verifica b TipoBool)
     then TipoBool
     else failwith "Ambos os tipos deveriam ser booleanos"
```

Note que a função também deve verificar que os operandos são do tipo `bool`.

Ambientes

- Uma variável pode ter *qualquer* um dos tipos disponíveis na linguagem.
- Em C ou Java, o tipo é determinado pela *declaração* da variável.
- Nas regras de inferência, as variáveis são agrupadas em um **ambiente**, também chamado de *contexto*.
- Em um compilador, o ambiente é uma **tabela de símbolos** que mapeia uma variável para seu tipo.
- Nas regras de inferência, o ambiente é denotado pela letra grega Γ, gama.
- A sentença para tipagem é generalizada para

$$\Gamma \vdash e : T$$

lida como: a expressão e possui tipo T no ambiente Γ.

Exemplo

$$\{x : \text{int},\ y : \text{int}\} \vdash x + y > y : \text{bool}$$

Significando que $x + y > y$ é uma expressão booleana em um ambiente onde x e y são variáveis inteiras.

Note a notação para ambientes:

$$\{x_1 : T_1, \ldots, x_n : T_n\}$$

Quando adicionarmos uma nova variável ao ambiente Γ, escreveremos

$$\Gamma + \{x : T\}$$

Observação

- Muitas sentenças possuem o mesmo Γ, pois o ambiente não muda.

$$\frac{\Gamma \vdash a : \text{bool} \qquad \Gamma \vdash b : \text{bool}}{\Gamma \vdash a \&\& b : \text{bool}}$$

- Entretanto, o ambiente mudará nas regras de tipagem para declarações.

Regra de tipagem para variáveis em expressões

- Isto é onde os ambientes são necessários.

$$\frac{}{\Gamma \vdash x : T} \text{ se } x : T \in \Gamma$$

- A condição *se* $x : T \in \Gamma$ não é uma sentença formal, mas uma sentença da metalinguagem (Português).
- No pseudocódigo, ela não se tornará uma chamada recursiva, mas uma função de *busca*:

infere(Γ, x)*:*
  t *:=* *busca*(Γ, x)
  retorna t

Regra de tipagem para variáveis em expressões

Em código Ocaml

```
let rec infere_exp amb exp = match exp with
| ExpVar (VarSimples nome) ->
   (try
       let tipo = busca amb nome in tipo
    with Not_found ->
       let msg = "A variável " ^ nome ^ " não foi declarada" in
       failwith msg
   )
```

Verificando a tipagem de aplicação de função

- Há a necessidade de buscar o tipo da função

$$\frac{\Gamma \vdash a_1 : T_1 \qquad \cdots \qquad \Gamma \vdash a_n : T_n}{\Gamma \vdash f(a_1, \ldots, a_n) : T} \text{ se } f : (T_1, \ldots, T_n) \to T \in \Gamma$$

Provas em um sistema de tipo

- Uma *árvore de prova* é um resumo dos passos que um verificador de tipos realiza, construído regra por regra.

$$\dfrac{\dfrac{\dfrac{}{\{x : \text{int},\ y : \text{int}\} \vdash x : \text{int}}\,x \qquad \dfrac{}{\{x : \text{int},\ y : \text{int}\} \vdash y : \text{int}}\,y}{\{x : \text{int},\ y : \text{int}\} \vdash x + y : \text{int}}\,+ \qquad \dfrac{}{\{x : \text{int},\ y : \text{int}\} \vdash y : \text{int}}\,y}{\{x : \text{int},\ y : \text{int}\} \vdash x + y > y : \text{bool}}\,>$$

- Cada sentença é uma conclusão de uma das sentenças acima, usando a regra indicada à direita da linha.
- Essa árvore usa a regra para variável e também as regras para $+$ e $>$:

$$\dfrac{}{\Gamma \vdash x : T}\,x \qquad \dfrac{\Gamma \vdash a : \text{int} \qquad \Gamma \vdash b : \text{int}}{\Gamma \vdash x + y : \text{int}}\,+ \qquad \dfrac{\Gamma \vdash a : \text{int} \qquad \Gamma \vdash b : \text{int}}{\Gamma \vdash x > y : \text{bool}}\,>$$

Sobrecarga

- As operações aritméticas (`+ - *`) e as comparações (`== != < > <= >=`) são *sobrecarregadas* em muitas linguagens, ou seja, usáveis com diferentes tipos.
- Se os tipos possíveis são `int`, `double` e `string`, então as regras de tipagem se tornam:

$$\frac{\Gamma \vdash a : t \qquad \Gamma \vdash b : t}{\Gamma \vdash a + b : t} \text{ se } t \text{ é } \texttt{int}, \texttt{double} \text{ ou } \texttt{string}$$

$$\frac{\Gamma \vdash a : t \qquad \Gamma \vdash b : t}{\Gamma \vdash a == b : \mathrm{bool}} \text{ se } t \text{ é } \texttt{int}, \texttt{double} \text{ ou } \texttt{string}$$

Inferência de tipo para sobrecarga

- Primeiro infira o tipo do primeiro operando, depois verifique o segundo operando em relação a esse tipo:

  infere$(a+b)$*:*
  $\quad t :=$ *infere(a)*
  $\quad$ *// verifica se* $t \in \{int, double, string\}$
  $\quad$ *verifica(b,t)*
  $\quad$ **retorne** t

- Para outras operações aritméticas, somente `int` e `double` são possíveis.

Conversões de tipo

- Em algumas linguagens, um inteiro pode ser *convertido* em um double, ou seja, usado como um double.
- Em muitas máquinas, inteiros e doubles possuem representações binárias totalmente diferentes, assim como diferentes conjuntos de instruções.
- Assim, o compilador normalmente tem que gerar uma instrução especial para conversões entre tipos.

Convertendo de um tipo menor para um maior

- Sem perda de informação

$$\frac{\Gamma \vdash a : t \qquad \Gamma \vdash b : u}{\Gamma \vdash a + b : \max(t, u)} \text{ se } t, u \in \{\text{int}, \text{double}, \text{string}\}$$

- Assumindo o ordenamento seguinte

$$\text{int} < \text{double} < \text{string}$$

- Por exemplo:

$$\max(\text{int}, \text{string}) = \text{string}$$

- Assim `2 + "oi"` resulta em `"2oi"`, pois a adição de strings está no máximo.

Validade de comandos

- Ao se realizar a verificação de tipos de um comando não se está interessado em um tipo, mas apenas se o comando é **válido**.
- Uma nova sentença da forma

$$\Gamma \vdash s \text{ válido}$$

  lida como: um comando s é válido em um ambiente Γ.
- Exemplo: comando `while`

$$\frac{\Gamma \vdash e : \text{bool} \qquad \Gamma \vdash s \text{ válido}}{\Gamma \vdash \text{while } (e) \ s \text{ válido}}$$

- Verificar a validade pode, assim, envolver verificar o tipo de algumas expressões.

Comandos expressões

- Não é importante o tipo da expressão, desde que ela tenha um.
- Ou seja, pode-se inferir um tipo para ela

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e;\ \text{válido}}$$

- Esse caso pode cobrir chamadas de funções ou atribuições.

Validade de definicões de funções

$$\frac{\{x_1 : T_1, \cdots, x_m : T_m\} \vdash s_1 \cdots s_n \text{ válido}}{T\ f(T_1\ x_1, \cdots, T_m\ x_m)\{s_1 \cdots s_n\} \text{ válido}}$$

- As variáveis declaradas como parâmetros definem o ambiente.
- Os comandos do corpo $s_1 \cdots s_n$ são verificados nesse ambiente.
- O ambiente pode ser alterado dentro do corpo devido às declarações.
- O verificador também deve se certificar que todas as variáveis na lista de parâmetros são distintas.

Comandos return

- Ao se verificar a definição de uma função, pode-se verificar se o corpo da função contém um comando `return` do tipo esperado.
- Uma versão mais sofisticada poderia também verificar retornos em ramificações do `if`, como em:

  ```
  if (fail()) return 1; else return 0;
  ```

Declarações e estruturas de blocos

- Cada declaração possui um *escopo*, que está dentro de um certo *bloco*.
- Dois princípios regulam o uso de variáveis:
    1. Uma variável declarada em um bloco é visível até o final deste bloco.
    2. Uma variável pode ser declarada novamente em um bloco aninhado, mas não no mesmo bloco.

Exemplo

```
{
  int x;
  {
    x = 3;    // x : int
    double x; // x : double
    x = 3.14;
    int z;
  }
  x = x + 1;  // x : int, recebe o valor 3 + 1
  z = 8;      // ILEGAL! z não está mais no escopo
  double x;   // ILEGAL! x não pode ser redeclarado
  int z;      // legal, pois z não está mais no escopo
}
```

Pilhas de ambientes

- Podemos refinar a noção de ambiente para lidar com blocos.
- Ao invés de usar uma única tabela de símbolos, Γ pode ser *uma pilha de tabelas de símbolos*.
- Separaremos as tabelas com pontos.
- Por exemplo

  $$\Gamma_1 \cdot \Gamma_2$$

  onde Γ_1 é o antigo ambiente, ou seja, o mais externo, e Γ_2 é um novo ambiente.
- O ambiente mais interno está no topo da pilha.

Refinando a busca para funcionar com uma estrutura de blocos

- Com apenas um ambiente, a busca funciona como antes.
- Com uma pilha de ambientes, ela inicia procurando no contexto no topo da pilha e se aprofunda na pilha apenas se ela não encontrar a variável.

Refinando a regra de declaração

- Uma declaração introduz uma nova variável no escopo atual.
- A variável deve ser nova nesse ambiente.
- Como expressar que uma nova variável será adicionada ao ambiente tal que os comandos seguindo-a usarão esse novo ambiente?
- Precisamos de regras para *sequências de comandos*, não apenas para comandos individuais:

$$\Gamma \vdash s_1 \cdots s_n \text{ válido}$$

Refinando a regra de declaração

- A declaração estende o ambiente usado para verificar os comandos que a seguem:

$$\frac{\Gamma + \{x : T\} \vdash s_2 \cdots s_n \text{ válido}}{\Gamma \vdash T x; s_2 \cdots s_n \text{ válido}} \; x \text{ não está no contexto do topo de } \Gamma$$

- Em outras palavras, uma declaração seguida por alguns comandos $s_2 \cdots s_n$ é válida se esses comandos são válidos em um ambiente onde a variável declarada é adicionada.
- Essa adição faz com que o verificador de tipos reconheça o efeito da declaração.

Verificação de comandos em blocos

- Empilhe um novo ambiente na pilha

$$\frac{\Gamma . \vdash r_1 \cdots r_m \text{ válido} \qquad \Gamma \vdash s_2 \cdots s_n \text{ válido}}{\Gamma \vdash \{r_1 \cdots r_m\} s_2 \cdots s_n \text{ válido}}$$

- Somente os comandos dentro do bloco são afetados pela declaração e não os comandos seguintes.