Compiladores – Análise de Tipos

Fabio Mascarenhas – 2018.1

http://www.dcc.ufrj.br/~fabiom/comp

Tipos

- Um *tipo* é:
  - Um conjunto de valores
  - Um conjunto de operações sobre esses valores
- Os tipos de uma linguagem podem ser pré-definidos, mas normalmente as linguagens também permitem que o programador defina seus tipos
- Os tipos de uma linguagem formam sua própria mini-linguagem

Sistema de Tipos

- O *sistema de tipos* de uma linguagem especifica a *sintaxe* dos tipos, e quais operações são válidas nesses tipos
- O compilador usa as regras do sistema de tipos para fazer a *verificação de tipos* do programa
- O objetivo é rejeitar programas que contêm operações inválidas
- Várias linguagens adiam essa verificação até o momento em que o programa está executando

Tipagem estática/dinâmica e forte/fraca

- Uma linguagem tem *tipagem forte* se a verificação de tipos sempre é feita para todas as operações
  - A maior parte das linguagens (incluindo Java) tem tipagem forte, pois ela tem implicação direta na *segurança* dos programas
  - A linguagem C tem tipagem fraca, pois o sistema de tipos é facilmente "desligado", podendo-se manipular diretamente os bytes da memória
- Uma linguagem é *estaticamente tipada* se quase toda a verificação de tipos é feita pelo compilador antes do programa ser executado, e *dinamicamente tipada* se quase toda a verificação é feita no momento de execução

Verificação de Tipos Estática

- Poderíamos dar todas as regras de verificação de tipos de uma linguagem informalmente, mas existem formalismos que tornam essa especificação mais precisa

- A especificação das regras de verificação de tipo de uma linguagem se dá através de *regras de dedução*

- As regras de dedução dão um esquema de como podemos *deduzir* o tipo de uma expressão dados os tipos de suas subexpressões

- Os *axiomas* do sistema de tipos dão a tipagem dos literais e identificadores que aparecem no programa

Regras de Dedução

- Tradicionalmente usamos uma notação “barra” para as regras de dedução, em que as hipóteses da regra ficam acima de uma barra horizontal e a conclusão abaixo dessa barra

- Tanto as hipóteses quanto a conclusão são escritas da forma ⊢ ***e*: *t***, onde *e* é uma expressão, *t* um tipo e o símbolo ⊢ é a “catraca”

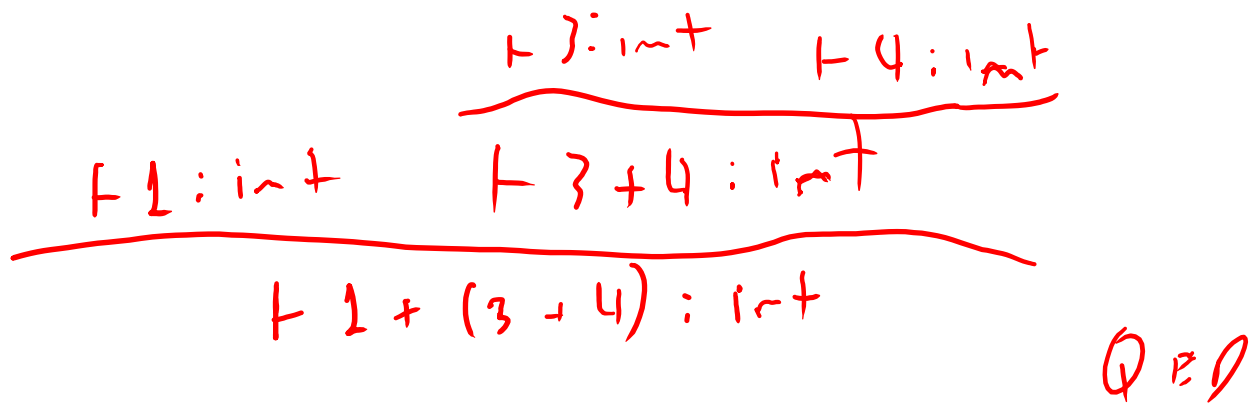
  - Lê-se “pode-se provar que *e* tem tipo *t*”

$$\vdash n : \text{int}$$

$$\frac{\vdash e_1 : \text{int} \quad \vdash e_2 : \text{int}}{\vdash e_1 + e_2 : \text{int}}$$

Exemplo – tipagem de expressões simples

- Vamos deduzir o tipo de 1 + (3 + 4):



Consistência e completude

- Como todo sistema lógico, podemos falar na *consistência* e *completude* de um sistema de tipos

- Um sistema de tipos é *consistente* se tudo que ele consegue provar é verdade, ou seja, se todo valor que uma expressão *e* com $\vdash e : t$ produz em tempo de execução tem tipo *t*

- Um sistema de tipos é *completo* se podemos tipar todos os programas corretos

- Em geral queremos que os sistemas de tipos sejam consistentes, mas dificilmente eles são completos

Exemplo - consistência

- A regra abaixo é consistente? Por quê?

$$\frac{\vdash e_1 : \text{int} \quad \vdash e_2 : \text{int}}{\vdash e_1 / e_2 : \text{boolean}}$$

- E quanto à regra abaixo?

$$\frac{\vdash e_1 : \text{int} \quad \vdash e_2 : \text{int}}{\vdash e_1 / e_2 : \text{int}}$$

- Consistência depende do comportamento da linguagem!

Tipagem de variáves

- Qual o tipo de uma variável?
- Não podemos determinar esse tipo sintaticamente, ele depende do *contexto*
- Vamos dar esse contexto usando uma *tabela de símbolos* que irá associar cada nome ao seu tipo declarado:

$$\mathrm{T} \vdash id : \mathrm{T.procurar(id)}$$

- Declarações de variáveis inserem os tipos na tabela
- A verificação de tipos pode ser feita em paralelo com a análise de escopo!

Tipos em TINY

- Atualmente todas as variáveis em TINY são números inteiros, e a própria sintaxe da linguagem está garantindo que todas as operações do programa são válidas

- Vamos mudar a linguagem para ter três tipos, `int`, `real` e `bool`, com conversão implícita de `int` para `real`, incluindo declaração de tipos na própria linguagem

```
VAR  -> var DECLS ;
      |
DECLS -> DECLS , DECL
       | DECL
DECL -> IDS : TIPO
IDS  -> IDS , id
      | id
```

```
TIPO -> int
      | real
      | bool
```

Tipagem de expressões

- As expressões aritméticas possem tipo inteiro se ambos os operandos forem inteiros; um operando real faz elas terem tipo real

- O verificador de tipos pode inserir *casts* (conversões de tipo) explícitos nos pontos em que precisa usar conversão, para facilitar o trabalho do gerador de código

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \oplus e_2 : int} \qquad \frac{\Gamma \vdash e_1 : real \quad \Gamma \vdash e_2 : real}{\Gamma \vdash e_1 \oplus e_2 : real}$$

$$\frac{\Gamma \vdash e : int}{\Gamma \vdash cast(e) : real}$$

Tipagem de comandos

- Os comandos TINY por si só não têm tipos, mas as regras de tipagem garantem que toda expressão usada dentro de um comando está consistente

$T \vdash c$ ≡ c está bem tipado

$T_1 \vdash b : T_2$ b está bem tipado e produz a tabela de símbolos T_2

$$\frac{T \vdash b : T' \quad T' \vdash e : bool}{T \vdash repeat\ b\ until\ e}$$

$$\frac{T \vdash e : bool \quad T \vdash b_1 : T' \quad T \vdash b_2 : T''}{T \vdash if\ e\ then\ b_1\ else\ b_2}$$

$$\frac{T' = T[v_1 \to t_1, \ldots, v_m \to t_m] \quad T' \vdash c_1 \ldots T' \vdash c_m}{T \vdash var\ v_1 : t_1 \ldots v_m : t_m;\ c_1 \ldots c_m : T'}$$

Tipagem de procedimentos

- Os procedimentos que colocamos em TINY não possuem parâmetros, então não faz sentido falar de verificação de tipos nas chamadas de procedimentos

- Mas e se colocamos parâmetros e retorno?

```
PROC  -> procedure id ( [DECLS] ) [: TIPO] CMDS end
CMD   -> id ( [EXPS] )
       | …
EXPS  -> EXPS , EXP
       | EXP
EXP   -> id ( [EXPS] )
       | ...
```

- Seguindo a linhagem Pascal, definimos o valor de retorno de um procedimento com uma atribuição para uma variável com o nome do procedimento

Tipagem de procedimentos – linhas gerais

- Como os procedimentos estão em um espaço de nomes separado das variáveis, seu contexto de tipagem também é diferente

- A ideia é representar o tipo de um procedimento como combinação dos tipos de seus parâmetros e do seu tipo de retorno

$$(t_1, \ldots, t_m) : t_n$$

- A verificação da chamada checa o número de parâmetros, e o tipo de cada um versus o tipo dos argumentos

$$\frac{T[p_1 \rightsquigarrow t_1 \ldots p_n \rightsquigarrow t_n, i2 \rightsquigarrow (t_1 \ldots t_n) : t_n], p \vdash b : T'}{T, p \vdash \text{procedure } i2(p_1 : t_1 \ldots p_n : t_n) : t_n \text{ b end}}$$

- A verificação do *corpo* do procedimento põe o tipo de cada parâmetro e da variável de retorno no ambiente de tipos de variáveis

$$\frac{p(i2) = (t_1, \ldots t_m) : t_n \quad m = n \quad T, p \vdash a_1 : t_1 \ldots T, p \vdash a_n : t_n}{T, p \vdash i2(a_1, \ldots, a_n) : t_n}$$

Tipagem de procedimentos - regras

tipo de procedimento

$$(t_1, \ldots, t_m) : t_n$$

declaração

$$\frac{T[p_1 \rightarrow t_1 \ldots p_m \rightarrow t_m, i2 \rightarrow t_n], p \vdash b : T'}{T, p \vdash \text{procedure } i2(p_1 : t_1 \ldots p_m : t_m) : t_n \text{ b end}}$$

chamada

$$\frac{p(i2) = (t_1, \ldots t_m) : t_n \quad m = n \quad T, p \vdash a_1 : t_1 \ldots T, p \vdash a_n : t_n}{T, p \vdash i2(a_1, \ldots, a_m) : t_n}$$