

Compiladores – Análise de Tipos

Fabio Mascarenhas – 2017.2

<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 ^{Q ver no}estática/dinâmica e ^{O Q ver}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 $\vdash e: t$, onde e é uma expressão, t um tipo e o símbolo \vdash é a “catraca”
- Lê-se “pode-se provar que e tem tipo t ”

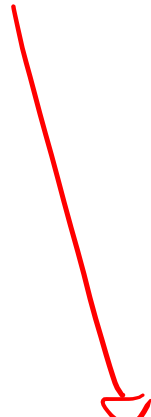
$$\frac{}{\vdash \text{num} : \text{int}} \qquad \frac{\vdash e_1 : \text{int} \quad \vdash e_2 : \text{int}}{\vdash e_1 + e_2 : \text{int}}$$

$$\vdash \text{true} : \text{bool}$$

$$\vdash \text{false} : \text{bool}$$

Exemplo – tipagem de expressões simples

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


$$\frac{\frac{\frac{}{\vdash 1 : \text{int}}}{\vdash 1 : \text{int}} \quad \frac{\frac{\frac{}{\vdash 3 : \text{int}}}{\vdash 3 : \text{int}} \quad \frac{\frac{}{\vdash 4 : \text{int}}}{\vdash 4 : \text{int}}}{\vdash 3 + 4 : \text{int}}}{\vdash 1 + (3 + 4) : \text{int}}$$

Handwritten derivation of the type for the expression $1 + (3 + 4)$. The derivation shows the types of the sub-expressions and how they are combined to deduce the type of the entire expression. The final result is $\vdash 1 + (3 + 4) : \text{int}$.

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 variáveis

- 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:

$T \vdash id: T(id)$
contexto e tipos

- 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 possuem 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

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

$$\frac{\tau \vdash e_1 : \text{real} \quad \tau \vdash e_2 : \text{int}}{\tau \vdash e_1 \oplus e_2 : \text{real}}$$

$$\frac{\tau \vdash e_1 : \text{int} \quad \tau \vdash e_2 : \text{real}}{\tau \vdash e_1 \oplus e_2 : \text{real}}$$

$$\frac{\tau \vdash e_1 : \text{real} \quad \tau \vdash e_2 : \text{real}}{\tau \vdash e_1 \oplus e_2 : \text{real}}$$