Compiladores – Análise de Tipos

Fabio Mascarenhas - 2013.2

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

inteiro, +, -, *, /
string, concat, tamanho
vetor de —, index.

sintaxe e semântica

structs
uniões
classes
interfaces

int + int → int

3 + 2 → 5

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

Java - quase tudo é estaticamente tipado

\ downcasts e escritas em vetores são dinamicamente tipados

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 subexpresões

- 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 $\vdash \boldsymbol{e}\boldsymbol{:}\ \boldsymbol{t}$, onde *e* é uma expressão, *t* um tipo e o símbolo $\vdash$ é a “roleta”

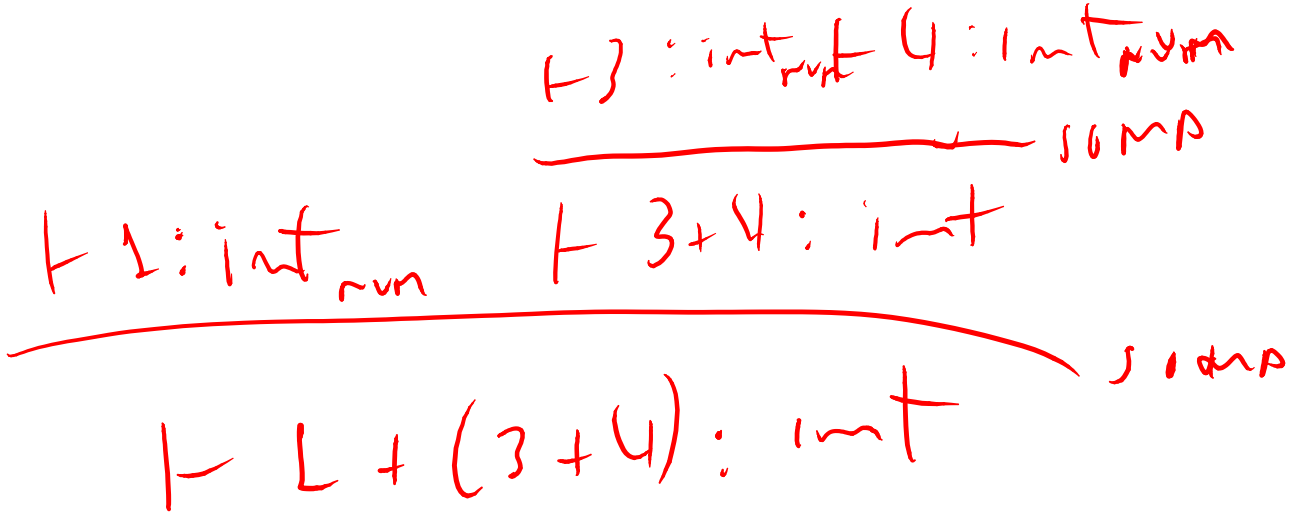
  - Lê-se “pode-se provar que *e* tem tipo *t*”

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

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

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

Vão existir programas corretos que serão rejeitados

Exemplo - consistência

- A regra abaixo é consistente? Por quê?

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

NÃO!

- E quanto à regra abaixo?

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

DEPENDE

- Consistência depende do comportamento da linguagem!

CONSISTÊNCIA DEPENDE DA SEMÂNTICA

Subtipagem

- O conjunto de valores de um tipo pode ser um subconjunto do conjunto de valores de outro tipo
- Podemos querer expressar isso no sistema de tipos através de uma *relação de subtipagem* ≤ $\subseteq$ $t_1 \leq t_2 \rightarrow t_1$ é subtipo de t_2
- Em uma linguagem OO essa relação é declarada pelo programador; em Java ela é dada pelas cláusulas *extends* e *implements*, e em MiniJava pela *extends*
- A relação de subtipagem é *simétrica* (t ≤ t) e *transitiva* (r ≤ s e s ≤ t implica r ≤ t)
- Podemos usar a relação de subtipagem explicitamente nas regras, ou podemos introduzir uma *regra de subsunção*

$$[\text{SUB}]\ \frac{\vdash e : t_1 \quad t_1 \leq t_2}{\vdash e : t_2} \qquad \frac{\vdash l : t_1 \quad \vdash l := e}{\vdash e : t_1} \qquad \frac{\vdash l : t_1 \quad \vdash e : t_2 \quad t_2 \leq t_1}{\vdash l := e}$$

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:

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

- 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 `int` ≤ `real`, incluindo declaração de tipos na linguagem e juntando expressões condicionais com as outras expressões:

```
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* explícitos nos pontos em que precisa usar subsunção, para facilitar o trabalho do gerador de código

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \; (+ \cdot \text{int})$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2 \quad t_1 \leq \text{real} \quad t_2 \leq \text{real}}{\Gamma \vdash e_1 + e_2 : \text{real}} \; (+ \cdot \text{real})$$

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

$$\frac{\Gamma \vdash con_2 : bool \quad \Gamma \vdash c_{if} \quad \Gamma \vdash c_{else}}{\Gamma \vdash \text{if } con_2 \text{ then } c_{if} \text{ else } c_{else}}$$

$\Gamma \vdash c$ válido

comando é válido

$$\frac{\Gamma \vdash id : t_1 \quad \Gamma \vdash e : t_2 \quad t_2 \leq t_1}{\Gamma \vdash id := e}$$

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 quanto a TINYPy?
  - A ideia é representar o tipo de um procedimento como uma tupla de tipos dos parâmetros, mais o tipo do seu valor de retorno
  - A verificação da chamada checa o número de parâmetros, e o tipo de cada um versus o tipo dos argumentos
  - A verificação do *corpo* do procedimento põe o tipo de cada parâmetro no ambiente de tipos de variáveis, e checa as expressões de retorno vs o tipo de retorno

Tipagem de procedimentos - regras

$$\frac{F, V \vdash f : t_n(t_1, \ldots, t_{n-1}) \quad F, V \vdash e_1 : t_1 \quad \ldots \quad F, V \vdash e_{n-1} : t_{n-1}}{F, V \vdash f(e_1, \ldots, e_{n-1}) : t_n}$$

F é o ambiente de funções

V é o ambiente de variáveis

$$\frac{F, V + \{p_1 : t_1, \ldots, p_m : t_m, f : t_n\} \vdash c}{F, V \vdash t_n\ f(t_1\ p_1, \ldots, t_m\ p_m) : c}$$

Procedimentos e subtipagem

- Em linguagens com subtipagem, já a questão de se podemos passar valores para um procedimento com tipos diferentes dos tipos dos parâmetros
- Argumentos podem ser *subtipos* dos tipos dos parâmetros
- O contrário (argumentos como *supertipos*) poderia ser inconsistente!

$$\frac{\Gamma \vdash f : t_n(t_1, \ldots, t_n) \quad \Gamma \vdash e_1 : te_1 \quad \ldots \quad \Gamma \vdash e_n : te_n}{\Gamma \vdash f(e_1, \ldots, e_n) : t_n}$$

$$te_1 \leq t_1$$
$$\vdots$$
$$te_n \leq t_n$$