Compiladores – Análise de Tipos

Fabio Mascarenhas – 2015.1

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 subexpresões

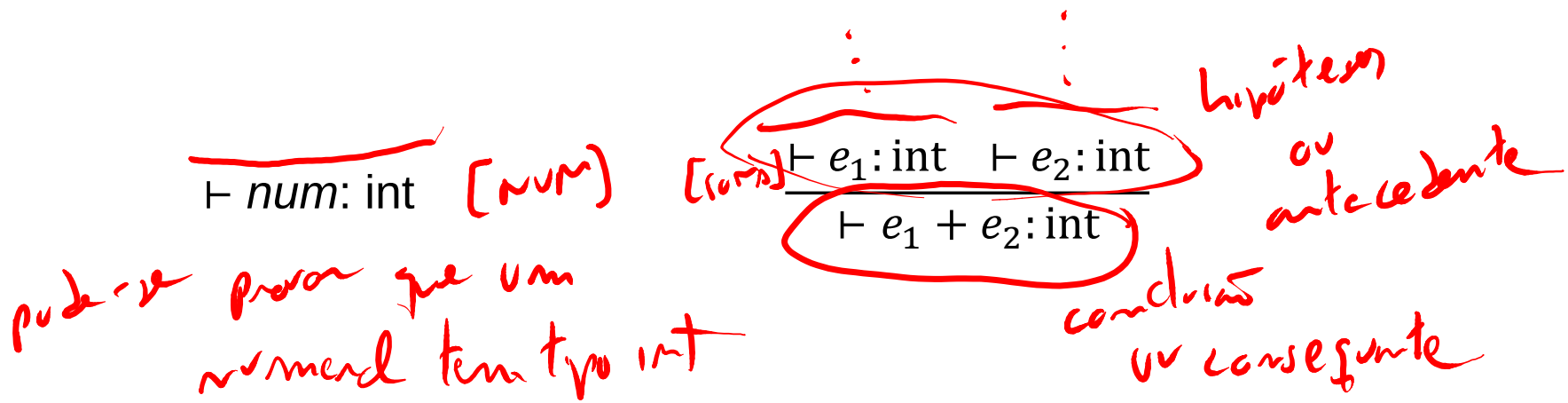
- Os *axiomas* do sistema de tipos dão a tipagem dos literais e identificadores que aparecem no programa

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

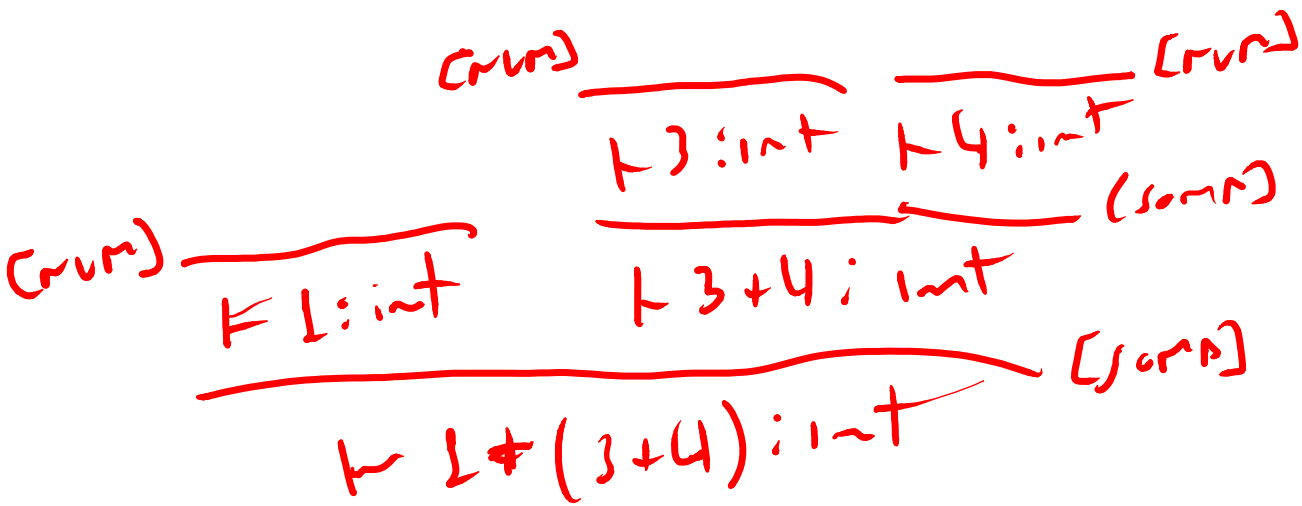
- Tanto as hipóteses quanto a conclusão são escritas da forma ⊢ ***e*: *t***, onde *e* é uma expressão, *t* um tipo e o símbolo ⊢ é a “catraca”

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



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?

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

NÃO

- E quanto à regra abaixo?

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

PODE SER

- Consistência depende do comportamento da linguagem!

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* ≤
- 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*

Subtipagem explícita vs. subsunção

$int \leq real$

- Subtipagem explícita:

$$\frac{\vdash e_1 : t_1 \quad \vdash e_2 : t_2 \quad t_1 \leq real \quad t_2 \leq real}{\vdash e_1 + e_2 : real}$$

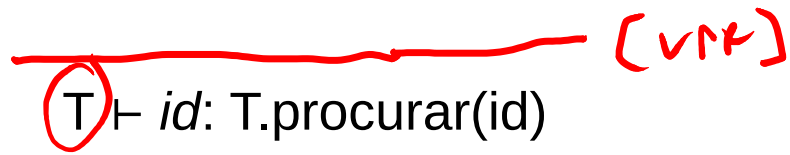
- Subsunção:

$$\frac{\vdash e_1 : real \quad \vdash e_2 : real}{\vdash e_1 + e_2 : real} \qquad \frac{\vdash e : t_1 \quad t_1 \leq t_2}{\vdash e : t_2} \text{ (SUBSUNÇÃO)}$$

- Usar subsunção deixa o sistema mais sintético, usar a subtipagem explícita deixa ele mais fácil de implementar

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:



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

$$\frac{T \vdash e_1 : int \quad T \vdash e_2 : int}{T \vdash e_1 + e_2 : int} \qquad \frac{T \vdash e_1 : real \quad T \vdash e_2 : real}{T \vdash e_1 + e_2 : real}$$

$$\frac{T \vdash e_1 : t_1 \quad T \vdash e_2 : t_2 \quad t_1 \leq t_3 \quad t_2 \leq t_3}{T \vdash e_1 = e_2 : bool} \qquad \frac{T \vdash e_1 : real \quad T \vdash e_2 : real}{T \vdash e_1 < e_2 : bool}$$

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

$$\frac{T \vdash e : t}{T \vdash \text{print } e} \qquad \frac{T.\text{procurar}(id) = t}{T \vdash \text{read } id}$$

$$\frac{T \vdash e : t \quad t \leq T.\text{procurar}(id)}{T \vdash id := e}$$

$$\frac{T \vdash b \quad T \vdash e : \text{bool}}{T \vdash \text{repeat } b \text{ until } e} \qquad \frac{\overline{T[x:t] \vdash c}}{T \vdash \text{var } \overline{x:t}\,;\, \overline{c}}$$

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 tipo 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 mais o seu tipo 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 e da variável de retorno no ambiente de tipos de variáveis

$V, P \vdash e : t \quad P.\mathrm{procura}(i2).\mathrm{params} = \overline{u} \quad \overline{t \leq u}$

$V, P \vdash i2(\overline{e}) : P.\mathrm{procura}(i2).\mathrm{ret}$

Procedimentos e subtipagem

- Em linguagens com subtipagem, há 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!

Tipagem de procedimentos - regras

$$\frac{V, P \vdash e : t \qquad P.\mathit{procura}(i2).\mathit{params} = \overline{u} \qquad \overline{t \leq u}}{V, P \vdash i2(\overline{e}) : P.\mathit{procura}(i2).\mathit{ret}}$$

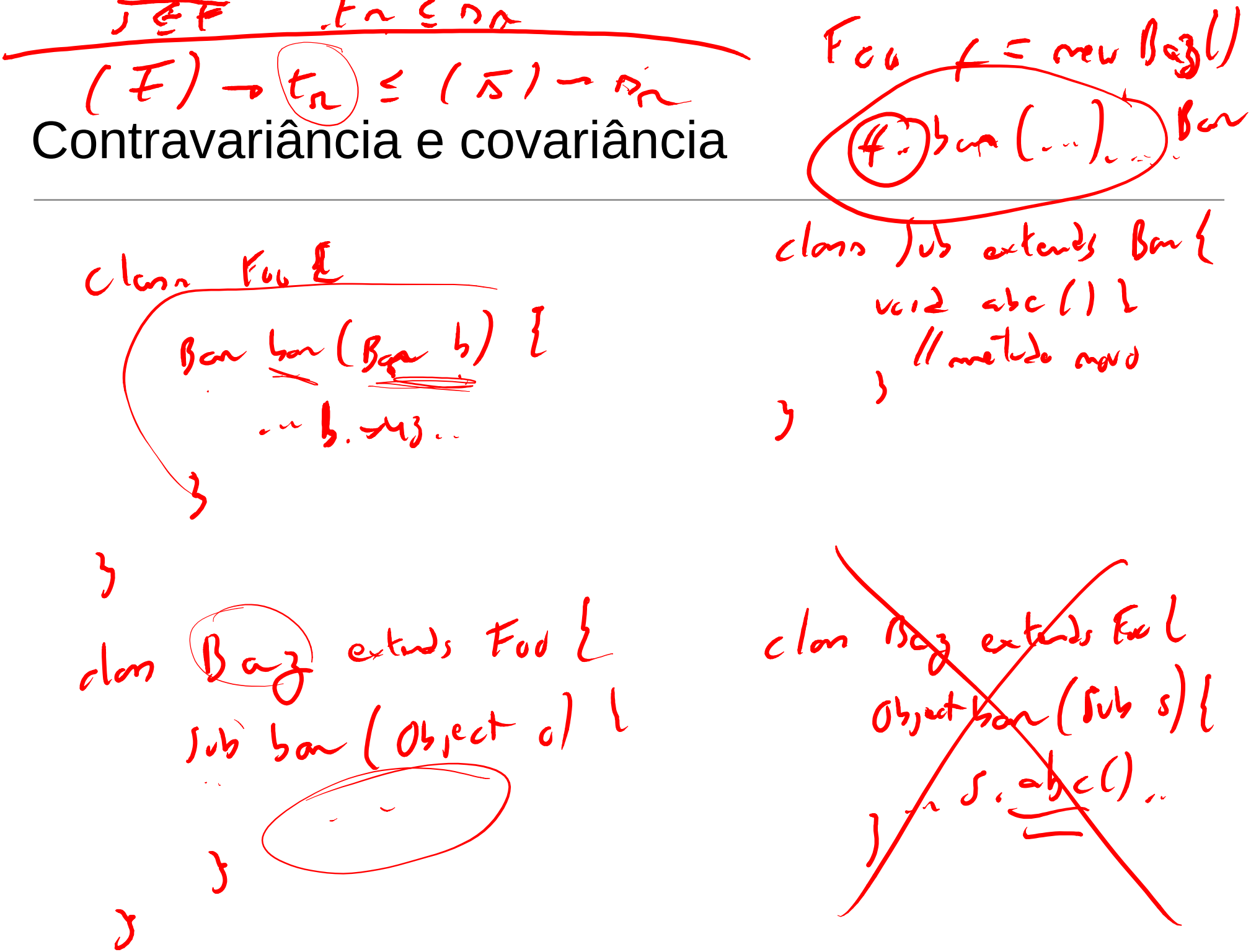
$$\frac{V[\overline{x:t}, i2 : t_n], P \vdash b}{V, P \vdash \mathit{procedure}\ i2(\overline{x:t}) : t_n\ b}$$

Tipagem em MiniJava – classes e métodos

- Em MiniJava, quando o programador define uma classe ele está adicionando um novo tipo à linguagem
- Uma classe é só uma versão mais complicada dos tipos de procedimento que vimos para TINY
- Ele tem quatro partes: o nome da classe, o nome da superclasse, os seus campos e os tipos associados, e os seus métodos e tipos associados
- O tipo de cada método é como o tipo dos procedimentos de TINY

$$\frac{T, C, self \vdash e : t \qquad T, C, self \vdash e_{obj} : t_{obj} \qquad C.procura(t_{obj}).metodo(id) = m}{T, C, self \vdash e_{obj}.id(\bar{e}) : m.t_r} \quad t \leq m.t_{param}$$

Contravariância e covariância



Subtipagem em MiniJava

- Determinar uma classe é subtipo de outra é fácil com um algoritmo recursivo
- O topo da hierarquia de classes só é subtipo dela mesma
- Uma classe é subtipo dela mesma
- Uma classe é subtipo da sua superclasse direta
- Se não for nenhum dos casos base acima, uma classe é subtipo de outra classe se a sua superclasse direta for subtipo dessa outra classe

Contextos de tipagem em MiniJava

- Ao contrário de TINY, a verificação de tipos para expressões e comandos MiniJava precisa de um único contexto de tipagem
- Chamadas de métodos obtêm o tipo do método a partir da classe do objeto alvo
- A classe associada a `this` pode ficar no mesmo contexto das outras variáveis e campos
- A verificação precisa de duas passadas: a primeira cria as classes, sem se importar com a consistência dos tipos nos corpos dos métodos, e a segunda verifica os corpos dos métodos