Compiladores
(IF688)

Leopoldo Teixeira
lmt@cin.ufpe.br | @leopoldomt

Sistema de Tipos

O que é um tipo?

Tipos

- Um tipo é uma categoria de elementos de programação
- Identifica um dentre vários tipos de dados, como inteiros, strings, booleanos
- Disciplina o uso, pois determina possíveis valores, operações, etc.
- Operação não suportada em um determinado tipo constitui um erro de tipo

Tipos

```
float salary;

void turn (float degrees)
          throws Exception {…}

Figure fig = new Circle(x,y,r);
```

Propósitos de Sistemas de Tipos

Sistema de Tipos

- Coleção de regras que limitam como programas podem ser escritos
- Visa reduzir a quantidade de erros
- Checagem de tipos verifica que as regras estão sendo respeitadas
- Strong - sistemas que nunca permitem erros de tipo
- Weak - podem permitir erros de tipo em runtime

Guerras

- Debate **eterno** sobre o sistema **certo**
- Dinâmico é mais fácil/rápido de desenvolver protótipos vs. Estático tem menos bugs
- Strong é mais robusto vs. Weak é mais rápido
- Não é o foco da disciplina!

Type-checking

- Estaticamente
  - Analisar o programa em tempo de compilação para verificar ausência de erros de tipo
  - Evitar que coisas ruins aconteçam em tempo de execução
- Dinamicamente
  - Checar operações em tempo de execução
  - Geralmente mais precisa que estáica
- Nenhuma
  - #caos

Sistema de Tipos Simples

$P \rightarrow D \; ; \; E$

$D \rightarrow D \; ; \; D \mid \textbf{id} : T$

$T \rightarrow \textbf{char} \mid \textbf{integer} \mid \textbf{array} \; [\; \textbf{num} \;] \; \textbf{of} \; T \mid \hat{} T$

$E \rightarrow \textbf{literal} \mid \textbf{num} \mid \textbf{id} \mid E \; \textbf{mod} \; E \mid E \; [\; E \;] \mid E \hat{}$

Sistema de Tipos Simples

$P \rightarrow D \; ; \; E$

$D \rightarrow D \; ; \; D \mid \textbf{id} : T$

$T \rightarrow \textbf{char} \mid \textbf{integer} \mid \textbf{array} \; [\; \textbf{num} \;] \; \textbf{of} \; T \mid \hat{} T$

$E \rightarrow \textbf{literal} \mid \textbf{num} \mid \textbf{id} \mid E \; \textbf{mod} \; E \mid E \; [\; E \;] \mid E\hat{}$

```
k: integer;
k mod 1985
```

Produção	Regra Semântica
$P \rightarrow D \; ; \; E$	
$D \rightarrow D \; ; \; D$	
$D \rightarrow \textbf{id} : T$	{ addtype(**id**.entry, T.type) }
$T \rightarrow \textbf{char}$	{ T.type = **char** }
$T \rightarrow \textbf{integer}$	{ T.type = **integer** }
$T \rightarrow \textbf{array} [\textbf{num}] \textbf{ of } T_1$	{T.type = **array**(1..**num**.val, T_1.type)}
$T \rightarrow \hat{}T_1$	{T.type = **pointer**(T_1.type)}
$E \rightarrow \textbf{literal}$	{E.type = **char**}
$E \rightarrow \textbf{num}$	{E.type = **integer**}
$E \rightarrow \textbf{id}$	{E.type = **lookup**(**id**.entry)}

Produção	Regra Semântica
$E \rightarrow E_1$ **mod** E_2	{ E.type = **if** (E_1.type == **integer** ∧ E_2.type == **integer**) **integer** **else type_error** }
$E \rightarrow E_1$ [E_2]	{ E.type = **if** (E_2.type == **integer** ∧ E_1.type == **array(s,t)**) **t** **else type_error** }
$E \rightarrow E_1$^	{ E.type = **if** (E_1.type == **pointer(t)**) **t** **else type_error** }

Adicione Regras para

$T \rightarrow$ **boolean**

$S \rightarrow$ **id** = E

$S \rightarrow$ **if** E **then** S

$S \rightarrow$ **while** E **do** S

Produção	Regra Semântica
T → **boolean**	{ T.type = **boolean** }
S → **id** = E	{ **if** (**lookup**(**id**.entry) != E.type) **type_error** }
S → **if** E **then** S_1	{**if** (E.type != **boolean**) **type_error** }
S → **while** E **do** S_1	{if (E.type != **boolean**) **type_error** }

Adicione Regras para

Considere agora definições e chamadas de métodos contendo apenas um argumento, como descrito abaixo

$$T \rightarrow T \mapsto T$$

$$E \rightarrow E\,(\,E\,)$$

Produção	Regra Semântica
$T \rightarrow T_1 \mapsto T_2$	$\{T.type = T_1.type \mapsto T_2.type\}$
$E \rightarrow E_1 \, (\, E_2 \,)$	$\{ E.type = \textbf{if} \, (E_2.type == \textbf{s} \wedge E_1.type == \textbf{s} \mapsto \textbf{t}) \, \textbf{t} \, \textbf{else type\_error} \}$

Tabelas de Símbolos

Tabelas de Símbolos

- Análise semântica é caracterizada pela manutenção de tabelas de símbolos
  - também chamadas de *environments*
- Estas tabelas geralmente fazem o mapeamento de identificadores com o seu significado
- Ao processar declarações de tipos, variáveis, e funções, associamos os identificadores com o seu significado na tabela
- Ao processar usos de tais identificadores, fazemos o lookup na tabela

Variáveis e Escopo

- Já discutimos escopo, cada variável vai ter uma visibilidade definida
- Em MiniJava, para um método m(), todos os parâmetros formais e variáveis locais em m são visíveis até o fim de m.
- Ao encerrar o método, os bindings locais para aquele identificador são descartados

Environments

- Um environment pode ser descrito como um conjunto de bindings (associações) denotados pela seta ↦
- Por exemplo, o environment σ_0 consiste dos bindings `{g ↦ string, a ↦ int}`

Exemplo Java

```
class C {
   int a; int b; int c;
   public void m() {
      System.out.println(a+c);
      int j = a+b;
      String a = "hello";
      System.out.println(a);
      System.out.println(b);
      System.out.println(j);
   }
}
```

Adaptado de Andrew Appel
Modern Compiler Implementation in Java

```
class C {
   int a; int b; int c;
   public void m() {
      System.out.println(a+c);
      int j = a+b;
      String a = "hello";
      System.out.println(a);
      System.out.println(b);
      System.out.println(j);
   }
}
```

σ_0

$\sigma_1 = \sigma_0 + \{\texttt{a} \mapsto \texttt{int}, \texttt{b} \mapsto \texttt{int}, \texttt{c} \mapsto \texttt{int}\}$

$\sigma_2 = \sigma_1 + \{\texttt{j} \mapsto \texttt{int}\}$

$\sigma_3 = \sigma_2 + \{\texttt{a} \mapsto \texttt{String}\}$ ⟵ $\sigma_a + \sigma_b \neq \sigma_b + \sigma_a$

Adaptado de Andrew Appel
Modern Compiler Implementation in Java

Implementação

- Funcional: mantemos environments enquanto criamos os novos
  - precisa de mais memória para armazenar
  - geralmente implementados com BSTs
- Imperativo: modificamos environment para produzir novo
  - é necessário manter uma pilha de modificações para desfazê-las
  - geralmente implementados com hash tables

```
class Bucket {String key; Object binding; Bucket next;
        Bucket(String k, Object b, Bucket n) {key=k; binding=b; next=n;}
}

class HashT {
   final int SIZE = 256;
   Bucket table[] = new Bucket[SIZE];

   private int hash(String s) {
        int h=0;
        for(int i=0; i<s.length(); i++)
           h=h*65599+s.charAt(i);
        return h;
   }

   void insert(String s, Binding b) {
        int index=hash(s)%SIZE
        table[index]=new Bucket(s,b,table[index]);
   }

   Object lookup(String s) {
        int index=hash(s)%SIZE
        for (Binding b = table[index]; b!=null; b=b.next)
          if (s.equals(b.key)) return b.binding;
        return null;
   }

   void pop(String s) {
        int index=hash(s)%SIZE
        table[index]=table[index].next;
   }
}
```

Adaptado de Andrew Appel
Modern Compiler Implementation in Java

Generalizando…

- Evita comparações de strings
- Compara símbolos
- Diferentes noções de bindings, portanto put associa símbolos com Object

```
package Symbol;

public class Symbol {
  public String toString();
  public static Symbol symbol(String s);
}

public class Table {
  public Table();
  public void put(Symbol key, Object value);
  public Object get(Symbol key);
  public void beginScope();
  public void endScope();
  public java.util.Enumeration keys();
}
```

Adaptado de Andrew Appel
Modern Compiler Implementation in Java

o que é necessário coletar
para fazer type-checking
de um programa MiniJava?

Bindings em MiniJava

- A tabela de símbolos deve ser preenchida com todas as informações de tipos
  - cada nome de variável e parâmetro formal deve ser associado (bound) ao seu tipo
  - cada nome de método deve ser associado com seus parâmetros, tipo de retorno, e variáveis locais
  - cada nome de classe deve ser associado com seus atributos e declarações de métodos

Então, como fazer o type-checking?

Type-checking MiniJava

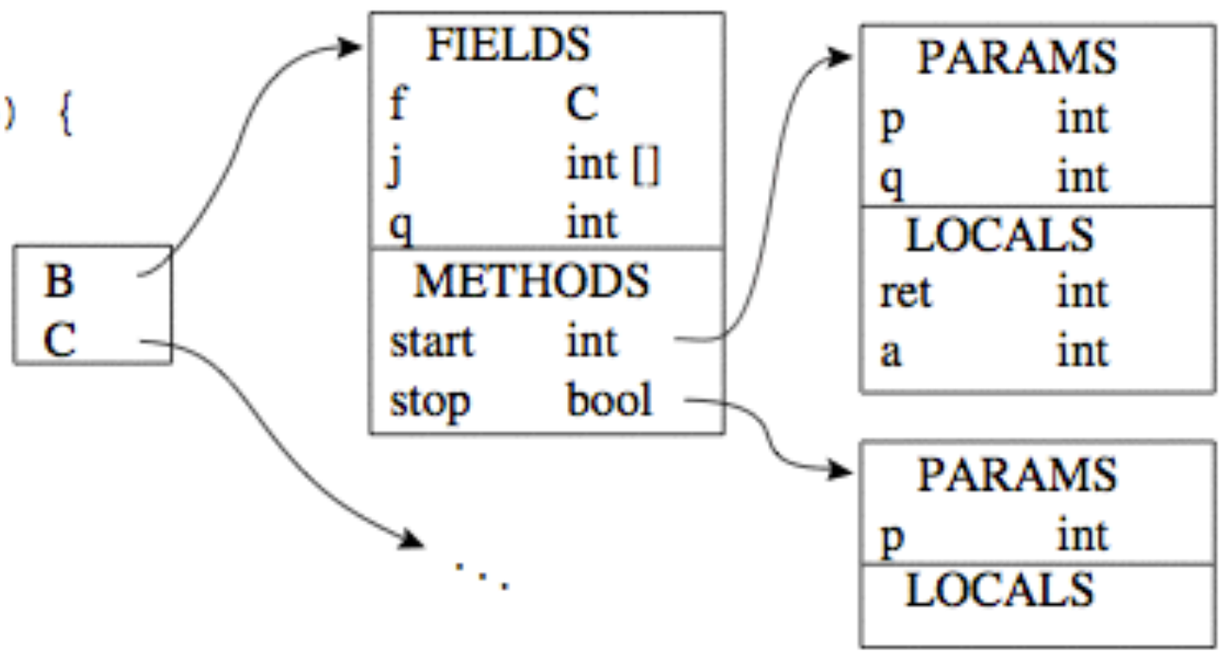
- (i) Construir a tabela de símbolos
- (ii) Fazer a checagem de tipos das instruções e expressões

- Durante a segunda fase, a tabela de símbolos é utilizada para consultar os tipos de identificadores

- Importante termos duas fases, para considerar recursão mútua

```
class B {
    C f;  int [] j;   int q;
    public int start(int p, int q) {
         int ret;   int a;
         /* ... */
         return ret;
    }
    public boolean stop(int p) {
              /* ... */
         retun false;
    }
}

class C {
   /* ... */
}
```



Adaptado de Andrew Appel
Modern Compiler Implementation in Java

```
class ErrorMsg {
  boolean anyErrors;
  void complain(String msg) {
     anyErrors = true;
     System.out.println(msg);
  }
}

// Type t;
// Identifier i;
public void visit(VarDecl n) {

  Type t =  n.t.accept(this);
  String id =  n.i.toString();

  if (currMethod == null) {
      if (!currClass.addVar(id,t))
         error.complain(id + "is already defined in " + currClass.getId());
  } else if (!currMethod.addVar(id,t))
         error.complain(id + "is already defined in "
              + currClass.getId() + "." + currMethod.getId());
}
```

Adaptado de Andrew Appel
Modern Compiler Implementation in Java

Type-checking MiniJava

- Em MiniJava, expressões de adição forçam que ambos os operandos sejam inteiros
  - em muitas linguagens esta operação é *overloaded*: o operador serve tanto para inteiros como reais, e o compilador converte implicitamente entre estes tipos
- Para atribuições, checar compatibilidade de tipos do lado esquerdo e lado direito
  - ao permitir herança, requisito é um pouco relaxado
- Chamadas de métodos?

```
// Exp e1,e2;
public Type visit(Plus n) {
  if (! (n.e1.accept(this) instanceof IntegerType) )
     error.complain("Left side of LessThan must be of type integer");
  if (! (n.e2.accept(this) instanceof IntegerType) )
     error.complain("Right side of LessThan must be of type integer");
  return new IntegerType();
}
```

Adaptado de Andrew Appel
Modern Compiler Implementation in Java