

Compiladores

Verificação de tipos

Resumo da última aula

- Análise semântica
- Implementação:
 - Esquemas S-atribuídos:
 - Mecanismo bottom-up direto
 - Esquemas L-atribuídos:
 - Mecanismo top-down:
 - Necessita gramática não recursiva
 - Mecanismo bottom-up:
 - Deve se introduzir não-terminais artificiais
- Cálculo de árvores de sintaxe abstrata

Análise Semântica e checagem de tipos

- Introdução: o que são tipos, para que servem...?
 - Representação de tipos
- Exemplo de verificação de tipos:
 - Uma linguagem simples
 - Declarações, expressões, instruções, funções...
 - Uso de regras semânticas para verificar o tipo de expressões
 - Uso de regras semânticas para verificar o tipo de comandos
 - Uso de regras semânticas para verificar o tipo de funções

Tipos

- Definições:
 - “Um tipo é uma coleção de valores computáveis que compartilham alguma propriedade estrutural” (Mitchell)
 - “Uma coleção de valores que um fragmento de programa pode assumir durante execução” (Cardelli)
- Porque ter tipos?
 1. Em um sistema tipado (conjunto de regras) é possível checar em pré-processamento para evitar erros de execução
 2. Estrutura do programa & documentação
 3. Manutenção & engenharia de software
 4. Otimizações

Regras semânticas

- Exemplos:
 - Se ambos operandos de operações aritméticas são inteiros, então o resultado é inteiro
 - O resultado de um operador unário & é um ponteiro para o objeto descrito pelo operando
- Denotar o tipo de uma construção de linguagem:
 - Expressão de Tipo:
 - tipo básico
 - tipo estruturado:
 - formado ou formado através da aplicação de um operador de construção de tipos

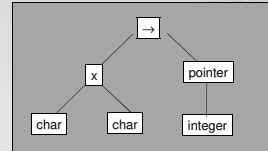
Expressões de tipos

1. Um tipo básico é um expressão de tipo.
 - Ex: boolean, char, integer, real, etc. Um tipo especial, type_error sinaliza um erro durante checagem
2. Se tipos de expressões podem ter nomes, um tipo de nome é uma expressão de tipo.
 - Ex. Registros/struct
4. Construtores de tipos aplicados a tipos de expressões são expressões de tipos
6. Expressões de Tipos podem conter variáveis cujos valores são expressões de tipos

Construtores de tipos

1. Arranjos:
 - Se T é um expressão de tipo, então array (I, T) é uma expressão de tipo.
 - Ex: var A: array[1..10] of integer;
2. Produtos:
 - Se T1 e T2 são expressões de tipos, então o produto cartesiano T1 x T2 é uma expressão de tipo.
3. Registros:
 - Um registro é formado por campos com nomes
4. Ponteiros:
 - Se T é uma expressão de tipo, então ponteiro(T) representa o tipo relativo ao ponteiro de um objeto de tipo T
5. Funções:
 - Mapeamento de um tipo de domínio D em um intervalo de domínio R.
Ex: function f(a, b: char) : pointer to integer
char x char -> pointer(integer)

Representação Gráfica



function f(a, b: char) : pointer to integer

Erros de tipos: Exemplos

```
#include <stdlib.h>
/* double atof(const char *); */
```

```
int main(int argc, char *argv[]) {
    printf("%g\n", atof(argv[1]));
}
```

- Se o include não for feito, imprime lixo!
- No C assume parâmetros "inteiros"

Tipos de erros: test1.c

```
#include <stdio.h>
#include <stdlib.h>
float a, b;
void readDouble(double *p, char *s) {
    *p = atof(s);
}
int main(int argc, char *argv[]) {
    readDouble( (double*)&a, argv[1]);
    readDouble( (double*)&b, argv[2]);
    printf("%g %g\n", a, b);
    return(0);
}
```

Um área double possui 8
Um área float possui 4 bytes

Tipos de erros: test2.c

```
#include <stdlib.h>
#include <stdio.h>
float a, b;
void readDouble(double *p, char *s) {
    *p = atof(s);
}
int main(int argc, char *argv[]) {
    double fa, fb;
    readDouble( &fa, argv[1]);
    readDouble( &fb, argv[2]);
    a = fa; b = fb;
    printf("%g %g\n", a, b);
    return 0;
}
```

Compatibilidade entre tipos

Propriedades de Sistemas Tipados

- Verificar se é decidível
 - Existe um algoritmo que encontra erros
- Transparente
 - Programador deve saber porque o programa não compila ou produz erro

Linguagens Fortemente Tipadas

- Ou fortemente verificadas (Cardelli)
- Nenhum erro de tipo passa despercebido
 - Corretude
- Exemplos:
 - ML, ADA, Pascal (maior parte), Java
- Verificação é não trivial:
 - Linguagens fortemente tipadas
 - Linguagens fracamente tipadas

13

Linguagens Fracamente Tipadas

- Ou fracamente checadas, ou não tipo-seguras
- Erros de tipos podem ocorrer:
 - C/C++: usando conversão, aritmética envolvendo ponteiros, passando final de um array etc
- Balanço:
 - Linguagens fracamente checadas precisam de alguma forma de "garbage collection" (podem impactar o tempo de execução)
 - Mas são mais fáceis de escrever programas (vantagem sobre aspecto de tempo de desenvolvimento)

14

Checagem compilação vs. execução

- Checagem dinâmica x estática
- Tempo de compilação
 - Antes da execução ("para todas as entradas")
 - Compilador maior, restringe flexibilidade, menos expressiva
- Tempo de execução
 - Durante a execução ("com uma dada entrada")
 - Pode ser muito tarde!
 - Cara, mas mais flexível

15

Checagem dinâmica

- `table: array[0..255] of integer;`
- `i: integer;`
- Calcular `table[i]`
 - Compilador não pode garantir que `i` vai estar na faixa de valores do array em tempo de compilação

16

Exemplo de checagem de Tipos

- Uma linguagem de programação simples:
 - Variáveis são definidas antes de serem usadas
 - Semelhante a Pascal
- ```
N : integer;
N mod 1999
```
- $P \rightarrow D ; E$
  - $D \rightarrow D ; D \mid id : T$
  - $T \rightarrow char \mid integer \mid array [ num ] \text{ of } T \mid \uparrow T$
  - $E \rightarrow literal \mid num \mid id \mid E \text{ mod } E \mid E [ E ] \mid E \uparrow$

17

## Regras semânticas

- $P \rightarrow D ; E$
- $D \rightarrow D ; D$
- $D \rightarrow id : T$
- $T \rightarrow char \quad \{ T.type := char \}$
- $T \rightarrow integer \quad \{ T.type := integer \}$
- $T \rightarrow \uparrow T_1$
- $T \rightarrow array [ num ] \text{ of } T_1$

18

## Regras semânticas

- $P \rightarrow D ; E$
- $D \rightarrow D ; D$
- $D \rightarrow id : T$  { addtype(id.lexema, T.type) }
- $T \rightarrow char$  { T.type := char }
- $T \rightarrow integer$  { T.type := integer }
- $T \rightarrow \uparrow T_1$
- $T \rightarrow array [num] of T_1$

19

## Regras semânticas

- $P \rightarrow D ; E$
- $D \rightarrow D ; D$
- $D \rightarrow id : T$  { addtype(id.lexema, T.type) }
- $T \rightarrow char$  { T.type := char }
- $T \rightarrow integer$  { T.type := integer }
- $T \rightarrow \uparrow T_1$  { T.type := pointer(T<sub>1</sub>.type) }
- $T \rightarrow array [num] of T_1$

20

## Regras semânticas

- $P \rightarrow D ; E$
- $D \rightarrow D ; D$
- $D \rightarrow id : T$  { addtype(id.lexema, T.type) }
- $T \rightarrow char$  { T.type := char }
- $T \rightarrow integer$  { T.type := integer }
- $T \rightarrow \uparrow T_1$  { T.type := pointer(T<sub>1</sub>.type) }
- $T \rightarrow array [num] of T_1$  { T.type := array(1..num.val, T<sub>1</sub>.tipo) }

21

## Verificação de Tipos de Expressões

- $E \rightarrow literal$
  - $E \rightarrow num$
  - $E \rightarrow id$
  - $E \rightarrow E1 mod E2$
  - $E \rightarrow E1 [E2]$
  - $E \rightarrow E \uparrow$
- Como avaliar tipos de expressões ?

22

## Verificação de Tipos de Expressões

- $E \rightarrow literal$  { E.type := char }
- $E \rightarrow num$  { E.type := integer }
- $E \rightarrow id$
- $E \rightarrow E1 mod E2$
- $E \rightarrow E1 [E2]$
- $E \rightarrow E \uparrow$

1. As constantes literal e num possuem tipos caractere e inteiro

## Verificação de Tipos de Expressões

- $E \rightarrow literal$  { E.type := char }
- $E \rightarrow num$  { E.type := integer }
- $E \rightarrow id$  { E.type := lookup(id.lexema) }
- $E \rightarrow E1 mod E2$
- $E \rightarrow E1 [E2]$
- $E \rightarrow E \uparrow$

2. O tipo associado a um identificador está definido na tabela de símbolos

### Verificação de Tipos de Expressões

- E -> literal { E.type := char }
- E -> num { E.type := integer }
- E -> id { E.type := lookup(id.lexema) }
- E -> E1 mod E2 { if E1.type == integer and E2.type == integer then E.type = integer else E.type = type\_error }
- E -> E1 [E2]
- E -> E ↑

3. O operador mod somente é aplicado a duas sub-expressões inteiras, retornando um valor inteiro

### Verificação de Tipos de Expressões

- E -> literal { E.type := char }
- E -> num { E.type := integer }
- E -> id { E.type := lookup(id.lexema) }
- E -> E1 mod E2 { if E1.type == integer and E2.type == integer then E.type = integer else E.type = type\_error }
- E -> E1 [E2] { if E2.type == integer and E1.type == array(s, t) then E.type := t else E.type := type\_error }
- E -> E ↑

4. Em uma referência a array E1[E2], a expressão E2 precisa ter tipo inteiro, caso em que o resultado é o elemento de tipo t obtido a partir do tipo array(s, t)

### Verificação de Tipos de Expressões

- E -> literal { E.type := char }
- E -> num { E.type := integer }
- E -> id { E.type := lookup(id.lexema) }
- E -> E1 mod E2 { if E1.type == integer and E2.type == integer then E.type = integer else E.type = type\_error }
- E -> E1 [E2] { if E2.type == integer and E1.type == array(s, t) then E.type := t else E.type := type\_error }
- E -> E ↑ { if E1.type == pointer(t) then E.type := t else E.type := type\_error }

5. O operador postfixo E↑ refere-se ao tipo t do objeto apontado pelo apontador E

### Checagem de comandos

- P -> D ; S
- S -> id := E
- S -> if E then S1
- S -> while E do S1
- S -> S1 ; S2

1. A comandos que não produzem valores, associar um tipo vazio (void)
2. Caso erros aconteçam, associar um tipo-erro (type\_error)
3. Verificar se o lado esquerdo e direito da atribuição possuem mesmo tipo
4. Expressões de comandos condicionais precisam ter tipo booleano

### Checagem de comandos

- P -> D ; S
- S -> id := E { if id.type == E.type then S.type = void else S.type = type\_error }
- S -> if E then S1
- S -> while E do S1
- S -> S1 ; S2

1. A comandos que não produzem valores, associar um tipo vazio (void)
2. Caso erros aconteçam, associar um tipo-erro (type\_error)
3. Verificar se o lado esquerdo e direito da atribuição possuem mesmo tipo

### Checagem de comandos

- P -> D ; S
- S -> id := E { if id.type == E.type then S.type := void else S.type := type\_error }
- S -> if E then S1 { if E.type == boolean then S.type := S1.type else S.type := type\_error }
- S -> while E do S1 { if E.type == boolean then S.type := S1.type else S.type := type\_error }
- S -> S1 ; S2

4. Expressões de comandos condicionais precisam ter tipo booleano

## Checagem de comandos

- $P \rightarrow D ; S$
- $S \rightarrow id := E$      { if  $id.type == E.type$   
                          then  $S.type := void$   
                          else  $S.type := type\_error$ }
- $S \rightarrow \text{if } E \text{ then } S1$  { if  $E.type == boolean$   
                          then  $S.type := S1.type$   
                          else  $S.type := type\_error$ }
- $S \rightarrow \text{while } E \text{ do } S1$  { if  $E.type == boolean$   
                          then  $S.type := S1.type$   
                          else  $S.type := type\_error$ }
- $S \rightarrow S1 ; S2$         { if  $S1.type = void$  and  $S2.type = void$   
                          then  $S.type := void$   
                          else  $S.type := type\_error$  }

1. A comandos que não produzem valores, associam um tipo vazio (void)

## Checagem de Funções

$E \rightarrow E (E)$

- Permitir a definição de tipos de funções:
  - $T \rightarrow T_1 \rightarrow T_2$
- Revisitando:
  - $E \rightarrow E_i (E_i)$
- Exemplo:
  - $sort: (array(int, T) X (TXT \rightarrow boolean) \rightarrow void)$
  - $function\ sort( A:array(int, T);$   
                   $function\ menor(T, T): boolean ) : void$

## Checagem de Funções

$E \rightarrow E (E)$

- Permitir a definição de tipos de funções:

- $T \rightarrow T1 \rightarrow T2$  {  $T.type := T1.type \rightarrow T2.type$ }

– Revisitando:

- $E \rightarrow E_i (E_i)$  { if  $E2.type == s$  and  $E1.type == s \rightarrow t$   
  then  $E.type := t$   
  else  $E.type := type\_error$  }

## O teste '==' entre tipos

- Como verificar se dois tipos são iguais ?

- Equivalência de tipos
  - Equivalência estrutural
  - Equivalência de nomes
- Compatibilidade de tipos
  - Conversão de tipos
  - Subtipos

## Equivalência por Nome

- Dois tipos são os mesmos se eles possuem o mesmo nome

```
type binario = record
 val: int;
 esq: ↑ binario;
 dir: ↑ binario;
end;
type no = ↑ binario;
type folha = ↑ binario;
var
 first: no;
 last: no;
 p: ↑ binario;
 q: ↑ binario;
 leaf: folha;
```

## Equivalência Estrutural

- Dois tipos são iguais se eles possuem a mesma estrutura:
  - possuem o mesmo tipo básico
  - formados pela aplicação do mesmo construtor a tipos estruturalmente equivalentes

```
struct IOBuffer {
 int n;
 struct buffer {
 char bytes[256];
 buffer *next;
 } b[6];
} io[12];

struct NetworkPacket {
 int len;
 struct packet {
 char content[256];
 packet *next;
 } p[6];
} np[12];
```

## Como reconhecer a equivalência estrutural?

```
function sequiv(s, t) : boolean
begin
 if (s e t são o mesmo tipo básico) return true
 else if s = array(s1,s2) and t = array(t1,t2) then
 return sequiv(s1,t1) and sequiv(s2,t2)
 else if s = s1 x s2 and t = t1 x t2 then
 return sequiv(s1,t1) and sequiv(s2,t2)
 else if s = pointer(s1) and t = pointer(t1) then
 return sequiv(s1,t1)
 else if s = s1 -> s2 and t = t1 -> t2 then
 return sequiv(s1,t1) and sequiv(s2,t2)
 else return false
end
```

37

## Exercício

### Checagem de Tipos:

Estruturas de dados como listas encadeadas são usualmente definidas de forma recursiva. Para a seguinte definição em Pascal, explique como é feita a checagem de tipos usando equivalência estrutural e por nome entre as variáveis p e q, citando as dificuldades introduzidas pela existência da recursividade.

```
type link = 1 cell;
cell = record
 info: integer;
 next: link;
end;
var p : link;
 q : 1 cell;
```

38

## Verificação de Tipos

- Conversão de tipos
  - Costuma ser empregada em código intermediário
  - Especifica as conversões necessárias para garantir a atribuição de valores, por exemplo
  - Um mesmo operador pode ser empregado a vários tipos distintos, neste caso, diz-se que o operador está “overloaded”

39

## Conversão de Tipos

- Exemplo de conversão de tipos
  - $x + i$ 
    - x é do tipo real
    - i é do tipo inteiro
  - diferentes tipos possuem representações diferentes no computador e diferentes instruções
  - converter um dos operandos para um mesmo tipo

40

## Conversão de Tipos

- Regras na linguagem definem quais as conversões possíveis
- int a;
- float b;
- a := b;
- b := a;

41

## Conversão de tipos

- Regras na linguagem definem quais as conversões possíveis
- int a;
- float b;
- a := b;
- b := a;

converter para o tipo da variável ao lado esquerdo da atribuição

42

## Conversão de Tipos

- Duas formas
  - **Conversão implícita** (coerção)
    - Realizada automaticamente pelo compilador
  - **Conversão explícita**
    - Realizada pelo programador, indicando a conversão que deve ser aplicada

43

## Conversão Implícita (coerção)

- Definidas em cada linguagem
- Normalmente limitadas a situações onde nenhuma informação é perdida
  - conversão inteiro para real ok
  - conversão real para inteiro inválida
- Em alguns casos perdas são aceitáveis
  - representar número real em número menor de bits
  - conversão de double para float

44

## Conversão de Tipos

- Costuma ser empregada em código intermediário
  - $x \text{ i inttoeal} +$ 
    - converte  $i$  de inteiro para real e em seguida o operador real  $+$  realiza a adição real

45

## Conversão Explícitas

- Informada na linguagem explicitamente
- Similares a aplicações de funções
  - `atoi`, `atof`, `casts` etc em C
  - `ord`, `chr`, etc em Pascal

46

## Conversões Implícitas

| PRODUÇÃO                            | REGRA SEMÂNTICA                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $E \rightarrow \text{num}$          | $E.\text{tipo} := \text{inteiro}$                                                                                                                                                                                                                                                                                                                                                                                                           |
| $E \rightarrow \text{num.num}$      | $E.\text{tipo} := \text{real}$                                                                                                                                                                                                                                                                                                                                                                                                              |
| $E \rightarrow \text{id}$           | $E.\text{tipo} := \text{procurar}(\text{id.lexema})$                                                                                                                                                                                                                                                                                                                                                                                        |
| $E \rightarrow E_1 \text{ op } E_2$ | $E.\text{tipo} :=$ IF $E_1.\text{tipo} = \text{inteiro}$ e $E_2.\text{tipo} = \text{inteiro}$<br>THEN inteiro<br>ELSE IF $E_1.\text{tipo} = \text{real}$ e $E_2.\text{tipo} = \text{real}$<br>THEN real<br>ELSE IF $E_1.\text{tipo} = \text{real}$ e $E_2.\text{tipo} = \text{inteiro}$<br>THEN real<br>ELSE IF $E_1.\text{tipo} = \text{inteiro}$ e $E_2.\text{tipo} = \text{real}$<br>THEN real<br>OBS: Resolvidas em tempo de compilação |

47

## Conversões Implícitas e Desempenho

- Pode melhorar tempo de execução
  - Observação de Bentley [1982] em Pascal
  - Com  $X$  sendo um array de reais:
    - for  $i := 1$  to  $n$  do  $X[i] := 1$ 
      - Leva  $48,4 \times n$  micro-secundos
    - for  $i := 1$  to  $n$  do  $X[i] := 1.0$ 
      - Leva  $5,4 \times n$  micro-secundos
  - Compiladores inteligentes convertem `1` para `1.0` em tempo de compilação

48

## Yacc, Tokens e Atributos decl.y

```
#{
#include <stdio.h>
#include <stdlib.h>
#}
%token NUM, ID, INT
%%
decl : type ID list
 { printf("Success!\n");
 } ;
list : ',' ID list
 | ';'
 ;
type : INT | CHAR | FLOAT
 ;
%%
```

49

## E os atributos?

- Cada símbolo tem um (ou vários) valor associado (atributo)
  - Valor numérico no caso de um número (42)
  - Ponteiro para um string ("Hello, World!")
  - Ponteiro para uma entrada na tabela de símbolos no caso de uma variável
- Quando usando lex junto com o Yacc, coloca-se o valor em `yylval`
  - Código típico lex:  
[0-9]+ {yylval = atoi(yytext); return NUM}
- Dois problemas:
  - Como fazer com mais de um atributos?
  - Como atribuir tipos aos atributos?

50

## A Union no YACC

- Por default, `YYSTYPE` é do tipo `int`
- `YYSTYPE` é o tipo de `yylval`
- Este tipo pode ser redefinido tipos de atributos

```
%union {
 double valdouble;
 int valint;
 char* valstring;
 /* etc ... */
}
```

**Gera:**  
typedef union {  
 double valdouble;  
 int valint;  
 char\* valstring;  
} YYSTYPE;  
extern YYSTYPE yylval;

- Assim, o YACC sabe qual é o tamanho a reservar na pilha.
- Tabelas de símbolos são usadas para guardar informações sobre tipos mais complexos:
  - Nomes, tipos, valores etc
  - Se retorna apenas um ponteiro para a entrada.

51

## Uso dos atributos tipados no Yacc

- No Yacc, os símbolos que estão na pilha podem ser acessados através do `$`:
  - `$$` = símbolo que está sendo reduzido (lado esquerdo de uma regra)
  - `$1`, `$2`, `$3...` = símbolos que aparecem à direita da regra.
  - `$x` tem atributos associados em função do tipo definido.
- Note que podem ser associados tipos aos símbolos: não trivial (!= `int`) um desses tipos:  
`%token<valdouble> DOUBLE`  
`%type <valdouble> dexpr`
  - Assim, a produção yacc:  
`dexp := DOUBLE { $$ = constante($1); }`
    - É automaticamente convertida em:  
`dexp.valdouble = constante(yyvsp[0].valdouble);`

topo da pilha!

52

## Bibliografia

- Livro do dragão, Capítulo 6
- Price & Toscani, Seções 5.2 e 5.3

53