Compiladores (CC3001)
Aula 9: Análise de tipos

Mário Florido

DCC/FCUP

2024

[dcc] FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

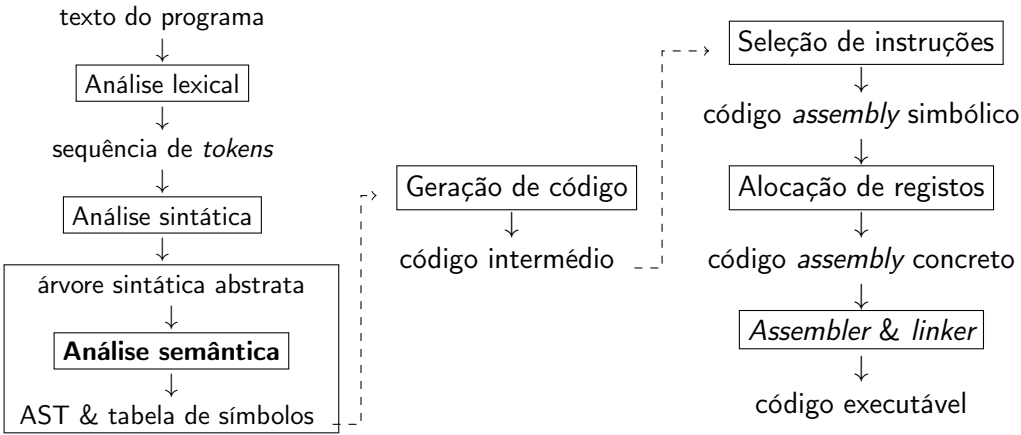
Esta aula

Sistemas de Tipos

Análise de Tipos
- Expressões
- Declarações e comandos
- Funções

Extras

Re-lembrar: fases dum compilador



Esta aula

Sistemas de Tipos

Análise de Tipos
- Expressões
- Declarações e comandos
- Funções

Extras

Um sistema de tipos é um conjunto de regras lógicas da linguagem que todos os programas devem respeitar.

Exemplos de regras:

- `+`, `-`, `*`, `/` só operam sobre *números*
- a condição de `if` deve ser um *boleano*
- numa atribuição `var = expr` a *variável* deve ser declarada com um tipo compatível com o da a *expressão*

Podemos classificar em dois eixos separados:

Estáticos verificação ocorre antes da execução do programa

vs.

Dinâmicos verificação ocorre durante a execução do programa

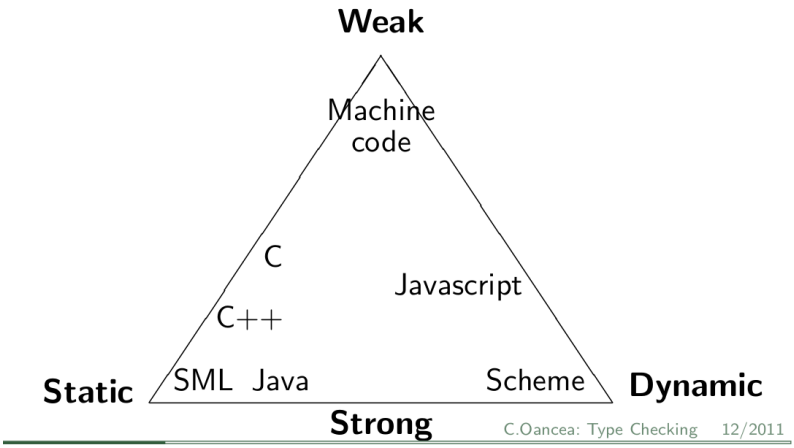
Fortes operações com erros de tipos são proibidas

vs.

Fracos operações com erros de tipos são permitidas

Exemplos:

- Haskell, SML, Java: tipos estáticos e fortes
- Python, Scheme: tipos dinâmicos e fortes
- C: tipos estáticos e fracos
- Estas classificações são graduais e não absolutas
  e.g. o sistema de tipos de Java é mais forte que o de C mas mais fraco que o de Haskell
- A distinção estático/dinâmico deixa de fazer sentido para código máquina



(Imagem: *Introduction to Compiler Design*, Torben Mogensen.)

- Efetuar a análise de tipos durante a compilação (estática):
  - evita verificações durante a execução do programa
  - permite gerar código mais eficiente
- Em qualquer caso: ajudam o programador a detetar erros lógicos nos programas

Esta aula

Sistemas de Tipos

Análise de Tipos
- Expressões
- Declarações e comandos
- Funções

Extras

- A análise de tipos pode ser feita como uma (ou mais) travesia da AST
- Como as ASTs são estruturas recursivas, a travesia será implementada como funções recursivas
- As traversias constroem atributos; exemplos:
  - o tipo de cada sub-expressão;
  - a tabela que associa cada identificador ao seu tipo (ambiente)
- Atributos sintetizados: calculados das *folhas* da AST para a *raiz*
- Atributos herdados: passados da *raiz* da AST para as *folhas*
- Atributos sintetizados numa parte da AST podem ser herdados noutra (e.g. a tabela de símbolos é sintetizada nas declarações e herdada nas expressões)

Esta aula

Sistemas de Tipos

Análise de Tipos
- Expressões
- Declarações e comandos
- Funções

Extras

- Uma linguagem com *variáveis*, *expressões aritméticas* e *lógicas* (comparações)
- Dois tipos: **int** (números inteiros) e **bool** (valores lógicos)

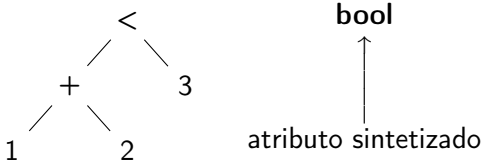
Exemplos de expressões:

```
1+2*3
(1+2)<3
1+x*3          (válida se x for int)
(1+x)<y        (válida se x for int e y for int)
```

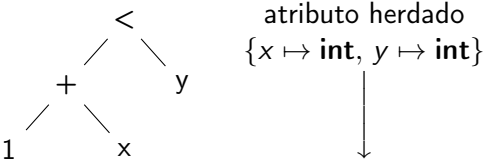
Um erro de tipos (não podemos somar inteiros com valores lógicos):

```
1+(2<3)
```

Determinamos o tipo da expressão a partir dos tipos das sub-expressões (atributo sintetizado).



Passamos uma tabela de símbolos com os tipos da variáveis (atributo herdado).



```
data Expr = Num Int                -- 1, 2, 3
          | Var Ident              -- x, y, z
          | Add Expr Expr          -- e1 + e2
          | LessThan Expr Expr     -- e1 < e2

data Type = TyInt | TyBool

type TypeEnv = Map Ident Type      -- ambiente (tabela de símbolos)

checkExpr :: TypeEnv -> Expr -> Type
-- sintetizar o tipo de uma expressão
-- argumentos: o ambiente (TypeEnv) e a expressão (Expr)
-- resultado: um tipo (ou erro)
-- definição por análise de casos
```

O tipo duma expressão é o resultado da função (atributo sintetizado):

```
checkExp env (Num n) = TyInt

checkExpr env (Add e1 e2)
   = let t1 = checkExpr env e1
         t2 = checkExpr env e2
     in if t1==TyInt && t2==TyInt then TyInt
     else error "type error in +"
```

O ambiente (tabela de símbolos) é um argumento da função (atributo herdado).

```
checkExpr env (Add e1 e2)
   = let t1 = checkExpr env e1
         t2 = checkExpr env e2
     in if t1==TyInt && t2==TyInt then TyInt
        else error "type error in +"
```

O ambiente inicial é definido na chamada de topo (raiz da AST):

```
> checkExpr (Map.fromList [("x",TyInt)])  (Add (Var "x") (Num 1))
TyInt
```

Esta expressão teria um erro de tipo num outro ambiente:

```
> checkExp (Map.fromList [("x", TyBool)] (Add (Var "x") (Num 1)))
type error in +
```

(Ver demonstração.)

```
checkExpr env (Num n) = TyInt
checkExpr env (Var x) = case Map.lookup x env of
    Nothing -> error "undeclared var"
    Just t -> t
checkExpr env (Add e1 e2)
   = let t1 = checkExpr env e1
         t2 = checkExpr env e2
     in if t1==TyInt && t2==TyInt then TyInt
        else error "type error in +"
checkExpr env (LessThan e1 e2)
   = let t1 = checkExpr env e1
         t2 = checkExpr env e2
     in if t1==TyInt && t2==TyInt then TyBool
        else error "type error in <"
```

Esta aula

Sistemas de Tipos

Análise de Tipos

- Expressões
- Declarações e comandos
- Funções

Extras

- Acrescentamos novos casos na sintaxe abstrata
- Vamos acrescentar definições para comandos e declações de variáveis

Declarações e comandos (cont.)


```
data Stm =  Assign Ident Expr          -- x := e
         | IfThenElse Expr Stm Stm     -- if cond then stm1 else stm2
         | IfThen Expr Stm             -- if cond then stm
         | While Expr Stm              -- while cond e
         | Block [Decl] [Stm]          -- { ... }

type Decl = (Ident, Type)
```

```
if x<y then x=y else x=0
```

```
IfThenElse
  (LessThan (Var "x") (Var "y"))
  (Assign "x" (Var "y"))
  (Assign "x" (Num 0))
```

```
{
  int n;
  n = 1;
  while (n < 10)
    n = n+1;
}
```

```
Block
  [("n", TyInt)]
  [ Assign "n" (Num 1)
  , While (LessThan (Var "n") (Num 10))
       (Assign "n" (Add (Var "n") (Num 1)))
  ]
```

Exemplos (cont.)

```
if (x<y) {
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

```
IfThen
  (LessThan (Var "x") (Var "y"))
  (Block
     [("temp",TyInt)]
     [ Assign "temp" (Var "x")
     , Assign "x" (Var "y")
     , Assign "y" (Var "temp")
     ])
```

Regras de tipos para comandos

Atribuição (*var* = *expr*) está bem tipado se a variável for declarada com o mesmo tipo da expressão

Condição (*if (cond) stm1 else stm2*)

1. a condição deve ter tipo **bool**
2. *stm*1 e *stm*2 devem estar bem tipados

Ciclo (*while (cond) stm*)

1. a condição *cond* deve ter tipo **bool**
2. o corpo *stm* deve estar bem tipado

Blocos ({decls; stms})

1. acrescentamos *decls* ao ambiente
2. verificamos que todos os comandos em *stm* estão bem tipados

```
checkStm :: TypeEnv -> Stm -> Bool
-- True: bem tipado
-- False ou erro de execução: erro de tipos
```

```
checkStm env (Assign id expr)
  = case Map.lookup id env of
      Just typ -> if checkExpr env expr == typ
                  then True
                  else error "type error in assign"
      Nothing -> error "undeclared variable"
```

```
checkStm env (IfThenElse cond stm1 stm2)
  = let tycond = checkExpr env cond
        check1 = checkStm env stm1
        check2 = checkStm env stm2
    in if tycond == TyBool && check1 && check2
       then True
       else error "type error in if then else"
```

(O caso *if-then* sem *else* é similar.)

```
checkStm env (While cond stm)
  = let tycond = checkExpr env cond
        check = checkStm env stm
    in if tycond == TyBool && check
       then True
       else error "type error in while"
```

```
checkStm env (Block decls stms)
  = let env' = extendEnv env decls
    in all (checkStm env') stms

-- função auxiliar do prelúdio
-- all :: (a -> Bool) -> [a] -> Bool

-- função auxiliar para extender um ambiente
extendEnv :: TypeEnv -> [Decl] -> TypeEnv
extendEnv env [] = env
extendEnv env ((v,t):rest) = extendEnv (Map.insert v t env) rest
```

Esta aula

Sistemas de Tipos

Análise de Tipos
- Expressões
- Declarações e comandos
- Funções

Extras

Vamos extender a linguagem com declarações e chamada de funções.

```
data Expr = ...
          | FunCall Ident [Expr]
           -- f(e1, e2, ...)
data Stm = ...
          | Return Expr

data FunDef = FunDef Ident [(Ident,Type)] Type Stm
    -- t f(t1 x1,..., tn xn)  stm
```

Exemplo

```
int incr(int x) {
    return x+1;
}
```

```
FunDef "incr"
    [("x",TyInt)] TyInt
    (Return (Add (Var "x") (Num 1)))
```

$$(t_1, t_2, \ldots, t_n) \to t_r$$

- $t_1, t_2, \ldots, t_n$ são os tipos dos argumentos
- t_r é o tipo do resultado

```
data Type = ...
          | TyFun [Type] Type
```

Para verificar uma chamada $f(e_1, \ldots, e_n)$

1. Procurar f no ambiente e obter um tipo $(t_1, \ldots, t_n) \to t_r$
2. Recursivamente sintetizar tipos para $e_1, \ldots, e_n$
3. Se os tipos sintetizados forem iguais a $t_1, \ldots, t_n$ então o tipo do resultado é t_r; caso contrário erro

Nova equação para `checkExpr`:

```
checkExpr env (FunCall f args)
    = case Map.lookup f env of
        Just (TyFun argTypes result) ->
             if map (checkExpr env) args == argTypes then result
             else error "invalid argument types"
        _ -> error "invalid function name"
```

Verificação de valores de retorno

- Para verificar `return` *expr*:
  1. Sintetizamos o tipo de *expr*;
  2. Testamos se é igual o tipo esperado.
- Logo: temos de passar o tipo esperado como *atributo herado*
- Como `return` só pode ocorrer dentro de uma função vamos passar um tipo opcional

```
checkStm :: TypeEnv -> Maybe Type -> Stm -> Bool
checkStm env (Just typ) (Return exp)
    = checkExpr env exp == typ
checkStm env Nothing (Return _)
    = error "return outside of function"
```

(Modificamos as restantes equações para propagar o tipo de retorno.)

Para verificar a definição de uma função:

1. Extendemos o ambiente com os tipos dos argumentos
2. No ambiente extendido: verificamos o corpo
3. Passamos o tipo esperado do resultado como argumento herdado
4. Sintetizamos o ambiente com extendido com o tipo da função

(Neste caso: o ambiente será simultaneamente herdado e sintetizado.)

Verificação de definições (cont.)


```
checkFunDef :: TypeEnv -> FunDef -> TypeEnv
checkFunDef env (FunDef fun decls tyret stm)
  = let tyargs = map snd decls
        env' = extendEnv env ((fun, TyFun tyargs tyret):decls)
    in if checkStm env' (Just tyret) stm then
         extendEnv env [(fun,TyFun tyargs tyret)]
       else error "type error in function definition"
```

- A regra anterior permite usos recursivos da função
- O corpo da função é tipado com o ambiente

  ```
  env' = extendEnv env ((fun,TyFun tyargs tyret):decls)
  ```

- Logo: podemos usar a função *fun* na sua definição (ver exemplo)

Esta aula

Sistemas de Tipos

Análise de Tipos
- Expressões
- Declarações e comandos
- Funções

Extras

Overloading utilizar o *mesmo operador* para operações em tipos diferentes (e.g. + em inteiros e fracionários)

```
1 + 2       -- int
1.0 + 2.5   -- float
```

Coercão conversão implícita ou explícita de tipos

```
int x = ...;
(float)x + 2.5  -- conversão explícita
x + 2.5         -- conversão implícita
```

- A verificação para tipos e operadores pré-definidos é simples (e.g. C, Pascal)
- Consideravelmente mais complexa se o programador pode definir operadores sobre tipos novos (e.g. C++, Haskell,...)

polimórfico (adjetivo): que se apresenta sob diversas formas.

Dicionário Online Priberam

Em programação: a possibilidade de escrever código que opera sobre diferentes tipos.

polimorfismo paramétrico SML/Haskell ou "generics" em Java/C#

```
reverse :: [t] -> [t]              -- em Haskell
void reverse(List<T> list);          // em Java
```

polimorfismo *ad-hoc* *overloading*, interfaces, herança...

(Tópicos abordados em *Fundamentos de Linguagens* e *Tópicos de Programação Funcional Avançada*.)

- Para alguns sistemas é possível inferir os tipos automaticamente em vez de apenas verificar declarações
- As linguagens funcionais fortemente tipadas usam inferência baseada no *algoritmo Damas-Milner* (com extensões)

(Tópicos abordados em *Fundamentos de Linguagens*)