Tipos

Linguagens de Programação
2018.2019

Teresa Gonçalves
tcg@uevora.pt

Departamento de Informática, ECT-UÉ

Sumário

Tipos

Inferência de tipos

Polimorfismo

Declaração de Tipos

Tipos

Tipo

O que é?

Uma coleção de valores computáveis que partilham propriedades estruturais

Exemplos

inteiro

string

int → bool

(int → int) → bool

Para que serve?

Organizar e documentar programas

- Definir tipos distintos para conceitos distintos
  - Objectivo: representar conceitos do domínio
- Indicar utilização para identificadores declarados
  - Objectivo: verificação

Identificar e prevenir erros

- Prevenir cálculos sem significado

Servir de suporte à otimização

- Exemplos
  - Requisitos de memória diferentes para tipos diferentes
  - Aceder à componente dum registo pelo seu “offset”

Verificação de tipos

Garantir que f(x) é uma função

Tempo de execução (verificação dinâmica)

- Verifica que f é função antes de a chamar
- Linguagens
  - Lisp e JavaScript

Tempo de compilação (verificação estática)

- Verifica que f:A→B e que x:A
- Linguagens
  - ML e Haskell

Vantagens e desvantagens

Ambos previnem erros de tipos...

Tempo de execução

- Penaliza velocidade de execução do programa

Tempo de compilação

- Elimina testes em tempo de execução
- Encontra erros antes da execução e dos testes
- Restringe a flexibilidade do programa
  - Lisp: cada elemento de uma lista pode ser de tipo diferente
  - ML: todos os elementos de uma lista são do mesmo tipo

Maioria das LP utilizam uma combinação...

Expressividade

Verificação dinâmica

Algumas execuções podem produzir erros, outras não!

Exemplo: JavaScript

```
function f(x) {return x<10 ? x : x();}
```

Verificação estática

É sempre conservadora!

não é possível decidir em tempo de compilação se um erro de execução vai acontecer!

Exemplo

```
if (funcao_bool_complicada)
then f(5);
else f(15);
```

Segurança de tipos

Linguagem não segura

- C, C++
- “cast” e aritmética de apontadores

Linguagem quase segura

- Algol, Pascal, Ada
- Apontadores pendentes
  - Nenhuma linguagem com libertação de memória explícita é segura!

Linguagem segura

- Lisp, ML, Haskell, SmallTalk, Java
- Verificação estática: ML, Haskell, Java
- Verificação dinâmica: Lisp, SmallTalk

Ferramentas de análise

Linguagem não segura

- Se diz que está correcto, pode não estar
- Se diz que está incorrecto então existe erro!

Linguagem segura

- Se diz que está correcto então está mesmo
- Se diz que não está correcto, pode estar!

Inferência de tipos

Porquê a inferência de tipos?

Sistema de Tipos

- Tem melhorado consistentemente desde o Algol 60
- Importante para a modularidade, segurança, compilação,...

Inferência de Tipos

- Reduz a sobrecarga sintática
- Garante a produção do tipo mais geral
- Inovação importante no desenho de LP
- Exemplo ilustrativo de um algoritmo de análise estática independente do fluxo

Inferência de tipos no ML

Exemplo

```
- fun f(x) = 2 + x;
> val it = fn : int → int
```

Qual o tipo de f?

+ tem 2 tipos possíveis

  `int → int  → int`

  `real → real → real`

2 tem um único tipo: `int`

Isto implica que `+:int → int → int`

Logo, `x:int`

Então `f(x:int)=2+x` tem tipo `int → int`

Algoritmo

Exemplo

– fun f(x) = 2 + x;

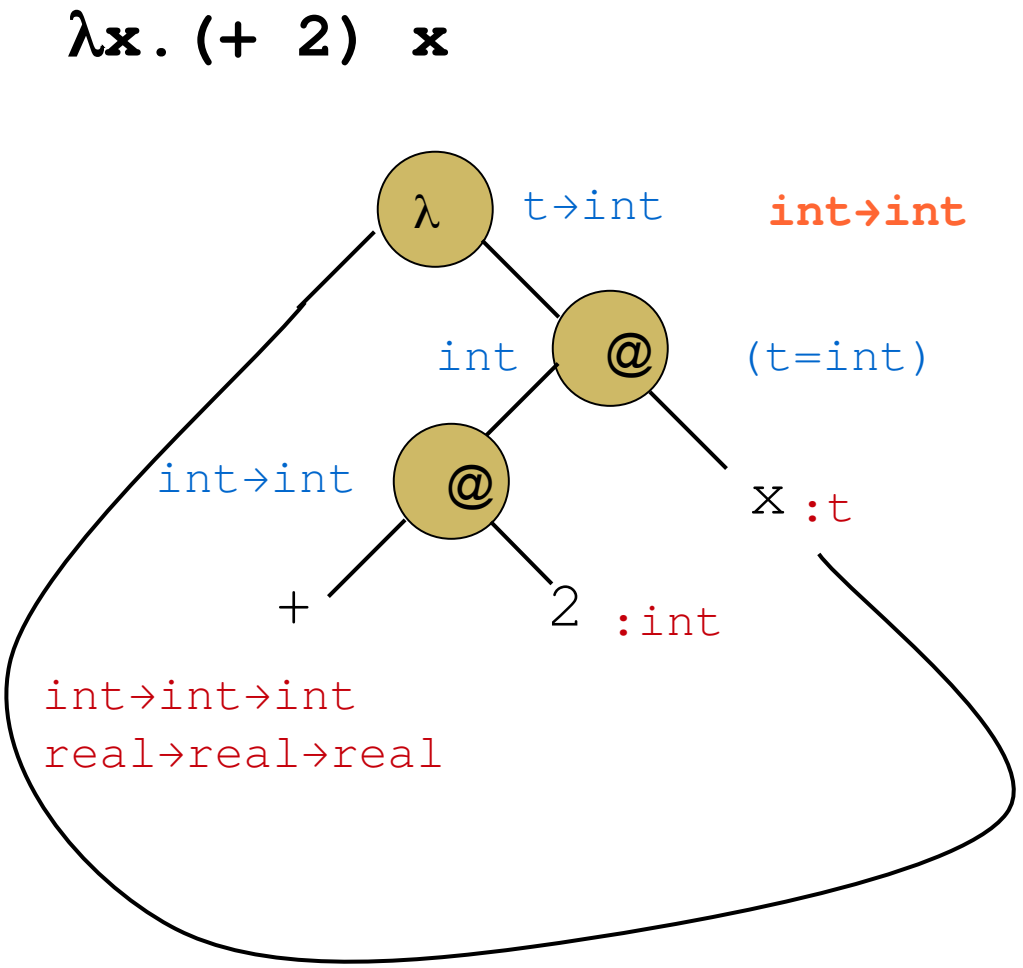
> val f = fn : int → int

Qual o tipo de f?

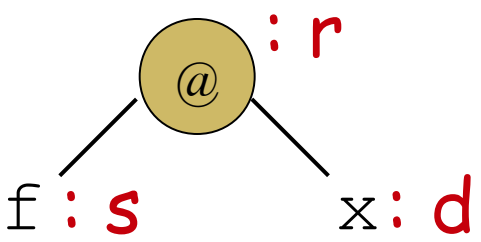
Definir o tipo das folhas

Propagar aos nós internos e gerar restrições

Resolver por substituição



Aplicação



s: domínio → contradomínio

domínio = d

contradomínio = r

Aplicação da função f ao argumento x

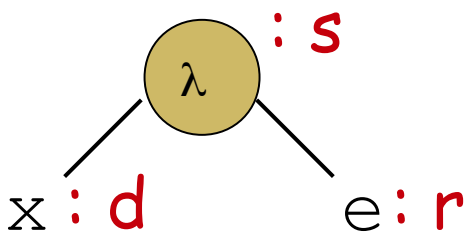
O tipo de f : s tem de ser uma função do tipo domínio → contradomínio

O domínio de f é o tipo do argumento x : d

O contradomínio de f é o tipo do resultado de f : r

Então, tem-se s = d → r.

Abstração



s: domínio → contradomínio

domínio = d

contradomínio = r

Expressão funcional `λx.e`

O tipo da abstração-lambda : s tem de ser uma função do tipo domínio → contradomínio

O domínio é o tipo da variável x : d

O contradomínio é o tipo do corpo da função e : r

Então, tem-se s = d → r

Tipos com variáveis de tipo

Exemplo

```
- fun f(g) = g(2);
> val it=fn : (int → t) → t
```

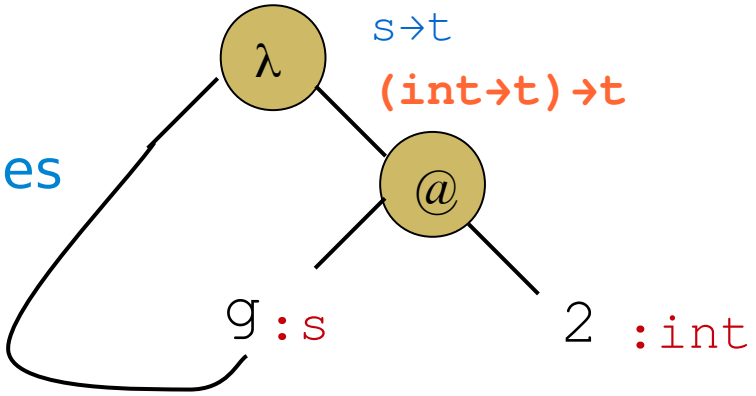
Qual o tipo de f?

Definir o tipo das folhas

Propagar aos nós internos e gerar restrições

Resolver por substituição

λg.g 2



Funções polimórficas

Função

```
- fun f(g) = g(2);
> val f : (int → t) → t
```

Aplicações possíveis

```
- fun add(x) = 2+x;
> val it = fn : int → int
- f(add);
> val it = 4 : int
```

```
- fun isEven(x) = ...;
> val it = fn : int → bool
- f(isEven);
> val it = true : bool
```

Deteção de erros de tipo

Função

```
- fun f(g) = g(2);
> val it = fn : (int → t) → t
```

Utilização incorreta

```
- fun not(x) = if x then false else true;
> val it = fn : bool → bool
```

```
- f(not);
Error: operator and operand don't agree
  operator domain: int -> 'Z
  operand:         bool -> bool
```

Erro de tipos

Impossível ter `bool→bool = int→t`

Outro exemplo de inferência de tipos

Definição de função

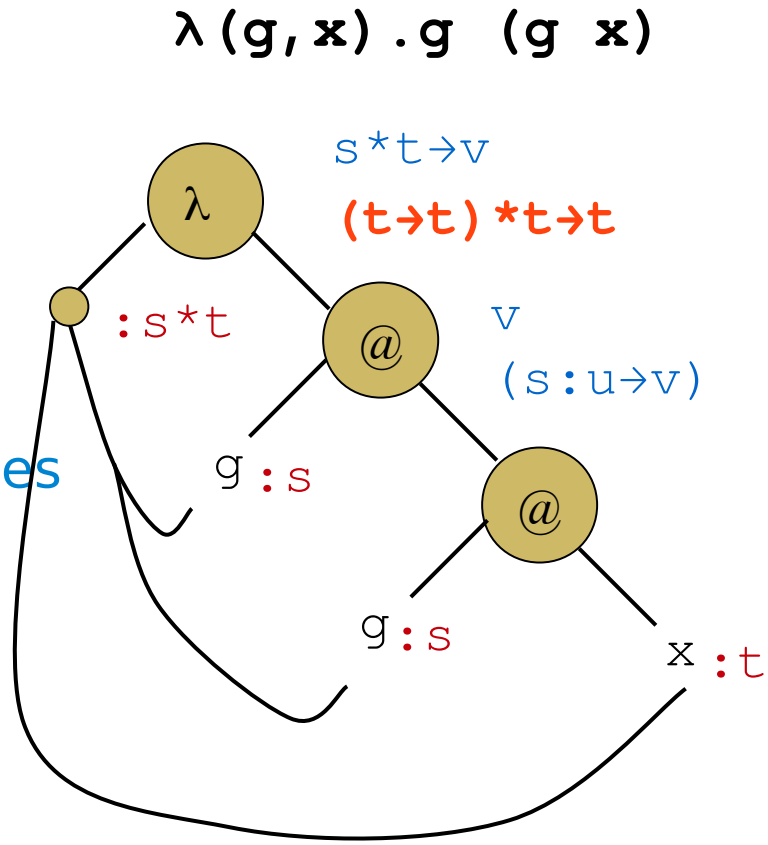
```
- fun f(g,x) = g(g(x));
> val f = fn : (t → t)*t → t
```

Qual o tipo de f?

Definir o tipo das folhas

Propagar aos nós internos e gerar restrições

Resolver por substituição



Tipos polimórficos

Tipo de dados com variável de tipo

```
datatype 'a list = nil
  | cons of 'a * ('a list)

> nil  : 'a list
> cons : 'a * ('a list) → 'a list
```

'a é a sintaxe para “variável de tipo a”

Função polimórfica

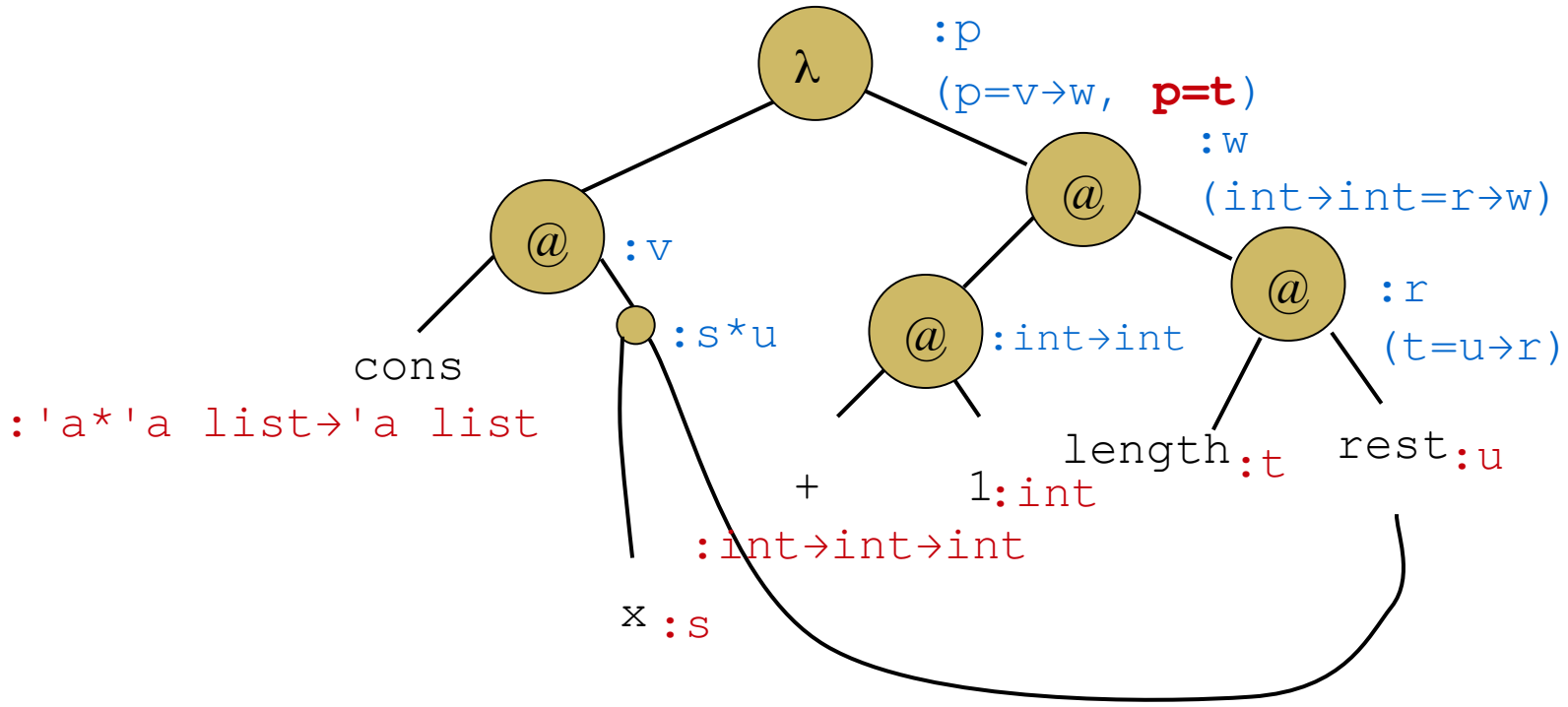
```
- fun length nil = 0
   | length(cons(x,rest))=1+length(rest)

> val length = fn : 'a list → int
```

Função recursiva

Exemplo

```
length(cons(x,rest)) = 1 + length(rest)
```



Restrições

```
p=t
p=v→w
int→int=r→w
t=u→r
'a*'a list→'a list = s*u → v
```

Solução

```
s = 'a
u = v ='a list
r = w = int
p = t = 'a list → int
```

Função com múltiplas cláusulas

Função

```
- fun append(nil,l) = l
  | append(x::xs,l) = x :: append(xs,l)

> val append=fn : 'a list * 'a list → 'a list
```

Qual o tipo de append?

Inferir o tipo de cada cláusula

1ª – append = `fn : 'a list * 'b → 'b`

2ª – append = `fn : 'a list * 'b → 'a list`

Se necessário, combinar tornando os dois tipos iguais

`fn : 'a list * 'a list → 'a list`

Tipo mais geral

A inferência de tipos garante a produção do tipo mais geral

```
fun map(f,nil) = nil
  | map(f, x::xs) = (f x) ::  map(f,xs)
> val map=fn : ('a → 'b) * 'a list → 'b list
```

A função pode ter outros tipos, menos gerais

```
map : ('a  → int) * 'a list   →  int list
map : (bool → 'b)  * bool list →  'b list
map : (char → int) * char list →  int list
```

Os tipos menos gerais são instâncias do tipo mais geral, também designado tipo principal

Inferência e erros no programa

Função

```
fun reverse (nil) = nil
  | reverse (x::xs) = reverse(xs);
```

Tipo principal

```
> val reverse = fn : 'a list → 'b list
```

O que significa?

Inverter uma lista não muda o seu tipo,

Tem de existir um **erro** na definição da função reverse!

Algoritmo de inferência de tipos

Algoritmo Hindley – Milner

Hindley, 1969 (extensão de Curry e Feys, 1958)

Milner, 1978 (algoritmo W)

Complexidade

Quando foi desenvolvido era desconhecida

em 1989 Kanellakis, Mairson e Mitchell [1] provaram ser um problema de complexidade exponencial...

...no entanto, na prática é linear

[1] Unification and ML Type Reconstruction. Computational Logic: Essays in Honor of Alan Robinson, ed. J.-L. Lassez and G.D. Plotkin, MIT Press, 1991, pages 444-478

Pontos chave

Calcula o tipo de expressões

- não requer a declaração do tipo das variáveis
- determina o tipo mais geral por resolução das restrições
- conduz ao **polimorfismo**

Pode originar melhor deteção de erros que a verificação de tipos

- O tipo pode indicar um erro de programação mesmo sem erro de tipos!

Custos

É mais difícil identificar a linha do programa que causa o erro

O ML requer sintaxe diferente para inteiros e reais

A implementação natural requer tamanhos de representação uniformes

Complicações relacionadas com a atribuição demoraram anos a resolver!

Polimorfismo

Polimorfismo

“ter múltiplas formas”

O que significa?

Construções que podem ser utilizadas com múltiplos tipos

Exemplo

Função para calcular o tamanho de uma lista

```
lenght: 'a list → int
```

Tipos de polimorfismo

Subtipo

Paramétrico

Ad-hoc (sobrecarga)

Polimorfismo de subtipo

O que é?

A relação de subtipo entre tipos permite que uma expressão possa ter vários tipos possíveis

Relacionado com programação orientada-a-objectos!

Polimorfismo paramétrico

O que é?

O tipo associado a uma função ou expressão é dado por uma expressão de tipos que contém variáveis de tipo

Característica

A função pode ter um número infinito de tipos

Exemplo

Função para ordenar listas

`sort: ('a*'a→bool)*'a list →'a list`

Pode ser aplicada a qualquer par `(função*lista)`;

- a função deve ter o tipo `'a*'a→bool`
- os elementos da lista são do tipo `'a`

A função argumento é uma operação “menor-que” utilizada para determinar a ordem dos elementos na lista ordenada

A função sort pode ser utilizada para ordenar

listas de inteiros

listas de listas de inteiros

...

Variações

Polimorfismo paramétrico implícito

- O texto do programa não contém tipos
- O algoritmo de inferência de tipos calcula quando uma função é polimórfica e a instanciação de variáveis de tipo quando necessário
- Exemplo
  - ML

Polimorfismo explícito

- O texto do programa tem variáveis de tipo
  - determina a forma como a função ou outro valor é tratado polimorficamente
- Muitas vezes involve instanciação explícita
  - indicando como as variáveis de tipo são substituidas por tipos específicos
- Exemplo
  - Templates C++

ML vs C++

Função polimórfica no ML

- As declarações não necessitam informação de tipo
- A inferência de tipos utiliza variáveis de tipo para “tipificar” as expressões
- A inferência de tipos substitui as variáveis conforme necessário para instanciar código polimórfico

Funções template no C++

- O programador tem de declarar os tipos dos argumento e resultado da função
- O programador tem de usar explicitamente parâmetros de tipo para expressar polimorfismo
- O verificador de tipos faz a instanciação de tipos

Exemplo

Trocar 2 valores

ML

```
fun swap(x,y) =
  let val z = !x in x := !y; y := z end;
val swap = fn : 'a ref * 'a ref -> unit
```

C++

```
template <typename T>
void swap(T& x, T& y){
   T z = x;  x=y;  y=z;
}
```

T: parâmetro de tipo

Quando aplicada a um tipo específico, o resultado é uma versão da função `swap` para esse tipo

Implementação

ML

- swap é compilado numa única função
- O verificador de tipos instancia a função a utilizar

C++

- swap é compilado num formato “ligador”
- O ligador (linker) duplica o código para cada tipo utilizado

Porquê as diferenças

ML

- a célula ref é passada por referência
- os parâmetros são apontadores para valores na heap sendo o seu tamanho constante

C++

- os argumentos são passados por referência
- os parâmetros estão na stack; o seu tamanho depende do tipo

Outro exemplo

Função sort polimórfica em C++

```
template <typename T>
void sort(int count, T * A[count]) {
  for (int i=0; i<count-1; i++)
    for (int j=i+1; j<count-1; j++)
      if (A[j]<A[i]) swap(A[i],A[j]);
}
```

Que partes do código dependem do tipo?

A indexação do array

O significado e implementação do operador menor (<)

Polimorfismo ad-hoc

O que é?

Duas ou mais implementações com diferentes tipos são referidas pelo mesmo nome

Também designado por sobrecarga

Exemplo

O algoritmo para implementar “<” depende do tipo envolvido

Diferenças

Polimorfismo paramétrico

- Um único algoritmo para muitos tipos
- Variavel de tipo pode ser substituida por qualquer tipo
- Se `f:t→t` então `f:int→int`, `f:bool→bool`, etc

Sobrecarga

- Um único símbolo pode referir mais que um algoritmo
- Cada algoritmo utiliza tipos diferentes
- A escolha do algoritmo é determinada pelo contexto
- Os tipos podem ser arbitrariamente diferentes
- + tem tipos `int*int→int`, `real*real→real`

Porquê a sobrecarga?

Muitas funções não são paramétricas!

Exemplos

Função member

```
member: [w] → w → bool
```

Apenas para tipos que suportam igualdade

Função sort

```
sort: [w] → [w]
```

Para tipos que suportam ordenação

Função serialize

```
serialize: w → String
```

Para tipos que suportam serialização

Função sumOfSquares

```
sumOfSquares: [w] → w
```

Para tipos que suportam operações numéricas

Sobrecarga aritmética – 1ª aproximação

Permitir que funções com sobrecarga de símbolos definam múltiplas funções

```
square x = x * x
```

Define duas funções: `int→int` e `real→real`

Mas

```
squares(x,y,z) = (square x, square y, square z)
```

Define 8 possíveis versões!

Não é muito utilizada devido ao crescimento exponencial do nº de versões

Sobrecarga aritmética – 2ª aproximação

Operações básicas podem ter sobrecarga, mas não as funções definidas em termos destas

```
square x = x * x
```

Escolhe uma implementação: `int→int`

Pouco ortogonal

A linguagem pode definir sobrecarga de operadores mas o programador não!

SML utiliza esta aproximação!

Sobrecarga da igualdade – 1ª aproximação

Igualdade definida apenas para tipos que admitem igualdade

tipos abstratos ou tipos que não contêm funções

```
3*3==9, 'a'=='b'
```

~~`\x->x == \y->y+1`~~

No SML é idêntica à dos operadores + e *

Não é possível definir funções com "=="

```
member [] y = False
member (x:xs) y = (x==y) || member xs y
```

Igualdade não está definida para qualquer tipo!

Sobrecarga da igualdade – 2ª aproximação

Tornar a igualdade totalmente polimórfica

```
(==): a → a → Bool
```

Tipo da função member

```
member: [a] → a → Bool
```

A linguagem Miranda utiliza esta apoximação

- A igualdade aplicada a uma função gera um **erro de execução**
- A igualdade aplicada a um tipo abstrato compara as suas representações
  - viola os princípios da abstração!

Sobrecarga da igualdade – 3ª aproximação

Tornar a igualdade polimórfica de forma limitada

$(==) : a_{(==)} \rightarrow a_{(==)} \rightarrow Bool$

onde $a_{(==)}$ é uma variável de tipo apenas para tipos que admitem igualdade

Tipo da função member

```
member: [a(==)] → a(==) → Bool
```

A versão actual do SML usa esta aproximação

O tipo $a_{(==)}$ é designado **eqtype** e escrito como `''a`

Classes de tipos

Resolvem estes problemas

- Permite aos utilizadores definir funções que utilizam operadores com sobrecarga
  - `squares`
  - `member`

Características

- Generaliza os *eqtypes* do ML para tipos arbitrários
- Fornece tipos para descrever sobrecarga de funções
- Permite aos utilizadores declarar novas coleções de funções com sobrecarga
  - Operadores de igualdade e aritmética não são privilegiados
- Pode ser utilizado na inferência de tipos
- Implementado como uma tradução fonte-para-fonte

Declaração de tipos

Classes de tipos

Transparente

Dá um nome alternativo a um tipo que também pode ser referido por outro nome

Opaca

É introduzido um novo tipo que não é igual a nenhum outro tipo

Declaração de tipos em ML

Declaração de tipos transparente

```
type <identificador> = <expressão_tipo>
```

Exemplo

```
type Celsius=real;
type Fahrenheit=real;
```

Declaração de tipos opaca

```
datatype <identificador>=
<clausula_cons> | … | <clausula_cons>
<clausula_cons>::=<cons> | <cons> of <tipo_args>
```

Exemplo

```
datatype A = C of int;
datatype B = C of int;
```

Declaração de tipos em C

Construção `typedef`

Geralmente é transparente

Se estiverem envolvidadas struct isso já não acontece!

Exemplo

```
typedef int A;
typedef int B;
A x; B y;
x=y
```
← **OK**

```
typedef struct{int m;} A;
typedef struct{int m;} B;
A x; B y;
x=y
```
← **erro de tipos**