Interpretação e Compilação (de Linguagens de Programação)

Unidade 5: Linguagens Imperativas

As expressões das linguagens consideradas até agora denotam sempre valores puros, que denotam sempre um mesmo valor ao longo da execução de um programa. No entanto, o paradigma de programação dominante é o paradigma imperativo, caracterizado pela mutação de estado (C, Java).

As operações fundamentais das linguagens imperativas são:

- A associação de identificadores a localizações de memória (var x:Integer)
- A modificação do conteúdo de localizações de memória (x := 2)

- Modelo de memória (cell: set e get)
- Ambiente versus memória
- Aliasing
- L-value e R-value
- Tempo de vida vs âmbito
- Manipulação de memória por apontadores, referências, etc
- Estrutura das linguagens imperativas. Família Algol vs família ML
- Sintaxe separada de comandos e expressões
- Zonas de memória (stack/heap)
- Representação interna de valores e objectos

Modelo de Memória

- Memória:
  é um conjunto (potencialmente infinito) de células cujo conteúdo é mutável.
- Cada célula de memória tem um designador único (a referência da célula) e pode conter qualquer valor da linguagem.
- As referências são valores de um tipo de dados especial ref que só podem ser usados no contexto da memória a que dizem respeito.
- Operações primitivas sobre uma memória $\mathcal{M}$

new: $\mathcal{M}$ × **void** → **ref**

set: $\mathcal{M}$ × **ref** × Value → **void**

get: $\mathcal{M}$ × **ref** → Value

free: $\mathcal{M}$ × **ref** → **void**

Modelo de Memória

- Operações sobre uma memória abstracta $\mathcal{M}$

new: $\mathcal{M}$ × **void** → **ref**

Devolve uma referência para uma nova célula livre, e define-a como estando "em uso".

set: $\mathcal{M}$ × **ref** × Value → **void**

Altera o conteúdo da célula referida para o valor indicado. O valor "antigo" perde-se irremediavelmente.

get: $\mathcal{M}$ × **ref** → Value

Devolve o valor contido na célula referida.

free: $\mathcal{M}$ × **ref** → **void**

Define a célula referida como estando livre, devolvendo-a ao gestor de memória, para ser reciclada.

Ambiente versus Memória

- O ambiente indica a denotação de cada identificador declarado num programa e reflecte a estrutura estática do programa.
- A associação estabelecida no ambiente entre um identificador e o seu valor denotado é fixa e imutável dentro do âmbito respectivo.
- A memória agrega o conteúdo das variáveis de estado mutáveis, indicando o valor contido em cada localização (ou referência).
- Uma variável de estado é visível nos programas através de identificadores.
- A associação entre o identificador de uma variável de estado e a sua localização de memória é imutável e é mantida pelo ambiente.

Ambiente versus Memória

Ambiente

Identificador	Valor
PI	3.14
x	loc_0
k	loc_1
j	loc_1
TEN	10

Memória

Localização	Valor
loc_0	25
loc_1	12
loc_2	loc_1
...	...
loc	0

Ambiente versus Memória

Ambiente

Identificador	Valor
PI	3.14
x	0x00FF
k	0x0100
j	0x0100
TEN	10

Memória

Endereço	Valor
0x00FF	25
0x0100	12
0x0102	0x0100
...	...
0xFFFF	0

Propriedades do modelo de memória

Ambiente

Identificador	Valor
PI	3.14
x	loc_0
k	loc_1
j	loc_1
TEN	10

Memória

Localização	Valor
loc_0	25
loc_1	12
loc_2	loc_1
...	...
loc	0

Uma mesma célula de memória pode ser referida por vários identificadores distintos (aliasing).

Propriedades do modelo de memória

Ambiente

Identificador	Valor
PI	3.14
x	loc_0
k	loc_1
j	loc_1
TEN	10

Memória

Localização	Valor
loc_0	25
loc_1	12
loc_2	loc_1
...	...
loc	0

Uma mesma célula de memória pode ser referida por vários identificadores distintos (aliasing).

Aliasing

- Dois identificadores diferentes que referem a mesma localização de memória.

```
class A {
  int x;
  boolean equals(A a) { return x == a.x}
}
A a = new A(); a.equals(a);
```

```
int x = 0;
void f(int* y) { *y = x+1; }
...
f(&x);
// x = ?
```

Propriedades do modelo de memória

Ambiente

Identificador	Valor
PI	3.14
x	loc_0
k	loc_1
j	loc_1
TEN	10

Memória

Localização	Valor
loc_0	25
loc_1	12
loc_2	loc_1
...	...
loc	0

Uma célula pode conter uma referência para outra célula, permitindo a construção de estruturas de dados dinâmicas.

Propriedades do modelo de memória

Ambiente

Identificador	Valor
PI	3.14
x	loc_0
k	loc_1
j	loc_1
TEN	10

Memória

Localização	Valor
loc_0	25
loc_1	12
loc_2	loc_1
...	...
loc	0

Uma célula pode conter uma referência para outra célula, permitindo a construção de estruturas de dados dinâmicas.

Operações Imperativas nas linguagens

- Reserva de uma nova variável / célula de memória e sua inicialização com o valor da expressão E

new E

- Afectação de novo valor (dado pela expressão F) a uma variável / célula de memória (dada pela expressão E)

E **:=** F

- Desreferenciação de variável / célula de memória dada a referência produzida por avaliação da expressão E.

!E

- Libertação de variável / célula de memória dada a sua referência produzida por avaliação da expressão E.

free E

Operações Imperativas nas linguagens

- Reserva de uma nova célula e sua inicialização com o valor da expressão E

new E

- Pode ser encontrada de diversas formas:

```
{
  int a;
  MyClass m;
  …
}

new int[10];

malloc(sizeof(int));

new MyClass();
```

Operações Imperativas nas linguagens

- Reserva de uma nova célula e sua inicialização com o valor da expressão E

new E

- Pode ser encontrada de diversas formas:

```
{
  int a;
  MyClass m;
  …
}
```

em Java tem um significado
em C++ tem outro,
qual a diferença?

```
new int[10];

malloc(sizeof(int));

new MyClass();
```

Operações Imperativas nas linguagens

- Afectação de um valor a uma variável dadas as expressões E e F

E := F

E denota uma referência para uma variável, F é uma expressão qualquer

- Pode ser encontrada de diversas formas:

```
a = 1

i := 2

b[x+2][b[x-2]] = 2

*(p+2) = y

myTable(i,j) = myTable(j,i)

Readln(MyLine);
```

Operações Imperativas nas linguagens

- Desreferenciação de variável / célula de memória dada a referência produzida por avaliação da expressão E.

!E

- Pode ser encontrada em diversas formas:

```
i := !i + 1                    (linguagem ML)

*p                             (linguagem C)

i = i + 1                      (linguagem C)

i++                            (linguagem C)
```

Operações Imperativas nas linguagens

- Desreferenciação de variável / célula de memória dada a referência produzida por avaliação da expressão E.

!E

- Pode ser encontrada em diversas formas:

```
i := !i + 1                    (linguagem ML)

*p                             (linguagem C)

i = i + 1                      (linguagem C)

i++                            (linguagem C)
```

referências

Operações Imperativas nas linguagens

- Desreferenciação de variável / célula de memória dada a referência produzida por avaliação da expressão E.

!E

- Pode ser encontrada em diversas formas:

```
i := !i + 1                    (linguagem ML)

*p                             (linguagem C)

i = i + 1                      (linguagem C)

i++                            (linguagem C)
```

valor

L-Value e R-Value

- Se uma expressão E tem por valor uma referência, a maior parte das linguagens de programação interpreta E de forma dependente do contexto

```
E := 2
```

- (Left-Value) À "esquerda" do símbolo de afectação, denota o seu valor efectivo (que é uma referência)

```
E := E + 1
```

- (Right-Value) À "direita" do símbolo de afectação, denota o conteúdo da célula referida, evitando-se escrever a desreferenciação explícita

```
E := !E + 1
```

L-Value e R-Value

- Se uma expressão E tem por valor uma referência, a maior parte das linguagens de programação interpreta E de forma dependente do contexto

```
E := 2
```

- (Left-Value) À "esquerda" do símbolo de afectação, denota o seu valor efectivo (que é uma referência)

```
E := E + 1
```

- (Right-Value) À "direita" do símbolo de afectação, denota o conteúdo da célula referida, evitando-se escrever a desreferenciação explícita

```
E := !E + 1
```

L-Value e R-Value

- Se uma expressão E tem por valor uma referência, a maior parte das linguagens de programação interpreta E de forma dependente do contexto.

```
A[A[2]] := A[2] + 1
```

- A terminologia "L-Value" e "R-Value" não é muito feliz. Por exemplo, na expressão acima as duas subexpressões da forma A[2], uma à esquerda e outra à direita, são ambas desreferenciadas implicitamente.

Desreferenciação

- A operação de desreferenciação !E torna a interpretação dos programas mais precisa e evita qualquer ambiguidade.

```
A[!A[2]] := !A[2] + 1
```

- Por outro lado, pode argumentar-se que torna os programas mais difíceis de ler.

A desreferenciação implícita pode ser vista como uma operação de coerção (conversão ou cast).

Idiomas Imperativos Básicos

- Todas as declarações e usos comuns de variáveis mutáveis podem exprimir-se usando as primitivas

new(E **)**	instanciação
free(E **)**	libertação
E **:=** E	afectação
! E	desreferenciação

```
{
/* linguagem C */

  const int k = 2;
  int a = k;
  int b = a + 2;
  …
  b = a * b
  …
}
```

```
decl
  k = 2
  a = new(k)
  b = new(!a+2)
in
  …
  b := !a * !b
  …
  free(a);
  free(b)
end
```

Idiomas Imperativos Básicos

- Todas as declarações e usos comuns de variáveis mutáveis podem exprimir-se usando as primitivas

new(E **)**	instanciação
free(E **)**	libertação
E **:=** E	afectação
! E	desreferenciação

```
{
/* linguagem C */

  const int k = 2;
  int a = k;
  int b = a + 2;
  …
  b = a * b
  …
}
```

```
decl
  k = 2
  a = new(k)
  b = new(!a+2)
in
  …
  b := !a * !b
  …
  free(a);
  free(b)
end
```

libertação implícita (das células atribuídas aos ids a e b)

Idiomas Imperativos Básicos

- Todas as declarações e usos comuns de variáveis mutáveis podem exprimir-se usando as primitivas

new(E **)**	instanciação
free(E **)**	libertação
E **:=** E	afectação
! E	desreferenciação

```
{
/* linguagem C */

   int k = 2;
   int *a = &k;
   … a = k+*a …

}
```

```
decl
   k = new(2)
   a = new(k)
in
   … !a := !k+!!a …
   free(k);
   free(a)
end
```

Idiomas Imperativos Básicos

- Todas as declarações e usos comuns de variáveis mutáveis podem exprimir-se usando as primitivas

new(E **)**	instanciação
free(E **)**	libertação
E **:=** E	afectação
! E	desreferenciação

```
{
/* linguagem C */

  int k = 2;
  const int *a = &k;
  int b = *a;
  … *a = k+b …

}
```

```
decl
  k = new(2)
  a = k
  b = new(!a)
in
  … a := !k+!b …
  free(k);
  free(a);
  free(b)
end
```

Idiomas Imperativos Básicos

- Todas as declarações e usos comuns de variáveis mutáveis podem exprimir-se usando as primitivas

new(E **)**	instanciação
free(E **)**	libertação
E **:=** E	afectação
! E	desreferenciação

```
{
/* linguagem C */

   int k = 2;
   int *a = &k;
   … k = k+*a …

}
```

```
decl
   k = new(2)
   a = new(k)
in
   … k := !k+!!a …
   free(k);
   free(a);
end
```

Tempo de Vida (de uma célula)

O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando **new(_)** e a sua libertação usando **free(_)**.

- Em muitas situações, o tempo de vida da célula concide com o âmbito do(s) seu(s) identificador.

criação da célula k

```
{
/* linguagem C */

   int k = 2;
   ...
}
```

```
int *f(int j) {
   int k = j;
   return &k;
}
```

libertação implícita da célula k

Tempo de Vida (de uma célula)

O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando **new(_)** e a sua libertação usando **free(_)**.

- Noutras situações, o tempo de vida da célula extravasa o âmbito do(s) seu(s) identificador.

âmbito de k

```
{
/* linguagem C */

  static int k = 2;
  ...
}
```

O tempo de vida da célula associada a k é o tempo do programa

reserva da célula para k

```
/* linguagem Java */
Integer f(int j) {
  Integer k = new Integer(j);
  return k;
}
```

reserva de novo objecto Integer

há libertação implícita da célula de k mas o objecto sobrevive ao bloco!

Tempo de Vida (de uma célula)

O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando **new(_)** e a sua libertação usando **free(_)**.

- Noutras situações, o tempo de vida da célula extravasa o âmbito do(s) seu(s) identificador.

âmbito de k

```
{
/* linguagem C */

  static int k = 2;
  ...
}
```

O tempo de vida da célula associada a k é o tempo do programa

reserva da célula para k

```
/* linguagem C */
int* f(int j) {
  int *k = malloc(sizeof(int));
  *k = 2;
  return k;
}
```

reserva de novo bloco de memória

há libertação implícita da célula de k mas a memória reservada predura.

Linguagens Imperativas

Linguagens da família do ALGOL (Pascal, C, …)

Assumem como princípio de desenho uma separação muito clara, logo ao nível sintáctico, entre expressões e comandos

- Expressões:

  Denotam valores puros (inteiros, booleanos, funções)

  A avaliação de expressões não deve ter efeitos (laterais)

- Comandos:

  Denotam efeitos (na memória)

  Um comando é executado pelo efeito que produz na memória: representa uma acção.

Uma linguagem de tipo ALGOL

- Definida com base em duas categorias sintácticas: Expressões (EXP) e comandos (COM):

num:	Integer → EXP
bool:	Boolean → EXP
id:	String → EXP
add:	EXP × EXP → EXP
and:	EXP × EXP → EXP
if:	EXP × COM × COM → COM
while:	EXP × COM → COM
assign:	EXP × EXP → COM
seq:	COM × COM → COM
var:	String × EXP × COM → COM
const:	String × EXP × COM → COM

Linguagens Imperativas

Linguagens da família do ML

Todas as construções pertencem a uma única categoria sintáctica, de expressões.

- Nestas linguagens, qualquer expressão pode potencialmente produzir um efeito lateral…
- Por exemplo, em OCAML a afectação x := E é uma expressão (de tipo **unit** (a.k.a. **void**)).
- N.B. Existem linguagens que combinam conceitos! Por exemplo, a linguagem C, contém expressões e comandos: a afectacão (x = y) é uma expressão, e expressões podem produzir efeitos (i++).

Uma linguagem tipo ML (microML)

- Consideramos uma só categoria sintáctica para expressões (EXP):

num:	Integer → EXP	
bool:	Boolean → EXP	
id:	String → EXP	
add:	EXP × EXP → EXP	
new:	EXP → EXP	
deref:	EXP → EXP	(!x)
if:	EXP × EXP × EXP → EXP	
while:	EXP × EXP → EXP	
assign:	EXP × EXP → EXP	(x := y + z)
seq:	EXP × EXP → EXP	(S1 ; S2)
decl:	String × EXP × EXP → EXP	

Semântica de microML

A semântica de uma linguagem imperativa pode ser caracterizada por uma função I que dá uma denotação a todos os programas abertos dado um ambiente e uma memória.

$$I : \mathrm{P} \times \mathrm{ENV} \times \mathrm{MEM} \rightarrow \mathrm{VAL} \times \mathrm{MEM}$$

P = Fragmentos de programa (abertos)

ENV = Ambientes (funções ID → VAL)

MEM = Memórias

VAL = Valores (Denotações)

O conjunto das denotações possíveis:

$$\mathrm{Val} = \mathrm{Boolean} \cup \mathrm{Integer} \cup \mathrm{Ref}$$

Esta função traduz a intuição que, em geral, um fragmento de programa P produz um valor e gera um efeito (na memória).

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$$\text{eval} : \text{microML} \times \text{ENV} \times \text{MEM} \rightarrow \text{VAL} \times \text{MEM}$$

eval(**add(**E1, E2**)** , *env* , *m0*) ≜**[** (*v1* , *m1*) = **eval**(E1, *env, m0*);

(*v2* , *m2*) = **eval**(E2, *env, m1*);

(*v1* + *v2* , *m2*) **]**

eval(**and(**E1, E2**)** , *env* , *m0*) ≜**[** (*v1* , *m1*) = **eval**(E1, *env, m0*);

(*v2* , *m2*) = **eval**(E2, *env, m1*);

(*v1* & *v2* , *m2*) **]**

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$$

eval(**new**(E) , env , m0) ≜ **[** (v1 , m1) = **eval**(E, env, m0);

(ref, m2) = m1.**new**(v1);

(ref, m2) **]**

eval(**deref**(E) , env , m0) ≜ **[** (ref , m1) = **eval**(E, env, m0);

(m1.**get**(ref) , m1) **]**

eval(**assign**(E1, E2) , env , m0) ≜ **[**(v1 , m1) = **eval**(E1, env, m0);

(v2 , m2) = **eval**(E2, env, m1);

m3 = m2.**set**(v1, v2) ;

(v1 , m3) **]**

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$$

eval(**seq**(E1, E2) , env , m0) ≜ **[**(v1 , m1) = **eval**(E1, env, m0);

(v2 , m2) = **eval**(E2, env, m1);

(v2 , m2) **]**

eval(**if**(E1, E2, E3) , env , m0) ≜

[(v1 , m1) = **eval**(E1, env, m0);

if (v1 = T) **then** (v2 , m2) = **eval**(E2, env, m1);

else (v2 , m2) = **eval**(E3, env, m1);

(v2 , m2) **]**

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$$

eval(**while**(E1, E2) , env , m0) ≜

[(v1 , m1) = **eval**(E1, env, m0);

if (v1 = T) **then** **[** (v2 , m2) = eval(E2, env, m1);

(v,m1) =**eval**(while(E1, E2), m2) **]**

else (F , m1) **]**

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$$

eval(**while**(E1, E2) , env , m0) ≜

[(v1 , m1) = **eval**(E1, env, m0);

if (v1 = T) **then** **[** (v2 , m2) = eval(E2, env, m1);

(v,m1) =**eval**(while(E1, E2), m2) **]**

else (F , m1) **]**

iteração interpretada em termos de recursão.

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$$

```
eval( decl(s, EI, EB) , env , m0) ≜
                    [(v1 , m1) = eval( EI, env, m0 );
                     env = env.BeginScope();
                     env.Assoc(s, v1);
                     (v2 , m2) = eval(EB, env, m1);
                     env = env.EndScope();
                     (v2 , m2) ]
```

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$$

```
decl a = new(2) in
decl b = new(!a) in
decl c = a in
(
  a := !b + 2;
  c := !c + 2
)
```

Semântica de microML

- Algoritmo eval para calcular o valor de uma expressão qualquer da linguagem microML:

$$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$$

```
decl a = new(2) in
decl b = new(!a) in
decl c = a in
(
  a := !b + 2;
  c := !c + 2
)
```

a e c são aliases (sinónimos), ou seja, referem a mesma célula de memória.