
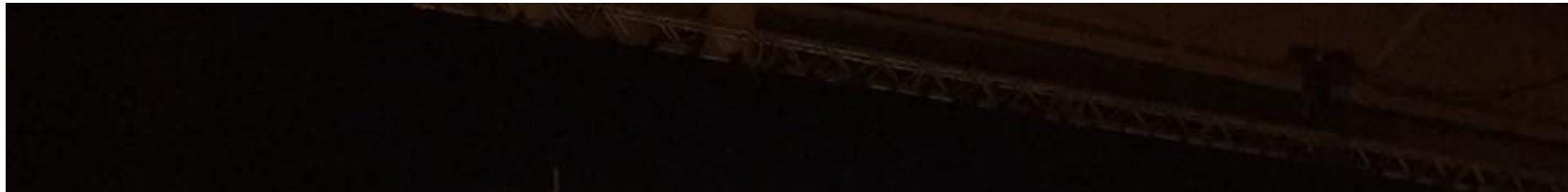


Lecture 08

Tipificação e compilação de abstrações

My students' best friend



907,415 REPUTATION

490 6584 7506

Jon Skeet top 0.01% overall

Senior Software Engineer at Google

Author of [C# in Depth](#).

Currently a software engineer at Google, London.

Usually a Microsoft MVP (C#, 2003-2010, 2011-)

Sites:

- [C# in Depth](#)
- [Coding blog](#)
- [C# articles](#)
- [Twitter updates \(@jonskeet\)](#)
- [Google+ profile](#)

33,439 answers **42** questions **~174.8m** people reached

📍 Reading, United Kingdom

🐦 [jonskeet](#)

🔗 [jskeet](#)

🌐 [csharpindepth.com](#)

🕒 Member for 8 years, 1 month

👁 1,285,139 profile views

🕒 Last seen 28 mins ago



Quiz

- Qual o valor (se existir) das seguintes expressões:

```
decl f = (fun x -> x+1) in
  decl g = (fun y -> y(2)) in g(f) end end

decl f = (fun x -> x(x)) in f(f) end
```

- Qual o valor da expressão seguinte quando avaliada pela regra dinâmica e pela regra estática de resolução de nomes:

```
decl x=2 in
  decl g = (fun y -> y-x) in
  decl x = 4 in g(x) end end end
```

- Considera que as duas expressões seguintes têm sempre o mesmo valor? Porquê?

```
decl id = E1 in E2 end
(fun id -> E2)( E1)
```

Quiz (solução)

- Considera que as duas expressões seguintes têm sempre o mesmo valor? Porquê?

```
decl id = E1 in E2 end  
(fun id -> E2)( E1 )
```

- Temos (aplicando as regras de avaliação):

```
eval(decl id = E1 in E2 end, env) =  
eval(E2, env.Assoc(id, eval(E1, env)))
```

- Por outro lado:

```
eval((fun id -> E2), env) = closure(id, E2, env)  
eval((fun id -> E2) (E1), env) =  
eval(E2, env.Assoc(id, eval(E1, env)))
```

Compilação de CALCF (1)

- A compilação de CALCF pode ser caracterizada por uma função:

$$\text{comp}: P \times ENV \rightarrow \text{CodeSeq}$$

P = Fragmento de programa (aberto)

ENV = Ambiente (função $\text{String} \rightarrow (\text{jumps}, \text{offset})$)

CodeSeq = Sequências de instruções

Objectos função

- A máquina JVM não dispõe de mecanismos para implementar closures directamente

```
fun x -> x + 1 end (1)
```

- A máquina JVM não dispõe de instruções para a chamada directa de funções a partir de valores (referências ou apontadores).
- A implementação pode ser feita recorrendo ao padrão dos objectos função:

```
interface Type_00 { int call (int); }
```

```
class Closure_00 implements Type_00 { int call(int x) { return x+1; } }
```

- concretizado em código jasmin

```
.source type_00.j  
.interface public type_00  
.super java/lang/Object  
.method public abstract call(I)I  
.end method
```

```
.method public call(I)I  
.limit locals 3  
.limit stack 256
```

```
; initialize new stackframe frame_1  
new frame_1  
dup  
invokespecial frame_1/<init>()V  
dup  
iload 1  
putfield frame_1/loc_00 I  
astore 2
```

Compilação de CALCF

Um programa CALCF contém abstrações anónimas em número finito e previsível. O corpo de uma abstracção é um pedaço de código que fica por avaliar e que nesta linguagem é referido por um valor. **Ideia geral:** Associar a cada abstracção CALCF uma classe com um método que contém o código compilado da expressão que a compõe.

```
decl
  x = 1
  f = (fun y -> y+x)
in
  decl
    g = (fun x -> f(x)+1)
  in
    decl
      h = g
      i = (fun y->(fun x -> x)(g(y)))
    in
      i(f(1))
    end
  end
```

```
.interface public type_00
.method public abstract call(I)I
...

.class public closure_01
.super java/lang/Object
.implements type_00
...

.class public closure_02
.super java/lang/Object
.implements type_00
...

.class public closure_03
.super java/lang/Object
.implements type_00
...

.class public closure_04
.super java/lang/Object
.implements type_00
...

.method public static main([Ljava/lang/String;)V
...

```

Compilação de CALCF

Um programa CALCF contém abstrações anónimas em número finito e previsível. O corpo de uma abstracção é um pedaço de código que fica por avaliar e que nesta linguagem é referido por um valor. **Ideia geral:** Associar a cada abstracção CALCF uma classe com um método que contém o código compilado da expressão que a compõe. No caso das closures, guardar um apontador para o ambiente.

```
decl
  x = 1
in
  decl
    f = (fun y -> y+x)
  in
    decl
      g = (fun h -> h(x)+1)
    in
      g (f)
    end
  end
end
```

```
.interface public type_00
.method public abstract call(I)I
...

.interface public type_01
.method public abstract call(type_00) I
...

.class public closure_01
.super java/lang/Object
.implements type_00
.field public SL Lframe_1;
...

.class public closure_02
.super java/lang/Object
.implements type_01
...

.method public static main([Ljava/lang/String;)V
...
```


Compilação de CALCF

- A compilação de CALCF pode ser caracterizada por uma função:

$\text{comp}: P \times \text{ENV} \rightarrow \text{CodeSeq} \times \text{CodeSeq list}$

P = Fragmento (tipificado) de programa (aberto)

ENV = Ambiente (função $\text{String} \rightarrow (\text{jumps}, \text{offset})$)

CodeSeq = Sequências de instruções

Tipificação de CALCF

- Algoritmo **typecheck** para calcular o tipo de uma expressão qualquer da linguagem CALCF:

typecheck : CALCF \times ENV \rightarrow TYPE

```
typecheck( fun(x, t, E) , env )  $\triangleq$  [ envlocal = env.BeginScope();  
                                     envlocal.assoc(x,t);  
                                     t1 = typecheck ( E, envlocal)  
                                     if (t1 == none )  
                                     then none  
                                     else t  $\Rightarrow$  t1]
```

TYPE = { int, bool, TYPE \Rightarrow TYPE, none }

Tipificação de CALCF

- Algoritmo **typecheck** para calcular o tipo de uma expressão qualquer da linguagem microML:

typecheck : CALCF \times ENV \rightarrow TYPE

```
typecheck( call(E1, E2) , env )  $\triangleq$  [ t1 = typecheck ( E1, env )  
                                         t2 = typecheck ( E2, env )  
                                         if ( t1 == (tp  $\Rightarrow$  tr ) ) and ( t2 == tp )  
                                         then tr  
                                         else none ]
```

TYPE = { **int**, **bool**, TYPE \Rightarrow TYPE, **none** }

Example

```
decl x = 1 in fun x:int => x+1 end (1) end
```

```
new frame_1
```

```
dup
```

```
invokespecial frame_1/<init>()V
```

```
dup
```

```
sipush 1
```

```
putfield frame_1/loc_00 I
```

```
astore 1 ; SP is now at local variable 1, 0 is reserved for “this”
```

```
new closure_01
```

```
dup
```

```
invokespecial closure_01/<init>()V
```

```
dup
```

```
aload 1 ; SP
```

```
putfield closure_01/SL Lframe_1;
```

```
checkcast type_00
```

```
sipush 1
```

```
invokeinterface type_00/call(I)I 2
```

Definições Recursivas

- Na construção de declaração local de identificadores

```
decl id = E1 in E2 end
```

o nome `id` **não é ligado** às ocorrências do mesmo nome em E_1

- Por exemplo, na expressão

```
decl x=1 in  
  decl x=x+1 in  
    x+1  
  end  
end
```

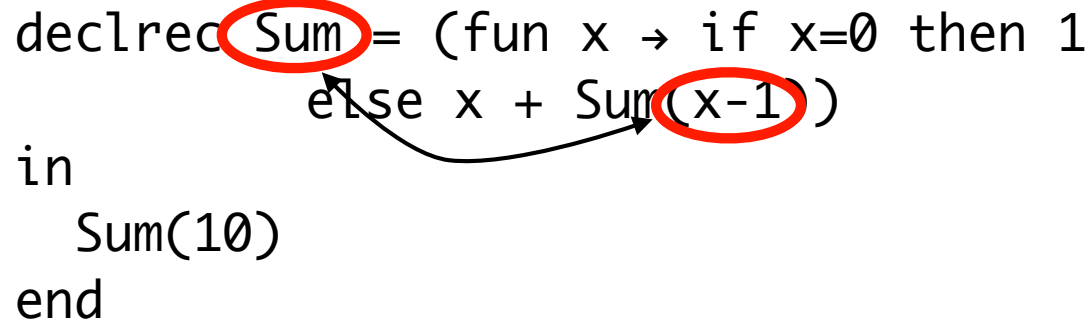
a ocorrência de `x` na inicialização do segundo `decl` está ligada à primeira declaração `x = 1`.

- A segunda definição de `x` não é “recursiva” !!

Definições Recursivas

- Uma definição recursiva é uma declaração onde o identificador declarado pode ocorrer dentro do corpo da definição:

```
declrec Sum = (fun x → if x=0 then 1
                    else x + Sum(x-1))
in
  Sum(10)
end
```



Mais rigorosamente: numa declaração recursiva a ocorrência ligante de um identificador também liga as ocorrências livres do mesmo identificador na expressão de inicialização.

- Problema:** como introduzir e como definir a semântica de definições recursivas (de funções) ?

A Linguagem RECF

- A linguagem RECF é a extensão da linguagem CALCF com expressões condicionais e definições recursivas.

num:	<code>integer → RECF</code>
var:	<code>string → RECF</code>
add:	<code>RECF × RECF → RECF</code>
...	
if:	<code>RECF × RECF × RECF → RECF</code>
declrec:	<code>string × RECF × RECF → RECF</code>
fun:	<code>string × RECF → RECF</code>
call:	<code>RECF × RECF → RECF</code>

- Nota: a construção `decl` não existe na linguagem RECF, mas pode ser codificada da forma apresentada atrás:

$(\text{decl } x = E1 \text{ in } E2) = (\text{fun } x \rightarrow E2)(E1)$

A Linguagem RECF

- Exemplo de programa em RECF:

```
declrec
  fact = fun n → if n then n*fact(n-1) else 1 end
in
  fact(2)
end
```

- Abreviaturas:

$$\text{decl id} = E_1 \text{ in } E_2 \triangleq \text{call}((\text{fun id} \rightarrow E_2), E_1)$$
$$E_1(E_2) \triangleq \text{call}(E_1, E_2)$$
$$\text{if } E_1 \text{ then } E_1 \text{ else } E_3 \triangleq \text{if}(E_1, E_2, E_3)$$

A Linguagem RECF

- Exemplo de programa em RECF:

```
declrec  
  fact = fun n → if n then n*fact(n-1) else 1 end  
in  
  fact(2)  
end
```

- Qual o ambiente no qual a abstração deve ser avaliada/definida?

Note que:

- fact é uma ocorrência livre na abstração
- O identificador fact deverá estar definido no ambiente ativo no momento em que a abstração for avaliada
- Por outro lado, o valor a associar a fact nesse mesmo ambiente deverá ser a própria função.

A Linguagem RECF

- Exemplo de programa em RECF:

```
declrec
  fact = fun n → if n then n*fact(n-1) else 1 end
in
  fact(2)
end
```

- Qual o ambiente no qual a abstração deve ser avaliada/definida?
 - As definições recursivas introduzem uma “circularidade” na construção do ambiente:
 - O ambiente E resultante da declaração de `fact` deverá ter uma ligação `fact` \rightarrow `closure(n , if ... end , E)` que associe ao nome `fact` um fecho cuja componente ambiente é o próprio E
 - Em geral, um ambiente E deverá poder conter ligações entre identificadores e fechos com referências para (partes de) o próprio ambiente E

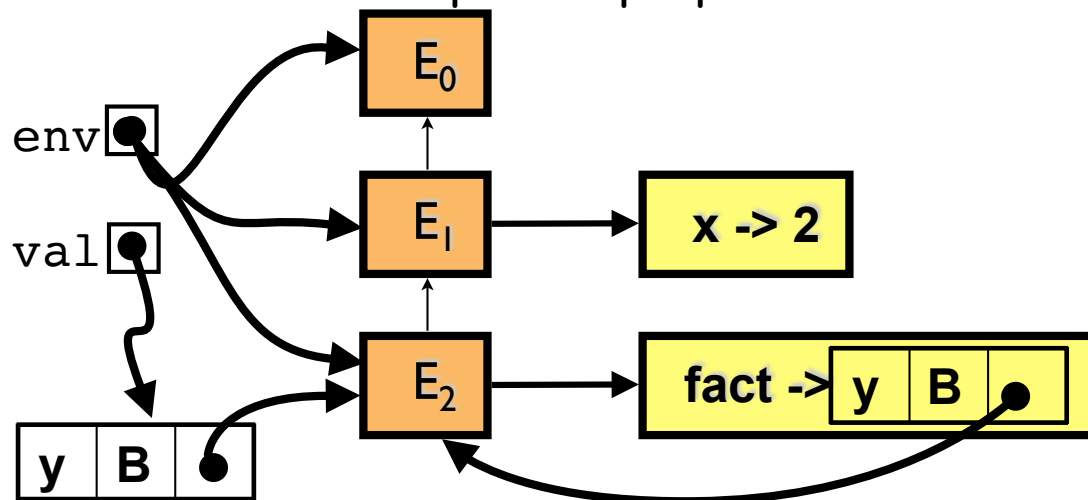
Ambiente Mutável (revisitado)

- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (**por alteração imperativa**) a ligação existente para o identificador id pelo valor val.
- Se o valor val contiver uma referência para o ambiente env, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();  
env = BeginScope();  
env.Assoc("x", 2);  
env = env.BeginScope();  
env.Assoc("fact", null);  
val = closure("y", B, env);  
env.Update("fact", val);
```



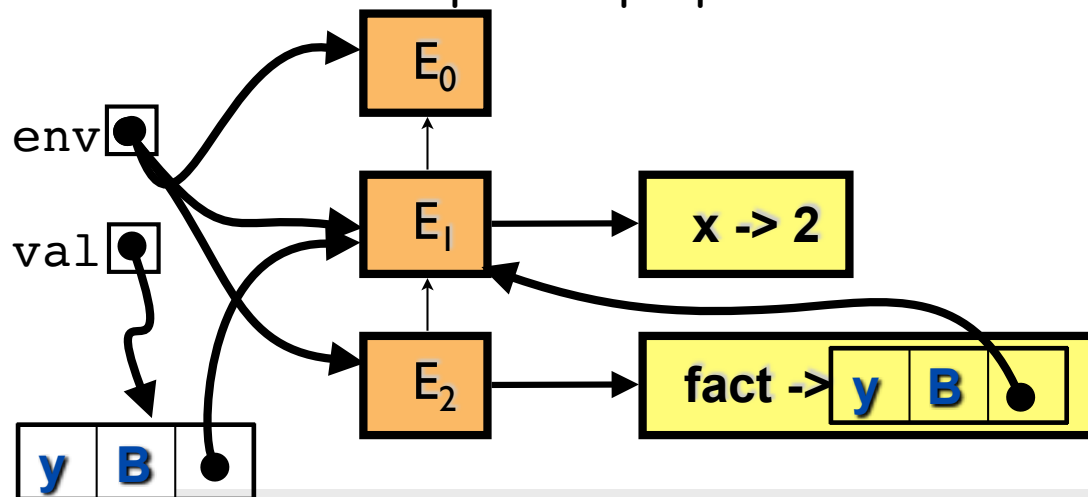
Ambiente Mutável (revisitado)

- Para suportar a criação de ambientes circulares, adicionamos uma nova primitiva aos ambientes que permite modificar uma ligação já efectuada

```
ENV Update(String id, Value val)
```

- Esta operação devolve o mesmo ambiente, mas substitui (**por alteração imperativa**) a ligação existente para o identificador id pelo valor val.
- Se o valor val contiver uma referência para o ambiente env, este passará a conter uma ligação mencionando uma referência para si próprio.

```
env = new Environment();  
env = env.BeginScope();  
env.Assoc("x", 2);  
val = closure(n, B, env);  
env.BeginScope();  
env.Assoc("fact", val);
```



Aqui, durante a avaliação do corpo B , o ambiente não refere a definição de $fact$ (apenas de x) ...

Semântica de RECF

- A função semântica I de RECF:

$$I : \text{RECF} \times \text{ENV} \rightarrow \text{RESULT}$$

RECF = conjunto dos programas abertos

ENV = conjunto dos ambientes válidos

RESULT = conjunto dos significados

- Os resultados podem ser valores inteiros, fechados (uma abstração + um ambiente), ou um erro.

$$\text{RESULT} = \text{integer} \cup \text{closure} \cup \{ \text{error} \}$$

Semântica de RECF

- Algoritmo `eval` para calcular a denotação de qualquer expressão `E` de RECF num ambiente `env`:

$\text{eval} : \text{RECF} \times \text{ENV} \rightarrow \text{RESULT}$

Dado um ambiente **env**

Se `E` é da forma `if(E1, E2, E3)`:
 `c = eval(E1, env)`;
 if `c != 0` **then** `eval(E, env) ≐ eval(E2, env)`
 else `eval(E, env) ≐ eval(E3, env)`

Se `E` é da forma `declrec(id, E1, E2)`:
 [`envloc = env.BeginScope()`;
 `envloc.Assoc(id, null)`;
 `val = eval(E1, envloc)`;
 `envloc.Update(id, val)`;
 `v = eval(E2, envloc)`;
 `envloc.EndScope()`]

eval(`E, env`) $\triangleq v$

Auto-avaliação...

- Avalie o programa P escrito na linguagem **RECF** usando a semântica apresentada:

```
declrec
  fact = fun n → if n then n*fact(n-1) else 1 end
in
  fact(3)
end
```

$E1 = [\text{fact} \rightarrow \text{closure}(n, (\text{if } \dots) , E_1)]$

$\text{eval}(P, \emptyset) =$

$\text{eval}(\text{fact}(3), E1) = \dots ?$

Se E é da forma **declrec**(id, E1, E2):

```
[ envloc = env.BeginScope();
  envloc.Assoc(id, null);
  val = eval(E1, envloc);
  envloc.Update(id, val);
  v = eval(E2, envloc);
  envloc.EndScope() ]
eval(E, env) ≜ v
```