Interpretação e Compilação de Linguagens de Programação

Luís Caires, João Costa Seco

Edição 2011-2012

Regência: João Costa Seco (joao.seco@di.fct.unl.pt)

Licenciatura em Engenharia Informática

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Unidade 9: Linguagens Orientadas por Objectos

O paradigma de programação dominante é o paradigma orientado por objectos. O conceito de objecto agrega estado e funcionalidade num contexto de visibilidade próprio. Os conceitos básicos para termos uma linguagem orientada por objectos são: Dynamic lookup, Abstraction, Subtyping, Inheritance.

Um objecto pode ser modelado por um registo com constantes, variáveis de estado, métodos (funções), e a definição recursiva de um identificador especial “self” ou “this”. Os objectos podem ser criados através de padrões bem definidos em “classes”.

- Valores Produto e Produtos Etiquetados
- Tipos Produto (Regras de tipificação)
- Conceitos básicos de objectos e classes
- Linguagens Orientadas por objectos
- Objectos na linguagem CORE
- Semântica por tradução na linguagem CORE
- Classes na linguagem CORE
- Tipificação de objectos
- Subtipificação

Valores Produto

- Correspondem aos records da linguagem Pascal

```
Type PersonName = packed array[1..20] of char;
     PersonInfo = Record
                    name:PersonName;
                    age: Integer;
                  end;
Var person1: PersonInfo;
```

```
person1.age := 25;
```

- structs da linguagem C,

```
typedef struct { char[20] name; int age; } person_info;
...
person_info p;
p.age := 25;
```

- ou produtos etiquetados na linguagem ML.

```
# type person_info = { name:string; age:int };;
type person_info = { name : string; age : int; }
# fun p -> p.name;;
- : person_info -> string = <fun>
```

Valores Produto

- Um valor de tipo produto, também chamado registo, é um valor estruturado que vários "agrega" vários valores, de tipos possivelmente diferentes, numa só entidade.
- O constructor básico é o operador binário de produto cartesiano

```
# (1,2);;
- : int * int = (1, 2)
# ("ola",("mundo",0));;
- : string * (string * int) = ("ola", ("mundo", 0))
```

- Construção de registos

```
v = [ nome = "rita", idade = 1 ]
c = [ real = 0.5+0.5, imag = 2.0 ]
```

- Manipulação de registos (selecção de campo)

```
v.nome = "rita"
c.real = 1.0
v.cor = ?                    (erro: campo não existente)
```

Valores Produto

- **Constructores para as expressões sobre registos:**

newrecord:	(string × EXP) list → EXP
selectfield:	EXP × string → EXP

- **Exemplo de uso**

```
decl
  p1 = [nome="Albert"; QI=var(250)]
in decl
  p2 = [nome="Hulk"; QI=var(50)] in
do
  p2.QI := !(p2.QI)+2
return
  !(p2.QI) + !(p1.QI)
end
end
end
```

Tipos Produto

- As operações disponíveis são a construção de registo e a selecção de campo de registo.
- Tuple(id_1:$\mathcal{T}_1$, … , id_n:$\mathcal{T}_n$):
  É o tipo dos registos com campos id_1, … , id_n, de tipo $\mathcal{T}1$, … , $\mathcal{T}n$.

$$\frac{Env \vdash E : \text{Tuple}(id_1:\mathcal{T}_1, \ldots, id_n:\mathcal{T}_n)}{Env \vdash E.id_j : \mathcal{T}_j} \textbf{(Select)}$$

$$\frac{Env \vdash E_1 : \mathcal{T}_1 \quad \ldots \quad Env \vdash E_n : \mathcal{T}_n}{Env \vdash [\, id_1 = E_1, \ldots, id_n = E_n \,] : \text{Tuple}(id_1:\mathcal{T}_1, \ldots, id_n:\mathcal{T}_n)} \textbf{(Record)}$$

Tipos Produto (Quiz)

- Qual o tipo da expressão?

$$\textbf{decl } c = [\ succ = \textbf{fun } x \rightarrow x{+}1 \textbf{ end},\ \ loc = \textbf{var}(0)\ ] \textbf{ in } c \textbf{ end}$$

- A expressão seguinte está bem tipificada?

$$\textbf{decl } c = [\ succ = \textbf{fun } x \rightarrow x{+}1 \textbf{ end},\ \ loc = \textbf{var}(0)\ ] \textbf{ in } \text{c.succ(!c.loc)} \textbf{ end}$$

Tipos Produto (Quiz)

- Qual o tipo da expressão?

$$\textbf{decl}\ c = [\ \mathit{succ} = \textbf{fun}\ x \rightarrow x{+}1\ \textbf{end},\ \ \mathit{loc} = \textbf{var}(0)\ ]\ \textbf{in}\ c\ \textbf{end}$$

$$\dfrac{\dfrac{\dfrac{x{:}\textbf{int} \vdash x : \textbf{int} \qquad x{:}\textbf{int} \vdash 1 : \textbf{int}}{x{:}\textbf{int} \vdash x{+}1 : \textbf{int}}}{\varnothing \vdash \textbf{fun}\ x{:}\textbf{int} \rightarrow x{+}1\ \textbf{end} : \textbf{Fun(int) int}}}{\varnothing \vdash [\ \mathit{succ} = \textbf{fun}\ x{:}\textbf{int} \rightarrow x{+}1\ \textbf{end},\ \ \mathit{loc} = \textbf{var}(0)\ ] : ?}}{\varnothing \vdash \textbf{decl}\ c = [\ \mathit{succ} = \textbf{fun}\ x{:}\textbf{int} \rightarrow x{+}1\ \textbf{end},\ \ \mathit{loc} = \textbf{var}(0)\ ]\ \textbf{in}\ c\ \textbf{end} : ?}$$

Tipos Produto (Quiz)

- Qual o tipo da expressão?

$$\textbf{decl}\ c = [\ succ = \textbf{fun}\ x \rightarrow x{+}1\ \textbf{end},\ \ loc = \textbf{var}(0)\ ]\ \textbf{in}\ c\ \textbf{end}$$

$$\dfrac{\dfrac{}{\varnothing \vdash \textbf{fun}\ \ldots : \textbf{Fun(int) int}} \qquad \dfrac{\dfrac{}{\varnothing \vdash 0 : \textbf{int}}}{\varnothing \vdash \textbf{var}(0) : \textbf{Ref(int)}}}{\varnothing \vdash [\ succ = \textbf{fun}\ x{:}\textbf{int} \rightarrow x{+}1\ \textbf{end},\ \ loc = \textbf{var}(0)\ ] : ?}$$

$$\varnothing \vdash \textbf{decl}\ c = [\ succ = \textbf{fun}\ x{:}\textbf{int} \rightarrow x{+}1\ \textbf{end},\ \ loc = \textbf{var}(0)\ ]\ \textbf{in}\ c\ \textbf{end} : ?$$

Tipos Produto (Quiz)

- Qual o tipo da expressão?

$$\textbf{decl } c = \textbf{[} succ = \textbf{fun } x \rightarrow x{+}1 \textbf{ end}, \;\; loc = \textbf{var}(0) \textbf{] in } c \textbf{ end}$$

Resposta: $\textbf{Tuple}(succ : \textbf{Fun(int)int}, loc : \textbf{Ref(int)})$

$$\frac{\varnothing \vdash \textbf{[...]} : \textbf{Tuple}(succ : \textbf{Fun(int)int}, loc : \textbf{Ref(int)})}{\varnothing \vdash \textbf{decl } c = \textbf{[} succ = \textbf{fun } x{:}\textbf{int} \rightarrow x{+}1 \textbf{ end}, \;\; loc = \textbf{var}(0) \textbf{] in } c \textbf{ end} : ?}$$

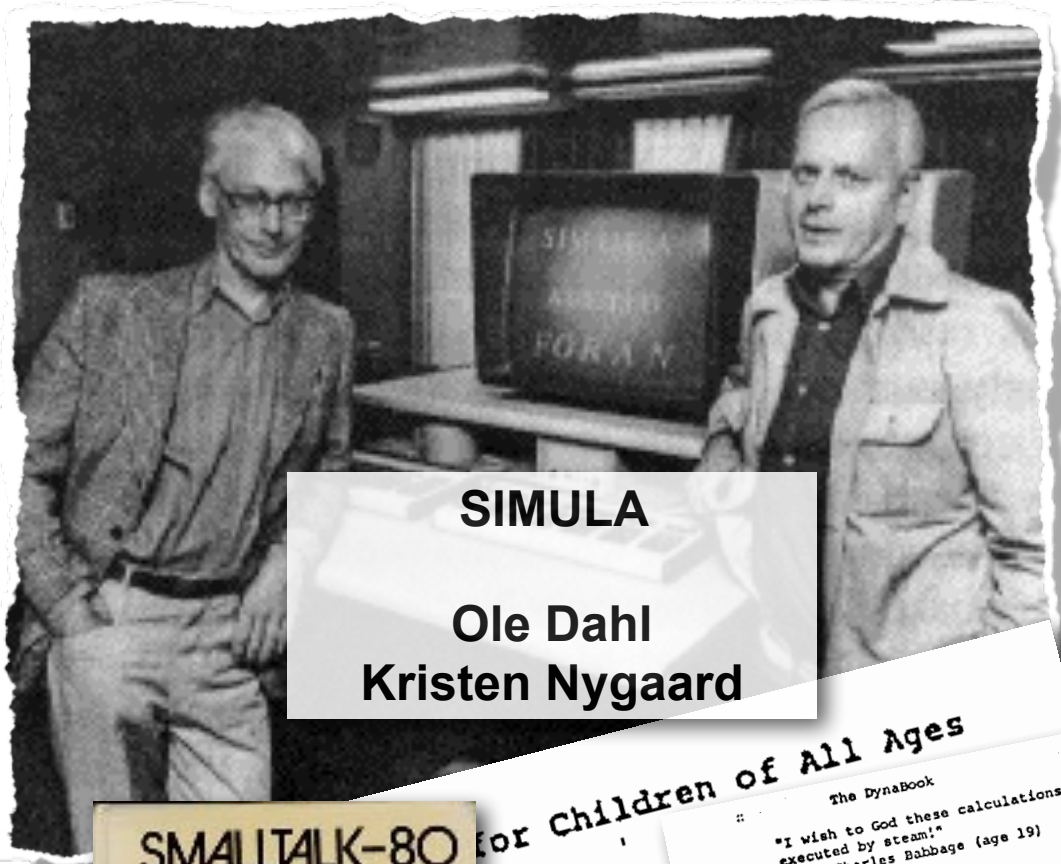
$$c : \text{Tuple(succ:Fun(int)int, loc=Ref(int))} \vdash c : ?$$

Tipos Produto (Quiz)

- A expressão seguinte está bem tipificada?

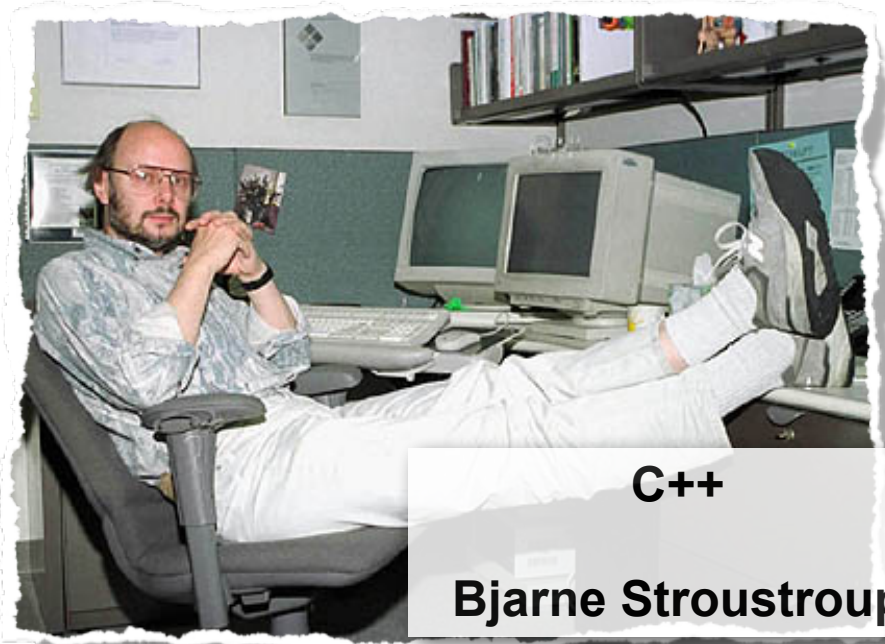
decl c = **[** $succ$ = **fun** $x \rightarrow x+1$ **end**, loc = **var**(0) **]** **in** c.succ(!c.loc) **end**

Objectos e Classes



SIMULA

Ole Dahl
Kristen Nygaard

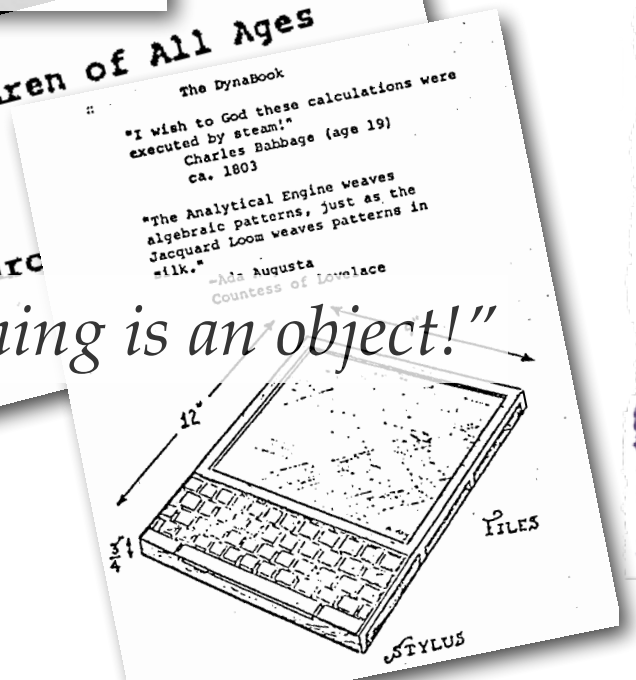


C++

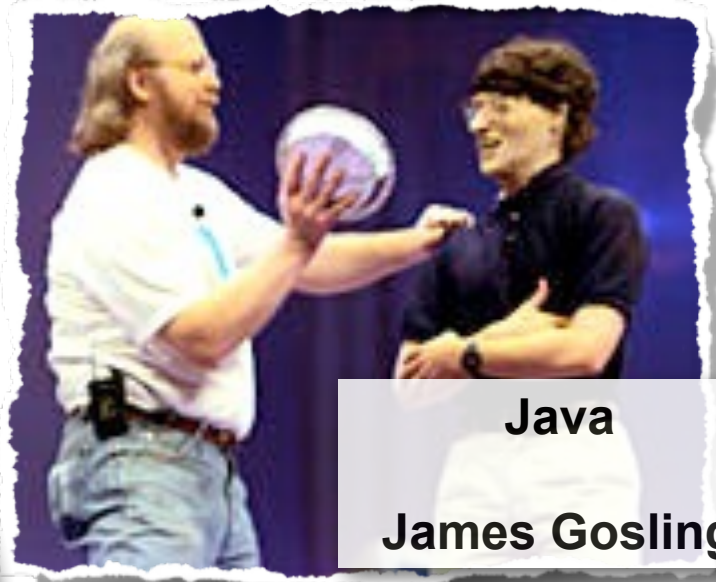
Bjarne Stroustroup

Smalltalk

Adele Goldberg
David Robson



"Everything is an object!"



Java

James Gosling

Simula 67

```
Begin
   Class Glyph;
      Virtual: Procedure print Is Procedure print;;
   Begin
   End;

   Glyph Class Char (c);
      Character c;
   Begin
      Procedure print;
        OutChar(c);
   End;

   Glyph Class Line (elements);
      Ref (Glyph) Array elements;
   Begin
      Procedure print;
      Begin
         Integer i;
         For i:= 1 Step 1 Until UpperBound (elements, 1) Do
            elements (i).print;
         OutImage;
      End;
   End;

   Ref (Glyph) rg;
   Ref (Glyph) Array rgs (1 : 4);

   ! Main program;
   rgs (1):- New Char ('A');
   rgs (2):- New Char ('b');
   rgs (3):- New Char ('b');
   rgs (4):- New Char ('a');
   rg:- New Line (rgs);
   rg.print;
End;
```

$$Z = \sum_{i=1}^{100} \frac{1}{(i+a)^2}$$

```
Real Procedure Sigma (l, m, n, u);
   Name l, u;
   Integer l, m, n; Real u;
Begin
   Real s;
   l:= m;
   While l <= n Do Begin s:= s + u; l:= l + 1;
End;
   Sigma:= s;
End;

Z:= Sigma (i, 1, 100, 1 / (i + a) ** 2);
```

- Baseada no Algol 60
- Objectos, Classes, Subclasses, métodos virtuais, corotinas, simulação de eventos e garbadge collection.
- Call by name

Smaltalk 80

- “Everything is an Object”
- Reflexion mechanism, dynamically typed
- Syntax minimalista
- No real keywords (true, false, nil, self, super, thisContext)
- Construções: message sends, assignment, method return and literais.
- Os programas são editados no próprio ambiente de execução, os sistemas de suporte guardam-se em imagens (cf. máquinas virtuais).

```
3 factorial + 4 factorial between: 10 and: 100
```

```
result := a > b
    ifTrue:[ 'greater' ]
    ifFalse:[ 'less or equal' ]
```

```
| window |
  window := Window new.
  window label: 'Hello'.
  window open
```

```
Object subclass: #MessagePublisher
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Smalltalk Examples'
```

Conceitos base de uma linguagem OO

- *Dynamic Lookup*: Quando uma mensagem é enviada a um objecto (executar uma função sobre um conjunto de dados) o código a ser executado depende da implementação concreta e não de uma ligação estabelecida estaticamente com base nos identificadores do programa.
- *Abstracção* (de dados e funcionalidade): Os detalhes de implementação da funcionalidade e da representação de dados são escondidos dentro de um subprograma com um interface bem determinado.
- *Subtyping*: Se um objecto tem a mesma, e possivelmente mais funcionalidades que outro, pode substituí-lo em qualquer contexto.
- *Inheritance*: A capacidade de utilizar um tipo de objectos para definir outros tipos de objectos.

Conceitos base de uma linguagem OO

- *Dynamic Lookup*: Quando uma mensagem é enviada a um objecto (executar uma função sobre um conjunto de dados) o código a ser executado depende da implementação concreta e não de uma ligação estabelecida estaticamente com base nos identificadores do programa.

```
interface I { void doIt(); }

class A implements I {
  void doIt(){ System.out.println("A");}
}

class B implements I {
  void doIt(){ System.out.println("B");}
}

class C implements I {
  void doIt(){ System.out.println("C");}
}
```

VS

```
#define A 0
#define B 1
#define C 2

typedef struct {
  int kind; ...
} I;

void doIt(I o) {
  switch(o.kind) {
  case A: printf("A");
  case B: printf("B");
  case C: printf("C");
  }
}
```

Conceitos base de uma linguagem OO

- *Abstracção* (de dados e funcionalidade): Os detalhes de implementação da funcionalidade e da representação de dados são escondidos dentro de um subprograma com um interface bem determinado.

```
interface Map<K,T> {
  void put(K key, T element);
  T get(K key);
}

class HashMap implements Dictionary {...}

class TreeMap implements Dictionary {...}

class PairList implements Dictionary {...}
```

Conceitos base de uma linguagem OO

- *Subtyping*: Se um objecto tem a mesma, e possivelmente mais funcionalidades que outro, pode substituí-lo em qualquer contexto.

```
interface I { void m1(); }

interface J extends I { void m2(); }

J j = ...
I i = j;
```

Conceitos base de uma linguagem OO

- *Implementation Inheritance*: A capacidade de utilizar um tipo de objectos para definir outros tipos de objectos.

```
class Point {
  int x, y;

  Point(int x, int y) {...}
  ...
}

class ColouredPoint extends Point {
  ...
}
```

A classe *Counter* em Java (primeira versão)

- Uma classe de objectos "contadores"

```
class Counter implements ICounter {
  int val;
  Counter() { val = 0; }
  void inc() { val = val + 1; }
  int get() { return val; }
}

  ...
  ICounter c = new Counter();
  c.inc();
  c.inc();
  System.out.println(c.get());
```

Objectos na linguagem CORE (com registos)

- Um objecto da classe Counter, representado na linguagem CORE por um registo contendo variáveis para os membros de dados (primeira versão)

```
decl c = [
    val = var(0),
    inc = proc => val := !val + 1 end,
    get = fun => !val end
  ]
in
  c.inc();
  c.inc();
  print(c.get());
end
```

Objectos na linguagem CORE (com registos)

- Um objecto da classe Counter, representado na linguagem CORE por um registo contendo variáveis para os membros de dados (primeira versão)

```
decl c = [
    val = var(0),
    inc = proc => val := !val + 1 end,
    get = fun => !val end
  ]
in
  c.inc();
  c.inc();
  print(c.get());
end
```

o campo val é visível do exterior, é "público"!

val não é visível a partir dos métodos!!!

Objectos na linguagem CORE (com registos)

- Um objecto da classe Counter, representado na linguagem CORE por um registo contendo variáveis para os membros de dados (segunda versão)

```
decl c =
  decl val = var(0) in
  [
    inc = proc => val := !val + 1 end,
    get = fun => !val end
  ]
  end
in
  c.inc();
  c.inc();
  print(c.get());
end
```

Objectos na linguagem CORE (com registos)

- Um objecto da classe Counter, representado na linguagem CORE por um registo contendo variáveis para os membros de dados (segunda versão)

```
decl c =
  decl val = var(0) in
  [
    inc = proc => val := !val + 1 end,
    get = fun => !val end
  ]
  end
in
  c.inc();
  c.inc();
  print(c.get());
end
```

campo val "privado"!

O âmbito de val inclui os métodos do objecto.

Objectos na linguagem CORE (com registos)

- Um objecto da classe Counter, representado na linguagem CORE por um registo contendo variáveis para os membros de dados (segunda versão)

```
decl c =
  decl val = var(0) in
  [
    inc = proc => val := !val + 1 end,
    get = fun => !val end
  ]
  end
in
  c.inc();
  c.inc();
  print(c.get());
end
```

```
E0
val - l0
```

```
E1

c - { inc = (, val := !val+1, E0 ),
      get = (, !val, E0 ) }
```

Classes na linguagem CORE (com registos)

- A classe Counter é representada na linguagem CORE por uma função geradora de objectos (primeira versão)

```
decl
  Counter = fun =>
    decl val = var(0) in
    [
      inc = proc => val := !val + 1 end,
      get = fun => !val end
    ]
    end
  end
in
  decl c = Counter() in
    c.inc();
    c.inc();
    print(c.get());
  end
end
```

A classe *Counter* em Java (segunda versão)

- Uma classe de objectos "contadores" com constructores...

```
class Counter implements ICounter {
  int val;
  Counter(int n) { val = n; }
  void inc() { val = val + 1; }
  int get() { return val; }
}

  ...
  Counter c = new Counter(0);
  Counter d = new Counter(2);
  c.inc();
  d.inc();
  System.out.println(c.get()+d.get());
```

Classes na linguagem CORE (com registos)

- A classe Counter, é representada na linguagem CORE por uma função geradora de objectos (segunda versão)

```
decl
  Counter = fun n =>
    decl val = var(n) in
    [
      inc = proc => val := !val + 1 end,
      get = fun => !val end
    ]
    end
  end
in
  decl c = Counter(0)
       d = Counter(2) in
    c.inc();
    d.inc();
    print(c.get()+d.get());
  end
end
```

Classes na linguagem CORE (com registos)

- Em geral, os métodos de um objecto devem poder chamar-se uns aos outros recursivamente. Nesta codificação não, porquê? dup chama inc...

```
decl
  Counter = fun n =>
    decl val = var(n) in
    [
      inc = proc => val := !val + 1 end,
      get = fun => !val end
      dup = proc => inc(); inc() end
    ]
    end
  end
in
  decl c = Counter(0) in
    c.dup();
    ...
  end
end
```

inc não é visível!!!

Classes na linguagem CORE (com registos)

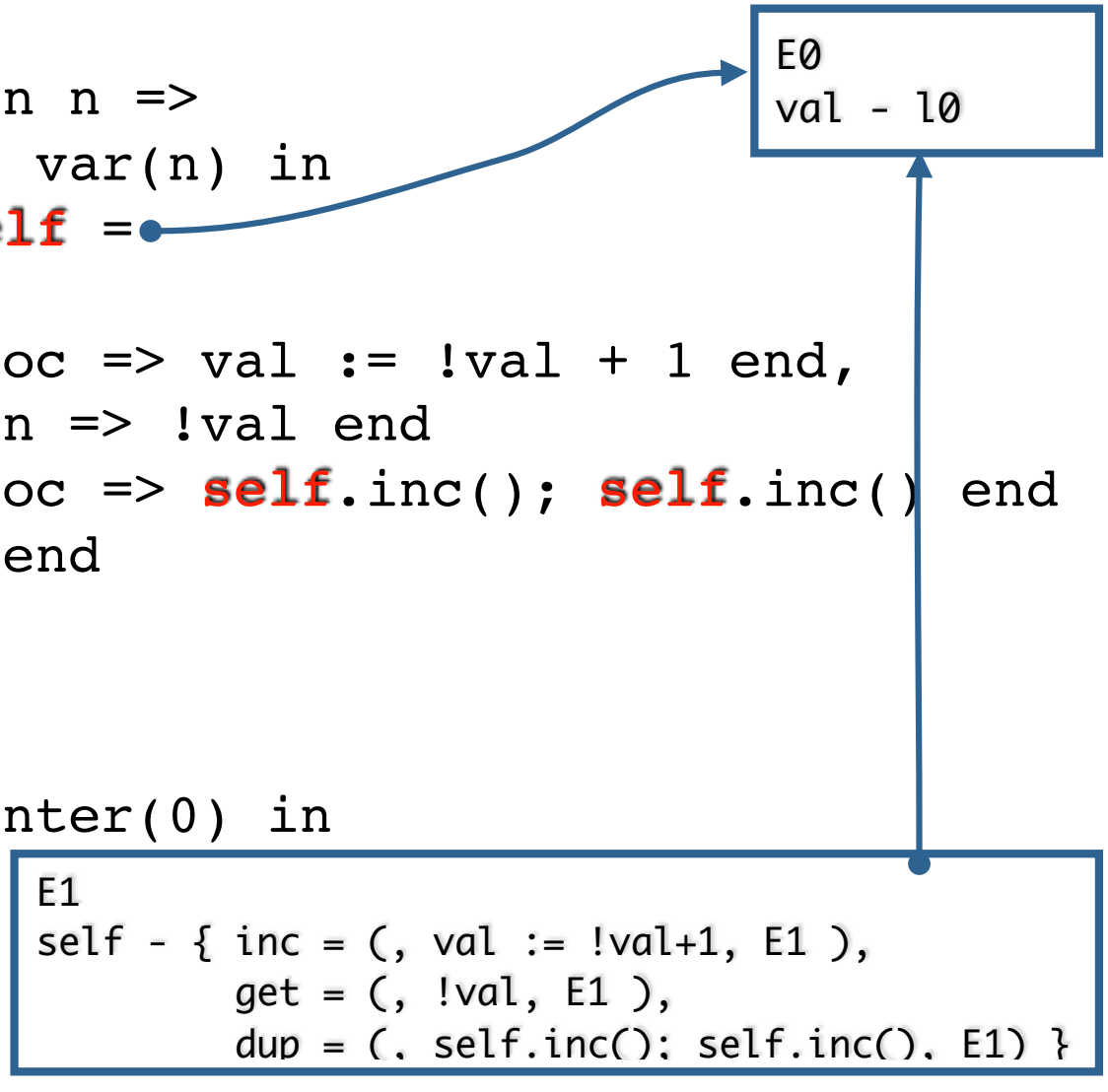
- Em geral, os métodos de um objecto devem poder chamar-se uns aos outros recursivamente. Nesta codificação não, porquê? dup chama inc...

```
decl
  Counter = fun n =>
    decl val = var(n) in
    declrec self =
    [
      inc = proc => val := !val + 1 end,
      get = fun => !val end
      dup = proc => self.inc(); self.inc() end
    ] in self end
    end
  end
in
  decl c = Counter(0) in
    c.dup();
    ...
  end
end
```

Classes na linguagem CORE (com registos)

- A solução consiste na definição de um registo recursivo. Note-se que as referências ao nome "self" apenas ocorrem no corpo de abstracções.

```
decl
  Counter = fun n =>
    decl val = var(n) in
    declrec self =
    [
      inc = proc => val := !val + 1 end,
      get = fun => !val end
      dup = proc => self.inc(); self.inc() end
    ] in self end
    end
  end
in
  decl c = Counter(0) in
    c.dup();
    ...
  end
end
```



Uma sintaxe para Classes

- Classes

```
class
  id1 := E1
  ...
  idp := Ep
methods
  M1
  ...
  Mn
end
```

- Métodos

```
proc id() = C
fun  id() = E
```

```
decl Counter =
  class
    val := 0
  methods
    proc inc() = val := !val + 1 end
    fun get() = !val end
    proc dup() = self.inc(); self.inc() end
  end
in
  decl c = new Counter()
    in c.inc(); c.dup(); print(c.get())
  end
end
```

Tradução para Core

- Construções novas podem ser implementadas por tradução nas construções já existentes na linguagem base.

```
decl Counter =
  class
    val := 0
  methods
    proc inc() = v
    fun get() = !v
    proc dup() = s
  end
in
  decl c = new Cou
    in c.inc(); c.
  end
end
```

```
decl Counter =
  fun =>
    decl val = var(0) in
    declrec self =
    [
      inc = proc => val := !val + 1 end,
      get = fun => !val end
      dup = proc => self.inc(); self.inc() end
    ] in self end
  end
in
  decl c = Counter()
    in c.inc(); c.dup(); print(c.get())
  end
end
```

Tipos para Objectos

Uma sintaxe para Classes

- Classes

```
class
  id1 := E1
  ...
  idp := Ep
methods
  M1
  ...
  Mn
end
```

- Métodos

```
proc id() = C
fun  id() = E
```

```
decl Counter =
  class
    val := 0
  methods
    proc inc() = val := !val + 1 end
    fun get() = !val end
    proc dup() = self.inc(); self.inc() end
  end
in
  decl c = new Counter()
    in c.inc(); c.dup(); print(c.get())
  end
end
```

Tipos para Objectos (1)

- As únicas operações disponíveis nos objectos são as chamadas de métodos (cf. selecção de campo).

$$\mathrm{Obj}(\mathrm{id}_1{:}\mathcal{T}_1, \ldots, \mathrm{id}_n{:}\mathcal{T}_n)$$

- Tipo dos objectos com métodos $\mathrm{id}_1, \ldots, \mathrm{id}_n$, respectivamente dos tipos funcionais $\mathcal{T}_1, \ldots, \mathcal{T}_n$.

$$\frac{Env \vdash \mathrm{E} : \mathrm{Obj}(\mathrm{id}_1{:}\mathcal{T}_1, \ldots, \mathrm{id}_n{:}\mathcal{T}_n) \qquad \mathcal{T}_j = \mathrm{Fun}(\mathcal{V})\mathcal{R} \qquad Env \vdash \mathrm{F} : \mathcal{V}}{Env \vdash \mathrm{E}.\mathrm{id}_j(\mathrm{F}){:}\mathcal{R}} \text{(InvokeF)}$$

Tipos para Objectos (1)

- As únicas operações disponíveis nos objectos são as chamadas de métodos (cf. selecção de campo).

$$\mathrm{Obj}(\mathrm{id}_1{:}\mathcal{T}_1, \ldots, \mathrm{id}_n{:}\mathcal{T}_n)$$

- Tipo dos objectos com métodos $\mathrm{id}_1, \ldots, \mathrm{id}_n$, respectivamente dos tipos funcionais $\mathcal{T}_1, \ldots, \mathcal{T}_n$.

$$\frac{Env \vdash \mathrm{E} : \mathrm{Obj}(\mathrm{id}_1{:}\mathcal{T}_1, \ldots, \mathrm{id}_n{:}\mathcal{T}_n) \qquad \mathcal{T}_j = \mathrm{Proc}(\mathcal{V}) \qquad Env \vdash \mathrm{F} : \mathcal{V}}{Env \vdash \mathrm{E}.\mathrm{id}_j(\mathrm{F})\ \mathbf{ok}}\ \textbf{(InvokeP)}$$

Tipos para Objectos (1)

- A única operações disponível numa classe é a instanciação de novos objectos.

$$\text{Class}(\text{id}_1{:}\mathcal{T}_1, \ldots, \text{id}_n{:}\mathcal{T}_n)$$

- Tipo das classes que geram objectos com o tipo

$$\text{Obj}(\text{id}_1{:}\mathcal{T}_1, \ldots, \text{id}_n{:}\mathcal{T}_n)$$

$$\frac{Env \vdash \text{E} : \text{Class}(\text{id}_1{:}\mathcal{T}_1, \ldots, \text{id}_n{:}\mathcal{T}_n)}{Env \vdash \textbf{new}\ \text{E}(): \text{Obj}(\text{id}_1{:}\mathcal{T}_1, \ldots, \text{id}_n{:}\mathcal{T}_n)}\ \textbf{(new)}$$

Tipos para Classes (1)

- A única operações disponível numa classe é a instanciação de novos objectos.

$$\text{Class}(\text{id}_1{:}\mathcal{T}_1, \ldots, \text{id}_n{:}\mathcal{T}_n)$$

- Tipo das classes que geram objectos com o tipo

$$\text{Obj}(\text{id}_1{:}\mathcal{T}_1, \ldots, \text{id}_n{:}\mathcal{T}_n)$$

$$\frac{\begin{array}{l} Env \vdash \text{E} : \mathcal{T} \\ Env, v{:}\text{Ref}[\mathcal{T}], \text{self}{:}?, x_1{:}\,\mathcal{T}_1 \vdash \text{B}_1 : \text{R}_1 \\ \ldots \\ Env, v{:}\text{Ref}[\mathcal{T}], \text{self}{:}?, x_n{:}\mathcal{T}_n \vdash \text{B}_n : \text{R}_n \end{array}}{\begin{array}{l} Env \vdash \textbf{class}\ \text{v} := \text{E} \\ \qquad \textbf{methods fun}\ \text{m}_1(x_1{:}\mathcal{T}_1) = \text{B}_1 \ \ldots\ \textbf{fun}\ \text{m}_n(x_n{:}\mathcal{T}_n) = \text{B}_n\ \textbf{end}{:} \\ \qquad\quad \text{Class}(\text{m}_1{:}\text{Fun}(\mathcal{T}_1)\text{R}_1, \ldots, \text{m}_n{:}\text{Fun}(\mathcal{T}_n)\text{R}_n) \end{array}}$$

Tipos para Classes (1)

- A única operações disponível numa classe é a instanciação de novos objectos.

$$\text{Class}(\text{id}_1{:}\mathcal{T}_1, \ldots, \text{id}_n{:}\mathcal{T}_n)$$

- Tipo das classes que geram objectos com o tipo

$$\text{Obj}(\text{id}_1{:}\mathcal{T}_1, \ldots, \text{id}_n{:}\mathcal{T}_n)$$

É necessário antecipar o tipo de "self", antes de validar o corpo dos métodos ! …

$$\frac{\begin{array}{l} Env \vdash \text{E} : \mathcal{T} \\ Env, v{:}\text{Ref}[\mathcal{T}], \text{self:?}\ x_1{:}\ \mathcal{T}_1 \vdash \text{B}_1 : \text{R}_1 \\ \ldots \\ Env, v{:}\text{Ref}[\mathcal{T}], \text{self:?}, x_n{:}\mathcal{T}_n \vdash \text{B}_n : \text{R}_n \end{array}}{\begin{array}{l} Env \vdash \textbf{class}\ \text{v} := \text{E} \\ \quad \textbf{methods fun}\ \text{m}_1(x_1{:}\mathcal{T}_1) = \text{B}_1 \ \ldots\ \textbf{fun}\ \text{m}_n(x_n{:}\mathcal{T}_n) = \text{B}_n\ \textbf{end}{:} \\ \quad\quad \text{Class}(\text{m}_1{:}\text{Fun}(\mathcal{T}_1)\text{R}_1, \ldots, \text{m}_n{:}\text{Fun}(\mathcal{T}_n)\text{R}_n) \end{array}}$$

Tipos para Classes (1)

- O identificador self denota um objecto com o interface de um objecto da classe. Podemos chegar ao tipo completo do objecto recolhendo todos tipos dos métodos da classe.

$$J \triangleq (m_1{:}Fun(\mathcal{T}_1)R_1, \ldots, m_n{:}Fun(\mathcal{T}_n)R_n)$$

$$\frac{\begin{array}{l} Env \vdash E : \mathcal{T} \\ Env, v{:}\mathrm{Ref}[\mathcal{T}], \mathrm{self}{:}\mathrm{Obj}(J), x_1{:}\mathcal{T}_1 \vdash B_1 : R_1 \\ \ldots \\ Env, v{:}\mathrm{Ref}[\mathcal{T}], \mathrm{self}{:}\mathrm{Obj}(J), x_n{:}\mathcal{T}_n \vdash B_n : R_n \end{array}}{\begin{array}{l} Env \vdash \textbf{class}\ v := E \\ \qquad \textbf{methods}\ \textbf{fun}\ m_1(x_1{:}\mathcal{T}_1) = B_1 \ \ldots\ \textbf{fun}\ m_n(x_n{:}\mathcal{T}_n) = B_n\ \textbf{end}{:}\ \mathrm{Class}(J) \end{array}}$$

Tipos para Classes (1)

- O identificador self denota um objecto com o interface de um objecto da classe. Podemos chegar ao tipo completo do objecto recolhendo todos tipos dos métodos da classe.

$$J \triangleq (m_1{:}Fun(\mathcal{T}_1)R_1, \ldots, m_n{:}$$

O tipo de "self" pode ser obtido a partir das declarações de tipos nos métodos.

$$\frac{\begin{array}{l} Env \vdash E : \mathcal{T} \\ Env, v{:}\mathrm{Ref}[\mathcal{T}], \mathrm{self}{:}\mathrm{Obj}(J), x_1{:}\mathcal{T}_1 \vdash B_1 : R_1 \\ \ldots \\ Env, v{:}\mathrm{Ref}[\mathcal{T}], \mathrm{self}{:}\mathrm{Obj}(J), x_n{:}\mathcal{T}_n \vdash B_n : R_n \end{array}}{\begin{array}{l} Env \vdash \mathbf{class}\ v := E \\ \qquad \mathbf{methods}\ \mathbf{fun}\ m_1(x_1{:}\mathcal{T}_1) = B_1 \ \ldots\ \mathbf{fun}\ m_n(x_n{:}\mathcal{T}_n) = B_n\ \mathbf{end}{:}\ \mathrm{Class}(J) \end{array}}$$

Tipos para Classes (1)

- Esta regra de tipificação permite tipificar as referências ao identificador self. No entanto não permite tipificar classes que referem objectos da mesma classe.

```
decl Counter =
       class
         val := 0
       methods
         proc inc() = val := !val + 1 end
         fun get():int = !val end
         fun equal(c:?):bool = (c.get() = self.get()) end
       end
in …
```

Tipos para Classes (1)

- Esta regra de tipificação permite tipificar as referências ao identificador self. No entanto não permite tipificar classes que referem objectos da mesma classe. Só é possível tipificar essas classes com tipos recursivos.

```
decl Counter =
       class
         val := 0
       methods
         proc inc() = val := !val + 1 end
         fun get():int = !val end
         fun equal(c:T) = (c.get() = self.get()) end
       end
in …
```

$$T \triangleq Obj(inc: Proc(), get : Fun()int, equal : Fun(T)bool)???$$

Quiz

- Qual é o ambiente de tipificação da expressão self.inc() ?

```
decl Counter =
  class
    val := 0
  methods
    proc inc() = val := !val + 1 end
    fun get() = !val end
    proc dup() = self.inc(); self.inc() end
  end
in
  decl c = new Counter()
    in c.inc(); c.dup(); print(c.get())
  end
end
```

val:**Ref**[**int**], self : **Obj**(*inc*:**Proc**(), *get*:**Fun**()**int**, *dup*:**Proc**()) ⊢ self.inc() ok

Quiz

- Qual é o ambiente de tipificação da expressão self.inc() ?

```
decl Counter =
  class
    val := 0
  methods
    proc inc() = val := !val + 1 end
    fun get() = !val end
    proc dup() = self.inc(); self.inc() end
  end
in
  decl c = new Counter()
    in c.inc(); c.dup(); print(c.get())
  end
end
```

$$\varnothing \vdash \text{class ... end: } \mathbf{Class}(\, \mathit{inc}{:}\mathbf{Proc}(), \mathit{get}{:}\mathbf{Fun}()\mathbf{int}, \mathit{dup}{:}\mathbf{Proc}() \,)$$

Quiz

- Qual é o ambiente de tipificação da expressão self.inc() ?

```
decl Counter =
  class
    val := 0
  methods
    proc inc() = val := !val + 1 end
    ...
  end
```

$$\dfrac{\dfrac{}{\varnothing \vdash 0 : \mathbf{int}} \qquad \dfrac{\dfrac{}{\mathrm{val}{:}\mathrm{Ref}[\mathbf{int}],\ \mathrm{self}{:}\ \mathbf{Obj}(\ \ldots\ ) \vdash \mathrm{val} : \mathrm{Ref}[\mathbf{int}]} \qquad \dfrac{}{\ldots \vdash\ !\mathrm{val} + 1 : \mathbf{int}}}{\mathrm{val}{:}\mathrm{Ref}[\mathbf{int}],\ \mathrm{self}{:}\ \mathbf{Obj}(\ \ldots\ ) \vdash \mathrm{val} := !\mathrm{val} + 1\ \mathbf{ok}}}{\varnothing \vdash \mathrm{class}\ \ldots\ \mathrm{end}{:}\ \mathbf{Class}(\ \mathit{inc}{:}\mathbf{Proc}(), \mathit{get}{:}\mathbf{Fun}()\mathbf{int}, \mathit{dup}{:}\mathbf{Proc}()\ )}$$

Quiz

- Qual é o ambiente de tipificação da expressão self.inc() ?

```
decl Counter =
  class
    val := 0
  methods
    ...
    fun get() = !val end
    ...
  end
```

$$\frac{\varnothing \vdash 0 : \mathbf{int} \qquad \dfrac{\text{val:Ref[}\mathbf{int}\text{], self: }\mathbf{Obj}(\ \ldots\ ) \vdash \text{val} : \text{Ref[}\mathbf{int}\text{]}}{\text{val:Ref[}\mathbf{int}\text{], self: }\mathbf{Obj}(\ \ldots\ ) \vdash \text{!val} : \mathbf{int}\ \mathbf{ok}}}{\varnothing \vdash \text{class ... end: } \mathbf{Class}(\ \mathit{inc}\text{:}\mathbf{Proc}(), \mathit{get}\text{:}\mathbf{Fun}()\mathbf{int}, \mathit{dup}\text{:}\mathbf{Proc}()\ )}$$

Quiz

- Qual é o ambiente de tipificação da expressão self.inc() ?

```
decl Counter =
  class
    val := 0
  methods
    ...
    proc dup() = self.inc(); self.inc() end
  end
```

$$\dfrac{\varnothing \vdash 0 : \textbf{int} \qquad \dfrac{\text{val:Ref[}\textbf{int}\text{], self: }\textbf{Obj}(\ \textit{inc}\text{:}\textbf{Proc}(), \ldots\ ) \vdash \text{ self.inc() } \textbf{ok}}{\text{val:Ref[}\textbf{int}\text{], self: }\textbf{Obj}(\ \ldots\ ) \vdash \text{ self.inc(); self.inc() } \textbf{ok}}}{\varnothing \vdash \text{class ... end: } \textbf{Class}(\ \textit{inc}\text{:}\textbf{Proc}(), \textit{get}\text{:}\textbf{Fun}()\textbf{int}, \textit{dup}\text{:}\textbf{Proc}()\ )}$$

Quiz

- Quais são os tipos e os ambientes de tipificação da expressões Counter, new Counter() e c.get()?

```
decl Counter =
  class
    val := 0
  methods
    proc inc() = val := !val + 1 end
    fun get() = !val end
    proc dup() = self.inc(); self.inc() end
  end
in
  decl c = new Counter()
  in c.inc(); c.dup(); print(c.get())
  end
end
```

Env0 ⊢ Counter : **Class**(*inc*:**Proc**(), *get*:**Fun**()**int**, *dup*:**Proc**())

Env1 ⊢ **new** Counter():**Obj**(*inc*:**Proc**(), *get*:**Fun**()**int**, *dup*:**Proc**())

Env1 ⊢ c.get(): **int**

Tipos para Classes (2)

- Esta regra de tipificação permite tipificar as referências ao identificador self. No entanto não permite tipificar classes que referem objectos da mesma classe. Só é possível tipificar essas classes com tipos recursivos.

```
decl Counter =
       class
         val := 0
       methods
         proc inc() = val := !val + 1 end
         fun get():int = !val end
         fun equal(c:T) = (c.get() = self.get()) end
       end
in …
```

$$T \triangleq \text{Obj}(\text{inc: Proc}(), \text{get} : \text{Fun}()\text{int}, \text{equal} : \text{Fun}(T)\text{bool})$$

Tipos para Classes (2)

- Esta regra de tipificação permite tipificar as referências ao identificador self. No entanto não permite tipificar classes que referem objectos da mesma classe. Só é possível tipificar essas classes com tipos recursivos.

```
declrec Counter =
        class
          val := 0
        init(v:int)
          val := v
        methods
          proc inc() = val := !val + 1 end
          fun get()int = !val end
          fun clone()T = new Counter(!val) end
        end
in …
```

$$T \triangleq \text{Obj}(\text{inc: Proc(), get : Fun()int, equal : Fun()}T)$$

Tipos Recursivos

- Os tipo recursivos permitem especificar estruturas de tipos “infinitas”.

$$T \triangleq Obj(inc: Proc(), get : Fun()int, clone : Fun()$$
$$Obj(inc: Proc(), get : Fun()int, clone : Fun()$$
$$Obj(inc: Proc(), get : Fun()int, clone : Fun()$$
$$Obj(inc: Proc(), get : Fun()int, clone : Fun()$$
$$Obj(inc: Proc(), get : Fun()int, clone : Fun()$$
$$Obj(inc: Proc(), get : Fun()int, clone : Fun()$$
$$\ldots$$
$$)$$
$$)$$

$$T \triangleq Obj(X)(inc: Proc(), get : Fun()int, clone : Fun()X)$$

$$)$$
$$)$$
$$)$$

X é uma variável de tipo: representa o tipo do self.

Tipos Recursivos

- Os tipo recursivos permitem especificar estruturas de tipos “infinitas”.

$$\mathrm{T} \triangleq \mathrm{Obj}(\mathcal{X})(\mathrm{inc}\colon \mathrm{Proc}(), \mathrm{get} : \mathrm{Fun}()\mathrm{int}, \mathrm{clone} : \mathrm{Fun}()\mathcal{X})$$

Os tipo recursivos satisfazem a seguinte igualdade estrutural, chamada “desdobramento” (*unfolding*).

$$\mathrm{Obj}(\mathcal{X})(\mathrm{id}\colon\mathrm{T}) \equiv \mathbf{Obj}(\mathbf{subst}(\mathcal{X}, \mathbf{Obj}(\mathcal{X})(\mathrm{id}\colon\mathrm{T}), \mathrm{id}\colon\mathrm{T}))$$

- Em $\mathrm{Obj}(\mathcal{X})(\mathrm{id}\colon\mathrm{T})$ a variável $\mathcal{X}$ é ligada em $\mathrm{id}\colon\mathrm{T}$
- $\mathbf{subst}(\mathcal{X}, \mathcal{T}, \mathrm{J})$ denota a substituição de todas as ocorrências livres de $\mathcal{X}$ em J pelo tipo $\mathcal{T}$.

Tipos para Classes (1)

- O identificador self denota um objecto com o interface de um objecto da classe. Podemos chegar ao tipo completo do objecto recolhendo todos tipos dos métodos da classe. Para além disso, é um tipo recursivo

$$J(Self) \triangleq (m_1{:}Fun(\mathcal{T}_1)R_1, \ldots, m_n{:}Fun(\mathcal{T}_n)R_n)$$

$$\frac{\begin{array}{l} Env \vdash E : \mathcal{T} \\ Env, v{:}\mathrm{Ref}[\mathcal{T}], \mathrm{self}{:}\mathrm{Obj}(Self)(\,J(Self)\,), x_1{:}\,\mathcal{T}_1 \vdash B_1 : R_1 \\ \ldots \\ Env, v{:}\mathrm{Ref}[\mathcal{T}], \mathrm{self}{:}\,\mathrm{Obj}(Self)(\,J(Self)\,), x_n{:}\mathcal{T}_n \vdash B_n : R_n \end{array}}{\begin{array}{l} Env \vdash \textbf{class}\ v := E \\ \qquad \textbf{methods fun}\ m_1(x_1{:}\mathcal{T}_1) = B_1 \ \ldots\ \textbf{fun}\ m_n(x_n{:}\mathcal{T}_n) = B_n\ \textbf{end}{:}\ \mathrm{Class}(\,J\,) \end{array}}$$

Tipos para Classes (2)

- Usando tipos recursivos, podemos tipificar uma grande classe de programas.

$$CType \triangleq Obj(Self)(\ inc{:}Proc(),\ get{:}Fun()int,\ clone{:}Fun()\ Self\ )$$

```
declrec Counter =
        class
          val := 0
        init(v:int)
          val := v
        methods
          proc inc() = val := !val + 1 end
          fun get()int = !val end
          fun equal(c:CType):bool = c.get() = self.get() end
          fun clone() CType = new Counter(!val) end
        end
in …
```

Tipos para Classes (2)

- Usando tipos recursivos, podemos tipificar uma grande classe de programas.

$$CType \triangleq Obj(Self)(\ inc{:}Proc(),\ get{:}Fun()int,\ clone{:}Fun()\ Self\ )$$

```
declrec Counter =
       class(CType)
         val := 0
        init(v:int)
         val := v
       methods
         proc inc() = val := !val + 1 end
         fun get()int = !val end
         fun equal(c:CType):bool = c.get() = self.get() end
         fun clone() CType = new Counter(!val) end
       end
in …
```

Tipos para Classes (2)

- Usando tipos recursivos, podemos tipificar uma grande classe de programas.

$$CType \triangleq Obj(Self)(\ inc{:}Proc(),\ get{:}Fun()int,\ clone{:}Fun()\ Self\ )$$

```
declrec Counter =
       class(Counter)
         val := 0
        init(v:int)
         val := v
       methods
         proc inc() = val := !val + 1 end
         fun get()int = !val end
         fun equal(c:Counter):bool = c.get() = self.get() end
         fun clone() Counter = new Counter(!val) end
       end
in …
```

Tipos para Classes (2)

- Usando tipos recursivos, podemos tipificar uma grande classe de programas.

```
interface CType {
  void inc();
  int get();
  bool equal(CType c);
  CType clone();
}

class Counter {
  private int val = 0;
  Counter (int c) { val = c; }
  public void inc() { val = val + 1; }
  public int get() { return val; }
  public bool equal(CType c) { return (c.get() == this.get()); }
  public CType clone() { return new Counter(val); }
}
```

Subtipificação

- Considere os tipos

$$\text{Point} \triangleq \text{Obj}(\mathit{getx}\text{:Fun()int}, \mathit{gety}\text{:Fun()int})$$

$$\text{ColorPoint} \triangleq \text{Obj}(\mathit{getx}\text{:Fun()int}, \mathit{gety}\text{:Fun()int}, \mathit{setc}\text{:Proc(C)})$$

- e a declaração de função

```
decl
  norm = fun p:Point =>
           sqrt(p.getx()*p.getx() + p.gety()*p.gety())
         end
in
  ...
end
```

- **Intuitivamente, a função `norm` também pode ser aplicada seguramente a valores de tipo ColorPoint.**

Subtipificação

- Podemos observar que sempre que dois tipos de objecto T1 e T2 tais que

  $T1 \triangleq Obj(m1:T1, \ldots, mn:Tn)$

  $T2 \triangleq Obj(m1:T1, \ldots, mn:Tn, n1:R1, \ldots, pk:Rk)$

  todo o objecto de tipo T2 pode ser usado seguramente em qualquer contexto onde é "esperado" um objecto de tipo T1.

- Um objecto de tipo ColorPoint pode ser sempre usado em vez de um objecto de tipo Point.

- Os métodos adicionais não são utilizados, podendo ser ignorados.

- O tipo T2 é mais específico que T1.

Subtipificação

- A relação "mais específico que" é capturada formalmente por uma relação de subtipificação.
- Sempre que

$$T1 \triangleq Obj(m1{:}T1, \ldots, mn{:}Tn)$$

$$T2 \triangleq Obj(m1{:}T1, \ldots, mn{:}Tn, n1{:}R1, \ldots, pk{:}Rk)$$

podemos afirmar uma asserção de subtipificação da forma

$$T2 <: T1$$

- Por exemplo, a asserção seguinte é válida.

$$ColorPoint <: Point$$

Subtipificação

- A relação de subtipificação <: pode ser definida através de um conjunto de regras de inferência, tal como as relações de tipificação já estudadas.

$$\text{Env} \vdash T <: T \qquad \textbf{(Reflexivity)}$$

$$\text{Env} \vdash \text{Obj}(m_1{:}T_1, \ldots, m_n{:}T_n, n_1{:}R_1, \ldots, p_k{:}R_k) <: \text{Obj}(m_1{:}T_1, \ldots, m_n{:}T_n) \qquad \textbf{(Object)}$$

$$\text{Env} \vdash \text{Class}(m_1{:}T_1, \ldots, m_n{:}T_n, n_1{:}R_1, \ldots, p_k{:}R_k) <: \text{Class}(m_1{:}T_1, \ldots, m_n{:}T_n) \qquad \textbf{(Class)}$$

Subtipificação

- A relação de subtipificação pode ser usada na relação de tipificação, através da regra geral chamada "regra da promoção" (subsumption).

$$\frac{\text{Env} \vdash \text{E} : \text{T} \quad \text{T} <: \text{U}}{\text{Env} \vdash \text{E} : \text{U}} \qquad \textbf{(Subsumption)}$$

- Usando esta regra, podemos por exemplo derivar

$$\frac{\text{Env} \vdash \text{o} : \text{ColorPoint} \quad \text{ColorPoint} <: \text{Point}}{\text{Env} \vdash \text{o} : \text{Point}}$$

Subtipificação (1)

- Alternativamente, a promoção de tipo pode ser incorporada nas regras de função e de método:

$$\frac{Env \vdash N : V \quad Env \vdash M : Fun(T\ )U \quad V <: T}{Env \vdash callf(\ M, N\ ) : U} \textbf{(subcall)}$$

$$\frac{Env \vdash E : Obj(id_1{:}T_1, \ldots, id_n{:}T_n) \quad T_j = Fun(U\ )R \qquad Env \vdash F: V \quad V <: U}{Env \vdash E.\ id_j\ (F): R} \textbf{(subinvokef)}$$

- Com a regra (subcall), é possível validar a chamada da função norm com um ColorPoint .

$$\frac{Env' \vdash p: ColorPoint \quad Env' \vdash norm:Fun(Point)int}{Env' \vdash norm(p) : int}$$

Subtipificação (2)

- Considere os tipos

$$\text{Look} \triangleq \text{Obj}(\text{get:Fun}()\text{Point})$$
$$\text{Cell} \triangleq \text{Obj}(\text{get:Fun}()\text{Point}, \text{set:Proc}(\text{Point}))$$
$$\text{ColCell} \triangleq \text{Obj}(\text{get:Fun}()\,\text{ColorPoint}, \text{set:Proc}(\text{ColorPoint}))$$

Temos certamente $\text{Cell} <: \text{Look}$ [porquê?]

Será que também se pode ter $\text{ColCell} <: \text{Look}$?

```
decl
  extractx = fun p:Look => p.get().getx() end
in
  extractx(cel)
end
```

$\text{Env} \vdash \text{cell} : \text{Cell}$

Subtipificação (2)

- Considere os tipos

$$\text{Look} \triangleq \text{Obj}(\text{get:Fun}()\text{Point})$$
$$\text{Cell} \triangleq \text{Obj}(\text{get:Fun}()\text{Point}, \text{set:Proc}(\text{Point}))$$
$$\text{ColCell} \triangleq \text{Obj}(\text{get:Fun}()\,\text{ColorPoint}, \text{set:Proc}(\text{ColorPoint}))$$

Temos certamente $\text{Cell} <: \text{Look}$ [porquê?]

Será que também se pode ter $\text{ColCell} <: \text{Look}$?

```
decl
  extractx = fun p:Look => p.get().getx() end
in
  extractx(colcell)
end
```

$\text{Env} \vdash \text{colcell} : \text{Cell}$

Subtipificação (2)

- Podemos observar que sempre que temos dois tipos de objecto T1 e T2 tais que

$$T1 \triangleq Obj(m_1:Fun(U_1)T_1, \ldots, m_n:Fun(U_n)T_n)$$

$$T2 \triangleq Obj(m_1:Fun\ (U_1)R_1, \ldots, m_n:Fun(U_n)R_n)$$

  e $R_i <: T_i$ para todo o i=1..n,

  todo o objecto de tipo T2 pode ser usado seguramente em qualquer contexto onde é "esperado" um objecto de tipo T1 (T2 <: T1).

- Um objecto de tipo ColCell pode ser usado em vez de um objecto de tipo Look.

Subtipificação (2)

- Pode-se então considerar as regras de validação seguintes para objectos e classes:

$$\frac{R_1 <: T_1 \quad \ldots \quad R_n <: T_n}{\begin{array}{c} Env \vdash Obj(m_1{:}Fun(U_1)R_1, \ldots, m_n{:}Fun(U_n)R_n, n_1{:}S_1, \ldots, p_k{:}S_k) \\ <: Obj(m_1{:}Fun(U_1)T_1, \ldots, m_n{:}Fun(U_n)T_n) \end{array}}$$

$$\frac{R_1 <: T_1 \quad \ldots \quad R_n <: T_n}{\begin{array}{c} Env \vdash Class(m_1{:}Fun(U_1)R_1, \ldots, m_n{:}Fun(U_n)R_n, n_1{:}S_1, \ldots, p_k{:}S_k) \\ <: Class(m_1{:}Fun(U_1)T_1, \ldots, m_n{:}Fun(U_n)T_n) \end{array}}$$

Subtipificação (2)

- Considere os tipos

Put ≜ Obj(set:Proc(Point))

Cell ≜ Obj(get:Fun()Point, set:Proc(Point))

ColCell ≜ Obj(get:Fun()ColorPoint, set:Proc(ColorPoint))

Temos certamente Cell <: Put [porquê?]

Será que também se pode ter ColCell <: Put ?

```
decl fill =
  proc c:Put => c.set(pt) end
in
  ... fill(cell)... cell.get().set_colour(red) ...
end
```

Env ⊢ pt : Point

Env ⊢ cell : ColorCell

Subtipificação (2)

- Considere os tipos

Put ≜ Obj(set:Proc(Point))

Cell ≜ Obj(get:Fun()Point, set:Proc(Point))

ColCell ≜ Obj(get:Fun()ColorPoint, set:Proc(ColorPoint))

Temos certamente Cell <: Put [porquê?]

Será que também se pode ter ColCell <: Put ?

```
decl fill =
  proc c:Put => c.set(pt) end
in
  ... fill(cell)... cell.get().set_colour(red) ...
end
```

Env ⊢ pt : Point

Erro! Método indefinido

Env ⊢ cell : ColorCell

Subtipificação (2)

- Considere os tipos

Put ≜ Obj(set:Proc(Point))
ColCell1 ≜ Obj(get:Fun()Point, set:Proc(ColorPoint))
ColCell2 ≜ Obj(get:Fun()ColorPoint, set:Proc(Point))

Temos certamente Cell <: Put [porquê?]
Será que se pode ter ColCell2 <: ColCell1 ?

```
decl fill =
  proc c:Colcell1 => c.set(pt) end
in
  ... fill(cell) ... cel.get().set_colour(red) ...
end
```

Env ⊢ pt : ColorPoint

Ok!

Env ⊢ cell : ColorCell2

Subtipificação (2)

Podemos observar que sempre que temos dois tipos de objecto T1 e T2 tais que

$$T1 \triangleq Obj(m_1:Fun(U_1)T_1, \ldots, m_n:Fun(U_n)T_n)$$

$$T2 \triangleq Obj(m_1:Fun(V_1)R_1, \ldots, m_n:Fun(V_n)R_n)$$

com $R_i <: T_i$ e $U_i <: V_i$ para todo o i=1..n,

todo o objecto de tipo T2 pode ser usado seguramente em qualquer contexto onde é "esperado" um objecto de tipo T1 (T2 <: T1).

Um objecto de tipo Put pode ser usado como argumento em vez de um objecto de tipo ColCell.

Subtipificação (2)

- Pode-se então considerar as regras de validação seguintes (mais gerais) para objectos e classes:

$$\frac{U_1 <: T_1 \ \ldots \ U_n <: T_n \qquad U_1 <: V_1 \quad \ldots \ U_n <: V_n}{\begin{array}{c}\text{Env} \vdash \text{Obj}(m_1\text{:Fun}(V_1)U_1, \ \ldots \ , m_n\text{:Fun}(V_n)U_n, n_1\text{:}R_1, \ \ldots \ , p_k\text{:}R_k) \\ <: \text{Obj}(m_1\text{:Fun}(U_1)T_1, \ \ldots \ , m_n\text{:Fun}(U_n)T_n)\end{array}} \ \textbf{(Object)}$$

$$\frac{U_1 <: T_1 \ \ldots \ U_n <: T_n \qquad U_1 <: V_1 \quad \ldots \ U_n <: V_n}{\begin{array}{c}\text{Env} \vdash \text{Class}(m_1\text{:Fun}(V_1)U_1, \ \ldots \ , m_n\text{:Fun}(V_n)U_n, n_1\text{:}R_1, \ \ldots \ , p_k\text{:}R_k) \\ <: \text{Class}(m_1\text{:Fun}(U_1)T_1, \ \ldots \ , m_n\text{:Fun}(U_n)T_n)\end{array}} \ \textbf{(Class)}$$

Co e Contravariância

- Os tipos funcionais são sempre covariantes no tipo do resultado e contravariantes no tipo dos argumentos.

$$\frac{U_1 <: T_1 \ \ldots \ U_n <: T_n \qquad U_1 <: V_1 \quad \ldots \ U_n <: V_n}{\begin{array}{c} Env \vdash Obj(m_1{:}Fun(V_1)U_1, \ \ldots \ , m_n{:}Fun(V_n)U_n, n_1{:}R_1, \ \ldots \ , p_k{:}R_k) \\ <: Obj(m_1{:}Fun(U_1)T_1, \ \ldots \ , m_n{:}Fun(U_n)T_n) \end{array}} \quad \textbf{(Object)}$$

$$\frac{U_1 <: T_1 \ \ldots \ U_n <: T_n \qquad U_1 <: V_1 \quad \ldots \ U_n <: V_n}{\begin{array}{c} Env \vdash Class(m_1{:}Fun(V_1)U_1, \ \ldots \ , m_n{:}Fun(V_n)U_n, n_1{:}R_1, \ \ldots \ , p_k{:}R_k) \\ <: Class(m_1{:}Fun(U_1)T_1, \ \ldots \ , m_n{:}Fun(U_n)T_n) \end{array}} \quad \textbf{(Class)}$$