

Linguagens de Programação 2

Licenciatura em Engenharia Informática
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Luis Caires

Luis.Caires@di.fct.unl.pt

Objectivos da Disciplina

- ◆ Descrever, analisar, justificar e comparar as características das várias linguagens de programação usando um vocabulário e um conjunto de conceitos universais;
- ◆ Caracterizar propriedades dos programas (ex: segurança de tipos), descrever e implementar algoritmos de verificação das mesmas.
- ◆ Conceber pequenos interpretadores / compiladores para linguagens de programação;
- ◆ Conhecer ambientes de suporte à execução modernos (Java, .NET)

"Mapa da Estrada"

Em LP2 estudaremos os conceitos fundamentais das linguagens de programação, abordando os seus aspectos **sintácticos** e **semânticos** tanto do ponto de vista dos fundamentos teóricos, como do ponto de vista da sua implementação:

- Expressões e valores
- Ligação e mecanismos de nomeação
- Estado
- Abstracção funcional e procedimental
- Tipos e sistemas de tipos
- Abstracção de dados
- Objectos, Classes e Módulos

Avaliação

- ◆ Trabalho prático (0...20)
 - de grupo (até 2 elementos)
 - 1 trabalho (de implementação, em Java)
 - A componente prática vale 20%
 - Notas: 0,1,2,3,4,5
 - Frequência: Nota do trabalho > 0.
- ◆ Exame final (0...20)
 - Vale 80% da nota final
 - Nota mínima : 8 (9.5 em 20)

Linguagem de Programação

- ◆ Linguagem de programação: linguagem para descrever *processos computacionais*.
- ◆ O que são "processos computacionais"?

Linguagem de Programação

- ◆ Linguagem de programação: linguagem para descrever *processos computacionais*.
- ◆ O que são "processos computacionais"?
- ◆ Duas grandes **categorias**:

Processos "transformadores", transformam uma *entrada* num *saída* (*input / output*), e terminam (*morrem*)



São os processos mais elementares que se podem imaginar.

Linguagem de Programação

- ◆ Linguagem de programação: linguagem para descrever *processos computacionais*.
- ◆ O que são "processos computacionais"?
- ◆ Duas grandes **categorias**:
Processos "interactivos"



Interagem com o ambiente, mudando de estado em consequência. Podem nunca terminar (ex: webserver).

Linguagem de Programação

- ◆ Linguagem de programação: linguagem para descrever *processos computacionais*.
- ◆ O que são "processos computacionais"?
- ◆ Vamos focar **apenas** processos transformadores.
Processos "transformadores", transformam uma *entrada* num *saída (input / output)*, e terminam (*morrem*)



São os processos mais **simples** que se podem imaginar.

- ◆ Aparentemente são apenas **funções**, no sentido usual matemático do termo.

O que é um "processo computacional"?

- ◆ O resultado de uma computação pode ser indefinido por dois motivos:

Operação não definida (ex: divisão por zero)

- A expressão "3/0" não (mesmo) tem valor
- A implementação pode decidir abortar a execução

Não-Terminação

- $f(x) \triangleq \text{if } x=0 \text{ then } 1 \text{ else } f(x-2)$
- f é uma função *parcial*: não está definida em todos os argumentos
- A terminação não pode ser detectada em tempo de compilação (uma instância do chamado "halting problem")

- ◆ Estas duas situações são

- "Matematicamente" equivalentes, mas
- Operacionalmente diferentes

Funções Parciais vs. Totais



Função total: $f(x)$ é definido para todo o valor x .

Função parcial: $g(x)$ é indefinido para alguns valores x .

(imagens retiradas de [J.Mitchell,2002])

Funções como "gráficos"

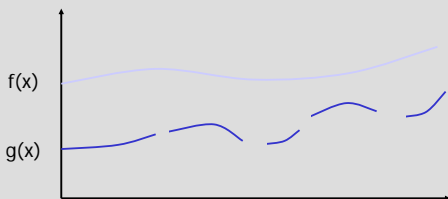


Gráfico de $f = \{ \langle x, y \rangle \mid y = f(x) \}$

Gráfico de $g = \{ \langle x, y \rangle \mid y = g(x) \}$

Uma função é um conjunto de pares ordenados (**gráfico da função**)

Funções Parciais vs Totais

- ◆ Uma função **total** $f:A \rightarrow B$ é um subconjunto $f \subseteq A \times B$ tq.
 - Para todo $x \in A$, existe $y \in B$ tal que $\langle x, y \rangle \in f$ (totalidade)
 - Se $\langle x, y \rangle \in f$ e $\langle x, z \rangle \in f$ então $y = z$ (unicidade)
- ◆ Uma função **parcial** $f:A \rightarrow B$ é um subconjunto $f \subseteq A \times B$ tq.
 - Se $\langle x, y \rangle \in f$ e $\langle x, z \rangle \in f$ então $y = z$ (unicidade)
- ◆ Os **programas** definem **funções parciais** por duas razões
 - Existem operações parciais (como a divisão, desempilhar pilha vazia)
 - Não-terminação
 $f(x) = \text{if } x=0 \text{ then } 1 \text{ else } f(x-2)$

Computabilidade

- ◆ **Definição:**
Uma função f diz-se *computável* se existe um programa P que calcula f .
Ou seja,
Para todo o *input* x , se $f(x)$ está definido, então a execução de $P(x)$ *termina* e *fornece* o *output* $f(x)$
- ◆ **Terminologia:**
"Funções recursivas parciais" =
funções parciais computáveis
- ◆ **Questão:** Será que todas as funções parciais são computáveis? Por outras palavras: será que é possível programar toda e qualquer função parcial que exista?

A Função *Halt*

- ◆ Decide se um programa termina, dado um input.
Dado um programa P e um *input* x ,

$$Halt(P, x) \triangleq \begin{cases} \text{true} & \text{se } P(x) \text{ termina} \\ \text{false} & \text{caso contrário} \end{cases}$$

A Função *Halt*

- ◆ Decide se um programa termina, dado um input.
Dado um programa P e um *input* x ,

$$Halt(P, x) \triangleq \begin{cases} \text{true} & \text{se } P(x) \text{ termina} \\ \text{false} & \text{caso contrário} \end{cases}$$

A função *Halt* aceita um programa P como input.
Não se baralhe com isto, existem muitas funções que aceitam programas como input (por exemplo, a função computada pelo comando `javac`).

A Função *Halt*

- ◆ Decide se um programa termina dado um input
Dado um programa P e um *input* x ,

$$Halt(P, x) \triangleq \begin{cases} \text{true} & \text{se } P(x) \text{ termina} \\ \text{false} & \text{caso contrário} \end{cases}$$

Teorema (Turing, 1931):

Não existe um programa para *Halt*
(*Nem Deus pode implementar Halt*)

Alan Turing (1912-1954)

Indecidibilidade de *Halt*

1. Assuma que P implementa (uma variante) *Halt*
Dado input Q , seja

$$P(Q) \triangleq \begin{cases} \text{true} & \text{se } Q(Q) \text{ termina} \\ \text{false} & \text{caso contrário} \end{cases}$$

2. Defina um programa D assim

$$D(Q) \triangleq \begin{cases} \text{if } P(Q) \text{ then while(true)\{ } \\ \text{else stop} \end{cases}$$

Indecidibilidade de *Halt*

- Assuma que P implementa (uma variante) *Halt*
Dado input Q, seja

$$P(Q) \triangleq \begin{cases} \text{true} & \text{se } Q(Q) \text{ termina} \\ \text{false} & \text{caso contrário} \end{cases}$$
- Defina um programa D assim

$$D(Q) \triangleq \begin{cases} \text{while(true)\{}} & \text{se } Q(Q) \text{ termina} \\ \text{stop} & \text{se } Q(Q) \text{ não termina} \end{cases}$$

Que pode D(D) fazer ?

Se D(D) termina, então D(D) não termina.

Se D(D) não termina, então D(D) termina.

CONTRADIÇÃO! Onde está o erro?

Ideias a reter!

- ◆ Algumas funções parciais são computáveis, outras não ...
 - *Halting problem* (e outros)
- ◆ Impacto na implementação de linguagens
 - É possível detectar e assinalar um erro no caso de uma operação estar indefinida (pilha vazia, overflow, divisão por zero)
 - Nem sempre é possível determinar se um programa vai terminar ou não, se vai executar uma operação indefinida, qual a memória que um programa vai necessitar, etc, etc, etc ...
- ◆ Mas, é possível *aproximar* algumas destas análises (ex: usando sistemas de tipos).

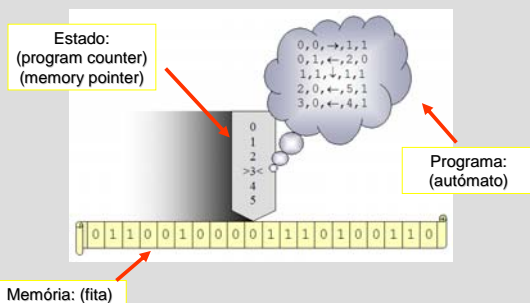
Ideias a reter!

- ◆ Processo computacional = Função parcial calculável por
 - Máquina de Turing
 - Cálculo Lambda [Church]
 - Sistemas de equações recursivas [Godel]
 - ... todos estes modelos são equivalentes!
- ◆ Terminologia:
 - Uma linguagem de programação diz-se **Turing-completa** se permite programar todas as funções efectivamente computáveis.

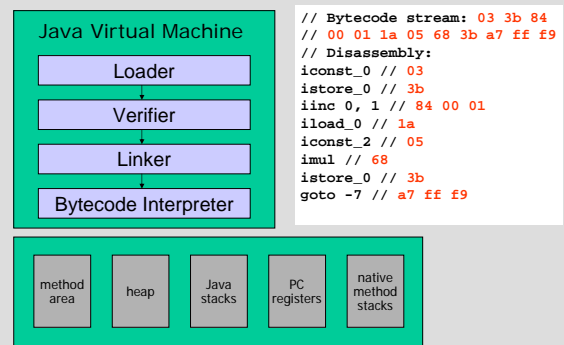
Máquinas Virtuais (*software processors*)

- ◆ Máquina de Turing (Turing, 1931)
 - A primeira ...
- ◆ SECD (Landin, 1962)
 - Stack, Environment, Code, Dump
- ◆ P-Machine (Wirth 1972)
 - Máquina de Pilha, Pascal p-code compiler
- ◆ JVM (Sun, 1995)
 - Multi-threaded, tipificada, dedicada à linguagem Java
- ◆ CLR (Microsoft, 2000)
 - Multi-threaded, tipificada, Multi linguagem

Máquina de Turing



Java Virtual Machine



Ideias a reter!

Uma linguagem de programação diz-se

Turing-completa

se permite programar todas as funções parciais computáveis.

Questão: **Será que todas as funções parciais são computáveis?** Por outras palavras: será que é possível programar toda e qualquer função parcial que exista?

"Some believe that we lacked the programming language to describe your perfect world ..."
Agent Smith



Linguagem de Programação

- ◆ Linguagem de programação: linguagens para descrever processos computacionais
- ◆ O que são "processos computacionais"?
- ◆ Linguagem = Sintaxe + Semântica
- ◆ Qualquer linguagem de programação decente tem uma **sintaxe** e uma **semântica** precisas, não ambíguas, e descritas formalmente
- ◆ Exemplo, da linguagem C:

Que significa a frase "f(2)+g(3)"?

Qual o valor da expressão ?

```
int a = 0;

int f(int x) {
    a = a + 1;
    return x;
}
int g(int y) {
    return y+a
}
```

"f(2)+g(3)"

Sintaxe e Semântica (1)

Sintaxe

- ◆ A **sintaxe** caracteriza a **forma** como se escrevem os programas da linguagem, sem atender ao seu significado.
- ◆ Exemplo de observação sobre "sintaxe"
"Enquanto na linguagem C os blocos são delimitados por chavetas { e }, na linguagem Pascal usam-se os delimitadores begin end"
- ◆ A sintaxe **concreta** de uma linguagem de programação pode ser descrita precisa e formalmente usando gramáticas (LFA)

Semântica

- ◆ A **semântica** descreve o significado das frases sintacticamente válidas de uma linguagem
- ◆ Exemplos de observações sobre "semântica":
"Em C, um vector é modelado por um apontador, mas em Pascal um vector é um valor primitivo"
"A linguagem ML (OCaml) é uma linguagem imperativa, mas centrada no uso de funções"
"Na linguagem Java os argumentos são sempre passados por valor, mas em Pascal também podem ser passados por referência"

Semântica

- ◆ A semântica de uma linguagem pode ser caracterizada por uma **função computável I** que atribui um **significado** a cada programa (ou fragmento de programa)

$$I : \text{PROG} \rightarrow \text{DENOT}$$

PROG = conjunto dos programas
DENOT = conjunto dos significados (denotações)

Semântica

- ◆ A função semântica I pode ser vista como um algoritmo que “sabe como interpretar” **todos** os programas (sintacticamente correctos) de uma linguagem, determinando o seu **valor ou efeito**

$$I : \text{PROG} \rightarrow \text{DENOT}$$

PROG = conjunto dos programas
DENOT = conjunto dos significados (denotações)

Semântica

- ◆ Quem consegue especificar (e implementar) a função semântica I para uma linguagem é **obrigado** a compreender detalhadamente o funcionamento de **todos** os mecanismos usados na linguagem de programação em causa
- ◆ É também conduzido naturalmente a reflectir sobre as razões que levam a que as linguagens sejam como são, e a imaginar alternativas ...
- ◆ A sua maturidade aumentará muito: passará de um mero “utilizador de linguagens”, para um “insider” informado, a par daqueles que as conceberam

Sintaxe e Semântica (1)

- ◆ **Conceito chave:** definição da semântica de uma linguagem **por indução** na estrutura sintática
- ◆ **Conceito chave:** definição da semântica de uma linguagem por meio de um algoritmo interpretador
- ◆ Como definir a sintaxe de uma linguagem de programação como um tipo de dados ?
- ◆ Como definir a semântica de uma linguagem de programação como um algoritmo interpretador ?

Tipos de Dados

- ◆ Um **tipo de dados** pode ser representado como um **conjunto de valores decidível**.
- ◆ Tal conjunto de valores pode ser infinito, mas cada valor é “finito”.
- ◆ Exemplos:

- Booleanos (true / false)
- Números inteiros (..., -2, -1, 0, 1, 2, ...)
- Listas
- Árvores binárias
- etc ...

Definição de tipos de dados

- ◆ Quando o conjunto de valores possíveis é finito, podemos definir os valores do tipo por enumeração :
 - Boolean \triangleq { true, false }
 - BasicColor \triangleq { red, green, blue }
- ◆ Como definir o conjunto dos valores de um tipo quando o número de valores possível é infinito ?
 - NaturalNumber \triangleq ?
 - List \triangleq ?
 - Tree \triangleq ?

Definição indutiva de dados

- ◆ O mecanismo de definição “universalmente” usado em informática é a definição **indutiva**.

- ◆ Tipo *NaturalNumber* :

1. **0** é um número natural (caso base)
2. se n é um número natural, então **succ**(n) é também um número natural
3. não existem mais números naturais, excepto os construídos de acordo com as regras 1 e 2 acima.

Definição indutiva de dados

- ◆ O mecanismo de definição “universalmente” usado em informática é a definição **indutiva**.

- ◆ Tipo *List* (de elementos de tipo T):

1. **nil** é uma lista (a lista vazia)
2. se x é um valor de tipo T e L é uma lista, então **cons**(x, L) é também uma lista
3. não existem mais listas, excepto as construídas de acordo com as regras 1 e 2 acima.

Definição indutiva de dados

- ◆ Todas as definições indutivas de tipos de dados obedecem ao mesmo padrão:

- ◆ Tipo *LabeledTree* (de valores de tipo T)

1. **empty** é uma árvore (a árvore vazia)
2. se x é um valor de tipo T, e T_1 e T_2 são árvores, então **node**(T_1, x, T_2) também é uma árvore, com raiz etiquetada por x , cuja subárvore esquerda é T_1 e cuja subárvore direita é T_2 .
3. Não existem mais ... 1. e 2.

Definição indutiva de dados

- ◆ Todas as definições indutivas de tipos de dados obedecem ao mesmo padrão:

- ◆ Tipo *Stack* (de valores de tipo T)

1. **empty** é uma pilha (a pilha vazia)
2. se x é um valor de tipo T, e S é uma pilha, então **push**(x, S) também é uma pilha, cujo topo contém o valor x , por trás do qual está a pilha S .
3. Não existem mais ... 1. e 2.

Definição indutiva de dados

- ◆ Os ingredientes da definição indutiva de um tipo de dados são:

1. o **nome** do tipo T
2. um conjunto de **construtores**

- ◆ Um **construtor** é uma função que permite construir novos valores do tipo T, a partir de valores **já construídos** do mesmo tipo ou de outros tipos.

- ◆ Cada construtor tem uma **assinatura**, que indica os tipos dos seus argumentos.

- ◆ O tipo do resultado de cada construtor do tipo T é T

Definição indutiva de dados

- ◆ Elementos da definição indutiva do tipo *lista* de valores de tipo *integer*

- ◆ o nome do tipo: **Listint**

- ◆ um conjunto de construtores: **nil, cons**

nil: \rightarrow Listint
cons: Integer \times Listint \rightarrow Listint

Definição indutiva de dados

- ◆ Elementos da definição indutiva do tipo *árvore etiquetada* de valores de tipo **integer**
- ◆ o nome do tipo: **Treelnt**
- ◆ um conjunto de construtores: **empty, node**

empty: \rightarrow **Treelnt**

node: **Treelnt** \times **Integer** \times **Treelnt** \rightarrow **Treelnt**

Definição indutiva de algoritmos

- ◆ É fácil definir algoritmos operando sobre tipos de dados de tipo indutivo, por **análise de casos** no construtores.
- ◆ Dado um valor v qualquer de um tipo **T**, o algoritmo
 - **verifica** qual é o último construtor aplicado na construção de v (por exemplo, v é `cons(3, cons(2,nil))`);
 - **extrai** os componentes aos quais o construtor foi aplicado (neste caso, o inteiro 3 e a lista `cons(2,nil)`);
 - aplica-se recursivamente aos componentes de tipo **T**. Assim, obtêm-se os resultados intermédios que permitem produzir o resultado final.

Definição indutiva de algoritmos

- ◆ Tipo **Listint**
- ◆ Construtores: **nil, cons**
- ◆ Algoritmo **length(L)** para calcular o comprimento de uma lista qualquer **L**:

se **L** é da forma **nil**

$\text{length}(L) \triangleq 0$

se **L** é da forma **cons(x,L')**

$\text{length}(L) \triangleq 1 + \text{length}(L')$

Definição indutiva de algoritmos

- ◆ Tipo **Listint**
- ◆ Construtores: **nil, cons**
- ◆ Algoritmo **sum(L)** para calcular a soma dos valores numa lista qualquer **L**:

se **L** é da forma **nil**

$\text{sum}(L) \triangleq 0$

se **L** é da forma **cons(x,L')**

$\text{sum}(L) \triangleq x + \text{sum}(L')$

Definição indutiva de algoritmos

- ◆ Tipo **Treelnt**
- ◆ Construtores: **empty, node**
- ◆ Algoritmo **numnodes(T)** para calcular o número de nós numa árvore qualquer **T**:

se **T** é da forma **empty**

$\text{numnodes}(L) \triangleq 0$

se **L** é da forma **node(L',x,L'')**

$\text{numnodes}(L) \triangleq 1 + \text{numnodes}(L') + \text{numnodes}(L'')$

Definição indutiva de algoritmos

- ◆ Tipo **Treelnt**
- ◆ Construtores: **empty, node**
- ◆ Algoritmo **depth(T)** para calcular a altura de uma árvore qualquer **T**:

se **T** é da forma **empty**

$\text{depth}(T) \triangleq 0$

se **T** é da forma **node(L',x,L'')**

$\text{depth}(T) \triangleq 1 + \max(\text{depth}(L'), \text{depth}(L''))$

Definição indutiva de programas

- ◆ O conjunto de todos os programas de uma linguagem de programação pode ser visto como um tipo de dados
- ◆ É fácil definir **indutivamente** o conjunto de todos os programas de uma linguagem de programação
- ◆ A definição indutiva de linguagens **melhorou** o desenho das mesmas (Fortran/spaghetti code vs. Pascal/estrutura em blocos)
- ◆ **Discussão:** definição indutiva de algoritmos *versus* definição indutiva de programas (a que se referem as duas ocorrências do adjetivo "indutivo"?)

© Luis Caires

LP2 2005/06

49

Estruturação hierárquica

```
10 for i=1 to 10 do
20 if i<10 goto 40
40 for j=1 to 20 do
30 next I
40 next J
```



```
for i=1 to 10 do
begin
if i<10 then
begin
end
end;
end;
```

© Luis Caires

LP2 2005/06

50

Exemplo: A Linguagem CALC

- ◆ Uma linguagem simples de expressões aritméticas
- ◆ Cada programa da linguagem CALC é uma expressão algébrica construída com base em numerais inteiros, nos quatro operadores aritméticos básicos (+, -, ×, ÷) e nos parêntesis.
- ◆ Exemplos:


```
(21+32) * 42
2 / (7-2)
```
- ◆ A semântica pretendida para a linguagem CALC é a esperada: a denotação de cada expressão de CALC é o seu resultado.

© Luis Caires

LP2 2005/06

51

Semântica de CALC

- ◆ Em geral, a semântica de uma linguagem pode ser caracterizada por uma função I que atribui um significado (ou denotação) a cada programa (ou fragmento) sintacticamente correcto.
- ◆ No caso da linguagem CALC:

$I : \text{CALC} \rightarrow \text{integer}$
$\text{CALC} = \text{conjunto dos programas válidos}$
$\text{integer} = \text{conjunto dos significados (denotações)}$
- ◆ Como representar a linguagem CALC?

© Luis Caires

LP2 2005/06

52

A Linguagem CALC (como tipo indutivo)

- ◆ Tipo de dados: CALC
- ◆ construtores: **num, add, mul, div, sub**

```
num: integer → CALC
add: CALC × CALC → CALC
mul: CALC × CALC → CALC
div: CALC × CALC → CALC
sub: CALC × CALC → CALC
```

© Luis Caires

LP2 2005/06

53

Exemplo: A Linguagem CALC

- ◆ Cada valor do tipo indutivo CALC representa uma dada **expressão** da linguagem CALC
- ◆ O tipo indutivo CALC define a **sintaxe abstracta** da linguagem CALC
- ◆ A **sintaxe concreta** de uma linguagem:

Caracteriza a forma como as suas expressões e programas são **efectivamente escritos** em termos de sequências de caracteres, formatação, etc ...
- ◆ A **sintaxe abstracta** de uma linguagem:

Caracteriza a estrutura das suas expressões e programas, apresentando essa estrutura em termos de construtores abstractos.

© Luis Caires

LP2 2005/06

54

Sintaxe abstracta / Sintaxe concreta

- ◆ Constantes inteiras
 - em decimal: 12
 - em hexadecimal: 0Cx0
 - sintaxe abstracta: `num(12)`
- ◆ Variáveis
 - em Pascal: A
 - em bash: %A
 - sintaxe abstracta: `var("A")`
- ◆ Afectação
 - em C: `x = 2`
 - em Pascal: `x := 2`
 - sintaxe abstracta: `assign(var("x"),num(2))`

© Luis Caires

LP2 2005/06

55

Sintaxe abstracta / Sintaxe concreta

- ◆ Expressões algébricas
 - em C: `2*3+2`
 - em RPN: `2 3 * 2 +`
 - em Lisp: `(PLUS (TIMES 2 3) 2)`
 - sintaxe abstracta: `add(mul(num(2),num(3)),num(2))`
- ◆ Blocos
 - em C: `{S1 S2 ... Sn}`
 - em Pascal: `begin S1; S2; ...; Sn end`
 - sintaxe abstracta: `block(S1,S2,...,Sn)`
- ◆ Ciclo while:
 - em C: `while (C) S`
 - em Pascal: `while C do S`
 - sintaxe abstracta: `while(C,S)`

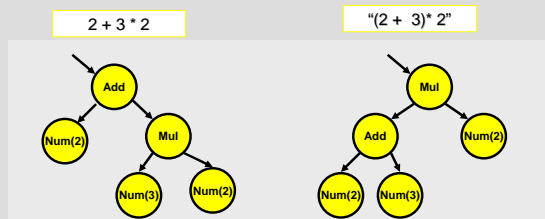
© Luis Caires

LP2 2005/06

56

Árvore Sintactica Abstracta

- ◆ Representa a estrutura de uma frase da linguagem em termos dos constructores abstractos.
- ◆ *Abstract Syntax Tree (AST)*



© Luis Caires

LP2 2005/06

57

Semântica de CALC

- ◆ A função semântica I pode ser definida por um algoritmo que "sabe como interpretar" todos os programas sintacticamente correctos de uma linguagem, determinando o seu valor ou efeito

$I : \text{CALC} \rightarrow \text{integer}$

CALC = conjunto dos programas válidos
integer = conjunto dos significados (denotações)

© Luis Caires

LP2 2005/06

58

Interpretores

- ◆ Um **interpretador** para uma linguagem de programação é um algoritmo que atribui um valor ou efeito a cada programa legítimo da linguagem
- ◆ Os programas fornecidos como *input* a um interpretador são representados por **valores da sintaxe abstracta** da linguagem a que pertencem
 - Programas exprimem algoritmos que processam dados
 - Programas também podem ser vistos como dados
 - Interpretador é um programa que processa programas
- ◆ Um algoritmo interpretador pode ser definido **indutivamente** na sintaxe abstracta da linguagem

© Luis Caires

LP2 2005/06

59

A Linguagem CALC (como tipo indutivo)

- ◆ Tipo de dados: CALC
- ◆ construtores: **Num, Add, Mul, Div, Sub**

Num: $\text{integer} \rightarrow \text{CALC}$
Add: $\text{CALC} \times \text{CALC} \rightarrow \text{CALC}$
Mul: $\text{CALC} \times \text{CALC} \rightarrow \text{CALC}$
Div: $\text{CALC} \times \text{CALC} \rightarrow \text{CALC}$
Sub: $\text{CALC} \times \text{CALC} \rightarrow \text{CALC}$

© Luis Caires

LP2 2005/06

60

Interpretador de CALC

- ◆ Algoritmo $\text{eval}(E)$ para calcular a denotação (valor inteiro) de uma expressão E qualquer de CALC:

$\text{eval} : \text{CALC} \rightarrow \text{integer}$

```
se E é da forma num(n):          eval(E) = n
se E é da forma add(E',E''):      v1 = eval(E'); v2 = eval(E'');
                                  eval(E)  $\triangleq$  v1+v2
se E é da forma mul(E',E''):      v1 = eval(E'); v2 = eval(E'');
                                  eval(E)  $\triangleq$  v1*v2
se E é da forma sub(E',E''):      v1 = eval(E'); v2 = eval(E'');
                                  eval(E)  $\triangleq$  v1-v2
se E é da forma div(E',E''):      v1 = eval(E'); v2 = eval(E'');
                                  eval(E)  $\triangleq$  v1/v2
```

© Luis Caires

LP2 2005/06

61

Interpretador de CALC

- ◆ Algoritmo $\text{eval}(E)$ para calcular a denotação (valor inteiro) de uma expressão E qualquer de CALC:

$\text{eval} : \text{CALC} \rightarrow \text{integer}$

```
eval(Num(n))           $\triangleq$  n
eval(Add(E',E''))      $\triangleq$  eval(E') + eval(E'')
eval(Mul(E',E''))      $\triangleq$  eval(E') * eval(E'')
eval(Sub(E',E''))      $\triangleq$  eval(E') - eval(E'')
eval(Div(E',E''))      $\triangleq$  eval(E') / eval(E'')
```

[notação mais legível, usando *pattern matching*]

© Luis Caires

LP2 2005/06

62

Interpretador de CALC

- ◆ Algoritmo $\text{eval}(E)$ para calcular o significado (valor inteiro) de uma expressão qualquer de CALC:

$\text{eval} : \text{CALC} \rightarrow \text{integer}$

- ◆ Note bem: a função de interpretação $\text{eval}(-)$ é definida **recursivamente na estrutura** do seu argumento!

Semântica composicional: o significado do todo só depende do significado das suas partes

- ◆ O mesmo não acontece nas linguagens "naturais":
 - *time flies like an arrow*
 - *fruit flies like a banana*

© Luis Caires

LP2 2005/06

63

Semântica Operacional Estrutural

- ◆ Sintaxe (abstracta) definida por um tipo indutivo, apresentada por um conjunto de construtores;
- ◆ Semântica definida por um algoritmo interpretador, que atribui um significado (valor) a cada expressão da linguagem, calculando-o **composicionalmente** a partir do significado das suas subexpressões;
- ◆ A esta **técnica de definição da semântica** de uma linguagem de programação chama-se:

semântica operacional estrutural

© Luis Caires

LP2 2005/06

64

Implementação em Java

- ◆ Usando uma linguagem baseada em objectos, um tipo de dados indutivo pode ser representado por **uma interface** (que representa o tipo indutivo), e por um **conjunto de classes** (em que cada classe representa um construtor do tipo indutivo)
- ◆ A interface pode declarar uma ou mais operações sobre o tipo indutivo, por exemplo:

```
public interface CALC { int eval(); }
```

- ◆ Ou seja, $\text{eval} : \text{CALC} \rightarrow \text{integer}$.

© Luis Caires

LP2 2005/06

65

Implementação em Java

- ◆ Cada classe representa **um (e um só)** dos construtores do tipo indutivo
- ◆ Cada classe fornece a implementação relativa ao construtor que representa, para cada operação definida sobre o tipo indutivo

```
public class Num implements CALC {
    private int value;
    Num(int v) { value = v; }
    int eval() { return value; }
}
```

© Luis Caires

LP2 2005/06

66

Implementação em Java

- ◆ Cada expressão é representada por uma árvore (*n*-ária) de objectos (uma AST);
- ◆ Cada construtor do tipo indutivo é aplicado chamando o construtor da classe respectiva:

```

CALC expr1 = new Add( new Num(2), new Num(3)) ;
int result1 = expr1.eval();
CALC expr2 = new Add( new Sub( Num(2),
                             Num(3)),
                    new Num(3));
int result2 = expr2.eval();
    
```

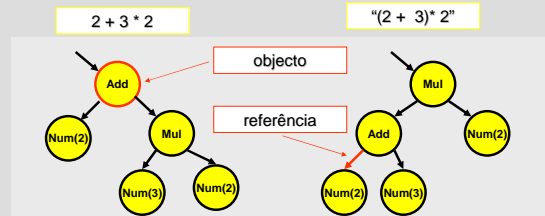
© Luis Caires

LP2 2005/06

67

Árvore Sintactica Abstracta

- ◆ Representa a estrutura de uma frase da linguagem em termos dos constructores abstractos.
- ◆ Implementada por uma estrutura de dados árvore



© Luis Caires

LP2 2005/06

68

Implementação em Java

- ◆ No nosso exemplo, temos as classes Num, Add, Sub, Mul e Div, implementando os construtores num, add, sub, mul e div.
- ◆ A definição das operações fica "dispersa" pelas várias classes (é mais cómodo acrescentar **novos construtores** a um tipo do que **novas operações**)

```

public class Add implements CALC {
    private CALC lhs;
    private CALC rhs;
    Add(CALC l, CALC r) { lhs = l; rhs = r; }
    int eval() { return lhs.eval()+rhs.eval(); }
}
    
```

© Luis Caires

LP2 2005/06

69

Compiladores

- ◆ Um **compilador** para uma linguagem de programação é um algoritmo que traduz cada programa legítimo da linguagem numa sua versão directamente executável por uma "máquina".
- ◆ A máquina pode ser física (Pentium) ou virtual (JVM, CLR)
- ◆ Pode também ser outro compilador já existente (gcc) [exemplo: tradutor de Java em C]
- ◆ A tradução preserva a semântica.
- ◆ Um algoritmo compilador pode ser definido **indutivamente** na sintaxe abstracta da linguagem

© Luis Caires

LP2 2005/06

70

Common Language Runtime

- ◆ Máquina Virtual do Common Language Runtime do ambiente .NET.
- ◆ Evolução da P-Machine (Pascal) e da JVM (JVM); mais se dirá adiante.
- ◆ Independente da linguagem fonte: suporta várias linguagens numa mesma arquitectura.
- ◆ Máquina de Pilha: todas as instruções consomem argumentos do topo da Pilha, e produzem um resultado no topo da Pilha (*stack machine*)
- ◆ "Primeiras" (5) instruções: ldc n, add, mul, div, sub.

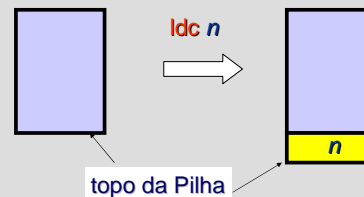
© Luis Caires

LP2 2005/06

71

Common Language Runtime

- ◆ "Primeiras" (5) instruções: ldc n, add, mul, div, sub.
- ◆ Load Constant (ldc n)



© Luis Caires

LP2 2005/06

72

Common Language Runtime

- ◆ “Primeiras” (5) instruções: *ldc n*, *add*, *mul*, *div*, *sub*.
- ◆ *Add (add)*

© Luis Caires LP2 2005/06 73

Common Language Runtime

- ◆ “Primeiras” (5) instruções: *ldc n*, *add*, *mul*, *div*, *sub*.
- ◆ *Add (add)*

O fundo da pilha é inalterado!

© Luis Caires LP2 2005/06 74

Compilador de CALC

- ◆ Algoritmo $comp(E)$ para traduzir uma expressão E qualquer de CALC numa sequência de instruções CLR

$comp : CALC \rightarrow CodeSeq$

se E é da forma num (n):	$comp(E) = \langle ldc.i4\ n \rangle$
se E é da forma add (E',E''):	$s1 = comp(E'); s2 = comp(E'');$ $comp(E) \triangleq s1 @ s2 @ \langle add \rangle$
se E é da forma mul (E',E''):	$v1 = comp(E'); v2 = comp(E'');$ $comp(E) \triangleq s1 @ s2 @ \langle mul \rangle$
se E é da forma sub (E',E''):	$v1 = comp(E'); v2 = comp(E'');$ $comp(E) \triangleq s1 @ s2 @ \langle sub \rangle$
se E é da forma div (E',E''):	$v1 = comp(E'); v2 = comp(E'');$ $comp(E) \triangleq s1 @ s2 @ \langle div \rangle$

© Luis Caires LP2 2005/06 75

Correcção do Compilador

- ◆ Algoritmo $comp(E)$ para traduzir uma expressão E qualquer de CALC numa sequência de instruções CLR

$comp : CALC \rightarrow CodeSeq$

- ◆ **Propriedade de Correcção:** Quando a sequência de instruções $comp(E)$ é executada num estado da máquina virtual em que a pilha está no estado ρ , quando termina deixa sempre a máquina no estado $push(v, \rho)$, em que v é o valor da expressão E.

© Luis Caires LP2 2005/06 76

Common Language Runtime

- ◆ “Primeiras” (5) instruções: *ldc n*, *add*, *mul*, *div*, *sub*.
- ◆ $Comp("2+2*(7-2)")$

```
ldc.i4 2
ldc.i4 2
ldc.i4 7
ldc.i4 2
sub
mul
add
```

© Luis Caires LP2 2005/06 77

Interpretador de CALC

- ◆ Algoritmo $eval(E)$ para calcular a denotação (valor inteiro) de uma expressão E qualquer de CALC:

$eval : CALC \rightarrow integer$

$eval(Num(n))$	$\triangleq n$
$eval(Add(E',E''))$	$\triangleq eval(E') + eval(E'')$
$eval(Mul(E',E''))$	$\triangleq eval(E') * eval(E'')$
$eval(Sub(E',E''))$	$\triangleq eval(E') - eval(E'')$
$eval(Div(E',E''))$	$\triangleq eval(E') / eval(E'')$

[notação mais legível, usando *pattern matching*]

© Luis Caires LP2 2005/06 78

Ligação, Âmbito e Ambiente

© Luis Caires

LP2 2005/06

79

Constantes e Identificadores

- ◆ Constantes (ou literais)
 - Referem entidades ou valores bem determinados em **qualquer contexto** onde ocorram
 - Nas linguagens “naturais”: são os “nome próprios”.
 - true, false, [] (linguagem ML)
 - 1, 1.0, 0xFF, “hello”, int (linguagem C)
- ◆ Identificadores (ou nomes)
 - Referem entidades que **dependem do contexto**
 - Nas linguagens “naturais”: são os “pronomes”.
 - x, Count, System.out (linguagem Java)
 - printf (linguagem C)
 - true, x, integer (linguagem Pascal)

© Luis Caires

LP2 2005/06

80

Ligação e Âmbito

- ◆ Tanto os **literais** como os **identificadores** denotam sempre uma entidade fixa bem determinada
- ◆ A entidade denotada por (ou valor de) um literal é fixa pelo próprio literal (23, “hi!”, etc).
- ◆ A associação entre um identificador e entidade por este denotada chama-se **ligação** (*binding*)
- ◆ Em geral, a ligação entre identificador e entidade denotada estabelece-se num certo contexto sintáctico e é introduzida por uma **declaração**
- ◆ Ao contexto sintáctico onde uma certa ligação tem efeito chama-se o **âmbito** (*scope*) dessa ligação

© Luis Caires

LP2 2005/06

81

Ligação e Âmbito

- ◆ O identificador **x** denota uma variável de estado (célula de memória)

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
        int x=j+y;
        z += x;
    }
    return z;
}
```

© Luis Caires

LP2 2005/06

82

Ligação e Âmbito

- ◆ O identificador **x** denota uma variável de estado (célula de memória)

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
        int x=j+y;
        z += x;
    }
    return z;
}
```

Ámbito da ligação

© Luis Caires

LP2 2005/06

83

Ligação e Âmbito

- ◆ O identificador **j** denota uma variável de estado (célula de memória)

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
        int x=j+y;
        z += x;
    }
    return z;
}
```

© Luis Caires

LP2 2005/06

84

Ligação e Âmbito

- ◆ O identificador `j` denota uma variável de estado (célula de memória)

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
        int x=j+y;
        z += x;
    }
    return z;
}
```

Âmbito da ligação

© Luis Caires

LP2 2005/06

85

Elementos de um âmbito

- ◆ A **ligação** entre um identificador e a respectiva entidade por este denotada (valor, posição de memória, etc) envolve os seguintes ingredientes:
 - Uma (única!) **ocorrência ligante** (que, em geral, corresponde à declaração do identificador)
 - O **âmbito** da ligação (que é a "parte/região/zona" do programa onde a ligação em causa tem efeito)
 - Várias (zero ou mais) ocorrências **ligadas** (que são todas as ocorrências do identificador, distintas da ocorrência ligante, que existem dentro do âmbito)

© Luis Caires

LP2 2005/06

86

Ocorrências ligantes e ligadas

- ◆ Ocorrências do identificador `x`

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
        int x=j+y;
        z += x;
    }
    return z;
}
```

Ocorrências ligantes

© Luis Caires

LP2 2005/06

87

Ocorrências ligantes e ligadas

- ◆ Ocorrências do identificador `x`

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
        int x=j+y;
        z += x;
    }
    return z;
}
```

Ocorrências ligadas

© Luis Caires

LP2 2005/06

88

Ocorrências ligadas

- ◆ Para cada ocorrência ligada existe uma e uma só ocorrência ligante (que ocorre na declaração)

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
        int x=j+y;
        z += x;
    }
    return z;
}
```

© Luis Caires

LP2 2005/06

89

Ocorrências livres

- ◆ Uma ocorrência de identificador que não é ligada nem ligante diz-se **livre**

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
        int x=j+y;
        z += x;
    }
    return z;
}
```

- ◆ Neste exemplo, apenas `y` tem uma ocorrência livre

© Luis Caires

LP2 2005/06

90

Expressões Abertas e Fechadas

- ◆ Uma subexpressão diz-se **aberta** se contém ocorrências livres de identificadores
- ◆ Uma subexpressão diz-se **fechada** se não contém ocorrências livres de identificadores
- ◆ Exemplos de expressões abertas:

```
void f(int x)
{
  int i;
  for(int i=0;i<TEN;i++) x+=i;
  printf("%d\n",x);
}
```

C

```
let x=1 in (f x)
```

OCaml

© Luis Caires

LP2 2005/06

91

Expressões Abertas e Fechadas

- ◆ Uma subexpressão diz-se **aberta** se contém ocorrências livres de identificadores
- ◆ Uma subexpressão diz-se **fechada** se não contém ocorrências livres de identificadores
- ◆ Exemplos de expressões abertas:

```
void f(int x)
{
  int i;
  for(int i=0;i<TEN;i++) x+=i;
  printf("%d\n",x);
}
```

livre

C

```
let x=1 in (f x)
```

livre

OCaml

© Luis Caires

LP2 2005/06

92

Semântica de expressões abertas

- ◆ A denotação de uma subexpressão de programa só pode ser calculada se se conhecer a denotação de cada identificador que nela ocorra livre.
- ◆ A definição de uma semântica composicional para linguagens com declarações de identificadores tem necessariamente que considerar expressões abertas. Por exemplo, a expressão OCaml

```
let x = 2
  in (x+x)
```

é fechada mas contém uma subexpressão aberta:

```
(x+x)
```

© Luis Caires

LP2 2005/06

93

Ambiente

- ◆ Um programa (fragmento fechado) pode conter no seu interior expressões abertas.
[Dê exemplos de linguagens de programação onde seja possível compilar um programa aberto]
- ◆ Um programa fechado fornece necessariamente ligações para todas as ocorrências livres de identificadores que ocorram nas suas subexpressões (através de declarações).
Para cada subexpressão E de um programa P , ao conjunto de todas as ligações no âmbito das quais E ocorre chama-se o **ambiente** de E em P

© Luis Caires

LP2 2005/06

94

Ambiente (Quiz)

- ◆ Qual o ambiente da subexpressão " $x+1$ "?

```
int f(int x)
{
  int z = (x+1);
  for(int j=0; j<10; j++){
    int x=j;
    z+=x;
  }
  return z;
}
```

- ◆ { $f \rightarrow$ "fun?", $x \rightarrow$ "var?", $z \rightarrow$ "var?" }

© Luis Caires

LP2 2005/06

95

Ambiente (Quiz)

- ◆ Qual o ambiente da subexpressão " $z+=x$ "?

```
int f(int x)
{
  int z = x+1;
  for(int j=0; j<10; j++){
    int x=j;
    (z+=x);
  }
  return z;
}
```

- ◆ { $f \rightarrow$ "fun?", $z \rightarrow$ "var?", $j \rightarrow$ "var?", $x \rightarrow$ "var?" }

© Luis Caires

LP2 2005/06

96

Ambiente (Quiz)

- ◆ Qual o ambiente da subexpressão "return z"?

```
int f(int x)
{
    int z = x+1;
    for(int j=0; j<10; j++){
        int x=j;
        z+=x;
    }
    return z;
}
```

- ◆ { f → "fun?", x → "var?", z → "var?" }

© Luis Caires

LP2 2005/06

97

Exemplo: A Linguagem CALC1

- ◆ A linguagem CALC1 estende a linguagem CALC com a possibilidade de se poderem introduzir e usar identificadores usando a construção **declare**:

```
decl Id = Expressão1 in Expressão2
```

- ◆ Numa expressão *decl1*, a primeira ocorrência de *Id* é **ligante**, no âmbito definido pela *Expressão2*
- ◆ Definimos os programas CALC1 como sendo as **expressões fechadas** de CALC1. Por exemplo:

```
(decl x=2 in decl y=x+2 in (x+y))
```

© Luis Caires

LP2 2005/06

98

A Linguagem CALC1 como tipo indutivo

- ◆ Tipo de dados: CALC1
- ◆ construtores: **num, add, mul, div, sub, id, decl**

```
num:   integer → CALC1
id:    string  → CALC1
add:   CALC1 × CALC1 → CALC1
mul:   CALC1 × CALC1 → CALC1
div:   CALC1 × CALC1 → CALC1
sub:   CALC1 × CALC1 → CALC1
decl:  string × CALC1 × CALC1 → CALC1
```

© Luis Caires

LP2 2005/06

99

Semântica de CALC1 (1)

- ◆ A função semântica *I* de CALC1 pode ser definida por um algoritmo que "sabe como interpretar" todas as expressões de CALC1, determinando o seu **valor ou efeito**, dado um ambiente contendo os valores dos seus identificadores livres.

$$I : \text{CALC1} \rightarrow \text{integer}$$

CALC1 = conjunto das expressões fechadas
integer = conjunto dos significados (denotações)

© Luis Caires

LP2 2005/06

100

Interpretador de CALC1

- ◆ Algoritmo **eval(E)** para calcular o valor de uma **expressão fechada** qualquer *E* de CALC1:

eval : CALC1 → integer

```
eval( num(n) ) ≙ return n
eval( add(E1,E2) ) ≙
    return eval(E1) + eval(E2)
...
eval( decl(s, E1, E2) ) ≙ [
    G = Subst(s, E2, eval(E1));
    return eval(G); ]
```

© Luis Caires

LP2 2005/06

101

A Função Subst

Subst(*s*, *E*, *F*)

- ◆ Calcula a expressão que resulta de substituir todas as ocorrências livres do identificador *s* pela expressão *F* na expressão *E*.

Subst(*s*, *s+s+2*, *y+z*) = (*y+z*)+(*y+z*)+2

Subst(*y*, decl *x=y* in decl *y=2* in *x+y*, *u*) =
 decl *x=u* in decl *y=2* in *x+y*

© Luis Caires

LP2 2005/06

102

Definição da Função Subst

```
Subst(s, num(n), F)  △ return num(n);
Subst(s, id(s), F)   △ return F;
Subst(s, add(E1, E2) △
    return add( Subst(s, E1, F), Subst(s, E2, F));
...
Subst(s, decl(s, E1, E2), F) △ [ /* caso s = s' */
    G = Subst(s, E1, F);
    return decl(s, G, E2); ]

Subst(s, decl(s', E1, E2), F) △ [ /* caso s ≠ s' */
    G = Subst(s, E1, F);
    return decl(s', G, Subst(s, E2, F)); ]
```

© Luis Caires

LP2 2005/06

103

Semântica de CALCI (2)

- ◆ A semântica da linguagem CALCI baseada em substituições é muito conveniente do ponto de vista da especificação pois é muito simples.

```
eval : CALCI → integer
```

- ◆ Já do ponto de vista operacional, é conveniente definir uma **semântica mais concreta**, recorrendo à manipulação de **ambientes**.
- ◆ A manipulação de ambientes também é mais conveniente como técnica de implementação de interpretadores.

© Luis Caires

LP2 2005/06

104

Semântica de CALCI (2)

- ◆ A função semântica I de CALCI pode ser definida por um algoritmo interpretador para expressões de CALCI, determinando o seu **valor ou efeito**, dado um ambiente contendo os valores dos seus identificadores livres.

```
I : CALCI × ENV → integer
```

CALCI = programas **abertos**

ENV = ambientes

integer = significados (denotações)

© Luis Caires

LP2 2005/06

105

Ambiente "mutável"

- ◆ Na prática, é conveniente implementar ambientes usando uma estrutura de dados mutável:

```
void Assoc(String id, Value val)
```

- Adiciona ao ambiente uma nova ligação que associa ao identificador `id` o valor `val` indicado.
- A ligação é adicionada ao último nível (mais recente) do ambiente.

```
Value Find(String id)
```

- Devolve o valor associado ao identificador `id` no ambiente.
- A pesquisa é efectuada do nível mais "recente" para o mais "antigo", de modo a respeitar o encaixe dos âmbitos das declarações.

© Luis Caires

LP2 2005/06

106

Ambiente "mutável"

- ◆ Na prática, é conveniente implementar ambientes usando uma estrutura de dados mutável:

```
Environ BeginScope()
```

- Cria um novo nível local vazio, onde serão colocadas as novas ligações.
- Não pode existir mais que uma ligação para um mesmo identificador no mesmo nível (Porquê?)

```
Environ EndScope()
```

- Coloca o ambiente no estado anterior à última operação `BeginScope()`.

- ◆ Funciona bem, pois numa linguagem estruturada em blocos encaixados hierarquicamente, as adições e remoções de ligações aos ambientes segue uma disciplina LIFO.

© Luis Caires

LP2 2005/06

107

A "interface" Ambiente

- ◆ Simule mentalmente:

```
env = new Environment();
env.Assoc("x", 2);
val = env.Find("x"); // devolve 2
env = env.BeginScope();
env.Assoc("y", 3);
env.Assoc("x", 4);
val = env.Find("y"); // devolve 3
val = env.Find("x"); // devolve 4
env=env.EndScope()
val = env.Find("x") // devolve 2
```

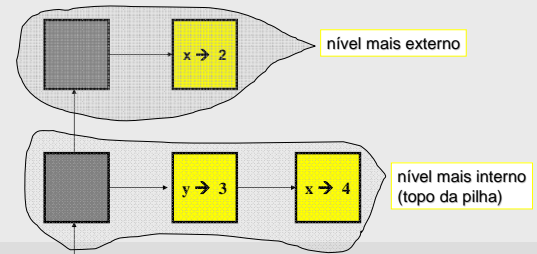
© Luis Caires

LP2 2005/06

108

Ambiente

- ◆ Implementado como pilha de dicionários ...



© Luis Caires

LP2 2005/06

109

Interpretador de CALCI

- ◆ Algoritmo $eval(E, env)$ para calcular o valor de uma expressão qualquer E da linguagem CALCI:

$eval : CALCI \times ENV \rightarrow integer$

```

eval( num(n) , env)  $\triangleq$  return n
eval( id(s) , env)  $\triangleq$  return env.Find(s)
eval( add(E1,E2) , env)  $\triangleq$  return eval(E1, env) + eval(E2, env)
...
eval( decl(s, E1, E2), env)  $\triangleq$  [
    env = env.BeginScope();
    env.Assoc(s,eval(E1, env));
    val = eval(E2, env);
    env = env.EndScope(); return val; ]
    
```

© Luis Caires

LP2 2005/06

110

Linguagens Imperativas

© Luis Caires

LP2 2005/06

111

Linguagens Imperativas

- ◆ As expressões das linguagens até agora consideradas denotam sempre **valores puros**
- ◆ Em particular, as variáveis denotam um valor que se mantém fixo durante a execução
- ◆ Actualmente, o paradigma de programação dominante é o paradigma imperativo, baseado na **mutação** de estado (C, Java).
- ◆ Uma linguagem de programação imperativa caracteriza-se por incluir primitivas que permitem:
 - associar posições de memória a variáveis (var x:integer)
 - alterar o estado da memória (x := E)

© Luis Caires

LP2 2005/06

112

Modelo de Memória

- ◆ Memória: conjunto (potencialmente infinito) de **células** cujo conteúdo é **mutável**.
- ◆ Cada célula tem um designador **único**, designado a **referência** da célula, e pode conter **qualquer valor**.
- ◆ 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:    void  $\rightarrow$  ref
set:    ref  $\times$  Value  $\rightarrow$  void
get:    ref  $\rightarrow$  Value
free:   ref  $\rightarrow$  void
    
```

© Luis Caires

LP2 2005/06

113

Modelo de Memória

- ◆ Operações sobre uma memória \mathcal{M}

```
new:    void  $\rightarrow$  ref
```

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

```
set:    ref  $\times$  Value  $\rightarrow$  void
```

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

```
get:    ref  $\rightarrow$  Value
```

Devolve o valor contido na célula referida.

```
free:   ref  $\rightarrow$  void
```

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

© Luis Caires

LP2 2005/06

114

Ambiente versus Memória

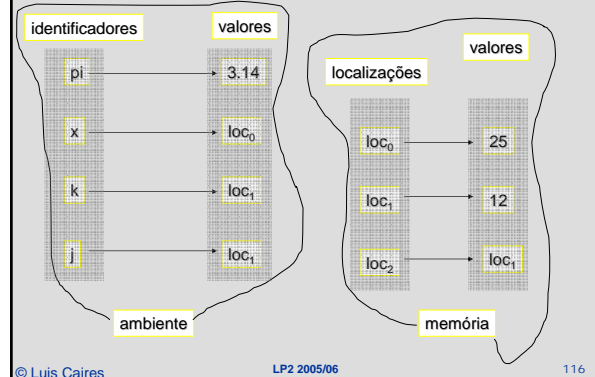
- ◆ O **ambiente** indica a denotação de cada identificador, e reflecte a estrutura estática do programa.
- ◆ A associação estabelecida no ambiente entre um identificador e o seu valor denotado é fixa e imutável dentro do âmbito respectivo.
- ◆ A **memória** agrega o conteúdo das variáveis de estado mutável, indicando o valor contido em cada localização (referência).
- ◆ O conteúdo de cada localização é mutável.

© Luis Caires

LP2 2005/06

115

Ambiente versus Memória



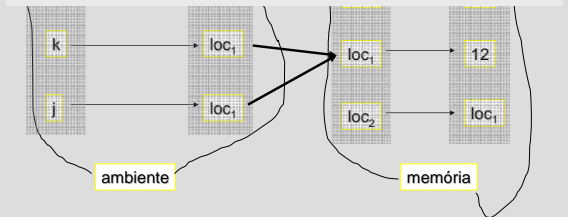
© Luis Caires

LP2 2005/06

116

Propriedades do Modelo

- ◆ Uma mesma célula de memória pode ser referida por vários identificadores distintos (**aliasing**).
- ◆ Uma célula pode conter uma referência para outra célula, permitindo a construção de estruturas de dados.



© Luis Caires

LP2 2005/06

117

Operações Imperativas

- ◆ Reserva de nova célula

cell(E)

E: expressão qualquer

- ◆ Exemplos:

```
{ int a; ... }
malloc(sizeof(int));
new MyClass();
```

© Luis Caires

LP2 2005/06

118

Operações Imperativas

- ◆ Afecção

E := F

E: expressão que denota uma referência

F: expressão qualquer

- ◆ Exemplos:

```
i := 2
b[x+2][b[x-2]] = 2
*(p+2) = y
myTable(i,j) = myTable(j,i)
Readln(MyLine);
```

© Luis Caires

LP2 2005/06

119

Operações Imperativas

- ◆ Desreferenciação

IE

E: expressão que denota uma referência

- ◆ Exemplos:

```
i := li + 1           (linguagem ML)
*p                    (linguagem C)

i = i + 1             (linguagem C)
i++                   (linguagem C)
```

© Luis Caires

LP2 2005/06

120

Operações Imperativas

◆ Desreferenciação

!E

E: expressão que denota uma referência

referência

◆ Exemplos:

i := !i + 1 (linguagem ML)
 *p (linguagem C)
 i = i + 1 (linguagem C)
 i++ (linguagem C)

valor

referência

© Luis Caires

LP2 2005/06

121

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

© Luis Caires

LP2 2005/06

122

L-Value e R-Value

- ◆ Se uma expressão E tem por valor uma referência, a maior parte das linguagens de programação interpreta E de maneira **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.

© Luis Caires

LP2 2005/06

123

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 muito mais difíceis de ler.
- ◆ A desreferenciação implícita pode ser vista como uma operação de **coerção** (conversão / cast).

© Luis Caires

LP2 2005/06

124

Operações Imperativas

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

newvar(_) instanciação
 free(_) libertação
 _ := _ afectação
 !_ desreferenciação

```

/* linguagem C */
const int k = 2;
int a = k;
int b = a + 2;
... b = a * b ...
}

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

© Luis Caires

LP2 2005/06

125

Operações Imperativas

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

cell(_) instanciação
 free(_) libertação
 _ := _ afectação
 !_ desreferenciação

```

/* linguagem C */
const int k = 2;
int a = k;
int b = a + 2;
... b = a * b ...
}

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

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

© Luis Caires

LP2 2005/06

126

Operações Imperativas

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

cell(_) instanciação
free(_) libertação
_ := _ afectação
! _ desreferenciação

```

/* linguagem C */
int k = 2;
int &a = k;
... a = k+a ...
}

decl
k = newvar(2)
a = newvar(k)
in ... !a := !k+!!a ...
free(k); free(a) end
    
```

© Luis Caires

LP2 2005/06

127

Operações Imperativas

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

newvar(_) instanciação
free(_) libertação
_ := _ afectação
! _ desreferenciação

```

/* linguagem C */
int k = 2;
int &a = k;
int b = a;
... a = k+b ...
}

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

© Luis Caires

LP2 2005/06

128

Operações Imperativas

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

newvar(_) instanciação
free(_) libertação
_ := _ afectação
! _ desreferenciação

```

/* linguagem C */
int k = 2;
int *a = &k;
... k = k+a ...
}

decl
k = newvar(2)
a = newvar(k)
in ... k := !k+!a ...
free(k); free(a) end
    
```

© Luis Caires

LP2 2005/06

129

Tempo de Vida (de uma célula)

- ◆ O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando **newvar(_)** e a sua libertação usando **free(_)**.
- ◆ Em muitas situações, o tempo de vida da célula **concide com o âmbito do(s) seu(s) identificador**.

```

/* linguagem C */
int k = 2;
...
}

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

reserva (de cell(k))
 libertação implícita (de cell(k))
 reserva (de cell(k))
 libertação implícita (de cell(k)) (retorna "dangling pointer")

© Luis Caires

LP2 2005/06

130

Tempo de Vida

- ◆ O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando **newvar(_)** e a sua libertação usando **free(_)**.
- ◆ Noutras situações, o tempo de vida da célula **extravasa o âmbito do(s) seu(s) identificador**.

```

/* linguagem C */
static int k = 2;
...
}

/* Java */
Integer foo(int j) {
Integer k = new Integer(2);
return k;
}
    
```

âmbito de k
 reserva (de cell(k))
 O tempo de vida de cell(k) é o do programa
 reserva de novo objecto Integer
 libertação implícita (de cell(k)) mas o objecto sobrevive ao bloco!

© Luis Caires

LP2 2005/06

131

Tempo de Vida

- ◆ O tempo de vida de uma célula é o tempo que medeia entre a sua criação / reserva usando **newvar(_)** e a sua libertação usando **free(_)**.
- ◆ Noutras situações, o tempo de vida da célula **extravasa o âmbito do(s) seu(s) identificador**.

```

/* linguagem C */
static int k = 2;
...
}

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

âmbito de k
 reserva (de cell(k))
 O tempo de vida de cell(k) é o do programa
 libertação implícita (de cell(k)) mas *k sobrevive ao bloco!

© Luis Caires

LP2 2005/06

132

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**.

© Luis Caires

LP2 2005/06

133

Uma linguagem de tipo ALGOL

◆ Definida com base em duas categorias sintácticas: EXP (expressões) e COM (comandos)

```
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
```

© Luis Caires

LP2 2005/06

134

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 afectação ($x = y$) é uma expressão, e expressões podem produzir efeitos ($i++$).

© Luis Caires

LP2 2005/06

135

Exemplo: A linguagem microML

◆ Consideramos uma só categoria sintáctica (EXP):

```
num:   integer → EXP
bool:  boolean → EXP
id:    string → EXP
add:   EXP × EXP → EXP
var:   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
```

© Luis Caires

LP2 2005/06

136

Semântica de microML

- ◆ A semântica de uma linguagem imperativa pode ser caracterizada por uma função:

$$I : P \times ENV \times MEM \rightarrow VAL \times MEM$$

P = Fragmentos de programa (abertos)
ENV = Ambientes (funções $ID \rightarrow VAL$)
MEM = Memórias
VAL = Valores (Denotações) = (**bool** \cup **int** \cup **ref**)

- ◆ Traduz a intuição que, em geral, um fragmento P produz um **valor** e gera um **efeito** (na memória).

© Luis Caires

LP2 2005/06

137

Semântica de microML

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

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

```
eval( num(n) , env , mem)  ≙ return ( n , mem )
eval( id(s) , env , mem)  ≙ return ( env.Find(s) , mem )

eval( true , env , mem)   ≙ (T , mem)
eval( false , env , mem)  ≙ (F , mem)
```

© Luis Caires

LP2 2005/06

138

Semântica de microML

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

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

```
eval( add(E1, E2), env, m0) ≜  
[  
  (v1, m1) = eval( E1, env, m0)  
  (v2, m2) = eval( E2, env, m1)  
  return (v1 + v2, m2);  
]
```

© Luis Caires

LP2 2005/06

139

Semântica de microML

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

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

```
eval( and(E1, E2), env, m0) ≜  
[  
  (v1, m1) = eval( E1, env, m0)  
  (v2, m2) = eval( E2, env, m1)  
  return (v1 & v2, m2);  
]
```

© Luis Caires

LP2 2005/06

140

Semântica de microML

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

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

```
eval( var(E), env, m0) ≜  
[ (v1, m1) = eval( E, env, m0);  
  (ref, m2) = m1.new(v1)  
  return (ref, m2); ]
```

© Luis Caires

LP2 2005/06

141

Semântica de microML

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

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

```
eval( deref(E), env, m0) ≜  
[  
  (v, m1) = eval( E, env, m0);  
  return (m1.get(v), m1);  
]
```

© Luis Caires

LP2 2005/06

142

Semântica de microML

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

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

```
eval( assign(E1, E2), env, m0) ≜  
[  
  (v1, m1) = eval( E1, env, m0);  
  (v2, m2) = eval( E2, env, m1);  
  m3 = m2.set(v1, v2);  
  return (v1, m3); ]
```

© Luis Caires

LP2 2005/06

143

Semântica de microML

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

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

```
eval( seq(E1, E2), env, m0) ≜  
[  
  (v1, m1) = eval( E1, env, m0);  
  (v2, m2) = eval( E2, env, m1);  
  return (v2, m2);  
]
```

© Luis Caires

LP2 2005/06

144

Semântica de microML

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

eval : microML × ENV × MEM → VAL × MEM

```
eval( if(E1, E2, E3) , env , m0) ≐
[
  (v1 , m1) = eval( E1 , env , m0);
  if (v1 = T) then (v2 , m2) = eval( E1 , env , m1);
  else (v2 , m2) = eval( E2 , env , m1);
  return (v2 , m2) ; ]
```

Semântica de microML

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

eval : microML × ENV × MEM → VAL × MEM

```
eval( while(E1, E2) , env , m0) ≐
[ (v1 , m1) = eval( E1 , env , m0);
  if (v1 = T) then [ (v2 , m2) = eval( E2 , env , m1);
    return eval( while(E1, E2), m2) ]
  else return (F , m1) ]
```

Semântica de microML

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

eval : microML × ENV × MEM → VAL × MEM

```
eval( while(E1, E2) , env , m0) ≐
[ (v1 , m1) = eval( E1 , env , m0);
  if (v1 = T) then [ (v2 , m2) = eval( E2 , env , m1);
    return eval( while(E1, E2), m2) ]
  else return (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 × ENV × MEM → VAL × MEM

```
eval( decl(s, E1, EB) , env , m0) ≐
[
  envlocal = env.BeginScope();
  (v1 , m1) = eval( E1 , envlocal , m0);
  envlocal.Assoc(s, v1);
  (v2 , m2) = eval(EB, envlocal , m1);
  env = envlocal.EndScope();
  return (v2 , m2) ; ]
```

Semântica de microML

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

eval : microML × ENV × MEM → VAL × MEM

```
decl a = newvar(2) in
decl b = newvar(!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 × ENV × MEM → VAL × MEM

```
decl a = newvar(2) in
decl b = newvar(!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.

Compilação de microML

- ◆ A compilação de microML pode ser caracterizada por uma função:

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

P = Fragmento de programa (aberto)
ENV = Ambiente (função $ID \rightarrow int$)
 Atribui a cada identificador uma posição (número inteiro) na pilha de chamada
CodeSeq = Sequências de instruções

© Luis Caires

LP2 2005/06

151

Compilação de microML

- ◆ Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

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

$\text{comp}(\text{num}(n), \text{env}) \triangleq \text{return} \langle \text{ldc.i4 } n \rangle$
 $\text{comp}(\text{id}(s), \text{env}) \triangleq \text{return} \langle \text{ldloc } \text{env.Find}(s) \rangle$
 $\text{comp}(\text{true}, \text{env}) \triangleq \text{return} \langle \text{ldc.i4 } 1 \rangle$
 $\text{comp}(\text{false}, \text{env}) \triangleq \text{return} \langle \text{ldc.i4 } 0 \rangle$

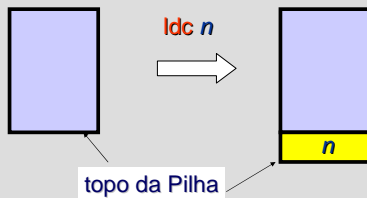
© Luis Caires

LP2 2005/06

152

Common Language Runtime

- ◆ “Primeiras” (5) instruções: **ldc n**, **add**, **mul**, **div**, **sub**.
- ◆ Load Constant (**ldc n**)



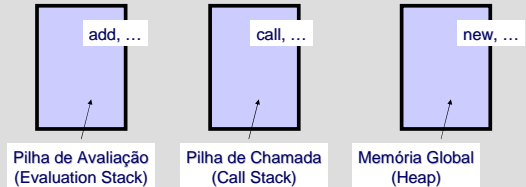
© Luis Caires

LP2 2005/06

153

Common Language Runtime

- ◆ Estruturas principais da máquina virtual do CLR
- ES:** guarda resultados temporários durante a avaliação de expressões;
CS: guarda variáveis locais e endereços de retorno;
Heap: guarda objectos e outras entidades dinâmicas



© Luis Caires

LP2 2005/06

154

Common Language Runtime

- ◆ Segundo “lote” de instruções:

ldloc n
 Empilha no ES o conteúdo da variável local na posição n

stloc n
 Desempilha o topo do ES e guarda-o na variável local na posição n

pop
 Desempilha o topo do ES, descartando-o.

brfalse label
 Desempilha o topo do ES e continua a execução a partir de *label*, se o valor desempilhado for zero.

br label
 Continua a execução a partir da *label*.

© Luis Caires

LP2 2005/06

155

Compilação de microML

- ◆ Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

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

$\text{comp}(\text{add}(E1, E2), \text{env}) \triangleq$
 [
 $s1 = \text{comp}(E1, \text{env})$
 $s2 = \text{comp}(E2, \text{env})$
 $\text{return } s1 @ s2 @ \langle \text{add} \rangle$
]

© Luis Caires

LP2 2005/06

156

Compilação de microML

- ◆ Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

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

```
comp( and(E1, E2), env) ≙
[
  s1 = comp( E1, env)
  s2 = comp( E2, env)
  return s1 @ s2 @ < and >
]
```

© Luis Caires

LP2 2005/06

157

Compilação de microML

- ◆ Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

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

```
comp( newvar(E), env) ≙
[
  s1 = comp(E, env);
  s2 = s1 @ < newobj instance void Cell::ctor(int32) >
  return s2;
]
```

© Luis Caires

LP2 2005/06

158

Common Language Runtime

- ◆ Terceiro “lote” de instruções:

- **newobj** *constructor-spec*

Empilha no ES uma referência para um novo objecto, criado no Heap, e tal como especificado pelo *constructor*.

Exemplo:

```
newobj instance void Cell::ctor(int32)
```

- **callvirt** *method-spec*

Chama o método *method*, os argumentos e objecto são desempilhados do topo do ES.

Exemplo:

```
ldloc.4 // load object reference
ldc.i4.s 2 // load parameter value
callvirt instance int Cell::set(int32) // call set method
```

© Luis Caires

LP2 2005/06

159

Implementação da “Memória”

- ◆ Representação das células de memória, a definir no ambiente de suporte à execução (*Runtime support system*), programado em C# (poderia ser programado directamente em MSIL!).

```
public class Cell {
  int v;
  public Cell(int n) { v = n; }
  public int get() { return v; }
  public int set(int n) { v = n; return v; }
}
```

© Luis Caires

LP2 2005/06

160

Compilação de microML

- ◆ Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

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

```
comp( deref(E), env) ≙
[
  s1 = comp( E, env);
  s2 = s1 @ < callvirt instance int32 Cell::get() >
  return s2;
]
```

© Luis Caires

LP2 2005/06

161

Compilação de microML

- ◆ Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

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

```
comp( assign(E1, E2), env) ≙
[
  s1 = comp( E1, env);
  s2 = comp( E2, env);
  s3 = s1 @ s2 @ < callvirt instance int Cell::set(int32) >
  return s3;
]
```

© Luis Caires

LP2 2005/06

162

Compilação de microML

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

comp : microML × ENV → CodeSeq

```
comp( seq(E1, E2), env ) ≙
[
  s1 = comp( E1, env );
  s2 = comp( E2, env );
  return s1 @ < pop > @ s2 ;
]
```

Descartar o valor de E1

© Luis Caires

LP2 2005/06

163

Compilação de microML

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

comp : microML × ENV → CodeSeq

```
comp( if(E1, E2, E3), env ) ≙
[
  s1 = comp( E1, env ); s2 = comp( E2, env );
  s3 = comp( E3, env );
  return s1 @ < brfalse L1 >
    @ s2 < br L2 > @ < L1: > s3 @ < L2: nop > ]
```

© Luis Caires

LP2 2005/06

164

Compilação de microML

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

comp : microML × ENV → CodeSeq

```
comp( while(E1, E2), env ) ≙
[
  s1 = comp( E1, env ); s2 = comp( E2, env );
  return < L1: > s1 @ < brfalse L2 >
    @ s2 < pop; br L1 > @ < L2: push 0 > ]
```

© Luis Caires

LP2 2005/06

165

Compilação de microML

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

comp : microML × ENV → CodeSeq

```
comp( decl(s, EI, EB), env ) ≙
[
  envlocal = env.BeginScope();
  s1 = comp( EI, envlocal );
  pos = env.getNumDecls();
  envlocal.Assoc(s, pos);
  s2 = eval(EB, envlocal);
  env = envlocal.EndScope();
  return s1 @ < stloc pos > @ s2; ]
```

© Luis Caires

LP2 2005/06

166

Compilação de microML

- Algoritmo comp para traduzir o valor de uma expressão qualquer da linguagem microML:

comp : microML × ENV → CodeSeq

```
comp( decl(s, EI, EB), env ) ≙
[
  envlocal = env.BeginScope();
  s1 = comp( EI, envlocal );
  pos = env.getNumDecls();
  envlocal.Assoc(s, pos);
  s2 = eval(EB, envlocal);
  env = envlocal.EndScope();
  return s1 @ < stloc pos > @ s2; ]
```

Número de ligações no ambiente env

© Luis Caires

LP2 2005/06

167

Compilação de microML

- O número de variáveis locais necessárias é igual à profundidade lexical do programa (ou seja, o número máximo de declarações encaixadas).

```
.locals init(
  object V_1,
  object V_2,
  ...
  object V_n)
```

Directiva para o assembler CLR reservar espaço para variáveis temporárias

n = Número máximo de ligações no ambiente durante a compilação

© Luis Caires

LP2 2005/06

168

Compilação de microML

◆ Exemplo 1

```
decl
  a = 2
in
  a + 4

.locals(object V_0)
ldc.i4 2
stloc.0
ldloc.0
ldc.i4 4
add
// <epilogo>
```

© Luis Caires

LP2 2005/06

169

Compilação de microML

◆ Exemplo 2

```
(decl x = 2
 in x + 4) *
(decl y = 3
 in y + y)

.locals(object V_0)
ldc.i4 2 // 2
stloc.0 // init x
ldloc.0 // fetch x
ldc.i4 4 // 4
add // x + 4

ldc.i4 3 // 3
stloc.0 // init y
ldloc.0 // fetch y
ldloc.0 // fetch y
add
mul
// <epilogo>
```

- ◆ Durante a compilação de "x+4" tem-se Env("x")=0
- ◆ Durante a compilação de "y+y" tem-se Env("y")=0

© Luis Caires

LP2 2005/06

170

Compilação de microML

◆ Exemplo 3

```
decl x = 2
in decl y = 3
  in x + y

.locals( object V_0,
         object V_1)
ldc.i4 2 // 2
stloc.0 // init x
ldc.i4 // 3
stloc.1 // init y
ldloc.0 // fetch x
ldloc.1 // fetch y
add
// <epilogo>
```

- ◆ Durante a compilação de "x+y" tem-se Env("x")=0 e Env("y")=1.

© Luis Caires

LP2 2005/06

171

Compilação de microML

◆ Exemplo 3

```
decl x = newvar(2)
in decl y = 3
  in !x + y

.locals( object V_0,
         object V_1)
ldc.i4 2 // 2
new Cell // create cell
stloc.0 // init x
ldc.i4 // 3
stloc.1 // init y
ldloc.0 // fetch x
callvirt Cell::get // deref
ldloc.1 // fetch y
add
// <epilogo>
```

- ◆ Durante a compilação de "x+y" tem-se Env("x")=0 e Env("y")=1.
- ◆ Durante a execução, V_0 vai conter uma referência para uma célula (objecto de classe Cell), e V_1 um valor inteiro.

© Luis Caires

LP2 2005/06

172

Compilação de microML

◆ Exemplo 3

```
decl x = newvar(2)
in decl y = 3
  in x := y + 10

.locals( object V_0,
         object V_1)
ldc.i4 2 // 2
new Cell // create cell
stloc.0 // init x
ldc.i4 3 // 3
stloc.1 // init y
ldloc.0 // fetch x (cell ref)
ldloc.1 // fetch y
ldc.i4 // 10
add
callvirt Cell::set // assign
// <epilogo>
```

- ◆ Durante a compilação de "x+y" tem-se Env("x")=0 e Env("y")=1.
- ◆ Durante a execução, V_0 vai conter uma referência para uma célula (objecto de classe Cell), e V_1 um valor inteiro.

© Luis Caires

LP2 2005/06

173

Compilação de microML

◆ Exemplo 4

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

.locals( object V_0,
         object V_1,
         object V_2)
ldc.i4 2 // 2
new Cell // create cell
stloc.0 // init a
ldloc.0 // fetch a
callvirt Cell::get // deref
new Cell // create cell
stloc.1 // init b
ldloc.0 // fetch a (cell ref)
stloc.2 // init c

ldloc.0 // a
ldloc.1 // b
ldc.i4 2 // 2
add
callvirt Cell::set // assign
pop // discard

ldloc.2 // fetch c
ldloc.2 // fetch c
callvirt Cell::get // deref
ldc.i4 2 // 2
add
callvirt Cell::set // assign
// <epilogo>
```

© Luis Caires

LP2 2005/06

174

Compilação de microML

◆ Exemplo 4

```
decl s = newvar(0) in (
  decl b = newvar(100) in
    while ( b > 0 ) do
      s := !s + !b;
      b := !b - 1;
    end;
  !s )
```

```
.Jocals( object V_0,
         object V_1)
ldc.i4 0 // 0
new Cell // create cell
stloc.0 // init s
ldc.i4 0 // 100
new Cell // create cell
stloc.1 // init b
L00:
ldloc.1 // fetch b
ldc.i4 0 // 0
cgt
brfalse L01:// done!
```

```
ldloc.0 // s
ldloc.0 // s
callvirt Cell:get // deref
ldloc.0 // b
callvirt Cell:get // deref
add
callvirt Cell:set // assign
pop // discard
ldloc.1 // b
ldloc.1 // b
callvirt Cell:get // deref
ldc.i4 1 // 1
sub
callvirt Cell:set // assign
pop
br L00 // loop
L01: ...
// <epilogo>
```

© Luis Caires

LP2 2005/06

Semântica de microML

◆ A função semântica

$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$
 está **indefinida** em várias situações.[Quais?]

© Luis Caires

LP2 2005/06

176

Semântica de microML

◆ A função semântica

$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$
 está **indefinida** em várias situações.[Quais?]

Aritmética com valores que não são inteiros:

```
( true + 2 )
decl v = newvar(2) in v + 2 end
```

© Luis Caires

LP2 2005/06

177

Semântica de microML

◆ A função semântica

$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$
 está **indefinida** em várias situações.[Quais?]

Operações lógicas com valores não booleanos:

```
( 2 and (2 > myconst) )
if newvar(2) then mynum := 2 else mynum := 0
```

© Luis Caires

LP2 2005/06

178

Semântica de microML

◆ A função semântica

$eval : microML \times ENV \times MEM \rightarrow VAL \times MEM$
 está **indefinida** em várias situações.[Quais?]

(Tentativas de) acesso a memória com valores que não são referências:

```
decl x = 2 in x := x + 1 end
decl x = 2 in 2 + !x end
```

© Luis Caires

LP2 2005/06

179

Semântica de microML

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

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

◆ Hum... a semântica definida por eval é parcial.

A função **eval** pode ser tornada **total** introduzindo um valor especial **error**, que denota as computações "sem sentido".

$VAL = \text{Valores (Denotações)} = (\text{bool} \cup \text{int} \cup \text{ref} \cup \{\text{error}\})$

Predicados para testar o domínio de um valor $v \in VAL$:

$is_integer(v); is_boolean(v); is_reference(v); is_error(v);$

© Luis Caires

LP2 2005/06

180

Semântica de microML

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

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

```
eval( add(E1, E2), env, m0) ≐  
[  
  (v1, m1) = eval( E1, env, m0);  
  (v2, m2) = eval( E2, env, m1);  
  if (is_integer(v1) & is_integer(v2))  
  then return (v1 + v2, m2);  
  else return (error, m2);  
]
```

© Luis Caires

LP2 2005/06

181

Semântica de microML

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

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

```
eval( assign(E1, E2), env, m0) ≐  
[  
  (v1, m1) = eval( E1, env, m0);  
  (v2, m2) = eval( E2, env, m2);  
  if (is_reference(v1))  
  then [m3 = m2.set(v1, v2); return (v1, m3)];  
  else return (error, m2);  
]
```

© Luis Caires

LP2 2005/06

182

Semântica de microML

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

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

```
eval( if(E1, E2, E3), env, m0) ≐  
[ (v1, m1) = eval( E1, env, m0);  
  if (not is_boolean(v1)) then return (error, m1)  
  else [ if (v1 = T)  
        then (v2, m2) = eval( E2, env, m1);  
        else (v2, m2) = eval( E3, env, m1);  
        return (v2, m2); ]  
]
```

© Luis Caires

LP2 2005/06

183

Sistemas de Tipos (1)

Análise Estática

- ◆ A análise estática é uma forma de interpretação “abstracta” que garante certas (boas) propriedades da interpretação “concreta” pretendida.
- ◆ A contrário da avaliação, que pode não terminar, a análise estática de um programa **termina sempre**.
- ◆ A propriedades garantida pela análise é a ausência de (certos tipos de) erros durante a execução.
- ◆ A técnica predominante de análise estática é a **verificação de tipos** (type checking).

© Luis Caires

LP2 2005/06

185

Duas Semânticas

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

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

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

$typchk : \text{microML} \times \text{TENV} \rightarrow \text{TYPE}$

$\text{TENV} : \text{ID} \rightarrow \text{TYPE}$

$\text{TYPE} = (\text{tipos}) = \{ \text{int}, \text{bool}, \text{ref}[\text{TYPE}], \text{none} \}$

- ◆ A função **typchk** pode ser vista como um interpretador que avalia um programa de acordo com uma semântica especial, mais abstracta, (em que os “tipos” são “valores”).

© Luis Caires

LP2 2005/06

186

Tipos para microML

- ◆ Algoritmo `typchk` para calcular o tipo de uma expressão qualquer da linguagem microML:

$\text{typchk} : \text{microML} \times \text{ENV} \rightarrow \text{TYPE}$

$\text{TYPE} = (\text{tipos}) = \{ \text{int}, \text{bool}, \text{ref}[\text{TYPE}], \text{none} \}$

- ◆ **int** é o tipo dos valores inteiros.
- ◆ **bool** é o tipo dos valores booleanos.
- ◆ **ref[τ]** é o tipo das referências para células que só podem conter valores de tipo τ . Exemplo: `ref[ref[int]]` é o tipo das referências para células que só podem conter referências para (células) com valores inteiros.
- ◆ **none** é o "tipo" das expressões indefinidas.

© Luis Caires

LP2 2005/06

187

Tipificação de microML

- ◆ Algoritmo `typchk` para calcular o tipo de uma expressão qualquer da linguagem microML:

$\text{typchk} : \text{microML} \times \text{TENV} \rightarrow \text{TYPE}$

$\text{typchk}(\text{num}(n), \text{tenv}) \triangleq \text{return int};$

$\text{typchk}(\text{id}(s), \text{tenv}) \triangleq \text{return env.Find}(s);$

$\text{typchk}(\text{true}, \text{tenv}) \triangleq \text{return bool};$

$\text{typchk}(\text{false}, \text{tenv}) \triangleq \text{return bool};$

© Luis Caires

LP2 2005/06

188

Tipificação de microML

- ◆ Algoritmo `typchk` para calcular o tipo de uma expressão qualquer da linguagem microML:

$\text{typchk} : \text{microML} \times \text{TENV} \rightarrow \text{TYPE}$

```
typchk( add(E1, E2), tenv )  $\triangleq$ 
[
  t1 = typchk ( E1, tenv )
  t2 = typchk ( E2, tenv )
  if ( t1 == int ) and ( t2 == int )
    then return int
    else return none ; ]
```

© Luis Caires

LP2 2005/06

189

Tipificação de microML

- ◆ Algoritmo `typchk` para calcular o tipo de uma expressão qualquer da linguagem microML:

$\text{typchk} : \text{microML} \times \text{TENV} \rightarrow \text{TYPE}$

```
typchk( and(E1, E2), tenv )  $\triangleq$ 
[ t1 = typchk ( E1, tenv )
  t2 = typchk ( E2, tenv )
  if ( t1 == bool ) and ( t2 == bool )
    then return bool
    else return none ; ]
```

© Luis Caires

LP2 2005/06

190

Tipificação de microML

- ◆ Algoritmo `typchk` para determinar o tipo de uma expressão qualquer da linguagem microML:

$\text{typchk} : \text{microML} \times \text{TENV} \rightarrow \text{TYPE}$

```
typchk( assign(E1, E2), tenv )  $\triangleq$ 
[ t1 = typchk( E1, tenv );
  t2 = typchk( E2, tenv );
  if ( t1 == ref[t2] )
    then return t2;
    else return none ; ]
```

© Luis Caires

LP2 2005/06

191

Tipificação de microML

- ◆ Algoritmo `typchk` para calcular o tipo de uma expressão qualquer da linguagem microML:

$\text{typchk} : \text{microML} \times \text{TENV} \rightarrow \text{TYPE}$

```
typchk( if(E1, E2, E3), tenv )  $\triangleq$ 
[ t1 = typchk( E1, tenv );
  if ( t1 != bool ) then return none
  else [ t2 = typchk( E2, tenv );
        t3 = typchk( E3, tenv );
        if ( t2 == none ) or ( t3 == none ) or ( t2 != t3 )
          then return none
          else return t2; ]
```

© Luis Caires

LP2 2005/06

192

Tipificação de microML

- ◆ Compare com o caso if do algoritmo eval (slide 124). Aqui: Os "ramos" E2 e E3 são **ambos** analisados. Impõe-se (como restrição) $t2 == t3$. [Porquê?]

```

typchk( if(E1, E2, E3) , tenv )  $\triangleq$ 
[ t1 = typchk( E1, tenv );
  if ( t1 != bool ) then return none
  else [ t2 = typchk( E2, tenv );
        t3 = typchk( E3, tenv );
        if ( t2 == none ) or ( t3 == none ) or ( t2 != t3 )
        then return none
        else return t2; ]

```

Tipificação de microML

- ◆ Algoritmo typchk para calcular o tipo de uma expressão qualquer da linguagem microML:

$\text{typchk} : \text{microML} \times \text{TENV} \rightarrow \text{TYPE}$

```

typchk( while(E1, E2) , tenv )  $\triangleq$ 
[ t1 = typchk( E1, tenv );
  if ( t1 != bool ) then return none
  else [ t2 = typchk( E2, tenv );
        if ( t2 == none ) then return none
        else return bool ]
]

```

Tipificação de microML

- ◆ Compare com o caso while do algoritmo eval. Aqui: O "ramo" E2 é sempre analisado exactamente **uma vez**. Impõe-se como tipo o tipo **bool**. [Porquê?]

```

typchk( while(E1, E2) , tenv )  $\triangleq$ 
[ t1 = typchk( E1, tenv );
  if ( t1 != bool ) then return none
  else [ t2 = typchk( E2, tenv );
        if ( t2 == none ) then return none
        else return bool ]
]

```

Tipificação de microML

- ◆ Algoritmo typchk para calcular o tipo de uma expressão qualquer da linguagem microML:

$\text{typchk} : \text{microML} \times \text{TENV} \rightarrow \text{TYPE}$

```

typchk( decl(s, E1, EB) , env )  $\triangleq$ 
[   envLocal = env.BeginScope();
    t1 = typchk( E1, envLocal );
    envLocal.Assoc(s, t1 );
    t2 = typchk( EB, envLocal );
    env = envLocal.EndScope();
    return t2 ; ]

```

Correcção da Tipificação

- ◆ Podemos verificar que, para todo o programa miniML P, e ambiente *env* que cubra todos os identificadores livres de P, a operação $\text{typchk}(P, \text{env})$ está bem definida e termina sempre [Porquê?]
- ◆ Podemos também demonstrar o seguinte teorema, que relaciona a tipificação com a avaliação de programas microML. [Como?]

Teorema: Para todo o programa μML P e tipo τ ,
Se $\text{typchk}(P, \emptyset) = \tau$ então $\text{eval}(P, \emptyset, \emptyset) = v \in \tau$.

Correcção da Tipificação

Teorema: Para todo o programa μML P e tipo τ ,
Se $\text{typchk}(P, \emptyset) = \tau$ então $\text{eval}(P, \emptyset, \emptyset) = v \in \tau$.

Em particular, se $\text{typchk}(P, \emptyset) \neq \text{none}$ então $\text{eval}(P, \emptyset, \emptyset) \neq \text{error}$.

Ou seja:

ou P não termina,

ou P termina e **não incorre** em erros de execução.

"Well-typed programs don't go wrong" (Robin Milner)

Correcção da Tipificação

Teorema: Para todo o programa μML P e tipo τ ,
Se $\text{typchk}(P, \emptyset) = \tau$ então $\text{eval}(P, \emptyset, \emptyset) = v \in \tau$.

Esta propriedade é em geral conhecida como a
"consistência do sistema de tipos". Garante que

"Well-typed programs don't go wrong" (Robin Milner)

Existem muitos programas P tais que $\text{eval}(P, \emptyset, \emptyset) = \text{int}$
mas (infelizmente) $\text{typchk}(P, \emptyset) = \text{error}$! [Exemplo?]
Não é possível obter sistemas de tipos mais precisos, se a
linguagem de programação for Turing-completa [Porquê?]

© Luis Caires

LP2 2005/06

199

Robin Milner

ACM Turing Award (1991)

For three distinct and complete achievements:

- 1) LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;
- 2) ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;
- 3) CCS, a general theory of concurrency. In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.



© Luis Caires

LP2 2005/06

200

Abstracção Funcional (1)

© Luis Caires

LP2 2005/06

201

Parametrização e Abstracção

- ◆ A possibilidade de se definirem entidades genéricas, introduzidas uma vez e utilizáveis várias para instanciações diferentes de parâmetros, é uma característica fundamental de todas as linguagens de programação:
 - Funções / procedimentos
 - Métodos
 - Tipos paramétricos
 - Classes paramétricas
- ◆ Os mecanismos comuns em todos estes exemplos são as noções de **parametrização** e **abstracção**.
- ◆ A parametrização e a abstracção são essenciais à modularidade dos programas, e em alguns casos, à expressividade computacional das linguagens.

© Luis Caires

LP2 2005/06

202

Parametrização

- ◆ É o mecanismo geral para declarar os nomes que se querem encarrar como genéricos, destacando-os sintacticamente através de notação conveniente:

- Whitehead and Russel: $\lambda x. x+1$
- C: `int f(int x) { return x+1; }`
- ML: `(fun x -> x+2)`
- Lisp: `(lambda (x) (add x 1))`
- Cálculo Lambda (Church): $\lambda x. x$

- ◆ Tecnicamente, chama-se **abstracção** a uma expressão sintáctica, explicitamente parametrizada num certo conjunto de nomes.

© Luis Caires

LP2 2005/06

203

Abstracção

- ◆ Uma **abstracção** é uma expressão sintáctica constituída por dois elementos fundamentais:
 - As **declarações** dos seus parâmetros
 - O seu **corpo** (uma qualquer expressão da linguagem)
- ◆ $(x \rightarrow x+y)$
 - Parâmetros declarados: x ; Corpo: $x+y$
- ◆ $(y \rightarrow (x \rightarrow x+y))$
 - Parâmetros declarados: y ; Corpo: $(x \rightarrow x+y)$
- ◆ $(y, x \rightarrow x+y)$
 - Parâmetros declarados: x, y ; Corpo: $x+y$
- ◆ Uma abstracção representa uma função anónima

© Luis Caires

LP2 2005/06

204

Observações

- ◆ As declarações dos parâmetros de uma abstracção são ocorrências **ligantes** dos nomes respectivos
- ◆ O âmbito das declarações dos parâmetros é (exactamente) o corpo da abstracção
- ◆ Os nomes livres numa abstracção são todos os nomes livres no seu corpo que não são parâmetros
 - $\text{NomesLivres}((x_1, \dots, x_n \rightarrow E) = \text{NomesLivres}(E) \setminus \{x_1, \dots, x_n\}$
 - $\text{NomesLivres}((x \rightarrow x+y) = \text{NomesLivres}(x+y) \setminus \{x\} = \{y\}$
- ◆ O significado de uma abstracção depende da interpretação dos seus nomes livres!

© Luis Caires

LP2 2005/06

205

Equivalência-alfa ($=\alpha$)

- ◆ Duas abstracções dizem-se *alfa-equivalentes* se uma pode ser obtida a partir da outra através da **renomeação** dos seus parâmetros, evitando conflitos com os seus nomes livres:
 - $(x \rightarrow x+y) =\alpha (z \rightarrow z+y)$
 - $(x \rightarrow x+y) \neq\alpha (y \rightarrow y+y)$
- ◆ Abstracções alfa-equivalentes são semanticamente equivalentes (são interpretadas como denotando a mesma função):
 - $(x \rightarrow x+1) =\alpha (z \rightarrow z+1)$
- ◆ Por isso, um interpretador pode traduzir os nomes dos parâmetros em algo mais conveniente ...

© Luis Caires

LP2 2005/06

206

Que denota uma abstracção?

- ◆ Uma abstracção é uma expressão sintáctica que denota uma certa **função**.
- ◆ Uma **função** é uma entidade semântica primitiva que suporta uma operação de aplicação, tal como um valor inteiro é uma entidade semântica primitiva que suporta a operação de adição, etc ...
- ◆ Uma linguagem pode suportar funções mas não abstracções (exemplo: C, C++, Pascal)
- ◆ Várias linguagens suportam abstracções (ML, Smalltalk, Python, Javascript, Java (usando objects anónimos))

© Luis Caires

LP2 2005/06

207

Abstracções vs. Declarações

- ◆ Em C e Pascal, as expressões que representam funções aparecem **sempre** no contexto de uma declaração:
 - `int f(int x) { return x+1; } (resto do programa)`
- ◆ Em ML, o mecanismo de declaração está **separado** do mecanismo de abstracção:
 - `let x=2 in (resto do programa)`
 - `let f = (fun x->x+1) in (resto do programa)`
 - `map (fun x-> x+1) [1;2;3] = [2;3;4]`
- ◆ Em ML, as funções são "*cidadãos de primeira classe*" (first-class citizens)

© Luis Caires

LP2 2005/06

208

Abstracções (exemplos)

- ◆ Smalltalk
 - Um bloco `[:x | E]` é uma abstracção
 - `1 to: 10 do: [:i | s ← s+i]`
- ◆ Abstracções em linguagens de objects
 - Uma abstracção pode ser representada por um objecto que implementa um método "apply"
 - Exemplo em Java:

```
f = new { int apply(int x) { return x+1; } } // f = λx. x+1
v = f.apply(2)                          // v = (f 2)
```

© Luis Caires

LP2 2005/06

209

Abstracções (mais exemplos)

- ◆ As abstracções podem realizar-se não apenas sobre identificadores de valores, mas também sobre outras entidades, como por exemplo **tipos**.
- ◆ C++
 - template <class T> class Stack { int push(const &T); ... }
- ◆ Java (5.0)
 - interface List<E> { void add(E x); Iterator<E> iterator(); }
 - interface Iterator<E> { E next(); boolean hasNext(); }
 - class LinkedList<E> implements List<E> { ... }

© Luis Caires

LP2 2005/06

210

Exemplo: A Linguagem CALCF

- ◆ A linguagem CALCF estende a linguagem CALCI com funções, representadas por **abstracções**:

```
(fun Id -> Expression)
```

- ◆ As funções podem ser aplicadas ao seu argumento, usando a expressão de **aplicação**:

```
Expression1(Expression2)
```

- ◆ Semântica pretendida: Se **Expression1** denota uma função f , e **Expression2** denota um valor qualquer v , então **Expression1(Expression2)** denota o valor que resulta de aplicar f a v .

© Luis Caires

LP2 2005/06

211

Exemplo: A Linguagem CALCF

- ◆ Um programa simples:

```
(fun x -> x+2)(4)
```

- ◆ Um outro exemplo:

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

- ◆ Uma sintaxe concreta tipo C traduzida em CALCF:

```
function f(x)= x+1
function g(y)= f(y)+2
  let x = g(2)
    in x+x
```

© Luis Caires

LP2 2005/06

212

A Linguagem CALCF

- ◆ Tipo de dados: CALCF

- ◆ Construtores:

```
num: integer -> CALCF
id: string -> CALCF
add: CALCF x CALCF -> CALCF
mul: CALCF x CALCF -> CALCF
div: CALCF x CALCF -> CALCF
sub: CALCF x CALCF -> CALCF
decl: string x CALCF x CALCF -> CALCF
fun: string x CALCF -> CALCF
call: CALCF x CALCF -> CALCF
```

© Luis Caires

LP2 2005/06

213

Semântica de CALCF (1)

- ◆ A função semântica I de CALCF:

```
I : CALCF -> RESULT
```

CALCF = conjunto dos programas abertos

RESULT = conjunto dos significados (denotações)

- ◆ significado: pode ser um valor inteiro ou uma função (representada por uma abstracção):

RESULT = integer + abstraction + error

© Luis Caires

LP2 2005/06

214

Resultados de CALCF

- ◆ Apresentação dos resultados da linguagem CALCF como um tipo indutivo

Tipo de dados: RESULT

- ◆ Construtores:

```
num: integer -> RESULT
abstraction: string x CALCF -> RESULT
error: RESULT
```

© Luis Caires

LP2 2005/06

215

Interpretador de CALCF (1)

- ◆ Algoritmo "de referência" $eval(E)$ para calcular o valor de uma **expressão fechada** E de CALCF:

```
eval : CALCF -> RESULT
```

```
eval( num(n) )  $\triangleq$  return n
eval( add(E1,E2) )  $\triangleq$ 
  return eval(E1) + eval(E2)
...
eval( decl(s, E1, E2) )  $\triangleq$  [
  G = Subst(E2, s, eval(E1));
  return eval(G); ]
```

© Luis Caires

LP2 2005/06

216

Interpretador de CALCF (1)

- ◆ Algoritmo “de referência” eval(E) para calcular o valor de uma expressão fechada E de CALCF:

eval : CALCF → RESULT

```
eval( fun(s, E) ) ≙ return abstraction(s, E)
eval( app(E1, E2) ) ≙ [
  arg = eval(E2);
  fun = eval(E1);
  if Fun = abstraction(param, body) then
    G = Subst(body, param, arg)
    return eval(G)
  else error ]
```

© Luis Caires

LP2 2005/06

217

Definição da Função Subst

```
Subst(s, num(n), F) ≙ return num(n);
Subst(s, id(s), F) ≙ return F;
Subst(s, add(E1, E2)) ≙
  return add( Subst(s, E1, F), Subst(s, E2, F) );
...
Subst(s, decl(s, E1, E2), F) ≙
  [ G = Subst(s, E1, F);
    return decl(s, G, E2); ]
Subst(s, decl(s', E1, E2), F) ≙ [ /* caso s ≠ s' */
  G = Subst(s, E1, F);
  newid = gen_fresh_id();
  E2' = Subst(s', E2, newid);
  return decl(newloc, G, Subst(s, E2', F)); ]
```

© Luis Caires

LP2 2005/06

218

Definição da Função Subst

```
Subst(s, num(n), F) ≙ return num(n);
Subst(s, id(s), F) ≙ return F;
Subst(s, add(E1, E2)) ≙
  return add( Subst(s, E1, F), Subst(s, E2, F) );
...
Subst(s, decl(s, E1, E2), F) ≙
  [ G = Subst(s, E1, F);
    return decl(s, G, E2); ]
Subst(s, decl(s', E1, E2), F) ≙
  G = Subst(s, E1, F);
  newid = gen_fresh_id();
  E2' = Subst(s', E2, newid);
  return decl(newloc, G, Subst(s, E2', F)); ]
```

Renomeação evita captura ilegal de eventuais ocorrências livres do identificador s' em F

© Luis Caires

LP2 2005/06

219

Definição da Função Subst

```
Subst(s, app(E1, E2)) ≙
  return app( Subst(s, E1, F), Subst(s, E2, F) );

Subst(s, abstraction(s, E), F) ≙ [
  return abstraction(s, E); ]

Subst(s, abstraction(s', E), F) ≙ [ /* caso s ≠ s' */
  newargid = gen_fresh_id();
  E' = Subst(s', E, newargid);
  return abstraction(newarg, Subst(s, E', F)); ]
```

© Luis Caires

LP2 2005/06

220

Definição da Função Subst

```
Subst(s, app(E1, E2)) ≙
  return app( Subst(s, E1, F), Subst(s, E2, F) );

Subst(s, abstraction(s, E), F) ≙ [
  return abstraction(s, E); ]

Subst(s, abstraction(s', E), F) ≙ [ /* caso s ≠ s' */
  newargid = gen_fresh_id();
  E' = Subst(s', E, newargid);
  return abstraction(newarg, Subst(s, E', F)); ]
```

Renomeação evita captura ilegal de eventuais ocorrências livres do identificador s' em F

© Luis Caires

LP2 2005/06

221

Interpretador de CALCF (1)

- ◆ Exemplo: avaliar o seguinte programa, usando a semântica simples de substituição.

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

© Luis Caires

LP2 2005/06

222

Semântica de CALCF (2)

- ◆ A função semântica I de CALCF:

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

CALCF = conjunto dos programas abertos
ENV = conjunto dos ambientes válidos
RESULT = conjunto dos significados (denotações)

- ◆ significado: pode ser um valor inteiro ou uma função (representada por uma abstracção):

RESULT = integer + abstraction + error

- ◆ N.B. Ambiente associa resultados a identificadores

© Luis Caires

LP2 2005/06

223

Interpretador de CALCF (2)

- ◆ Algoritmo $\text{eval}(E, \text{env})$ para calcular a denotação (valor inteiro) de uma expressão E de CALCF:

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

```
eval( id(ident), env ) : return env.Find(ident)
eval( fun(ident, B), env ) : return new abstraction(ident, B)
eval( call(E1, E2), env ) :
  if eval(E1, env) is abstraction(ident, B) then [
    env.BeginScope();
    env.Assoc(ident, eval(E2, env));
    val = eval(B, env);
    env.EndScope(); return val; ] else error
```

Nota. O resultado é indefinido se E_1 não denotar uma função!

© Luis Caires

LP2 2005/06

224

Interpretador de CALCF (2)

- ◆ Avaliação do programa:

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

© Luis Caires

LP2 2005/06

225

Interpretador de CALCF (2)

- ◆ Avaliação do programa:

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

- ◆ $\text{eval}(P1, 0) =$

© Luis Caires

LP2 2005/06

226

Interpretador de CALCF (2)

- ◆ Avaliação do programa:

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

- ◆ $\text{eval}(P1, 0) =$
- ◆ $\text{eval}(P2, [f \leftarrow \text{abs}(x, x+1)]) =$

© Luis Caires

LP2 2005/06

227

Interpretador de CALCF (2)

- ◆ Avaliação do programa:

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

- ◆ $\text{eval}(P1, 0) =$
- ◆ $\text{eval}(P2, [f \leftarrow \text{abs}(x, x+1)]) =$
- ◆ $\text{eval}(P3, [g \leftarrow \text{abs}(y, f(y)+2), f \leftarrow \text{abs}(x, x+1)]) =$
 $\text{eval}(x+x, [x \leftarrow ?, g \leftarrow \text{abs}(y, f(y)+2), f \leftarrow \text{abs}(x, x+1)])$
- ◆ $\text{eval}(g, [g \leftarrow \text{abs}(y, f(y)+2), f \leftarrow \text{abs}(x, x+1)]) = \text{abs}(y, f(y)+2)$
- ◆ $\text{eval}(g(2), E) =$
 $\text{eval}(f(y)+2, [y \leftarrow 2, g \leftarrow \text{abs}(y, f(y)+2), f \leftarrow \text{abs}(x, x+1)]) =$

© Luis Caires

LP2 2005/06

228

Interpretador de CALCF (2)

◆ Avaliação do programa:

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

- ◆ $\text{eval}(g(2), E) =$
 $\text{eval}(f(y)+2, [y \leftarrow 2, g \leftarrow \text{abs}(y, f(y)+2), f \leftarrow \text{abs}(x, x+1)]) =$
 $\text{eval}(x+1, [x \leftarrow 2, y \leftarrow 2, g \leftarrow \text{abs}(y, f(y)+2), f \leftarrow \text{abs}(x, x+1)]) + 2 =$
 $3+2 = 5$

© Luis Caires

LP2 2005/06

229

Interpretador de CALCF (2)

◆ Avaliação do programa:

```
decl f=(fun x ←x+1) in
  decl g = (fun y ← f(y)+2) in
    decl x = g(2)
      in x+x
```

- ◆ $\text{eval}(g(2), E) =$
 $\text{eval}(f(y)+2, [y \leftarrow 2, g \leftarrow \text{abs}(y, f(y)+2), f \leftarrow \text{abs}(x, x+1)]) =$
 $\text{eval}(x+1, [x \leftarrow 2, y \leftarrow 2, g \leftarrow \text{abs}(y, f(y)+2), f \leftarrow \text{abs}(x, x+1)]) + 2 =$
 $3+2 = 5$
- ◆ $\text{eval}(x+x, [x \leftarrow 5, g \leftarrow \text{abs}(y, f(y)+2), f \leftarrow \text{abs}(x, x+1)]) = 10$

© Luis Caires

LP2 2005/06

230

Interpretador de CALCF (2)

◆ Outro exemplo:

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

- ◆ $E = [g \leftarrow \text{abs}(x, x+f(x)); f \leftarrow \text{abs}(y, y+x); x \leftarrow 1]$
- ◆ $\text{eval}(g(2), E) =$
 $\text{eval}(x+f(x), [x \leftarrow 2; g \leftarrow \text{abs}(y, f(y)+2), f \leftarrow \text{abs}(x, y+x)]) =$
 $\text{eval}(f(x), [x \leftarrow 2; g \leftarrow \text{abs}(y, f(y)+2), f \leftarrow \text{abs}(x, y+x)]) + 2 =$
 $\text{eval}(y+x, [y \leftarrow 2; x \leftarrow 2; g \leftarrow \text{abs}(y, f(y)+2), f \leftarrow \text{abs}(x, y+x)]) +$
 $2 = 2+2+2 = 6$
- ◆ Qual é o valor intuitivo deste programa?

© Luis Caires

LP2 2005/06

231

Interpretador de CALCF (1)

◆ Outro exemplo:

```
decl x=1 in
  decl f = (fun y ←y+x) in
    decl g = (fun x ←x+f(x))
      in g(2)
```

- ◆ $E = [g \leftarrow \text{abs}(x, x+f(x)); f \leftarrow \text{abs}(y, y+x); x \leftarrow 1]$
- ◆ $\text{eval}(g(2), E) =$
 $\text{eval}(x+f(x), [x \leftarrow 2; g \leftarrow \text{abs}(y, f(y)+2), f \leftarrow \text{abs}(x, x+y)]) =$
 $\text{eval}(f(x), [x \leftarrow 2; g \leftarrow \text{abs}(y, f(y)+2), f \leftarrow \text{abs}(x, x+y)]) + 2 =$
 $\text{eval}(y+x, [y \leftarrow 2; x \leftarrow 2; g \leftarrow \text{abs}(y, f(y)+2), f \leftarrow \text{abs}(x, x+y)]) + 2 =$
 $2+2+2 = 6$
- ◆ O valor do programa deveria ser 5 !

© Luis Caires

LP2 2005/06

232

Resolução dinâmica de nomes

- ◆ A semântica simplificada (2) que definimos para a linguagem CALCF adota a "regra dinâmica" de resolução de nomes (*dynamic scoping*).
- ◆ Segundo esta regra, os valores dos nomes **não locais** são interpretados no contexto da chamada da função, em vez do contexto da definição.
- ◆ Historicamente, algumas linguagens de programação (ex: Lisp, JavaScript 1.0) adoptaram a resolução dinâmica de nomes, por ser de implementação mais simples.
- ◆ Actualmente, é considerado um mecanismo indesejável e até incorrecto (não respeita o princípio da substituição), apesar de às vezes "dar jeito" interpretar nomes não locais no contexto da chamada (por exemplo: "hostname").

© Luis Caires

LP2 2005/06

233

Princípio da substitutividade

- ◆ O valor de qualquer expressão permanece inalterado sempre que nela se substitui uma subexpressão por outra expressão com o mesmo significado / valor.
- ◆ A semântica da linguagem CALCF **viola** este princípio, pois os dois programa seguintes têm valores diferentes:

```
decl x=1 in
  decl f = (fun y ←y+x) in
    decl g = (fun x ←x+f(x))
      in g(2)
```

```
decl x=1 in
  decl f = (fun y ←y+1) in
    decl g = (fun x ←x+f(x))
      in g(2)
```

© Luis Caires

LP2 2005/06

234

Resolução estática de nomes

- ◆ Todas as linguagens de programação modernas adoptam a regra da resolução estática de identificadores (*static scoping*).
- ◆ Segundo esta regra, os valores dos identificadores livres que ocorram no corpo de abstrações são interpretados no contexto em que as abstrações ocorrem (na definição das funções), e não no contexto da chamada.
- ◆ Para implementar esta semântica de forma eficiente é necessário usar um domínio de resultados mais rico, em que as funções são representadas por entidades chamadas “fechos”.

© Luis Caires

LP2 2005/06

235

Semântica de CALCF (3)

- ◆ A (nova) função semântica I de CALCF:

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

CALCF = conjunto dos programas **abertos**

ENV = conjunto dos ambientes

RESULT = conjunto dos significados (denotações)

- ◆ Resultados: podem ser um valor inteiro, um fecho (uma abstracção + um ambiente), ou um erro:

RESULT = **integer** + **closure** + **error**

© Luis Caires

LP2 2005/06

236

Resultados de CALCF (3)

- ◆ Tipo de dados: **RESULT**

- ◆ Construtores:

```
num:      integer → RESULT
closure:  string × CALCF × ENV → RESULT
error:    RESULT
```

- ◆ Um **fecho** representa uma função através do seu parâmetro, do seu corpo, e do **ambiente** que regista os valores dos nomes livres no seu corpo
- ◆ Assim, ao contrário de uma abstracção, um fecho é efectivamente “fechado” (não depende de nenhum nome externo)

© Luis Caires

LP2 2005/06

237

Ambiente “mutável”

- ◆ Na prática, é conveniente implementar ambientes usando uma estrutura de dados mutável:

```
Environ BeginScope()
```

- Cria um **novo nível vazio**, onde serão colocadas as ligações de um novo âmbito local.
- Não pode existir mais que uma ligação para um mesmo identificador no mesmo nível.

```
Environ EndScope()
```

- Devolve o ambiente no estado anterior à última operação `BeginScope()`.
- Mas **não destrói o último nível** pois podem existir no contexto de execução fechos que o referem!

© Luis Caires

LP2 2005/06

238

Interpretador de CALCF (3)

- ◆ Algoritmo `eval(E, env)` para calcular o valor de uma expressão E de CALCF:

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

```
eval( id(ident), env ) : return env.Find(ident)
eval( fun(ident, B), env ) : return new closure(ident, env, B)
eval( call(E1, E2), env ) :
  if eval(E1, env) is closure(ident, envc, B) then [
    envloc = envc.BeginScope();
    envloc.Assoc(ident, eval(E2, env));
    return eval(B, envloc); ]
  else return error
```

© Luis Caires

LP2 2005/06

239

Avaliação de Funções

- ◆ Exemplo: avaliar o seguinte programa, na semântica usando ambientes e fechos.

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

© Luis Caires

LP2 2005/06

240

Avaliação de Funções

$E_0 \rightarrow x \rightarrow 1$

```

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

© Luis Caires 241

Avaliação de Funções

$E_0 \rightarrow x \rightarrow 1$
 $E_1 \rightarrow f \rightarrow (y \quad x \quad y+x)$

```

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

© Luis Caires 242

Avaliação de Funções

$E_0 \rightarrow x \rightarrow 1$
 $E_1 \rightarrow f \rightarrow (y \quad x \quad y+x)$
 $E_2 \rightarrow g \rightarrow (x \quad x+f(x))$

```

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

© Luis Caires 243

Avaliação de Funções

$E_0 \rightarrow x \rightarrow 1$
 $E_1 \rightarrow f \rightarrow (y \quad x \quad y+x)$
 $E_2 \rightarrow g \rightarrow (x \quad x+f(x))$
 $E_3 \rightarrow x \rightarrow 2$
 $E_4 \rightarrow y \rightarrow 2, x \rightarrow 2$

```

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

© Luis Caires 244

Avaliação de Funções

$E_0 \rightarrow x \rightarrow 1$
 $E_1 \rightarrow f \rightarrow (y \quad x \quad y+x)$
 $E_2 \rightarrow g \rightarrow (x \quad x+f(x))$
 $E_3 \rightarrow x \rightarrow 2$
 $E_4 \rightarrow y \rightarrow 2, x \rightarrow 2$
 $E_5 \rightarrow y \rightarrow 2, x \rightarrow 2$

```

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

© Luis Caires 245

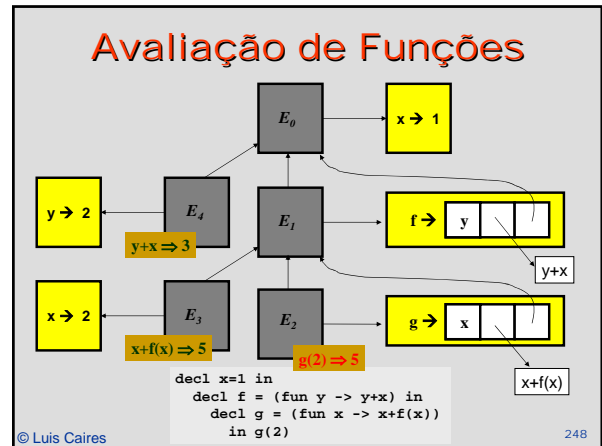
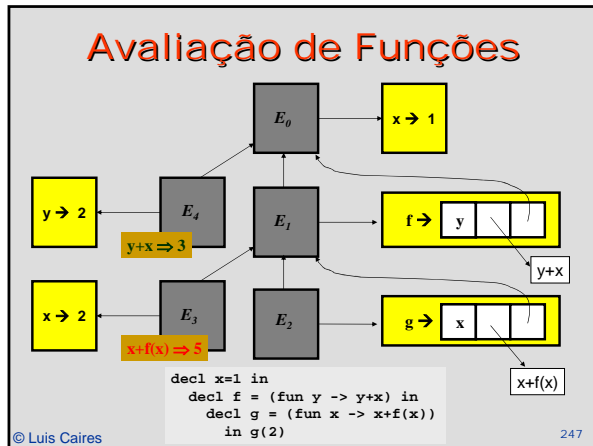
Avaliação de Funções

$E_0 \rightarrow x \rightarrow 1$
 $E_1 \rightarrow f \rightarrow (y \quad x \quad y+x)$
 $E_2 \rightarrow g \rightarrow (x \quad x+f(x))$
 $E_3 \rightarrow x \rightarrow 2$
 $E_4 \rightarrow y \rightarrow 2, x \rightarrow 2$
 $E_5 \rightarrow y \rightarrow 2, x \rightarrow 2$
 $E_6 \rightarrow y \rightarrow 2, x \rightarrow 2$

```

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

© Luis Caires 246



Avaliação de Funções

◆ Exemplo: avaliar o seguinte programa, na semântica usando ambientes e fechos.

```

decl c = (fun f,g -> (fun x -> f(g(x))))
in
  decl i = (fun x -> x+1) in
    decl d = c(i,i)
      in d(2)
  
```

© Luis Caires LP2 2005/06 249

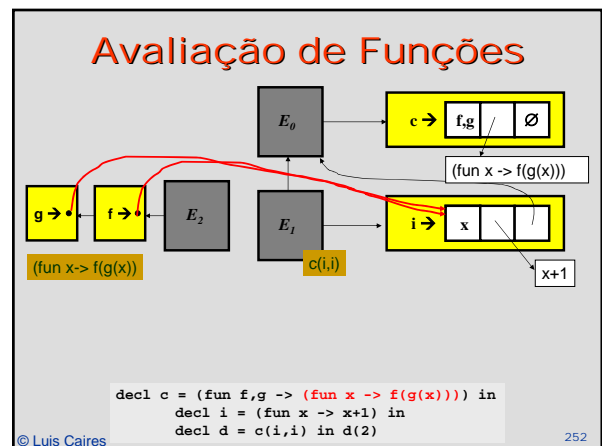
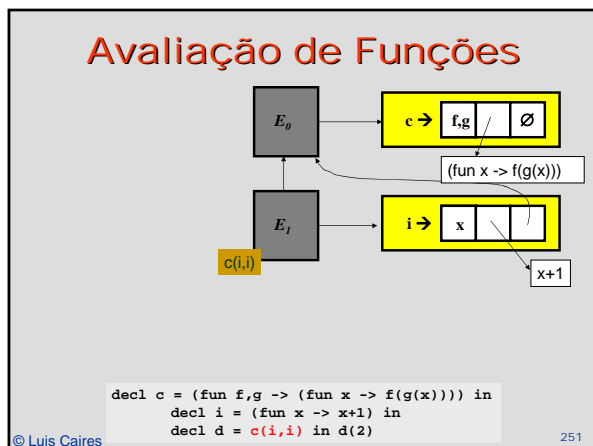
Avaliação de Funções

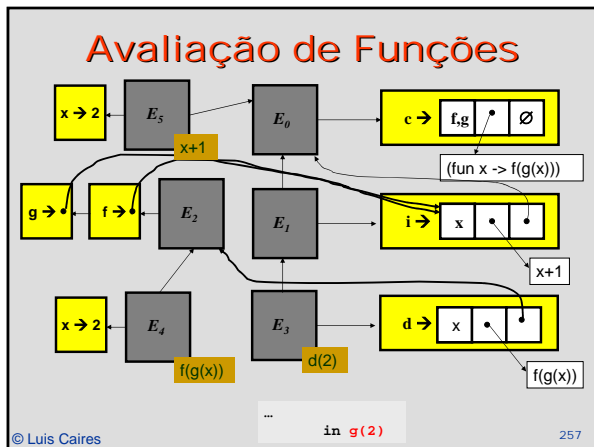
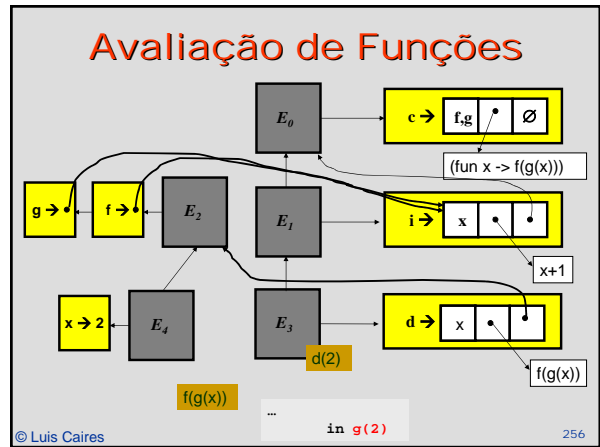
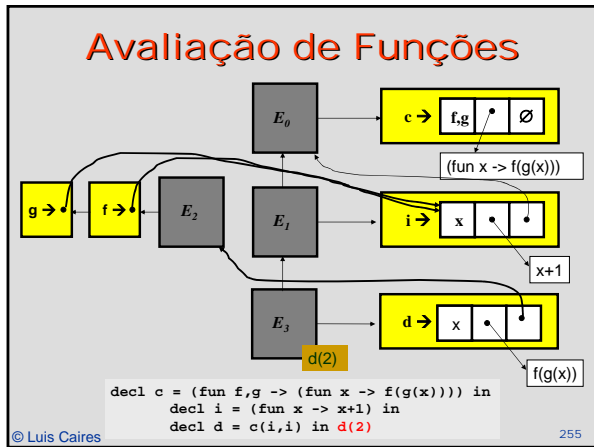
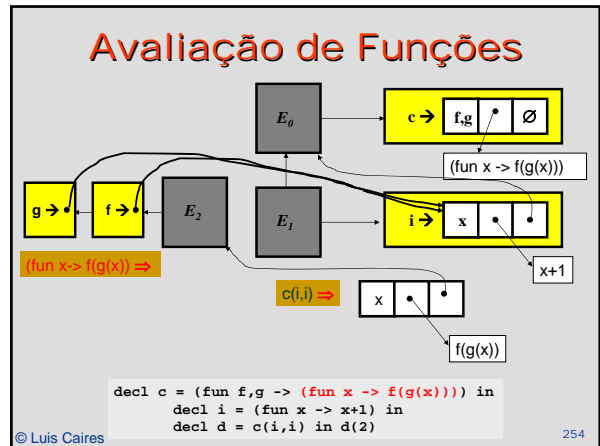
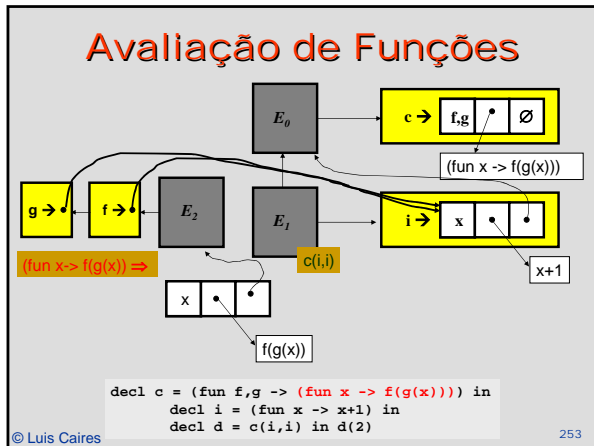
◆ Exemplo: avaliar o seguinte programa, na semântica usando ambientes e fechos.

```

decl comp = (fun f,g -> (fun x -> f(g(x))))
in
  decl inc = (fun x -> x+1) in
    decl d = comp(inc,inc)
      in d(2)
  
```

© Luis Caires LP2 2005/06 250





Mini Teste

- ◆ 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:
 - `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`
 - `call((fun id -> E2), E1)`

© Luis Caires LP2 2005/06 258

Mini Teste (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)))
```

© Luis Caires

LP2 2005/06

259

Abstracção Funcional (2)

© Luis Caires

LP2 2005/06

260

Definições Recursivas

- ◆ Na construção de declaração local

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

o nome introduzido *id* não é identificado com 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"!

© Luis Caires

LP2 2005/06

261

Definições Recursivas

- ◆ Uma definição recursiva é uma declaração onde o nome declarado pode "aparecer" dentro do corpo da definição:

```
declrec Sum =  
  (fun x -> if x=0 then 1  
            else x * Sum(x-1))  
in Sum(10)
```

- ◆ Mais rigorosamente: numa declaração recursiva o nome declarado também liga as suas ocorrências livres na expressão de inicialização.
- ◆ Como introduzir (e definir a semântica operacional) de definições recursivas (de funções) ?

© Luis Caires

LP2 2005/06

262

A Linguagem RECF

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

```
num: integer → RECF  
var: string → RECF  
add: RECF × RECF → RECF  
...  
if: RECF × RECF × RECF → RECF  
declr: string × RECF × RECF → RECF  
fun: string × RECF → RECF  
call: RECF × RECF → RECF
```

- ◆ Note: a construção decl não existe em RECF (pode ser codificada da forma apresentada atrás)

© Luis Caires

LP2 2005/06

263

A Linguagem RECF

- ◆ Exemplo de programa em RECF:

```
declr fact = (fun n ←  
  if n then n*fact(n-1) else 1  
  ) in fact(2)
```

- ◆ Abreviaturas:

```
- decl id = E1 in E2 ≙ call((fun id ← E2), E1)  
- E1(E2) ≙ call(E1, E2)  
- if E1 then E1 else E3 ≙ if(E1, E2, E3)
```

© Luis Caires

LP2 2005/06

264

A Linguagem RECF

◆ Exemplo de programa em RECF:

```
declr fact = (fun n ←  
  if n then 1  
  else n*fact(n-1)  
) in fact(2)
```

◆ Qual o ambiente no qual a abstracção (sombreada) deve ser avaliada? Note que:

- fact é um **nome livre** na abstracção
- Assim, o nome fact deverá estar definido no ambiente activo no momento em que a abstracção for avaliada
- Por outro lado, o valor a associar a fact nesse mesmo ambiente deverá ser a própria função

© Luis Caires

LP2 2005/06

265

Ambientes Circulares

◆ Exemplo de programa em RECF:

```
declr fact = (fun n ←  
  if n then 1  
  else n*fact(n-1)  
) in fact(2)
```

◆ As definições recursivas introduzem uma "circularidade" na construção do ambiente:

- O ambiente E resultante da declaração de fact deverá ter uma **ligação** fact ← clos(n, ..., 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

© Luis Caires

LP2 2005/06

266

Ambiente (revisitado)

◆ Adicionamos aos ambientes uma nova primitiva:

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

- Devolve o mesmo ambiente, mas **substitui (por alteração imperativa) a ligação existente para o identificador id pelo valor val**
- Esta afectação mutável é importante para obter a circularidade do ambiente pretendida:
- Se o valor val contiver uma referência para o ambiente env, este passará a conter uma ligação mencionando uma referência para si próprio.

© Luis Caires

LP2 2005/06

267

Ambiente Circular

◆ Exemplo:

```
env = new Environment();
```

```
env
```

© Luis Caires

LP2 2005/06

268

Ambiente Circular

◆ Exemplo:

```
env = new Environment();  
env.Assoc("x", 2);
```

```
env → x ← 2
```

© Luis Caires

LP2 2005/06

269

Ambiente Circular

◆ Exemplo:

```
env = new Environment();  
env.Assoc("x", 2);  
env.Assoc("fact", null);
```

```
env → fact ← null → x ← 2
```

© Luis Caires

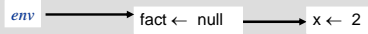
LP2 2005/06

270

Ambiente Circular

◆ Exemplo:

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



© Luis Caires

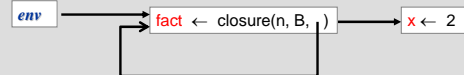
LP2 2005/06

271

Ambiente Circular

◆ Exemplo:

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



Durante qualquer avaliação do corpo B, o ambiente vai conter sempre a definição correcta de fact (e de x)

© Luis Caires

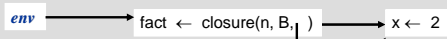
LP2 2005/06

272

Ambiente Circular

◆ Contraexemplo com declaração não recursiva

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



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

© Luis Caires

LP2 2005/06

273

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

◆ resultados: pode ser um valor inteiro, um fecho, ou um erro (uma abstracção + um ambiente):

RESULT = integer + closure + error

◆ Ambiente associa resultados a identificadores

© Luis Caires

LP2 2005/06

274

Semântica de RECF

◆ Algoritmo eval(E, env) para calcular a denotação de qualquer expressão E de RECF:

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

eval(if(E₁, E₂, E₃) , env) :

if eval(E₁, env) = 0 then return eval(E₂, env)

else return eval(E₃, env)

eval(declr(ident, E₁, E₂) , env) :

envloc = env .Assoc(ident, NULL);

valinit = eval(E₁, envloc)

return eval(E₂, envloc.Update(ident, valinit))

© Luis Caires

LP2 2005/06

275

TPC (Auto-Teste)

◆ Avalie o programa RECF usando a semântica

```
P ≜ letrec fact = (fun n ←
    if n then 1
    else n*fact(n-1)
) in fact(3)
```

eval(P, ∅) = E_1
 ↓
 eval(fact(3), [fact ← clos(n, (if ...) , E₁)]) =
 ... ?

© Luis Caires

LP2 2005/06

276

Passagem de argumentos

- ◆ Recorde a semântica da chamada de função adoptada nas linguagens CALCF e RECF:

```
eval( call(E1, E2), env ) :
  if eval(E1, env) = closure(ident, envc, B) then [
    envloc = envc.BeginScope();
    envloc.Assoc(ident, eval(E2, env));
    return eval(B, envloc); ] else return error
```

- ◆ A expressão E₂ que denota o argumento da função é avaliada **antes** da função denotada pela expressão E₁ ser efectivamente aplicada.
 - Qual o valor intuitivo/efectivo do programa seguinte?


```
declr f = (fun x ← f(x)) in declr g = (fun y ← 1) in g(f(1))
```

© Luis Caires

LP2 2005/06

277

Passagem de valor

- ◆ Qual o valor da expressão seguinte ?

```
declr f = (fun x ← f(x)) in
  declr g = (fun y ← 1) in g(f(2)) end end
```

- ◆ Valor de acordo com a semântica “declarativa”:
 - g é a função constante que devolve sempre o valor 1 independentemente do valor do argumento.
 - Então, o valor g(f(2)) deverá ser 1.
- ◆ Valor determinado pela semântica “operacional”:
 - A semântica apresentada não define valor para esta expressão, pois a avaliação de f(2) não termina!
- ◆ A regra de passagem de argumentos utilizada é a “passagem de valor” (*call-by-value*)

© Luis Caires

LP2 2005/06

278

Passagem de “nome”

- ◆ Qual o valor da expressão (1 ou nenhum)?

```
declr f = (fun x ← f(x)) in
  declr g = (fun y ← 1) in g(f(2)) end end
```

- ◆ Existe um modo de avaliação que permite sempre encontrar o valor “declarativo” das expressões.
- ◆ Consiste em suspender a avaliação dos argumentos das funções até ao momento em que estes se tornam necessários
- ◆ Corresponde em passar a **expressão** como argumento, em vez do seu resultado.
- ◆ A esta regra de avaliação de argumentos chama-se “passagem por nome” (*call-by-name*) (CBN)

© Luis Caires

LP2 2005/06

279

Resultados de RECF(CBN)

- ◆ Tipo de dados: RESULT

- ◆ Construtores:

```
num:      integer → RESULT
closure:  string × RECF × ENV → RESULT
susp:     RECF × ENV → RESULT
```

- ◆ Uma **suspensão** representa uma expressão **por avaliar**, juntamente com o ambiente respectivo.
- ◆ É necessário guardar na suspensão, junto com cada expressão, o ambiente correcto, pois em geral a expressão contém identificadores livres.

© Luis Caires

LP2 2005/06

280

Semântica de RECF (CBN)

- ◆ Interpretador para RECF (CBN)

eval : RECF × ENV → RESULT

```
eval( var(ident), env ):
  val = env.Find(ident)
  if val = susp(B, E) then return eval(B, E)
  else return val
```

```
eval( fun(ident, B), env ): return closure(ident, B, env)
```

```
eval( call(E1, E2), env ):
  if eval(E1, env) = closure(ident, B, ED) então
  then eval(B, ED.Assoc(ident, susp(E2, env)))
```

© Luis Caires

LP2 2005/06

281

Avaliação RECF (CBN)

- ◆ Qual o valor (CBN) da expressão?

```
P ≜ declr f = (fun x ← f(x)) in
  declr g = (fun y ← 1) in g(f(2)) end end
```

```
eval(P, ∅) =
eval(declr g = (fun y ← 1) in g(f(2)), [f ← clos(x, f(x), A1)] ) =
eval(g(f(2)), [g ← clos(y, 1, A1),
               f ← clos(x, f(x), A1)] ) =
eval(1, [y ← susp(f(2), A2),
         g ← clos(y, 1, A1),
         f ← clos(x, f(x), A1)] ) = 1
```

© Luis Caires

LP2 2005/06

282

Avaliação RECF (CBN)

Qual o valor (CBN) da expressão?

$P \triangleq \text{declr } y = 2 \text{ in declr } f = (\text{fun } z \leftarrow z+y) \text{ in } f(2+y)$

```
eval(P, ∅) =
eval( let f = (fun z ← z+y) in f(2+y), [y←2] ) =
eval( f(2+y), [f ← clos(z, z+y, A1), y ← 2] ) =
eval( z+y, [z ← susp(2+y, A2), f ← clos(z, z+y, A1), y ← 2] ) =
eval(z, [z ← susp(2+y, A2), f ← clos(z, z+y, A1), y ← 2] ) + 2 =
eval(2+y, A2) + 2 = 6
```

Note que a expressão argumento $2+y$ é avaliada apenas quando o valor do parâmetro z se torna necessário!

© Luis Caires

LP2 2005/06

283

Tipos Funcionais

© Luis Caires

LP2 2005/06

284

O Cálculo Lambda

© Luis Caires

LP2 2005/06

285

A Linguagem CALCF

◆ Tipo de dados: CALCF

◆ Construtores:

```
num: integer → CALCF
var: string → CALCF
add: CALCF × CALCF → CALCF
mul: CALCF × CALCF → CALCF
div: CALCF × CALCF → CALCF
sub: CALCF × CALCF → CALCF
decl: string × CALCF × CALCF → CALCF
fun: string × CALCF → CALCF
call: CALCF × CALCF → CALCF
```

© Luis Caires

LP2 2005/06

286

A Linguagem CALCF

◆ Tipo de dados: CALCF

◆ Construtores: Fragmento funcional da linguagem

```
num: integer → CALCF
var: string → CALCF
add: CALCF × CALCF → CALCF
mul: CALCF × CALCF → CALCF
div: CALCF × CALCF → CALCF
sub: CALCF × CALCF → CALCF
decl: string × CALCF × CALCF → CALCF
fun: string × CALCF → CALCF
call: CALCF × CALCF → CALCF
```

© Luis Caires

LP2 2005/06

287

O Cálculo Lambda (λ -Calculus)

◆ A linguagem funcional **minimal**, permitindo a definição de nomes locais e funções como entidades de "primeira classe".

◆ Inventada por Alonzo Church em 1936, é a base dos mecanismos de abstracção usados em **todas** as linguagens de programação

◆ Sintaxe do cálculo lambda:

```
var: string → LAMBDA x
abs: string × LAMBDA → LAMBDA ( $\lambda x.E$ )
app: LAMBDA × LAMBDA → LAMBDA ( $E_1 E_2$ )
```

◆ As expressões da linguagem constroem-se só com **variáveis** (var), **aplicação** (app), e **abstracção** (app)

© Luis Caires

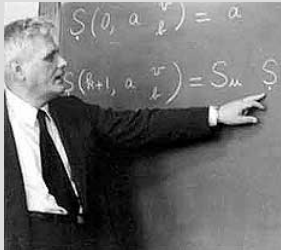
LP2 2005/06

288

Alonzo Church (1903-1995)

Tese de Church:

"As funções intuitivamente computáveis são exactamente as funções programáveis no cálculo lambda".



Church foi o orientador de tese de doutoramento de Alan Turing. Juntos, demonstraram que o cálculo lambda tem o mesmo poder computacional que a máquina de Turing.

© Luis Caires

LP2 2005/06

289

O Cálculo Lambda

- ◆ O cálculo lambda é computacionalmente **completo**: **qualquer algoritmo** pode ser programado no cálculo lambda (usando codificações mais ou menos complicadas)
- ◆ A ideia geral consiste em **traduzir** os mecanismos existentes nas linguagens de programação (tipos de dados, construções de controle, definições recursivas) em expressões do cálculo lambda.
- ◆ Um exemplo já conhecido (definições locais):

```
decl(id, E1, E2)      ≜ app( abs(id, E2), E1 )
decl x = E1 in E2 end ≜ (λx.E2) E1
```

© Luis Caires

LP2 2005/06

290

Semântica do Cálculo λ

- ◆ Algoritmo eval(E, env) para calcular o valor de uma expressão E de LAMBDA:
- ```
eval : LAMBDA → LAMBDA
```
- ◆ Resultados: subconjunto de LAMBDA (formas "normais").
  - ◆ Segue a regra de avaliação "Redução Topo-Esquerda"

```
eval(var(ident)) : return var(ident)
eval(abs(ident, B)) : return abs(ident, B)
eval(app(E1, E2)) :
 v = eval(E1);
 if (v = abs(ident, B) then
 return eval(Subst(ident, B, E2))
 else
 return app(v, E2))
```

© Luis Caires

LP2 2005/06

291

## Regra Beta

- ◆ A semântica operacional do cálculo lambda pode ser definida usando **uma única** regra de avaliação, chamada **redução beta**.

$$((\lambda x.E_1) E_2) = \text{subst}(x, E_1, E_2)$$

- ◆ Exemplo:

$$(\lambda x. x + x) 2 = \text{subst}(x, x + x, 2) = 2 + 2 = 4$$

© Luis Caires

LP2 2005/06

292

## Definição da Função Subst

```
Subst(s, var(s), E) ≜ return E;

Subst(s, var(x), E) ≜ return var(x) /* caso x ≠ s */

Subst(s, abs(s, M), E) ≜
 return abs(s, M);

Subst(s, abs(x, M), E) ≜ /* caso x ≠ s */
 return abs(f, Subst(s, Subst(x, B, f), M));

Subst(s, app(M, N), E) ≜
 return app(Subst(s, M, E), Subst(s, N, E));
```

© Luis Caires

LP2 2005/06

293

## Definição da Função Subst

```
Subst(s, var(s), E) ≜ return E;

Subst(s, var(x), E) ≜ Renomeação do parâmetro x com um
 identificador fresco f, para evitar ligação
 ilegal de ocorrências livres de x em E.
 return abs(s, B);

Subst(s, abs(x, B), E) ≜ /* caso x ≠ s */
 return abs(f, Subst(s, Subst(x, B, f), E));

Subst(s, app(M, N), E) ≜
 return app(Subst(s, M, E), Subst(s, N, E));
```

© Luis Caires

LP2 2005/06

294

## Exemplos

- ◆  $\text{eval} ( (\lambda x.x) y ) = \text{eval} ( \text{subst}(x, x, y) ) = y$
- ◆  $\text{eval}( (\lambda x.\lambda z.(xz)) f 2 ) = \text{eval}( (\lambda z.(fz))2 ) = (f 2)$
- ◆  $\text{eval}( (\lambda x.\lambda z.(x z)) (\lambda y.(z y)) 2 ) =$   
 $\text{eval}( (\lambda w.((\lambda y.(z y))w)) 2 ) =$   
 $\text{eval}( (\lambda y.(z y)) 2 ) = (z 2)$

## Potencial de não-terminação

- ◆ Resulta da possibilidade de exprimir a **auto-aplicação**: uma função é aplicada a ela própria.

$$\Omega \triangleq (\lambda x.(x x))$$

$$\text{eval} (\Omega \Omega) = \text{eval} (\lambda x.(x x), \Omega) =$$

$$\text{eval} (\text{subst}(x, (x x), \Omega)) = \text{eval}(\Omega \Omega) = \dots$$

$$\text{eval} (\Omega \Omega) = \text{eval} (\Omega \Omega) = \text{eval} (\Omega \Omega) = \dots$$

- ◆ Permite exprimir construções com potencial de computação Turing (ciclos while, recursividade, ...)

## Representação de Booleanos

- ◆ O tipo de dados **Boolean** consiste em dois valores diferentes **true** e **false**, que podem ser usados para seleccionar entre duas alternativas, usando a função

if C then T else E

que deve satisfazer as igualdades seguintes:

$$\text{eval} ( \text{if } \text{true} \text{ then } M \text{ else } E ) = \text{eval}(M)$$

$$\text{eval} ( \text{if } \text{false} \text{ then } M \text{ else } E ) = \text{eval}(E)$$

- ◆ Podemos definir:

$$\text{True} \triangleq (\lambda x.\lambda y. x)$$

$$\text{False} \triangleq (\lambda x.\lambda y. y)$$

$$\text{If} \triangleq (\lambda x.\lambda y.\lambda z. (x y z))$$

## Representação de Naturais

- ◆ O tipo de dados **Nat** consiste numa constante **zero** e numa função **succ**, que podem ser usados para construir qualquer valor natural.
- ◆ Para programar qualquer função aritmética, precisamos também de um predicado **isZero** tal que

$$\text{eval} (\text{isZero} (\text{zero})) = \text{true}$$

$$\text{eval} (\text{isZero} (\text{succ}(M))) = \text{false}$$

- ◆ Podemos definir (desafio ☺)

$$\text{zero} \triangleq$$

$$\text{succ} \triangleq$$

$$\text{isZero?} \triangleq$$

$$\text{Add} \triangleq$$

## Definições Recursivas

- ◆ A recursividade pode ser expressa internamente por meio do operador **Rec**.
- ◆ **Rec** satisfaz a igualdade semântica seguinte:  
 $\text{eval} (\text{Rec } E) = \text{eval}( E (\text{Rec } E) )$
- ◆ Usando este operador, a declaração recursiva  
 $\text{declr } f = E \text{ in } B \text{ end}$   
 pode ser expressa de forma **não recursiva** por  
 $\text{decl } f = \text{Rec} (\lambda f.E) \text{ in } B \text{ end}$
- ◆ Define-se **Rec** em LAMBDA (Church-Kleene):

$$\text{Rec} \triangleq (\lambda f. ( (\lambda x.\lambda f. f (x x)) (\lambda x.\lambda f. f (x x)) ) )$$

## Semântica do Cálculo $\lambda$ (CBV)

- ◆ Algoritmo  $\text{eval}(E, \text{env})$  para calcular o valor de uma expressão E de LAMBDA:

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

$$\text{eval}(\text{var}(ident), \text{env}) : \text{return } \text{env}.\text{Find}(ident)$$

$$\text{eval}(\text{abs}(ident, B), \text{env}) : \text{return } \text{closure}(ident, \text{env}, B)$$

$$\text{eval}(\text{app}(E_1, E_2), \text{env}) :$$

if  $\text{eval}(E_1, \text{env}) = \text{closure}(ident, \text{envc}, B)$  then [

$\text{envloc} = \text{envc}.\text{BeginScope}()$ ;

$\text{envloc}.\text{Assoc}(ident, \text{eval}(E_2, \text{env}))$  ;

return  $\text{eval}(B, \text{envloc})$  ;

else return error

## Tipos para o Cálculo $\lambda$

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem LAMBDA:

**typchk** : LAMBDA  $\times$  TENV  $\rightarrow$  TYPE

TYPE  $\triangleq$  { int, Fun[TYPE,TYPE], none }

- int** é o tipo dos valores inteiros.
- Fun[ $\mathcal{T}_1$ ,  $\mathcal{T}_2$ ]** é o tipo das funções que, dado um argumento de tipo  $\mathcal{T}_1$ , são garantidas devolver um resultado de tipo  $\mathcal{T}_2$ . Podemos abreviar **Fun[ $\mathcal{T}_1$ ,  $\mathcal{T}_2$ ]** por **( $\mathcal{T}_1 \rightarrow \mathcal{T}_2$ )**. Por exemplo: **(int  $\rightarrow$  int)** é o tipo da função sucessor succ. **((int  $\rightarrow$  int)  $\rightarrow$  int)** é o tipo da função  $(\lambda f. succ(f(2)))$ .
- none** é o "tipo" das expressões indefinidas.

© Luis Caires

LP2 2005/06

301

## Cálculo $\lambda$ Tipificado

- Para obter um algoritmo de tipificação simples, é necessário alterar a sintaxe do cálculo lambda, etiquetando os parâmetros das funções com **tipos**.

```
var: string \rightarrow LAMBDA
abs: string \times TYPE \times LAMBDA \rightarrow LAMBDA
app: LAMBDA \times LAMBDA \rightarrow LAMBDA
```

- Exemplos:

$\lambda x.int. x$

$\lambda y.int. \lambda f:int. f(y)$

© Luis Caires

LP2 2005/06

302

## Tipificação do Cálculo $\lambda$

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem Lambda:

**typchk** : LAMBDA  $\times$  TENV  $\rightarrow$  TYPE

```
typchk(num(n) , tenv) \triangleq return int ;
typchk(id(s) , tenv) \triangleq return tenv.Find(s) ;
```

*podemos ainda introduzir outros literais e tipos básicos:*

```
typchk(true , tenv) \triangleq return bool ;
typchk(false , tenv) \triangleq return bool ;
```

© Luis Caires

LP2 2005/06

303

## Tipificação do Cálculo $\lambda$

- Algoritmo **typchk** para calcular o tipo de uma expressão qualquer da linguagem Lambda:

**typchk** : LAMBDA  $\times$  TENV  $\rightarrow$  TYPE

```
typchk(abs(ident, T, B) , tenv) \triangleq
 envloc = tenv.BeginScope();
 envloc.Assoc(ident, T) ;
 return (T \rightarrow typchk(B, envloc))
```

```
typchk(app(M, N) , tenv) \triangleq
 tm = typchk(M, tenv);
 tn = typchk(N, tenv);
 if tm = (tn \rightarrow A) then return A else return none
```

© Luis Caires

LP2 2005/06

304

## Regras de Tipificação

- Asserções de Tipificação

Env [ Expr : Type (para expressões)

Env [ Stmt ok (para comandos)

- Env (o ambiente)

– Representa-se como uma lista de declarações da forma

$id_0 : Type_0, \dots, id_n : Type_n$

- Expr / Stmt

– É uma expressão da **sintaxe abstracta**

- Type

– É um **tipo** (indica o tipo da expressão Expr)

© Luis Caires

LP2 2005/06

305

## Regras de Tipificação

- Exemplos de tipos

int

string

int  $\rightarrow$  int

- Exemplos de asserções de tipificação válidas:

$\emptyset$  [ 1 : int

s: int [ 1+ s : int

x: int, y : int [ x + y > 1 : bool

x: Ref[int] [ while (!x<0) do x:=!x+1 end ok

y: int [ (fun x: int -> y + x) : int  $\rightarrow$  int

© Luis Caires

LP2 2005/06

306

## Regras de Tipificação

- ◆ Uma regra de tipificação é uma regra da forma

$$\frac{Env_0 [ Expr_0 : Type_0 \dots Env_n [ Expr_n : Type_n ]}{Env [ Expr : Type ]}$$

- ◆ As premissas e a conclusão são asserções de tipificação.
- ◆ Zero ou mais premissas, uma só conclusão.
- ◆ Um axioma é uma regra sem premissas.
- ◆ Leitura declarativa de uma regra: se as premissas são válidas, então a conclusão também é válida.

© Luis Caires

LP2 2005/06

307

## Regras de Tipificação

- ◆ Uma regra de tipificação é uma regra  $d$  (premissas)

$$\frac{Env_0 [ Expr_0 : Type_0 \dots Env_n [ Expr_n : Type_n ]}{Env [ Expr : Type ]}$$

- ◆ As premissas e a conclusão são asser (conclusão) tipificação.
- ◆ Zero ou mais premissas, uma só conclusão.
- ◆ Um axioma é uma regra sem premissas.
- ◆ Leitura declarativa de uma regra: se as premissas são válidas, então a conclusão também é válida.

© Luis Caires

LP2 2005/06

308

## Sistema de Tipos para LAMBDA

- ◆ Regras para o Cálculo Lambda tipificado

$$Env, x:A, Env' [ x : A \quad \text{(axioma)}$$

$$\frac{Env, x:A [ M : B \quad \text{(abstracção)}}{Env [ (\lambda x:A.M) : (A \rightarrow B) ]}$$

$$\frac{Env [ N : A \quad Env [ M : (A \rightarrow B) ]}{Env [ (M N) : B \quad \text{(aplicação)}}$$

© Luis Caires

LP2 2005/06

309

## Sistema de Tipos para LAMBDA

- ◆ Demonstração de  $z:int, f:int \rightarrow int [(\lambda x:int.(f x))z : int$

$$z:int, f:int \rightarrow int [(\lambda x:int.(f x))z : ?$$

© Luis Caires

LP2 2005/06

310

## Sistema de Tipos para LAMBDA

- ◆ Demonstração de  $z:int, f:int \rightarrow int [(\lambda x:int.(f x))z : int$

$$\frac{z:int, f:int \rightarrow int [z:int \quad z:int, f:int \rightarrow int [(\lambda x:A.(f x)):(int \rightarrow ?)]}{z:int, f:int \rightarrow int [(\lambda x:int.(f x))z : int}$$

© Luis Caires

LP2 2005/06

311

## Sistema de Tipos para LAMBDA

- ◆ Demonstração de  $z:int, f:int \rightarrow int [(\lambda x:int.(f x))z : int$

$$\frac{z:int, f:int \rightarrow int, x: int [(f x):?] \quad z:int, f:int \rightarrow int [z:int \quad z:int, f:int \rightarrow int [(\lambda x:int.(f x)):(int \rightarrow ?)]}{z:int, f:int \rightarrow int [(\lambda x:A.(f x))z : int}$$

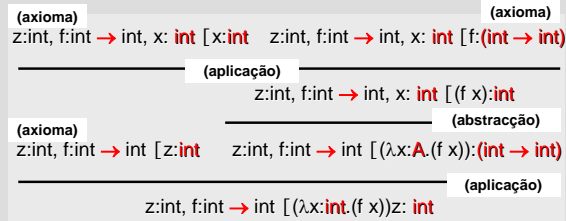
© Luis Caires

LP2 2005/06

312

## Sistema de Tipos para LAMBDA

### ◆ Demonstração de $z:\text{int}, f:\text{int} \rightarrow \text{int} \vdash [(\lambda x:\text{int}.(f\ x))z]: \text{int}$



## Correcção da Tipificação

**Teorema:** Para todo o programa lambda P, se

$$\emptyset \vdash [P : \mathcal{T}$$

é derivável, então  $\text{eval}(P, \emptyset) = v \in \mathcal{T} \setminus \{\text{none}\}$

◆ Ou seja, se P está bem tipificado, então a sua execução termina **sempre**, e **sem erros**.

*Well-typed programs don't go wrong* (Robin Milner)

◆ Como todos os seus programas terminam, o cálculo lambda tipificado **não é Turing-completo**.

◆ Impossível tipificar a auto-aplicação: não é possível em particular representar o combinador Rec.

## Sistemas de Tipos (2)

## A Linguagem CORE

### ◆ Expressões

num: integer  $\rightarrow$  EXP  
 id: string  $\rightarrow$  EXP  
 add: EXP  $\times$  EXP  $\rightarrow$  EXP  
 and: EXP  $\times$  EXP  $\rightarrow$  EXP  
 eq: EXP  $\times$  EXP  $\rightarrow$  EXP  
 var: EXP  $\rightarrow$  EXP

## A Linguagem CORE

### ◆ Expressões

deref: EXP  $\rightarrow$  EXP  
 do: COM  $\times$  EXP  $\rightarrow$  EXP  
 declc: string  $\times$  EXP  $\times$  EXP  $\rightarrow$  EXP  
 fun: string  $\times$  EXP  $\rightarrow$  EXP  
 proc: string  $\times$  COM  $\rightarrow$  EXP  
 callf: EXP  $\times$  EXP  $\rightarrow$  EXP

## A Linguagem CORE

### ◆ Comandos

assign: EXP  $\times$  EXP  $\rightarrow$  COM  
 if: EXP  $\times$  COM  $\times$  COM  $\rightarrow$  COM  
 while: EXP  $\times$  COM  $\rightarrow$  COM  
 declc: string  $\times$  EXP  $\times$  COM  $\rightarrow$  COM  
 callp: EXP  $\times$  EXP  $\rightarrow$  COM

## Um programa em CORE

```

declc
 Acum = (proc r,v => r := !r + v)
 inc = (proc j => Acum (j, 1))
 looper = (fun x,f =>
 declc
 i = var(0)
 s = var(0)
 in do
 while (li < x) do
 Acum(s, f(li)) ;
 inc(i)
 end
 return (ls) end
 end
 google = (fun x => x * x)
 in print (looper(10,google))
end

```

© Luis Caires

LP2 2005/06

319

## Sistema de Tipos para CORE

### ◆ Tipos ( $\mathcal{T}$ )

**int**

**bool**

**Ref( $\mathcal{T}$ )** : Tipo das referências que guardam valores de tipo  $\mathcal{T}$ .

**Fun( $\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_n$ ) $\mathcal{T}$**  : Tipo das funções que recebem  $n$  argumentos, de tipos resp.  $\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_n$ , e devolvem um valor de tipo  $\mathcal{T}$ .

**Proc( $\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_n$ )** : Tipo dos procedimentos que recebem  $n$  argumentos, de tipos respectivamente  $\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_n$ .

### ◆ Exemplos de asserções de tipificação válidas:

$x$ : Ref(int) [ **while** (!x<0) do x:=!x+1 **end ok**

$y$ : int [ (**fun** x: int -> y + x) : **Fun(int)int**

© Luis Caires

LP2 2005/06

320

## A Linguagem CORE

- ◆ A fim de se poder definir um algoritmo de tipificação **simples**, e garantir a propriedade de unicidade do tipo, **extendemos a linguagem CORE etiquetando os parâmetros das funções e procedimentos com o seu tipo pretendido.**

- ◆ **Etiquetam-se também as declarações de modo a facilitar a tipificação das chamadas recursivas.**

**declc:** string  $\times$  TYPE  $\times$  EXP  $\times$  COM  $\rightarrow$  EXP

**declc:** string  $\times$  TYPE  $\times$  EXP  $\times$  COM  $\rightarrow$  COM

**fun:** string  $\times$  TYPE  $\times$  EXP  $\rightarrow$  EXP

**proc:** string  $\times$  TYPE  $\times$  COM  $\rightarrow$  EXP

© Luis Caires

LP2 2005/06

321

## Sistema de Tipos para Core

### ◆ Tipificação de expressões (Cálculo lambda)

$Env, x: \mathcal{T}, Env' [ \text{id}(x) : \mathcal{T}$  (axioma)

$Env, x: \mathcal{T} [ E : \mathcal{V}$  (fun)  
 $\frac{Env, x: \mathcal{T} [ E : \mathcal{V} }{Env [ (\text{fun } x: \mathcal{T}E) : \text{Fun}(\mathcal{T})\mathcal{V} }$

$Env [ N : \mathcal{T} \quad Env [ M : \text{Fun}(\mathcal{T})\mathcal{V}$   
 $\frac{Env [ N : \mathcal{T} \quad Env [ M : \text{Fun}(\mathcal{T})\mathcal{V} }{Env [ \text{callf}(M, N) : \mathcal{V} }$  (callf)

© Luis Caires

LP2 2005/06

322

## Sistema de Tipos para Core

### ◆ Tipificação de expressões (Declarações)

$Env, x: \mathcal{T} [ M : \mathcal{T} \quad Env, x: \mathcal{T} [ N : \mathcal{V}$   
 $\frac{Env, x: \mathcal{T} [ M : \mathcal{T} \quad Env, x: \mathcal{T} [ N : \mathcal{V} }{Env [ \text{decl } x: \mathcal{T} = M \text{ in } N \text{ end} : \mathcal{V} }$  (decl)

- ◆ Note que para permitir declarações recursivas, necessitamos de declarar o tipo do nome introduzido.
- ◆ Se  $M$  não denotar um valor funcional, poderíamos omitir a declaração do seu tipo (porquê?).

© Luis Caires

LP2 2005/06

323

## Sistema de Tipos para Core

### ◆ Tipificação de expressões (inteiros)

$Env [ \text{num}(x) : \text{int}$  (num)

$Env [ N : \text{int} \quad Env [ M : \text{int}$  (add)  
 $\frac{Env [ N : \text{int} \quad Env [ M : \text{int} }{Env [ \text{add}(M, N) : \text{int} }$

$Env [ N : \text{int} \quad Env [ M : \text{int}$   
 $\frac{Env [ N : \text{int} \quad Env [ M : \text{int} }{Env [ \text{eq}(M, N) : \text{bool} }$  (eq)

© Luis Caires

LP2 2005/06

324

## Sistema de Tipos para Core

### ◆ Tipificação de expressões (booleanos)

(true) (false)

$$\frac{}{Env [ \text{true} : \text{bool} } \quad Env [ \text{false} : \text{bool}$$

(and)

$$\frac{Env [ N : \text{bool} }{Env [ \text{and}(M, N) : \text{bool}$$

(eq)

$$\frac{Env [ N : \text{bool} }{Env [ \text{eq}(M, N) : \text{bool}$$

## Sistema de Tipos para Core

### ◆ Tipificação de expressões (referências)

(var)

$$\frac{Env [ E : \mathcal{T} }{Env [ \text{var}(E) : \text{Ref}(\mathcal{T})}$$

(deref)

$$\frac{Env [ E : \text{Ref}(\mathcal{T}) }{Env [ \text{deref}(E) : \mathcal{T}$$

## Sistema de Tipos para Core

### ◆ Regras de tipificação para as expressões

(do)

$$\frac{Env [ C \text{ ok} }{Env [ \text{do } C \text{ return } E \text{ end} : \mathcal{V}$$

- ◆ Note que a forma da asserção de tipificação para o comando, não indica nenhum tipo, mas apenas a validade da asserção.

## Sistema de Tipos para Core

### ◆ Tipificação de comandos

(assign)

$$\frac{Env [ N : \text{Ref}(\mathcal{T}) }{Env [ \text{assign}(N, E) \text{ ok}$$

(if)

$$\frac{Env [ E : \text{bool} }{Env [ \text{if } E \text{ then } C \text{ else } C' \text{ end ok}$$

(while)

$$\frac{Env [ E : \text{bool} }{Env [ \text{while } E \text{ do } C \text{ end ok}$$

## Sistema de Tipos para Core

### ◆ Tipificação de comandos

(callp)

$$\frac{Env [ N : \mathcal{T} }{Env [ \text{callp}(M, N) \text{ ok}$$

(decl)

$$\frac{Env, x : \mathcal{T} [ E : \mathcal{T} }{Env [ \text{decl } x : \mathcal{T} = E \text{ in } C \text{ end}$$

- ◆ Podemos estender facilmente linguagem CORE com novos tipos primitivos e operações associadas (por exemplo, strings).

## Valores Produto

- ◆ Correspondem aos **records** da linguagem Pascal, **structs** da linguagem C, etc.

- ◆ Um valor de tipo produto, chamado um **registro**, "agrega" vários valores, de tipos possivelmente diferentes.

- ◆ Construção de registros

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

- ◆ Manipulação de registros (selecção de campo)

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

## Valores Produto

### ◆ Expressões

**newrecord:** (string × EXP)\* → 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 in
 !(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**( $id_1:\mathcal{T}_1, \dots, id_n:\mathcal{T}_n$ ): Tipo dos registos com campos  $id_1, \dots, id_n$  de tipo  $\mathcal{T}_1, \dots, \mathcal{T}_n$ .

$$\frac{\text{Env} [ E : \text{Tuple}(id_1:\mathcal{T}_1, \dots, id_n:\mathcal{T}_n) ]}{\text{Env} [ E. id_j : \mathcal{T}_j ]} \quad (\text{select})$$

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

## Tipos Produto

◆ Qual o tipo da expressão?

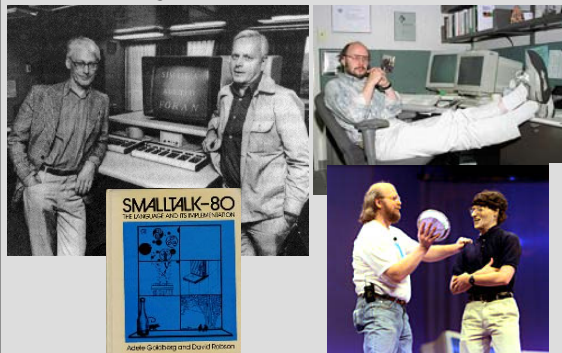
```
decl c = [succ = (fun x -> x+1); loc = var(0)]
in c end
```

◆ A expressão seguinte está bem tipificada?

```
decl c = [succ = (fun x -> x+1); loc = var(0)]
in c.succ(!c.loc) end
```

## Objectos e Classes

## Objectos e Classes



## Objectos e Classes



## A classe Java Counter (v1)

- ◆ Uma classe de “contadores”

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

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

© Luis Caires

LP2 2005/06

337

## Objectos em CORE

- ◆ Um objecto da classe Counter, representado na linguagem CORE por um registo (versão 1)

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

© Luis Caires

LP2 2005/06

338

## Objectos em CORE

- ◆ Um objecto da classe Counter, representado na linguagem CORE por um registo (versão 1).

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

Campo val “público”!

Os “métodos” não podem aceder ao campo val !

© Luis Caires

LP2 2005/06

339

## Objectos em CORE

- ◆ Um objecto da classe Counter, representado na linguagem CORE por um registo (versão 2)

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

© Luis Caires

LP2 2005/06

340

## Objectos em CORE

- ◆ Um objecto da classe Counter, representado na linguagem CORE por um registo (versão 2)

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

Campo val “privado”!

“métodos” definidos no âmbito da declaração de val

© Luis Caires

LP2 2005/06

341

## Objectos em CORE

- ◆ Um objecto da classe Counter, representado na linguagem CORE por um registo (versão 2)

```
decl c = decl
 val = var(0)
 in
 [
 inc = (proc _ -> val := !val + 1);
 get = (fun _ -> !val)
] end
in
 c ← { inc = pclos(∅, val := !val+1, env1),
 get = fclos(∅, !val, env1) }
 c.inc(); c.inc();
 print (c.get());
end
```

env1 → val ← ref(#0)

env2 → c ← { inc = pclos(∅, val := !val+1, env1), get = fclos(∅, !val, env1) }

© Luis Caires

LP2 2005/06

342

## Objectos em CORE

- ◆ A classe Counter, representada em CORE por uma **função geradora de objectos (versão 1)**

```

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

```

© Luis Caires

LP2 2005/06

343

## A classe Java Counter (v2)

- ◆ Uma classe de "contadores" :

```

class Counter {
 int val;
 Counter(n) { val = n; } ← Construtor com parâmetros
 void inc() { val = val + 1; }
 int get() { return val; }
}

Counter c = new Counter(0), d = new Counter(2);
c.inc();
d.inc();
println (c.get()+d.get());

```

© Luis Caires

LP2 2005/06

344

## Objectos em CORE

- ◆ A classe Counter, representada em CORE por uma **função geradora de objectos (versão 2)**

```

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

```

© Luis Caires

LP2 2005/06

345

## Objectos em CORE

- ◆ Em geral, os métodos de um objecto devem poder chamar-se uns aos outros recursivamente. Neste exemplo, **erróneo** (porquê?), dup chama inc.

```

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

```

© Luis Caires

LP2 2005/06

346

## Objectos em CORE

- ◆ Em geral, os métodos de um objecto devem poder chamar-se uns aos outros recursivamente. Neste exemplo, dup chama inc.

```

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

```

© Luis Caires

LP2 2005/06

347

## Objectos em CORE

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

```

decl Counter = (fun -> decl
 val = var(0)
 in declr self =
 [
 inc = (proc -> val := !val + 1),
 dup = (proc -> self.inc(); self.inc())
] in self end
 end)
env1 → val ← ref(#0)
env2 → self ← { inc = pclos(∅, val := !val+1, env2),
 dup = pclos(∅, self.inc();self.inc(), env2) }

```

© Luis Caires

LP2 2005/06

348

## Uma sintaxe para Classes

### ◆ Classes

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

### ◆ Métodos

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

### ◆ Exemplo

```
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 o = new Counter()
 in o.inc(); o.dup(); print(o.get())
 end
end
```

© Luis Caires

LP2 2005/06

349

## Uma sintaxe para Classes

### ◆ Classes

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

### ◆ Métodos

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

### ◆ Exemplo

```
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 o = new Counter()
 in o.inc(); o.dup(); print(o.get())
 end
end
```

O identificador self é declarado implicitamente neste âmbito

© Luis Caires

LP2 2005/06

350

## "Codificação" de Classes

### ◆ Exemplo

```
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 o = new Counter()
 in o.inc(); o.dup(); print(o.get())
 end
end
```

© Luis Caires

LP2 2005/06

351

### ◆ Tradução em CORE

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

## Ambiente em Pilha Simples

- ◆ Os interpretadores das linguagens estudadas (CALCF / CORE) utilizam a representação de ambientes conhecida por "spaghetti stack".
- ◆ Porquê não é possível usar uma representação de ambientes em pilha, reservando espaço para os âmbitos numa disciplina **estritamente** LIFO?
- ◆ Que condições poderíamos impor à linguagem CORE para que esta pudesse ser implementada usando um ambiente em Pilha simples?
- ◆ Poderiam tais condições ser verificadas por um sistema de tipos? Como?

© Luis Caires

LP2 2005/06

352

## Trabalho Final

### ◆ Linguagem de Programação (Comandos)

```
C ::= C;C
| decl id1=E1, ..., idn=En in C end
| E := E
| E(E1,...,En)
| while E do C end
| if E then C else C end
| if E then C end
| print(E)
| println()
```

© Luis Caires

LP2 2005/06

353

## Trabalho Final

### ◆ Linguagem de Programação (Expressões)

```
E ::= id | num | string | true | false
| (E) | E+E | E-E | E*E | E/E | E%E | -E
| E and E | E or E | not E
| E>E | E<=E | E=E | E<>E |
| var(E) | !E
| [id1=E1, ..., idn=En]
| E.id
| fun x1,...,xn => E end
| proc x1,...,xn => C end
| if E then E else E end
| do C return E end
| readInt()
| readString()
```

© Luis Caires

LP2 2005/06

354

## Trabalho Final

### ◆ Comandos do interpretador

```
load "filename";

define id = E ;

C;;

quit;
```

© Luis Caires

LP2 2005/06

355

## Trabalho Final

### ◆ Comandos do interpretador

```
> load "game";

> define prompt =
> fun s => do print(s) return readString() end end;;

> prompt("hello");
helloteste
teste

> quit;
```

© Luis Caires

LP2 2005/06

356

## Trabalho Final

### ◆ Trabalho Base: Interpretador (14 Valores)

### ◆ BONUS I: Sistema de Tipos (3 valores) Desafio: Concepção e implementação das regras de tipificação.

### ◆ BONUS II: Compilador CIL / CLR (3 Valores) Desafio: Compilação de valores funcionais.

### ◆ A + B = 20 valores

© Luis Caires

LP2 2005/06

357

## Tipos para Objectos

© Luis Caires

LP2 2005/06

358

## Uma sintaxe para Classes

### ◆ Classes

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

### ◆ Métodos

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

### ◆ Exemplo

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

© Luis Caires

LP2 2005/06

359

## Tipos para Objectos (1)

### ◆ As únicas operações disponíveis nos objectos são a chamada de método (cf., selecção de campo).

### ◆ $\text{Obj}(id_1:T_1, \dots, id_n:T_n)$ :

Tipo dos objectos com métodos  $id_1, \dots, id_n$ , respectivamente dos tipos funcionais  $T_1, \dots, T_n$ .

(invokef)

$$\text{Env} [ E : \text{Obj}(id_1:T_1, \dots, id_n:T_n) \quad T_j = \text{Fun}(U)\mathcal{R}$$

$$\text{Env} [ F:U$$


---


$$\text{Env} [ E, id_j(F): \mathcal{R}$$

© Luis Caires

LP2 2005/06

360

## Tipos para Objectos (1)

- ◆ As únicas operações disponíveis nos objectos são a **chamada de método** (cf., selecção de campo).

- ◆ **Obj**( $id_1:\mathcal{T}_1, \dots, id_n:\mathcal{T}_n$ ):

Tipo dos objectos com métodos  $id_1, \dots, id_n$ , respectivamente dos tipos funcionais  $\mathcal{T}_1, \dots, \mathcal{T}_n$ .

(invokep)

$$\frac{Env [ E : \mathbf{Obj}(id_1:\mathcal{T}_1, \dots, id_n:\mathcal{T}_n) \quad \mathcal{T}_j = \mathbf{Proc}(\mathcal{U})}{Env [ F:\mathcal{U}]} \\ Env [ E. id_j (F) \mathbf{ok}$$

## Tipos para Objectos (1)

- ◆ A única operação disponível numa classe é a **instanciação de novos objectos**.

- ◆ **Class**( $id_1:\mathcal{T}_1, \dots, id_n:\mathcal{T}_n$ ): Tipo das classes que geram objectos com o tipo **Obj**( $id_1:\mathcal{T}_1, \dots, id_n:\mathcal{T}_n$ ).

(new)

$$\frac{Env [ E : \mathbf{Class}(id_1:\mathcal{T}_1, \dots, id_n:\mathcal{T}_n)}{Env [ \mathbf{new} E(): \mathbf{Obj}(id_1:\mathcal{T}_1, \dots, id_n:\mathcal{T}_n)}$$

## Tipos para Classes (1)

- ◆ A única operação disponível numa classe é a **instanciação de novos objectos**.

- ◆ **Class**( $id_1:\mathcal{T}_1, \dots, id_n:\mathcal{T}_n$ ): Tipo das classes que geram objectos com o tipo **Obj**( $id_1:\mathcal{T}_1, \dots, id_n:\mathcal{T}_n$ ).

$$\frac{Env [ E : \mathcal{T} \\ Env, v:\mathbf{Ref}[\mathcal{T}], \mathbf{self}?:x_1:T_1 [ B_1 : \mathcal{R}_1 \\ \dots \\ Env, v:\mathbf{Ref}[\mathcal{T}], \mathbf{self}?:x_n:T_n [ B_n : \mathcal{R}_n ]}{Env [ \mathbf{class} v := E \\ \mathbf{methods} \mathbf{fun} m_1(x_1:T_1) = B_1 \dots m_n(x_n:T_n) = B_n \\ \mathbf{end} : \mathbf{Class}(m_1:\mathbf{Fun}(T_1)\mathcal{R}_1, \dots, m_n:\mathbf{Fun}(T_n)\mathcal{R}_n)}$$

## Tipos para Classes (1)

- ◆ A única operação disponível numa classe é a **instanciação de novos objectos**.

- ◆ **Class**( $id_1:\mathcal{T}_1, \dots, id_n:\mathcal{T}_n$ ): Tipo das classes que geram objectos com o tipo **Obj**( $id_1:\mathcal{T}_1, \dots, id_n:\mathcal{T}_n$ ).

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

$$\frac{Env [ E : \mathcal{T} \\ Env, v:\mathbf{Ref}[\mathcal{T}], \mathbf{self}?:x_1:T_1 [ B_1 : \mathcal{R}_1 \\ \dots \\ Env, v:\mathbf{Ref}[\mathcal{T}], \mathbf{self}?:x_n:T_n [ B_n : \mathcal{R}_n ]}{Env [ \mathbf{class} v := E \\ \mathbf{methods} \mathbf{fun} m_1(x_1:T_1) = B_1 \dots m_n(x_n:T_n) = B_n \\ \mathbf{end} : \mathbf{Class}(m_1:\mathbf{Fun}(T_1)\mathcal{R}_1, \dots, m_n:\mathbf{Fun}(T_n)\mathcal{R}_n)}$$

## Tipos para Classes (1)

- ◆ Uma solução possível: **etiquetar o tipo completo dos métodos** (parâmetros e resultado):

$$\frac{Env [ E : \mathcal{T} \\ Env, v:\mathbf{Ref}[\mathcal{T}], \mathbf{self}:\mathbf{Obj}(\mathbf{J}), x_1:T_1 [ B_1 : \mathcal{R}_1 \\ \dots \\ Env, v:\mathbf{Ref}[\mathcal{T}], \mathbf{self}:\mathbf{Obj}(\mathbf{J}), x_n:T_n [ B_n : \mathcal{R}_n ]}{Env [ \mathbf{class} v := E \\ \mathbf{methods} \mathbf{fun} m_1(x_1:T_1):\mathcal{R}_1=B_1 \dots m_n(x_n:T_n):\mathcal{R}_n=B_n \\ \mathbf{end} : \mathbf{Class}(\mathbf{J})}$$

## Tipos para Classes (1)

- ◆ Uma solução possível: **etiquetar o tipo completo dos métodos** (parâmetros e resultado):

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

$$\frac{Env [ E : \mathcal{T} \\ Env, v:\mathbf{Ref}[\mathcal{T}], \mathbf{self}:\mathbf{Obj}(\mathbf{J}), x_1:T_1 [ B_1 : \mathbf{Fun}(T_1)\mathcal{R}_1 \\ \dots \\ Env, v:\mathbf{Ref}[\mathcal{T}], \mathbf{self}:\mathbf{Obj}(\mathbf{J}), x_n:T_n [ B_n : \mathbf{Fun}(T_n)\mathcal{R}_n ]}{Env [ \mathbf{class} v := E \\ \mathbf{methods} \mathbf{fun} m_1(x_1:T_1):\mathcal{R}_1=B_1 \dots m_n(x_n:T_n):\mathcal{R}_n=B_n \\ \mathbf{end} : \mathbf{Class}(\mathbf{J})}$$

## Tipos para Classes (1)

- ◆ Esta regra de tipificação permite tipificar quaisquer referências ao self. No entanto não permite tipificar classes que gerem objectos com **tipo recursivo**.

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

© Luis Caires

LP2 2005/06

367

## Tipos para Classes (1)

- ◆ Esta regra de tipificação permite tipificar quaisquer referências ao self. No entanto, não permite tipificar classes que gerem objectos com tipo recursivo.

```
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 ...
```

Como tipificar equal?

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

© Luis Caires

LP2 2005/06

368

## Tipos para Classes e Objectos

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

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

© Luis Caires

LP2 2005/06

369

## Tipos para Classes e Objectos

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

Env<sub>0</sub> [ Counter : **Class**(inc:**Proc**(), get:**Fun**()int, dup:**Proc**())  
Env<sub>1</sub> [ new Counter():**Obj**(inc:**Proc**(), get:**Fun**()int, dup:**Proc**())  
Env<sub>1</sub> [ o.get(): **int**

© Luis Caires

LP2 2005/06

370

## Subtipificação (1)

- ◆ Considere os tipos

**Point**  $\triangleq$  **Obj**(getx:**Fun**()int, gety:**Fun**()int)  
**ColorPoint**  $\triangleq$  **Obj**(getx:**Fun**()int, gety:**Fun**()int,  
setc:**Proc**(C))

- ◆ Considere a declaração

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

© Luis Caires

LP2 2005/06

371

## Subtipificação (1)

- ◆ Considere os tipos e declaração

**Point**  $\triangleq$  **Obj**(getx:**Fun**()int, gety:**Fun**()int)  
**ColorPoint**  $\triangleq$  **Obj**(getx:**Fun**()int, gety:**Fun**()int,  
setc:**Proc**(C))

```
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**.

© Luis Caires

LP2 2005/06

372

## Subtipificação (1)

- ◆ Podemos observar que sempre que temos dois tipos de objecto **T1** e **T2** tais que
 
$$T1 \triangleq \text{Obj}(m_1:T_1, \dots, m_n:T_n)$$

$$T2 \triangleq \text{Obj}(m_1:T_1, \dots, m_n:T_n, n_1:R_1, \dots, p_k:R_k)$$
 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 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**.

© Luis Caires

LP2 2005/06

373

## Subtipificação (1)

- ◆ A relação "mais específico que" é capturada formalmente por uma **relação de subtipificação**.
- ◆ Sempre que
 
$$T1 \triangleq \text{Obj}(m_1:T_1, \dots, m_n:T_n)$$

$$T2 \triangleq \text{Obj}(m_1:T_1, \dots, m_n:T_n, n_1:R_1, \dots, p_k:R_k)$$
 podemos afirmar uma asserção de **subtipificação** da forma

$$T2 \triangleleft T1$$

- ◆ Por exemplo, a asserção seguinte é válida.

$$\text{ColorPoint} \triangleleft \text{Point}$$

© Luis Caires

LP2 2005/06

374

## Subtipificação (1)

- ◆ A relação de subtipificação  $\triangleleft$  pode ser definida através de um conjunto de regras de inferência, tal como as relações de tipificação já estudadas.

$$\frac{}{\text{Env} [ T \triangleleft T ]} \text{ (refl)}$$

$$\frac{\text{Env} [ \text{Obj}(m_1:T_1, \dots, m_n:T_n, n_1:R_1, \dots, p_k:R_k) ]}{\text{Env} [ \text{Obj}(m_1:T_1, \dots, m_n:T_n) ]} \triangleleft \text{Obj}(m_1:T_1, \dots, m_n:T_n) \text{ (obj)}$$

$$\frac{\text{Env} [ \text{Class}(m_1:T_1, \dots, m_n:T_n, n_1:R_1, \dots, p_k:R_k) ]}{\text{Env} [ \text{Class}(m_1:T_1, \dots, m_n:T_n) ]} \triangleleft \text{Class}(m_1:T_1, \dots, m_n:T_n) \text{ (class)}$$

© Luis Caires

LP2 2005/06

375

## Subtipificação (1)

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

$$\frac{\text{Env} [ E : T \quad T \triangleleft U ]}{\text{Env} [ E : U ]} \text{ (subsum)}$$

- ◆ Usando esta regra, podemos por exemplo derivar

$$\frac{\text{Env} [ o : \text{ColorPoint} \quad \text{ColorPoint} \triangleleft \text{Point} ]}{\text{Env} [ o : \text{Point} ]}$$

© Luis Caires

LP2 2005/06

376

## Subtipificação (1)

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

$$\frac{\text{Env} [ N : \mathcal{V} \quad \text{Env} [ M : \text{Fun}(\mathcal{I})\mathcal{V} \quad \mathcal{V} \triangleleft \mathcal{I} ] }{\text{Env} [ \text{callf}(M, N) : \mathcal{V} ]} \text{ (subcall)}$$

$$\frac{\text{Env} [ E : \text{Obj}(id_1:\mathcal{T}_1, \dots, id_n:\mathcal{T}_n) \quad \mathcal{T}_i = \text{Fun}(\mathcal{V})\mathcal{R} \quad \mathcal{V} \triangleleft \mathcal{V} ]}{\text{Env} [ F : \mathcal{V} ]} \text{ (subinvokef)}$$

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

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

© Luis Caires

LP2 2005/06

377

## Subtipificação (2)

- ◆ Considere os tipos

$$\text{Look} \triangleq \text{Obj}(\text{get}:\text{Fun}()\text{Point})$$

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

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

- ◆ Temos certamente **Cell**  $\triangleleft$  **Look** [porquê?]
- ◆ Será que também se pode ter **ColCell**  $\triangleleft$  **Look**?

```

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

```

Env [ cel : Cell

© Luis Caires

LP2 2005/06

378

## Subtipificação (2)

- ◆ Considere os tipos

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

- ◆ Temos certamente **Cell**  $<$ : **Look** [porquê?]
- ◆ Será que também se pode ter **ColCell**  $<$ : **Look**?

```

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

```

$Env [ colcel : ColCell ]$

© Luis Caires

LP2 2005/06

379

## Subtipificação (2)

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

**T1**  $\triangleq$  Obj( $m_1$ :Fun( $\mathcal{U}_1$ ) $T_1, \dots, m_n$ :Fun( $\mathcal{U}_n$ ) $T_n$ )  
**T2**  $\triangleq$  Obj( $m_1$ :Fun( $\mathcal{U}_1$ ) $R_1, \dots, m_n$ :Fun( $\mathcal{U}_n$ ) $R_n$ )

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**.

© Luis Caires

LP2 2005/06

380

## Subtipificação (2)

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

$$\frac{U_1 <: T_1 \dots U_n <: T_n}{Env [ Obj(m_1:Fun(\mathcal{U}_1)U_1, \dots, m_n:Fun(\mathcal{U}_n)U_n, n_1:R_1, \dots, p_k:R_k) <: Obj(m_1:Fun(\mathcal{U}_1)T_1, \dots, m_n:Fun(\mathcal{U}_n)T_n) }$$

$$\frac{U_1 <: T_1 \dots U_n <: T_n}{Env [ Class(m_1:Fun(\mathcal{U}_1)U_1, \dots, m_n:Fun(\mathcal{U}_n)U_n, n_1:R_1, \dots, p_k:R_k) <: Class(m_1:Fun(\mathcal{U}_1)T_1, \dots, m_n:Fun(\mathcal{U}_n)T_n) }$$

© Luis Caires

LP2 2005/06

381

## Subtipificação (2)

- ◆ Considere os tipos

**Put**  $\triangleq$  Obj(set:Proc(Point))  
**Cell**  $\triangleq$  Obj(get:Fun()Point, set:Proc(Point))  
**ColCell**  $\triangleq$  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(col) ... cel.get().setc(red) ...
end

```

$Env [ pt : Point ]$   
 $Env [ cel : ColorCell ]$

© Luis Caires

LP2 2005/06

382

## Subtipificação (2)

- ◆ Considere os tipos

**Put**  $\triangleq$  Obj(set:Proc(Point))  
**Cell**  $\triangleq$  Obj(get:Fun()Point, set:Proc(Point))  
**ColCell**  $\triangleq$  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.put(pt) end
in
 ... fill(col) ... cel.get().setc(red) ...
end

```

$Env [ pt : Point ]$   
 $Env [ cel : ColorCell ]$

Erro! Método indefinido

© Luis Caires

LP2 2005/06

383

## Subtipificação (2)

- ◆ Considere os tipos

**Put**  $\triangleq$  Obj(set:Proc(Point))  
**ColCell1**  $\triangleq$  Obj(get:Fun()Point, set:Proc(ColorPoint))  
**ColCell2**  $\triangleq$  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(col) ... cel.get().setc(red) ...
end

```

$Env [ pt : ColorPoint ]$   
 $Env [ cel : ColCell2 ]$

Ok!

© Luis Caires

LP2 2005/06

384

## Subtipificação (2)

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

**T1**  $\triangleq$  **Obj**( $m_1$ :**Fun**( $\mathcal{V}_1$ ) $T_1, \dots, m_n$ :**Fun**( $\mathcal{V}_n$ ) $T_n$ )

**T2**  $\triangleq$  **Obj**( $m_1$ :**Fun**( $\mathcal{V}'_1$ ) $R_1, \dots, m_n$ :**Fun**( $\mathcal{V}'_n$ ) $R_n$ )

com  $R_i <: T_i$  e  $\mathcal{V}_i <: \mathcal{V}'_i$  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 **ColCel**.

© Luis Caires

LP2 2005/06

385

## Subtipificação (2)

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

$$\frac{U_1 <: T_1 \dots U_n <: T_n \quad \mathcal{V}_1 <: \mathcal{V}'_1 \dots \mathcal{V}_n <: \mathcal{V}'_n}{\text{Env} [\text{Obj}(m_1:\text{Fun}(\mathcal{V}'_1)U_1, \dots, m_n:\text{Fun}(\mathcal{V}'_n)U_n, \rho_1:R_1, \dots, \rho_k:R_k) <: \text{Obj}(m_1:\text{Fun}(\mathcal{V}_1)T_1, \dots, m_n:\text{Fun}(\mathcal{V}_n)T_n)]}$$

(objsub)

$$\frac{U_1 <: T_1 \dots U_n <: T_n \quad \mathcal{V}_1 <: \mathcal{V}'_1 \dots \mathcal{V}_n <: \mathcal{V}'_n}{\text{Env} [\text{Class}(m_1:\text{Fun}(\mathcal{V}'_1)U_1, \dots, m_n:\text{Fun}(\mathcal{V}'_n)U_n, \rho_1:R_1, \dots, \rho_k:R_k) <: \text{Class}(m_1:\text{Fun}(\mathcal{V}_1)T_1, \dots, m_n:\text{Fun}(\mathcal{V}_n)T_n)]}$$

(classsub)

© Luis Caires

LP2 2005/06

386

## Co e Contravariância

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

$$\frac{U_1 <: T_1 \dots U_n <: T_n \quad \mathcal{V}_1 <: \mathcal{V}'_1 \dots \mathcal{V}_n <: \mathcal{V}'_n}{\text{Env} [\text{Obj}(m_1:\text{Fun}(\mathcal{V}'_1)U_1, \dots, m_n:\text{Fun}(\mathcal{V}'_n)U_n, \rho_1:R_1, \dots, \rho_k:R_k) <: \text{Obj}(m_1:\text{Fun}(\mathcal{V}_1)T_1, \dots, m_n:\text{Fun}(\mathcal{V}_n)T_n)]}$$

(objsub)

$$\frac{U_1 <: T_1 \dots U_n <: T_n \quad \mathcal{V}_1 <: \mathcal{V}'_1 \dots \mathcal{V}_n <: \mathcal{V}'_n}{\text{Env} [\text{Class}(m_1:\text{Fun}(\mathcal{V}'_1)U_1, \dots, m_n:\text{Fun}(\mathcal{V}'_n)U_n, \rho_1:R_1, \dots, \rho_k:R_k) <: \text{Class}(m_1:\text{Fun}(\mathcal{V}_1)T_1, \dots, m_n:\text{Fun}(\mathcal{V}_n)T_n)]}$$

(classsub)

© Luis Caires

LP2 2005/06

387

## Tipos para Classes (2)

- ◆ Esta regra de tipificação permite tipificar quaisquer referências ao **self**. No entanto, não permite tipificar classes que gerem objectos com tipo recursivo.

```
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 ...
```

Como tipificar equal()?

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

© Luis Caires

LP2 2005/06

388

## Tipos para Classes (2)

- ◆ Esta regra de tipificação permite tipificar quaisquer referências ao **self**. No entanto, não permite tipificar classes que gerem objectos com tipo recursivo

```
decl Counter =
class
 val := 0
 init (c:int)
 val := c
 methods
 proc inc() = val := !val + 1 end
 fun get():int = !val end
 fun clone():T = new Counter(!val); end
end in ...
```

Como tipificar clone()?

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

© Luis Caires

LP2 2005/06

389

## Tipos Recursivos

- ◆ Os tipos recursivos permitem especificar estruturas de tipos "infinitas".

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

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

© Luis Caires

LP2 2005/06

390

## Tipos Recursivos

- Os tipos recursivos permitem especificar estruturas de tipos "infinitas".

$\mathcal{X}$  é uma variável de tipo: representa o tipo do self.

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

```
T ≜ Obj(inc:Proc(), get:Fun()int, clone:Fun()T)
 Obj(inc:Proc(), get:Fun()int, clone:Fun()T)
 Obj(inc:Proc(), get:Fun()int, clone:Fun()T)
 Obj(inc:Proc(), get:Fun()int, clone:Fun()T)
 Obj(inc:Proc(), get:Fun()int, clone:Fun()T)
```

## Tipos Recursivos

- Os tipos recursivos permitem especificar estruturas de tipos "infinitas".

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

Os tipos recursivos satisfazem a seguinte igualdade estrutural, chamada "desdobramento" (*unfolding*).

$\text{Obj}(\mathcal{X})(\text{id}_n:\mathcal{T}_n) \equiv \text{Obj}(\text{Subst}(\text{id}_n:\mathcal{T}_n, \mathcal{X}, \text{Obj}(\mathcal{X})(\text{id}_n:\mathcal{T}_n)))$

- Em  $\text{Obj}(\mathcal{X})(\text{id}_n:\mathcal{T}_n)$  a variável  $\mathcal{X}$  é ligada em  $\text{id}_n:\mathcal{T}_n$
- $\text{Subst}(J, \mathcal{X}, T)$  denota a substituição de todas as ocorrências livres de  $\mathcal{X}$  em  $J$  pelo tipo  $T$ .

## Tipos para Classes (2)

- Uma solução possível: **etiquetar o tipo completo dos métodos (parâmetros e resultado)**, usando tipos recursivos:

$J(\text{Self}) \triangleq (m_1:\text{Fun}(T_1)R_1, \dots, m_n:\text{Fun}(T_n)R_n)$

```
Env [E : T
 Env, v:Ref[A], self:Obj(Self)(J(Self)), x_1:T_1 [B_1 : R_1
 ...
 Env, v:Ref[A], self:Obj(Self)(J(Self)), x_n:T_n [B_n : R_n

 Env [class v := E
 methods fun m_1(x_1:T_1):R_1 = B_1 ... m_n(x_n:T_n):R_n = B_n
 end: Class(Self)(J(Self))
```

## Tipos para Classes (2)

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

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

```
decl Counter =
 class
 val := 0
 init (c:int)
 val := c
 methods
 proc inc() = val := !val + 1 end
 fun get() = !val end
 fun equal(c: CType) = (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 OO.

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

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