Functions

Massimo Merro

24 October 2017

Most programming languages have some notion of *function*, *method*, or *procedure*

... to abstract a piece of code on formal parameters so that you can use that code multiple times with different parameters.

```
fun succ x = x +1
```

```
public int succ(int x) {
   x+1
}
```

```
<script type="text/vbscript">
 function succ(x)
 succ = x+1
 end function
</script>
```

Obbiettivo introdurre la parte funzionale al nostro programma, analizzando i vari problemi

Molti linguaggi di programmazione sono puramente funzionali

Functions - Examples

definiamo grammatica e typesystem (parte statica)
definiamo l'interprete (parte dinamica)

Thus, we will extend our language with expressions of this form:

- fn x : int $\Rightarrow x + 1$
- (fn x : int $\Rightarrow x + 1$)8
- fn y : int $\Rightarrow$ (fn x : int $\Rightarrow x + y$)
- (fn y : int $\Rightarrow$ (fn x : int $\Rightarrow x + y$))9

  funzione che dato un intero x ritorna x+9

  applica a quello di sinistra quello di destra
- fn x : int $\rightarrow$ int $\Rightarrow$ (fn y : int $\Rightarrow x(xy)$)
- (fn x : int $\rightarrow$ int $\Rightarrow$ (fn y : int $\Rightarrow x(xy)$))(fn x : int $\Rightarrow x + 1$)
- ((fn x : int $\rightarrow$ int $\Rightarrow$ (fn y : int $\Rightarrow x(xy)$))(fn x : int $\Rightarrow x + 1$))7

For simplicity,

- our functions are anonymous: they don't have a name
- they take always a single argument and return a single result
- they are always typed.

Functions - Extended Syntax

argomento per quella funzione

Variables $x \in \mathbb{X}$, for $\mathbb{X} = \{\mathrm{x}, \mathrm{y}, \mathrm{z}, \dots\}$

place holder

Expressions $e \quad ::= \quad \dots \quad | \quad \mathsf{fn}\ x : T \Rightarrow e \quad | \quad e\ e \quad | \quad x$

$f(x) = e$

parametro formale

funzione

Types

$$T \quad ::= \quad \mathsf{int} \quad | \quad \mathsf{bool} \quad | \quad \mathsf{unit} \quad | \quad T \rightarrow T$$

$\lambda x.T.e$

$$T_{loc} \quad ::= \quad \mathsf{intref}$$

$T_1 := \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit}$

$T := T_1 \mid T_1 \rightarrow T_1$

Limitata non posso fare funzioni funzioni composte ma solo da tipo primitivo 'in tipo primitivo

Conventions:

- Function application associates to the left: $e_1 e_2 e_3 = (e_1 e_2) e_3$
- Function type associates to the right:

$$T_1 \rightarrow T_2 \rightarrow T_3 = T_1 \rightarrow (T_2 \rightarrow T_3)$$

legata alla dichiarazione

- fn extends to the right as far as possible, thus $\mathsf{fn}\ \mathrm{x} : \mathsf{unit} \Rightarrow \mathrm{x}; \mathrm{x}$ corresponds to $\mathsf{fn}\ \mathrm{x} : \mathsf{unit} \Rightarrow (\mathrm{x}; \mathrm{x})$

libera

- and $\mathsf{fn}\ \mathrm{x} : \mathsf{unit} \Rightarrow \mathsf{fn}\ \mathrm{y} : \mathsf{int} \Rightarrow \mathrm{x}; \mathrm{y}$ has type $\mathsf{unit} \rightarrow \mathsf{int} \rightarrow \mathsf{int}$

Note that:

- Variables are not locations ($\mathbb{X} \cap \mathbb{L} = \emptyset$) so, $\mathrm{x} := 3$ is not allowed!
- you cannot abstract on locations: fn l : intref $\Rightarrow$ $!\mathrm{l} + 5$ is not in the syntax!
- the (non-meta) variables $\mathrm{x}, \mathrm{y}, \mathrm{z}$ are not the same as metavariables $x, y, z, \ldots$.
- The type grammar and the expression syntax suggest the language includes higher-order functions: you can abstract on a variable of any type
- If you wanted only *first-order functions* you should change the type grammar. How?

Variable shadowing

In a language with nested function definitions it is desirable to define a function without being aware of what are the variables in the surrouding space.
For instance,

sono due
x diverse

legata alla sua
definizione

$$\text{fn } x : \text{int} \Rightarrow (\text{fn } x : \text{int} \Rightarrow x + 1)$$

Variable shadowing is not allowed in Java!

F = fn x : int =>
((fn x : int -> x + 1) x
funzionale
F
F 5

```
class F {
  void m() {
    int y;
    {int y; .....} \\static error!!!
    ....
  }
}
```

Alpha conversion

In expressions of the form fn $x : T \Rightarrow e$ the variable x is a bound in e.

- x is the formal parameter of the function: any occurrence of x in e, which does not occur inside a nested function definition, means the same thing
- outside the term "fn $x : t \Rightarrow e$" the variable x does not mean anything!
- As a consequence, it does not matter which variable has been chosen as formal parameter: $\mathrm{fn}\ \mathrm{x} : \mathrm{int} \Rightarrow \mathrm{x} + 2$ and $\mathrm{fn}\ \mathrm{y} : \mathrm{int} \Rightarrow \mathrm{y} + 2$ denotes exactly the same function!

sono uguali

Free and bound variables

We will say that an occurrence of a variable x inside an expression e is free if x is not inside any term of the form fn $x : T \Rightarrow \dots$. For example, variable x is free in the following expressions:

- 21
- $\mathrm{x} + \mathrm{y}$
- fn $\mathrm{z} : T \Rightarrow \mathrm{x} + \mathrm{z}$

free aspetta di essere legata

bound

Notice that, in the last example the variable x if free but the variable z is bound by the closest enclosing function definition fn $\mathrm{z} : T \Rightarrow \dots$.

Notice also that in the expression

bound

$$\text{fn } \mathrm{x} : T' \Rightarrow \text{fn } \mathrm{z} : T \Rightarrow \mathrm{x} + \mathrm{z}$$

variable x is not free anymore but it is bound by the closest enclosing fn $\mathrm{x} : T' \Rightarrow \dots$.

Alpha conversion - The convention

Convention: we will allow ourselves to any time, in any expression

$$\ldots(\mathsf{fn}\ x : T \Rightarrow e)\ldots$$

to replace the binding x and all occurrences of x in e that are bound to that binder, by any other *fresh* variable that does not occur elsewhere:

- $\mathsf{fn}\ \mathrm{x} : T \Rightarrow \mathrm{x} + \mathrm{z} = \mathsf{fn}\ \mathrm{y} : T \Rightarrow \mathrm{y} + \mathrm{z}$
- $\mathsf{fn}\ \mathrm{x} : T \Rightarrow \mathrm{x} + \mathrm{y} \neq \mathsf{fn}\ \mathrm{y} : T \Rightarrow \mathrm{y} + \mathrm{y}$

free

bound

This is called "working up to alpha conversion".

NON devo fare cattura di nomi → le variabili globali diventano locali

Free variables, formally

The intuition is that free variables are not (yet) bound to some expression.

Let us define the following function by induction:

$\mathrm{fv}(\,) : \mathrm{Exp} \rightarrow 2^{\mathbb{X}}$

facciamo pattern matching di Exp

- $\mathrm{fv}(x) \stackrel{\text{def}}{=} \{x\}$
- $\mathrm{fv}(\mathrm{fn}\ x : T \Rightarrow e) \stackrel{\text{def}}{=} \mathrm{fv}(e) \setminus \{x\}$
- $\mathrm{fv}(e_1 e_2) = \mathrm{fv}(e_1; e_2) \stackrel{\text{def}}{=} \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)$
- $\mathrm{fv}(n) = \mathrm{fv}(b) = \mathrm{fv}(!l) = \mathrm{fv}(skip) \stackrel{\text{def}}{=} \emptyset$
- $\mathrm{fv}(e_1 \text{ op } e_2) = \mathrm{fv}(\text{while } e_1 \text{ do } e_2) \stackrel{\text{def}}{=} \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)$
- $\mathrm{fv}(l := e) = \mathrm{fv}(e)$
- $\mathrm{fv}(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) = \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \cup \mathrm{fv}(e_3)$

An expression e is said to be closed if $\mathrm{fv}(e) = \emptyset$.

↳ I programmi che eseguiamo

Substitution - Examples

The semantics for functions will involve *substituting* actual paramenter for formal parameters.

Write $e_2\{^{e_1}/_{x}\}$ for the result of substituting e_1 for all *free occurences* of x in e_2. For example,

$$
\begin{array}{rcl}
(\mathrm{x} \geq \mathrm{x})\{^{3}/_{\mathrm{x}}\} & = & (3 \geq 3) \\
((\mathsf{fn}\ \overset{\text{bound}}{\mathrm{x}} : \mathsf{int} \Rightarrow \overset{\text{bound}}{\mathrm{x}} + \mathrm{y})\overset{\text{free}}{\mathrm{x}})\{^{3}/_{\mathrm{x}}\} & = & ((\mathsf{fn}\ \overset{\text{bound}}{\mathrm{x}} : \mathsf{int} \Rightarrow \overset{\text{bound}}{\mathrm{x}} + \mathrm{y})3) \\
(\mathsf{fn}\ \mathrm{y} : \mathsf{int} \Rightarrow \mathrm{x} + \mathrm{y})\{^{\mathrm{y}+2}/_{\mathrm{x}}\} & = & \mathsf{fn}\ \mathrm{z} : \mathsf{int} \Rightarrow (\mathrm{y} + 2) + \mathrm{z}
\end{array}
$$

alpha conversion

Note that in the last substitution we "work up to alpha conversion" to avoid name capture!

devo ridenominare la variabile locale con valori fresh

Substitution - Definition

$\hat{e}\{^e/_x\}$: substitute expression e for each free occurrence of x in $\hat{e}$.

$$
\begin{array}{lcl}
n\{^e/_x\} & \stackrel{\text{def}}{=} & n \\
b\{^e/_x\} & \stackrel{\text{def}}{=} & b \\
skip\{^e/_x\} & \stackrel{\text{def}}{=} & skip \\
\mathrm{x}\{^e/_\mathrm{x}\} & \stackrel{\text{def}}{=} & e \\
\mathrm{y}\{^e/_\mathrm{x}\} & \stackrel{\text{def}}{=} & \mathrm{y} \\
(\mathsf{fn}\ \mathrm{x} : T \Rightarrow e_1)\{^e/_\mathrm{z}\} & \stackrel{\text{def}}{=} & (\mathsf{fn}\ \mathrm{x} : T \Rightarrow e_1\{^e/_\mathrm{z}\}) \ \text{ if } \mathrm{x} \notin \mathsf{fv}(e) \\
(\mathsf{fn}\ \mathrm{x} : T \Rightarrow e_1)\{^e/_\mathrm{z}\} & \stackrel{\text{def}}{=} & (\mathsf{fn}\ \mathrm{y} : T \Rightarrow (e_1\{^\mathrm{y}/_\mathrm{x}\})\{^e/_\mathrm{z}\}) \ \text{ if } \mathrm{x} \in \mathsf{fv}(e) \wedge \mathrm{y} \text{ fresh}^1 \\
(\mathsf{fn}\ \mathrm{x} : T \Rightarrow e_1)\{^e/_\mathrm{x}\} & \stackrel{\text{def}}{=} & (\mathsf{fn}\ \mathrm{x} : T \Rightarrow e_1) \\
(e_1 e_2)\{^e/_\mathrm{x}\} & \stackrel{\text{def}}{=} & (e_1\{^e/_\mathrm{x}\} e_2\{^e/_\mathrm{x}\})
\end{array}
$$

non utilizzata nel programma

si dice omomorfismo, l'operazione si distribuisce su tutti gli elementi

... on the other expressions substitution is an homomorphism.

[1] Here we do alpha conversion.

Simultaneous substitutions

- Substitutions can be easily generalised to replace more variables simultaneously
- More generally, a *simultaneous substitution* σ is a partial function $\sigma : \mathbb{X} \rightharpoonup \mathrm{Exp}$
- Given an expression e, we write $e\sigma$ to denote the expression resulting by the simultaneous substitution of each $x \in \mathrm{dom}(\sigma)$ by the corresponding expression $\sigma(x)$
- Notation: write σ as $\{e_1/x_1, \ldots, e_k/x_k\}$ instead of $\{x_1 \mapsto e_1, \ldots, x_k \mapsto e_k\}$.
- We will write $e\sigma$ to denote the expression e which has been affected by the substitution σ.

$$(x) \quad \left\{ {}^{y+2}/_{x},\ {}^{5}/_{y} \right\}$$

- una sostituzione per volta $\longrightarrow 7$
- simultaneamente $\longrightarrow y+2$

La sostituzione avviene tipicamente in simultanea

The λ-calculus: the core of sequential programming languages

definizione di funzione

- Our functions fn x:T $\Rightarrow$ e could be written in λ-calculus as $\lambda x{:}T.e$
- In the mid 1960s, Peter Landin observed that a complex programming language (such as ALGOL 60) can be understood by focussing on a tiny core calculus capturing the language's essential mechanisms ....
- ... together with a collection of convenient *derived constructs* whose behaviour is understood by translating them into the core calculus.
- The core language used by Landin was the λ-calculus, a formal system invented by Alonzo Church in the 1930's as a universal language of computable functions.
- In 1960, John McCarthy published the design of the programming language Lisp based on the λ-calculus.
- Since then, the λ-calculus has seen a widespread use in
  - the specification of programming language features
  - in language design and implementation
  - in the study of type systems.

Expressiveness of the λ-calculus

- The λ-calculus can be viewed as a very simple programming language in which computations can be described as mathematical objects.
- It is a formal language in which
  - every term denotes a function
  - any term (function) can be applied to any other term, so functions are inherently *higher-order*
- Despite its simplicitiy, it is a *Turing-complete* language: it can express computations on natural number as does any other known programming language.
- Church's Thesis: any conceivable notion of computable function on natural numbers is equivalent to the λ-calculus.
- The force of Church's Thesis is that it postulates that all future notions of computation will be equivalent in expressive power to the λ-calculus.

Encoding language features in λ-calculus

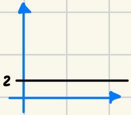
- The λ-calculus can be enriched in a variety of ways.
- It is often convenient to add special constructs for features like numbers, booleans, tuples, records, etc.
- However, all these features can be encoded in the λ-calculus, so they represent only "syntactic sugar".
- Such extensions lead eventually to programming languages such as Lisp (McCarthy, 1960), ML (Milner et al., 1990), Haskell (Hudak et al., 1992), or Scheme (Sussman and Steele, 1975).
- In the the previous slides, we have basically extended our language with the λ-calculus primitives.

The untyped λ-calculus

$$M \in Lambda \quad ::= \quad x \quad \big| \quad \lambda x.M \quad \big| \quad M\,M$$

- x is a variable, used to define formal parameters
- $\lambda x.M$: called λ-abstraction, define anonymous functions
- this construct is a binder as the variable x is bound in the body function M
- $M_1 M_2$: apply function M_1 to argument M_2
- Thus, $(\lambda x.M)N$ evolves in $M\{^N/_x\}$, where the argument N replaces each (free) occurrence of x in M
- In (pure) λ-calculus functions are the only values
- Integer, Boolean and other basic values can be easily codified: they are not primitive in λ-calculus.

$$(\text{fn } x : int \Rightarrow 2)\ \underset{}{\overset{\left(\begin{array}{c}\text{while true}\\ \text{do skip}\end{array}\right)}{\Omega}}$$



eager : la computazione diverge

lazy : risulta 2

$$\text{fn } x : T \Rightarrow \overset{\text{100 volte}}{\ldots x \ldots x \ldots x \ldots}$$

(Programma pesante)

(eager) : $(\ldots \tau \ldots \tau \ldots)$

(lazy) : $\underbrace{\ldots P \ldots P \ldots}_{\text{100 volte}}$

Function application: intuition

To evaluate $M_1 M_2$:

- First evaluate M_1 to a function $\lambda x.M$
- Then it depends on the evaluation strategy.

If Call-by-value:

- evaluate M_2 to a value v
- evaluate $M\{^{v}/_{x}\}$.

If Call-by-name:

- evaluate $M\{^{M_2}/_{x}\}$

Function application: formal semantics

$$(\text{App})\ \frac{M_1 \twoheadrightarrow M_1'}{M_1 M_2 \twoheadrightarrow M_1' M_2}$$

Call-by-value:

continuo a computare finché ottengo un valore

$$(\text{CBV.A})\ \frac{M_2 \twoheadrightarrow M_2'}{(\lambda x.M)M_2 \twoheadrightarrow (\lambda x.M)M_2'} \qquad (\text{CBV.B})\ \frac{-}{(\lambda x.M)v \twoheadrightarrow M\{^v/_x\}}$$

valore ottenuto

where $v \in Val \ ::= \ \lambda x.N$

Call-by-name

lazy, nessuna valutazione dell'argomento

$$(\text{CBN})\ \frac{-}{(\lambda x.M)M_2 \twoheadrightarrow M\{^{M_2}/_x\}}$$

where M_2 is a closed term (i.e. a program).

Self-application

Is it possible to express non-terminating programs in *Lambda*?

Yes, of course!

For example, the *divergent* combinator

$$\Omega \stackrel{\text{def}}{=} (\lambda x.xx)(\lambda x.xx)$$

contains just one *redex*, and reducing this redex yields exactly Ω again!

$$\Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \ldots\ldots\Omega\ldots\ldots$$

Non-termination is built-in in *Lambda*!

Call-by-name vs. call-by-value

non vanno di pari passo

Note that unlike the previous language, having function definitions among the constructs may lead to different results!!!

They give different results:

- $(\lambda x.0)(\Omega) \rightarrow_{cbn} 0$
- $(\lambda x.0)(\Omega) \rightarrow_{cbv} (\lambda x.0)(\Omega) \rightarrow_{cbv} \ldots \rightarrow_{cbv} \ldots$

Even more surprisingly:

macro per la funzione identità

$\lambda x = x$

applicata a zero ritorna zero

- $(\lambda x.\lambda y.x)(Id\ 0) \rightarrow^{*}_{cbn} \lambda y.(Id\ 0)$
- $(\lambda x.\lambda y.x)(Id\ 0) \rightarrow^{*}_{cbv} \lambda y.0$

For different evaluation strategies see Benjamin C. Pierce's "Types and Programming Languages" at pp. 56.

Back to our language: function behaviour

Consider the expression:

$$e \stackrel{\text{def}}{=} (\text{fn } \mathrm{x}{:}\text{unit} \Rightarrow (\mathrm{l} := 1); \mathrm{x})\, (\mathrm{l} := 2)$$

then consider a run in a store where l is associated to 0:

$\langle e, \{\mathrm{l} \mapsto 0\}\rangle \rightarrowtriangle^* \langle skip, \{\mathrm{l} \mapsto ???\}\rangle$

What is the resulting store?
This is not a trivial questions as there are a number of different possibilities for evaluating function calls!

Evaluating function calls (1)

How to evaluate a function call $e_1 e_2$?

Call-by-value (also called eager evaluation):

- Evaluate e_1 to a function fn $x{:}T \Rightarrow e$
- Evaluate e_2 to a value v
- Substitute actual parameter v, for formal parameter x in the body function e
- Evaluate $e\{^v/_x\}$

Used in many languages such as C, Scheme, ML, OCaml, Java, etc (there are several variants of call-by-value)

Evaluation function calls (2)

There is at leat another way to evaluata $e_1 e_2$:

Call-by-name (also called lazy evaluation):

- Evaluate e_1 to a function fn $x{:}T \Rightarrow e$
- Substitute the argument e_2, without evaluating it, for formal parameter x in the body function
- Evaluate $e\{^{e_2}/_x\}$

$\ell_2 = \text{if } \ell > 5 \text{ then } 0 \text{ else } 1$

$e \equiv \ldots x \ldots x \ldots x \ldots$ 1 2 3 ℓ_2 ℓ_2

Variants of call-by-name have been used in some well-known programming languages, notably Algol-60 (Naur et al., 1963) and Haskell (Hudak et al., 1992).

Haskell actually uses an optimised version known as call-by-need (Wadsworth, 1971) that, instead of re-evaluating an argument each time it is used, overwrites all occurrences of the argument with its value the first time it is evaluated.

Function Behaviour: Call-by-value

Let us evaluate our previous example in a call-by-value strategy:

programma che dato un programma fa $l := 1$ e poi esegue il programma. Il programma eseguito è $l := 2$

$$e \;=\; (\mathrm{fn}\ x\mathrm{:unit} \Rightarrow (l := 1); x)\,(l := 2)$$

then

$$\begin{aligned}
\langle e, \{l \mapsto 0\}\rangle \;&\rightarrow\; \langle (\mathrm{fn}\ x\mathrm{:unit} \Rightarrow (l := 1); x)\mathit{skip}\ ,\ \{l \mapsto 2\}\rangle \\
&\rightarrow\; \langle (l := 1; \mathit{skip})\ ,\ \{l \mapsto 2\}\rangle \\
&\rightarrow\; \langle \mathit{skip}; \mathit{skip}\ ,\ \{l \mapsto 1\}\rangle \\
&\rightarrow\; \langle \mathit{skip}\ ,\ \{l \mapsto 1\}\rangle
\end{aligned}$$

At the end of the evaluation the location l is associated to 1.

Function Behaviour: Call-by-name

ci sono side-effects

Let us evaluate our previous example in a call-by-name strategy:

$$e \;=\; (\mathsf{fn}\ \mathrm{x}{:}\mathsf{unit} \Rightarrow (\mathrm{l} := 1); \mathrm{x})\,(\mathrm{l} := 2)$$

then

$$\begin{array}{rcl}
\langle e, \{\mathrm{l} \mapsto 0\}\rangle & \twoheadrightarrow & \langle (\mathrm{l} := 1); \mathrm{l} := 2\ ,\ \{\mathrm{l} \mapsto 0\}\rangle \\
& \twoheadrightarrow & \langle (\mathit{skip}; \mathrm{l} := 2)\ ,\ \{\mathrm{l} \mapsto 1\}\rangle \\
& \twoheadrightarrow & \langle \mathrm{l} := 2\ ,\ \{\mathrm{l} \mapsto 1\}\rangle \\
& \twoheadrightarrow & \langle \mathit{skip}\ ,\ \{\mathrm{l} \mapsto 2\}\rangle
\end{array}$$

Which makes quite a difference with respect to a call-by-value strategy!

Call-by-value: small-step semantics

Values $v \quad ::= \quad b \quad | \quad n \quad | \quad skip \quad | \quad \mathsf{fn}\ x : T \Rightarrow e$

$$(\text{CBV-app1})\ \frac{\langle e_1, s\rangle \twoheadrightarrow \langle e_1', s'\rangle}{\langle e_1\, e_2\,,\ s\rangle \twoheadrightarrow \langle e_1'\, e_2\,,\ s'\rangle}$$

non ottengo né int, bool, ecc solo fun

$$(\text{CBV-app2})\ \frac{\langle e_2, s\rangle \twoheadrightarrow \langle e_2', s'\rangle}{\langle v e_2\,,\ s\rangle \twoheadrightarrow \langle v e_2'\,,\ s'\rangle}$$

siamo in un linguaggio tipato, può uscire solo una funzione

$$(\text{CBV-fn})\ \frac{}{\langle (\mathsf{fn}\ x : T \Rightarrow e) v\,,\ s\rangle \twoheadrightarrow \langle e\{^{v}/_{x}\}\,,\ s\rangle}$$

qui ho bisogno della x per fare la sostituzione

- Function evaluation does not touch the store: In a pure functional language we would not need a store!
- In $e\{^{v}/_{x}\}$ the value v would be copied in e as many times as there are
  è un place-holder
  free occurences of x in e. Real implementations don't do that!

Call-by-name: small-step semantics

$$(\text{CBN-app})\ \frac{\langle e_1, s\rangle \rightarrow \langle e_1', s'\rangle}{\langle e_1 e_2\ ,\ s\rangle \rightarrow \langle e_1' e_2\ ,\ s'\rangle}$$

$$(\text{CBN-fn})\ \frac{-}{\langle (\text{fn}\ x : T \Rightarrow e)e_2\ ,\ s\rangle \rightarrow \langle e\{{}^{e_2}/_{x}\}\ ,\ s\rangle}$$

Here, we don't evaluate the argument at all if it is not used in the function body:

$$\begin{array}{ll} & \langle (\text{fn}\ \mathrm{x} : \text{unit} \Rightarrow \mathit{skip})(\mathrm{l} := 2)\ ,\ \{\mathrm{l} \mapsto 0\}\rangle \\ \rightarrow & \langle \mathit{skip}\{(\mathrm{l} := 2/\mathrm{x})\}\ ,\ \{\mathrm{l} \mapsto 0\}\rangle \\ \rightarrow & \langle \mathit{skip}\ ,\ \{\mathrm{l} \mapsto 0\}\rangle \end{array}$$

but if it is used the we end up evaluating it repeatedly.

Call-by-value vs. call-by-name

Il linguaggio é Turing-Complete

$\text{loop} \stackrel{\text{def}}{=} \text{while } \textit{true} \text{ do } \textit{skip}$

viene scartato

$\text{fst} \stackrel{\text{def}}{=} \text{fn } x : \text{int} \Rightarrow \text{fn } y : \text{unit} \Rightarrow x$

é un funzionale o prende valori o li ritorna

Then,

- Call-by-name: $\langle \text{fst } 0 \text{ loop}, s\rangle \longrightarrow^* \langle 0, s\rangle$
- Call-by-value: $\langle \text{fst } 0 \text{ loop}, s\rangle \longrightarrow^* \ldots\ldots$ diverges!

non esco più dal while

$\text{dup} \stackrel{\text{def}}{=} \text{fn } y : \text{int} \Rightarrow y \times y$

contatore accumulatore

$\text{fact} \stackrel{\text{def}}{=} \text{fn } x : \text{int} \Rightarrow (l := 1; m := 1; \text{while } !l \leq x \text{ do } (m := !m * !l; l := !l + 1); !m)$

mi aspetto sia un intero

operazione pesante

Then,

- Call-by-name: in $\langle \text{dup}(\text{fact}(40)), s\rangle$, $\text{fact}(40)$ is evaluated twice
- Call-by-value: in $\langle \text{dup}(\text{fact}(40)), s\rangle$, $\text{fact}(40)$ is evaluated once
- Call-by-name: $\langle \text{fst}(\text{fact } 0), s\rangle \longrightarrow^* \langle \text{fn } y : \text{unit} \Rightarrow (\text{fact } 0), s\rangle$
- Call-by-value: $\langle \text{fst}(\text{fact } 0), s\rangle \longrightarrow^* \langle \text{fn } y : \text{unit} \Rightarrow 1, s\rangle$

A third semantics: Full beta

Here, the reduction relation includes the CBV and the CBN relations, and also reduction inside function definitions.

$$\text{(BETA-app1)}\ \frac{\langle e_1, s\rangle \twoheadrightarrow \langle e_1', s'\rangle}{\langle e_1 e_2\ ,\ s\rangle \twoheadrightarrow \langle e_1' e_2\ ,\ s'\rangle}$$

$$\text{(BETA-app2)}\ \frac{\langle e_2, s\rangle \twoheadrightarrow \langle e_2', s'\rangle}{\langle e_1 e_2\ ,\ s\rangle \twoheadrightarrow \langle e_1 e_2'\ ,\ s'\rangle}$$

$$\text{(BETA-fn1)}\ \frac{-}{\langle (\mathsf{fn}\ x : T \Rightarrow e) e_2\ ,\ s\rangle \twoheadrightarrow \langle e\{^{e_2}/_x\}\ ,\ s\rangle}$$

$$\text{(BETA-fn2)}\ \frac{\langle e, s\rangle \twoheadrightarrow \langle e', s'\rangle}{\langle \mathsf{fn}\ x : T \Rightarrow e\ ,\ s\rangle \twoheadrightarrow \langle \mathsf{fn}\ x : T \Rightarrow e'\ ,\ s'\rangle}$$

- Full beta: $\langle \mathsf{fst}(\mathsf{fact}\ 0)\ ,\ s\rangle \twoheadrightarrow^* \langle \mathsf{fn}\ y : \mathsf{unit} \Rightarrow 0\ ,\ s\rangle$

Typing functions (1)

Up to now a type environment Γ gives the type of store locations. From now on, it must also provide assumptions on the type of variables used in functions: e.g.

$$\Gamma = \{\mathrm{l}_1 : \mathsf{intref}, \ \mathrm{x} : \mathsf{int}, \ \mathrm{y} : \mathsf{bool} \to \mathsf{int}\}$$

Thus, we extend the set *TypeEnv* of type environments as follows:

$$\mathit{TypeEnv} \stackrel{\text{def}}{=} \mathbb{L} \cup \mathbb{X} \rightharpoonup \mathrm{T}_{\mathrm{loc}} \cup \mathrm{T}$$

such that:

- $\forall l \in \mathsf{dom}(\Gamma).\, \Gamma(l) \in \mathrm{T}_{\mathrm{loc}}$
- $\forall x \in \mathsf{dom}(\Gamma).\, \Gamma(x) \in \mathrm{T}$

Notations: if $x \notin \mathsf{dom}(\Gamma)$, write $\Gamma, x : T$ for the partial function which maps x to T but otherwise is like Γ.

Typing functions (2)

Notice that with the introduction of functions there are more stuck configurations (e.g. $2\ \mathit{true}$, $\mathit{true}\ \mathrm{fn}\ \mathrm{x} : T \Rightarrow e$, etc).
Our type system will reject these configurations by means of the following rules:

$$(\text{var})\ \frac{-}{\Gamma \vdash x : T}\ \text{if } \Gamma(x) = T$$

gamma é un insieme di che ci ritorna il tipo delle locazioni x

c'é solo una x e ha un determinato tipo ← unione disgiuntiva

se é già presente un altra variabile con lo stesso nome faccio Alpha-conversion

$$(\text{fn})\ \frac{\overbrace{\Gamma, x : T}^{\text{unione disgiuntiva}} \vdash e : T'}{\Gamma \vdash \mathrm{fn}\ x : T \Rightarrow e : T \to T'}$$

se e va in T'

La funzione va da T in T'

Lo si desume dal tipaggio del corpo della funzione

devono avere stesso tipo

$$(\text{app})\ \frac{\Gamma \vdash e_1 : T \to T' \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 e_2 : T'}$$

Typing functions - Examples

$$
(\text{app})\ \dfrac{(\text{fn})\ \dfrac{(\text{op}+)\ \dfrac{(\text{var})\ \dfrac{-}{\mathrm{x}:\mathsf{int}\vdash \mathrm{x}:\mathsf{int}}\quad (\text{int})\ \dfrac{-}{\mathrm{x}:\mathsf{int}\vdash 2:\mathsf{int}}}{\mathrm{x}:\mathsf{int}\vdash \mathrm{x}+2:\mathsf{int}}}{\emptyset\vdash(\mathsf{fn}\ \mathrm{x}:\mathsf{int}\Rightarrow \mathrm{x}+2):\mathsf{int}\to\mathsf{int}}\qquad \nabla}{\emptyset\vdash(\mathsf{fn}\ \mathrm{x}:\mathsf{int}\Rightarrow \mathrm{x}+2)2:\mathsf{int}}
$$

where ∇ is

$$
(\text{int})\ \dfrac{-}{\emptyset\vdash 2:\mathsf{int}}
$$

Note that sometimes you may need to work up to alpha conversion:

$$
\mathsf{fn}\ \mathrm{x}:\mathsf{int}\Rightarrow \mathrm{x}+(\mathsf{fn}\ \mathrm{x}:\mathsf{bool}\Rightarrow \mathsf{if}\ \mathrm{x}\ \mathsf{then}\ 3\ \mathsf{else}\ 4)\mathit{true}
$$

It is always a good idea to start typing with all binders different from each other and from all free variables.

Manca

$$\dfrac{\dfrac{\emptyset, x : int \vdash x+2 : int}{\emptyset \vdash (fn\ x : int \Rightarrow x+2) : int \to 2}\,\text{(fn)} \qquad \dfrac{-}{\emptyset \vdash 2 : int}\,\text{(int)}}{\emptyset \vdash (fn\ x . int \Rightarrow x+2)\, 2 \ : T'}\,\text{(App)}$$

$$\Gamma' = \Gamma, x : int$$
$$\Gamma'' = \Gamma', y : bool$$

$$\dfrac{\Gamma''(y) = bool}{\Gamma'' \vdash y : bool} \qquad \dfrac{-}{\Gamma'' \vdash 3 : int} \qquad \dfrac{-}{\Gamma'' \vdash 4 : int}$$

$$\dfrac{\Gamma'' \vdash y : bool \qquad \Gamma'' \vdash 3 : int \qquad \Gamma'' \vdash 4 : int}{\overbrace{\Gamma', y, bool}^{\Gamma''} \vdash \text{if } y \text{ then } 3 \text{ else } 4 : int}$$

$$\dfrac{\Gamma', y, bool \vdash \text{if } y \text{ then } 3 \text{ else } 4 : int}{\Gamma' \vdash \text{fn } x : bool \to \text{if } x \text{ then } 3 \text{ else } 4 : bool \to int} \qquad \dfrac{-}{\Gamma \vdash true : bool}$$

$$(var)\ \dfrac{-}{\Gamma' \vdash x : int} \qquad \dfrac{\Gamma' \vdash \text{fn } x : bool \to \text{if } x \text{ then } 3 \text{ else } 4 : bool \to int \qquad \Gamma \vdash true : bool}{\Gamma' \vdash (\text{fn } x : bool \Rightarrow \text{if } x \text{ then } 3 \text{ else } 4)\ true}$$

$$\dfrac{\Gamma' \vdash x : int \qquad \Gamma' \vdash (\text{fn } x : bool \Rightarrow \text{if } x \text{ then } 3 \text{ else } 4)\ true}{\overbrace{\Gamma, x : int}^{\Gamma'} \vdash x + (\text{fn } \ldots\ldots)\ true : int}\ (+)$$

$$\dfrac{\Gamma, x : int \vdash x + (\text{fn } \ldots\ldots)\ true : int}{\Gamma \vdash \text{fn } x : int \Rightarrow x + (\text{fn}(x : bool) \Rightarrow \text{if } x \text{ then } 3 \text{ else } 4)\ true}\ (fn)$$

Typing functions - Example

$$
(\text{app})\ \dfrac{(\text{fn})\ \dfrac{(\text{seq})\ \dfrac{(\text{ass})\ \dfrac{(\text{int})\ \dfrac{-}{l : \text{intref}\,,\ x : \text{unit} \vdash 1 : \text{int}}}{l : \text{intref}\,,\ x : \text{unit} \vdash (l := 1) : \text{unit}} \quad \nabla_1}{l : \text{intref}\,,\ x : \text{unit} \vdash (l := 1); x : \text{unit}}}{l : \text{intref} \vdash (\text{fn}\ x : \text{unit} \Rightarrow (l := 1); x) : \text{unit} \rightarrow \text{unit}} \quad \nabla_2}{l : \text{intref}\ \vdash\ (\text{fn}\ x : \text{unit} \Rightarrow (l := 1); x)(l := 2)\ :\ \text{unit}}
$$

where ∇_1 is

$$
(\text{var})\ \dfrac{-}{l : \text{intref}\,,\ x : \text{unit} \vdash x : \text{unit}}
$$

and ∇_2 is

$$
(\text{ass})\ \dfrac{(\text{int})\ \dfrac{-}{l : \text{intref} \vdash 2 : \text{int}}}{l : \text{intref} \vdash (l := 2) : \text{unit}}
$$

$$\Gamma''(\ell) = \mathrm{intref}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\ }}{\Gamma'' \vdash 1 : \mathrm{int}}}{\Gamma'' \vdash (\ell := 1) : \mathrm{unit}} \quad \dfrac{\Gamma''(x) = \mathrm{unit}}{\Gamma'' \vdash x : \cancel{T}\ \mathrm{unit}}}{\{\ell : \mathrm{intref}\}, x : \mathrm{unit} \vdash (\ell := 1); x : \cancel{T}\ \mathrm{unit}}\,(\mathrm{seq})}{\{\ell : \mathrm{intref}\} \vdash \mathrm{fn}\ x : \mathrm{unit} \Rightarrow \ell := 1; x : \overset{\mathrm{unit}}{\cancel{T}} \to T}\,(\mathrm{fn}) \qquad \dfrac{\dfrac{\dfrac{}{\ }}{\{\ell : \mathrm{int}\} \vdash 2 : \mathrm{int}}\,(\mathrm{int}) \quad \{\ell : \mathrm{intref}\}(\ell) = \mathrm{intref}}{\{\ell : \mathrm{intref}\} \vdash (\ell := 2) : \cancel{T}\ \mathrm{unit}}\,(\mathrm{Assign})}{\{\ell : \mathrm{intref}\} \vdash (\mathrm{fn}\ x : \mathrm{unit} \Rightarrow (l := 1); x)(l := 2) : T}\,(\mathrm{App})$$

Properties of Typing

We only consider executions of closed programs, with no free variables.

Theorem 15 (Progress)

If e closed and $\Gamma \vdash e : T$ and $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$ then either e is a value or there exist e', s' such that $\langle e, s \rangle \rightarrow \langle e', s' \rangle$.

Theorem 16 (Type preservation)

If e is closed and $\Gamma \vdash e : T$ and $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$ and $\langle e, s \rangle \rightarrow \langle e', s' \rangle$ then $\Gamma \vdash e' : T$ and e' closed and $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s')$.

This requires:

Substitution lemma

If $\Gamma \vdash e : T$ and $\Gamma, x : T \vdash e' : T'$ with $x \notin \mathrm{dom}(\Gamma)$ then $\Gamma \vdash e'\{^{e}/_{x}\} : T'$.

Normalization

Theorem 18 (Normalization)

In the sublanguages without whileloops or store operations, if $\Gamma \vdash e : T$ and e closed then there is a value v such that, for any store s,

$$\langle e, s \rangle \longrightarrow^* \langle v, s \rangle \ .$$

Said in other terms if we consider a pure functional language, like the λ-calculus, its typed version is not turing-complete anymore!

Local declarations

For readability, we want to be able to *name* expressions, and to restrict their scope.

Duplicate evaluations:

$(1 + 2) \geq (1 + 2) + 4)$

This is a very common feature of many programming languages.

New construct:

$$\texttt{let}\ \mathrm{y} : \mathsf{int} = 1 + 2\ \texttt{in}\ \mathrm{y} \geq (\mathrm{y} + 4)$$

Intuition:

- First evaluate $1 + 2$ to 3
- Then evaluate $\mathrm{y} \geq (\mathrm{y} + 4)$ with y replaced by 3
- Here, y is a binder, binding any free occurrence of y in $y \geq (\mathrm{y} + 4)$.

Local declarations: syntax and typing

Let us extend the syntax of our expressions:

$$e \ ::= \ \ldots \ \mid \ \texttt{let}\ x : T = e\ \texttt{in}\ e$$

Let us provide the typing rule of the new contruct:

$$(\text{let})\ \frac{\Gamma \vdash e_1 : T \qquad \Gamma, x : T \vdash e_2 : T'}{\Gamma \vdash \texttt{let}\ x : T = e_1\ \texttt{in}\ e_2 : T'}$$

Note that, since x is a local variable, Γ does not contain an entry for variable x. This means that, as for “fn”, typing a “let” construct may require alpha-conversion.

Local declarations: intuition

In a let construct variables are placeholders standing for unknown quantities.

Problem 1: Many expressions are meaningless

- $\texttt{let}\ \mathrm{y} : T = 2 + 3\ \texttt{in}\ \mathrm{y} + \mathrm{z}$
- $\texttt{let}\ \mathrm{z} : T = 2 + \mathrm{x}\ \texttt{in}\ \mathrm{z} \times \mathrm{z}$

Problem 2: Variables may be used in multiple roles

- $\texttt{let}\ \mathrm{z} : T = 2 + \mathrm{z}\ \texttt{in}\ \mathrm{z} \times \mathrm{y}$

Problem 3: Multiple declarations for the same value

- $\texttt{let}\ \mathrm{y} : T = 1\ \texttt{in}\ \texttt{let}\ \mathrm{y} : T' = (1 + 2)\ \texttt{in}\ \mathrm{y} \times (\mathrm{y} + 4)$

shadowing

Local declarations: Free and bound variables

Intuition:

In $\texttt{let}\ y : T = 2 + 3\ \texttt{in}\ y + z$

- y is **bound** – stands for expression $2 + 3$
- z is **free** – does not stand for any expression

In $\texttt{let}\ z : T = 2 + x\ \texttt{in}\ z \times (z + y)$

- z is **bound** – stands for expression $2 + x$
- x and y are **free** – do not stand for any expression

In $\texttt{let}\ z : T = 2 + z\ \texttt{in}\ z \times (z + y)$

- z has **bound** occurrences (in blue) – stands for expression $2 + z$
- z has a **free** occurrence (in red) – does not stand for any expression.

Local declarations: Alpha conversion

As the let construct is a binder for the local variable, we can use alpha-conversion when necessary.

Convention: we will allow ourselves to any time, in any expression

$$\ldots(\texttt{let } x : T = e_1 \texttt{ in } e_2)\ldots$$

to replace the binding x and all occurrences of x in e_2 that are bound to that binder, by any other *fresh* variable that does not occur elsewhere:

- $(\texttt{let } \mathrm{x} : T = e_1 \texttt{ in } e_2) =_\alpha (\texttt{let } \mathrm{y} : T = e_1 \texttt{ in } e_2\{^{\mathrm{y}}/_{\mathrm{x}}\})$

where y is a fresh variable, i.e. it does not occur neither in e_1 or in e_2.

Local declarations: free variables and substitution

The definition of free variables for the "let" construct is as expected:

$$\mathsf{fv}(\texttt{let}\ \mathrm{x} : T = e_1\ \texttt{in}\ e_2) \stackrel{\text{def}}{=} \mathsf{fv}(e_1) \cup (\mathsf{fv}(e_2) \setminus \{\mathrm{x}\})$$

As regards substitution we must be careful as we may need to work up to alpha-conversion:

$$(\texttt{let}\ \mathrm{x}{:}T = e_1\ \texttt{in}\ e_2)\{^{e}/_{\mathrm{z}}\} \stackrel{\text{def}}{=} (\texttt{let}\ \mathrm{x}{:}T = e_1\{^{e}/_{\mathrm{z}}\}\ \texttt{in}\ e_2\{^{e}/_{\mathrm{z}}\}) \quad \text{if } \mathrm{x} \notin \mathsf{fv}(e)$$

$$(\texttt{let}\ \mathrm{x}{:}T = e_1\ \texttt{in}\ e_2)\{^{e}/_{\mathrm{z}}\} \stackrel{\text{def}}{=} (\texttt{let}\ \mathrm{y}{:}T = e_1\{^{e}/_{\mathrm{z}}\}\ \texttt{in}\ (e_2\{^{\mathrm{y}}/_{\mathrm{x}}\})\{^{e}/_{\mathrm{z}}\}) \quad \text{if } \mathrm{x} \in \mathsf{fv}(e) \wedge \mathrm{y} \text{ fresh}^2$$

$$(\texttt{let}\ \mathrm{x}{:}T = e_1\ \texttt{in}\ e_2)\{^{e}/_{\mathrm{x}}\} \stackrel{\text{def}}{=} (\texttt{let}\ \mathrm{x}{:}T = \underbrace{e_1\{^{e}/_{\mathrm{x}}\}}_{\text{uniche x libere}}\ \texttt{in}\ e_2)$$

Our definitions uses variables and not meta-variables: hence, $\mathrm{x} \neq \mathrm{z}$!

[2] Here, y fresh means $\mathrm{y} \notin \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \cup \{\mathrm{x}, \mathrm{z}\}$.

free

let $x = 3+y$ in $x+x$ $\qquad$ $fv = y$

let $x = 3+x$ in $x+x$ $\qquad$ $fv = x$

free

Local declarations: small-step semantics

As for function application we can have at least a couple of different semantics for the "let" construct.

Call-by-value semantics

$$(\text{CBV-let1})\ \frac{\langle e_1, s\rangle \twoheadrightarrow \langle e_1', s'\rangle}{\langle \texttt{let}\ x{:}T = e_1\ \texttt{in}\ e_2\,,\ s\rangle \twoheadrightarrow \langle \texttt{let}\ x{:}T = e_1'\ \texttt{in}\ e_2\,,\ s'\rangle}$$

$$(\text{CBV-let2})\ \frac{-}{\langle \texttt{let}\ x{:}T = v\ \texttt{in}\ e_2\,,\ s\rangle \twoheadrightarrow \langle e_2\{{}^{v}/_{x}\}\,,\ s\rangle}$$

Call-by-name semantics

$$(\text{CBN-let})\ \frac{-}{\langle \texttt{let}\ x{:}T = e_1\ \texttt{in}\ e_2\,,\ s\rangle \twoheadrightarrow \langle e_2\{{}^{e_1}/_{x}\}\,,\ s\rangle}$$

Sequential composition vs. Local declarations vs. Function application

As said before, the λ-calculus is very expressive and it can encode most of the constructs of sequential languages.

Here we show how the constructs for local declarations and sequential composition can be encoded in terms of function application.

The 'let' construct is syntactic sugar for:

$$\texttt{let}\ x{:}T = e_1\ \texttt{in}\ e_2 \quad \rightsquigarrow \quad (\mathsf{fn}\ x : T \Rightarrow e_2)e_1$$

Similarly, in a call-by-value semantics[3]:

$$e_1; e_2 \quad \rightsquigarrow \quad \texttt{let}\ x{:}\mathsf{unit} = e_1\ \texttt{in}\ e_2 \quad \rightsquigarrow \quad (\mathsf{fn}\ x : \mathsf{unit} \Rightarrow e_2)e_1$$

if x does not occur free in e_2.

[3] Does it work also in a CBN semantics?

Recursion

- How do we model recursive functions?
- Are there in our language for free?
- Due to the Normalization theorem we know that without *while* or *store* we cannot express infinite computations, and hence recursion as well
- Using "if", "while" and store we can implement recursion, but it would become a bit heavy to use
- Actually, we already did it. Did you notice it?
- Let us define a new operator to define recursive functions
- Once again, let'us take inspiration from the λ-calculus.

Fixpoints → un punto in cui la funzione vale quello stesso punto

- In Mathematics, a fixpoint p (also known as an invariant point) of a function f is a point that is mapped to itself, i.e. $f(p) = p$.
- Fixpoints represent the core of what is known as recursion theory.
- Kleene's recursion theorems are a pair of fundamental results about the application of computable functions to their own descriptions.
- The two recursion theorems can be applied to construct fixed points of certain operations on computable functions, to generate quines [4], and to construct functions defined via recursive definitions.
- Kleene's recursion theorems is used to prove a fundamental result in computability theory: the Rice's Theorem!
- Rice's Theorem: "For any non-trivial property of partial functions, there is no general and effective method to decide whether an algorithm computes a partial function with that property".

[4] A quine is a computer program which produces a copy of its own source code as its only output.

Fixpoints via Turing's combinator (1)

So, it is very important to prove that the λ-calculus can express fixpoints.

Let us use Turing's combinator to derive fixpoints:

$$\begin{array}{rcl} A & \stackrel{\text{def}}{=} & \lambda x.\lambda y.y(xxy) \\ \mathit{fix} & \stackrel{\text{def}}{=} & AA \end{array}$$

fix is a recursive function that given a term M returns a fixpoint of M, denoted with fix M.

In fact, for any term M, using a call-by-name evaluation, we have:

$$\begin{array}{rcl} \text{fix } M & \rightarrow & (\lambda y.y(AAy))M \\ & \rightarrow & M(\text{fix } M) \ . \end{array}$$

$$\underbrace{fix\ M = (AA)M \longleftrightarrow (\lambda y.y\,(AAy))M \longleftrightarrow M\,(fix\ M)}_{=}$$

$$M = (\lambda f.\lambda x.B)(\text{funzione}) \rightarrow \lambda x.B\,\{\text{funzione}/f\}$$

Fixpoints via Turing's combinator (2)

vuole avere i calcolatori più piccoli possibili

Now, if you choose M of the form $\lambda f.\lambda x.B$, for some body B, then, in a call-by-name semantics we have:

Il puntofisso lo vediamo come una chiamata ricorsiva

$$\begin{aligned} \text{fix } (\lambda f.\lambda x.B) &\rightarrow^* (\lambda f.\lambda x.B)(\text{fix } (\lambda f.\lambda x.B)) \\ &\rightarrow \lambda x.B\{^{\text{fix } (\lambda f.\lambda x.B)}/_f\} \end{aligned}$$

Vuole tirare fuori i punti fissi di un funzionale M

Recursive definitions:

$$M(\text{fix } M) = \text{fix } M$$

Thus, if we define

$$\boxed{\text{rec } f.B \text{ as an abbreviation for fix } (\lambda f.\lambda x.B)}$$

then we can rewrite the previous reduction as:

$$\text{rec } f.B \rightarrow^* \lambda x.B\{^{\text{rec } f.B}/_f\}$$

Example: the factorial in call-by-name semantics

$$\begin{array}{rcl} Fact & \stackrel{\text{def}}{=} & \lambda f.\lambda z.\text{if } z = 0 \text{ then } 1 \text{ else } z \times f(z-1) \\ & = & \lambda f.\lambda z.F \\ fact & \stackrel{\text{def}}{=} & fix\ Fact \\ & = & fix(\lambda f.\lambda z.F) \\ & = & \text{rec}\, f.F \end{array}$$

é un funzionale del fattoriale

il punto fisso di fact si comporta come il fattoriale

Then,

Sviluppo del fact

andrebbe valutato alla fine e bisogna valutare prima fact

$$\begin{array}{rcl} fact\ 3 & \rightarrow^* & (\lambda z.\text{if } z = 0 \text{ then } 1 \text{ else } z \times fact(z-1))\ 3 \\ & \rightarrow^* & 3 \times fact\ 2 \\ & \rightarrow^* & 3 \times (\lambda z.\text{if } z = 0 \text{ then } 1 \text{ else } z \times fact(z-1))\ 2 \\ & \rightarrow^* & 3 \times 2 \times fact\ 1 \\ & \rightarrow^* & 3 \times 2 \times 1 \times fact\ 0 \\ & \rightarrow^* & 3 \times 2 \times 1 \times 1 \\ & \rightarrow^* & 6 \end{array}$$

Fixpoints in the call-by-value semantics (1)

So, we found a way to express recursive functions in λ-calculus, according to the call-by-name semantics.

Does this mechanism work also in call-by-value semantics? Let us try:

$$
\begin{array}{lcl}
\mathrm{rec}\, f.B & \stackrel{\mathrm{def}}{=} & \mathrm{fix}\ (\lambda f.\lambda x.B) \\
 & \rightarrowtriangle^* & (\lambda f.\lambda x.B)(\mathrm{fix}\ (\lambda f.\lambda x.B)) \\
 & \rightarrowtriangle^* & (\lambda f.\lambda x.B)(\lambda f.\lambda x.B)(\mathrm{fix}\ (\lambda f.\lambda x.B)) \\
 & \rightarrowtriangle^* & (\lambda f.\lambda x.B)(\lambda f.\lambda x.B)(\lambda f.\lambda x.B)(\mathrm{fix}\ (\lambda f.\lambda x.B)) \\
 & \rightarrowtriangle^* & \ldots\ldots\ \text{forever!}
\end{array}
$$

Solution: We need to stop the indefinite unfolding of fix $(\lambda f.\lambda x.B)$.

Fixpoints in the call-by-value semantics (2)

Question: What is the difference between

$$M \qquad \text{and} \qquad \lambda z.Mz$$

where z not free in M?

Answer: Not much!

And what about:

$$\text{fix } (\lambda f.\lambda x.B) \qquad \text{vs} \qquad \lambda z.\big(\text{fix } (\lambda f.\lambda x.B)\big)z$$

The second one is a value (a function) the first one is not!

Fixpoints in the call-by-value semantics (3)

Let us redefine the fixpoint combinators!

Call-by-name combinator

$A \stackrel{\text{def}}{=} \lambda x.\lambda y.y(xxy)$ $\quad$ $fix \stackrel{\text{def}}{=} AA$ $\quad$ $fix\, M \rightarrow^* M(fix\, M)$

Call-by-value combinator

$A_v \stackrel{\text{def}}{=} \lambda x.\lambda y.y(\lambda z.(xxy)z)$ $\quad$ $fix_v \stackrel{\text{def}}{=} A_v A_v$ $\quad$ $fix_v\, M \rightarrow^* M(\lambda z.(fix_v\, M)z)$

Call-by-value recursive functions

Let $\mathrm{rec_v}\, f.B$ be an abbreviation for $fix_v\,(\lambda f.\lambda x.B)$, then

$$\mathrm{rec_v}\, f.B \;\rightarrow^*\; \lambda x.B\{^{\lambda z.(\mathrm{rec_v}\, f.B)z}/_f\}.$$

It does not diverge anymore!

Some fun: Klop's fixpoint combinator

Jan Klop came up with this *ridiculous* one: if we define

$$L \stackrel{\text{def}}{=} \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$$

where

- $\lambda abcdef \ldots$ means $\lambda a.\lambda b.\lambda c.\lambda d.\lambda e.\lambda f \ldots$.
- *thisisafixe*... means $((((((((((th)i)s)i)s)a)f)i)x)e)$....

then

$$LLLLLLLLLLLLLLLLLLLLLLLLLL \quad \text{(26 times)}$$

is indeed a fixpoint combinator.

Exercise

Check that Klop's combinator works. Hint: the phrase "this is a fixed point combinator" contains 27 letters.

Back again to our language

- Do we adopt one of the previous fixpoint combinators?
- No, none of the previous combinators is well-typed. Remember the Normalization theorem...

Syntax

$e \ ::= \ \ldots \ \mid \ \mathsf{fix}.e$

In un linguaggio con store bisogna aggiungerlo e il punto fisso resta invariato.

Lo store?
È una computazione funzionale in genere non c'è lo store perché serve all'interno del corpo

Typing

$$\text{(T-Fix)} \ \frac{\Gamma \vdash e : (T_1 \to T_2) \to (T_1 \to T_2)}{\Gamma \vdash \mathsf{fix}.e : T_1 \to T_2}$$

Semantics:

non funziona in modalità cbv

devo farla stoppata in cbv

$$\text{(Fix-cbn)} \ \frac{-}{\mathsf{fix}.e \twoheadrightarrow e(\mathsf{fix}.e)} \qquad \text{(Fix-cbv)} \ \frac{e \equiv \mathsf{fn}\ f{:}T_1 \to T_2 \ \Rightarrow \ e_2}{\mathsf{fix}.e \twoheadrightarrow e(\mathsf{fn}\ x{:}T_1 \Rightarrow (\mathsf{fix}.e)\ x)}$$

Factorial in a CBN semantics

For simplicity, we omit types:

$$Fact \stackrel{\text{def}}{=} \text{fn } f.\text{fn } z.\text{if } z = 0 \text{ then } 1 \text{ else } z \times f(z-1)$$
$$fact \stackrel{\text{def}}{=} \text{fix}.Fact$$

$$fact = \text{fix}.Fact \twoheadrightarrow Fact(\text{fix}.Fact) \twoheadrightarrow^* \text{fn } z.\text{if } z{=}0 \text{ then } 1 \text{ else } z \times fact(z{-}1)$$

$$\begin{aligned}
fact\,3 &\twoheadrightarrow^* (\text{fn } z.\text{if } z = 0 \text{ then } 1 \text{ else } z \times fact(z-1))\,3\\
&\twoheadrightarrow^* 3 \times fact\,2\\
&\twoheadrightarrow^* 3 \times (\text{fn } z.\text{if } z = 0 \text{ then } 1 \text{ else } z \times fact(z-1))\,2\\
&\twoheadrightarrow^* 3 \times 2 \times fact\,1\\
&\twoheadrightarrow^* 3 \times 2 \times 1 \times fact\,0\\
&\twoheadrightarrow^* 3 \times 2 \times 1 \times 1\\
&\twoheadrightarrow^* 6
\end{aligned}$$

Rule (Fix-cnb) does not work in call-by-value semantics

If we use rule (Fix-cbn) in a call-by-value semantics we have:

$$\begin{array}{rcl} \mathit{fact}\ 1 & \twoheadrightarrow & \mathit{Fact}(\mathit{fact})\ 1 \\ & \twoheadrightarrow & \mathit{Fact}(\mathit{Fact}(\mathit{fact}))\ 1 \\ & \twoheadrightarrow & \ldots\ldots \\ & \twoheadrightarrow & \mathit{Fact}(\mathit{Fact}(\mathit{Fact}(\ldots \mathit{fact})))\ 1 \end{array}$$

- Call-by-value recursion needs a mechanism for stopping evaluation of next iteration
- That's why we defined a different rule, (Fix-cbv), to be used in call-by-value semantics.
- Exercise. Prove that, in a CBV semantics, using rule (Fix-cbv), we have $\mathrm{fix}.\mathit{Fact}\ 3 \twoheadrightarrow^* 6$.

Encoding while

Having recursion as a primitive we can remove the while construct if we wish so!

Let

chiamata ricorsiva ci metterò il punto fisso del while

$$W \stackrel{\text{def}}{=} \text{fn } w{:}\text{unit} \to \text{unit. fn } y{:}\text{unit. if } e_1 \text{ then } (e_2; (w\, \mathit{skip})) \text{ else skip}$$

for w and y not in $\text{fv}(e_1) \cup \text{fv}(e_2)$.

Then,

$$\text{while } e_1 \text{ do } e_2 \;\rightsquigarrow\; \text{fix}.W\, \mathit{skip}$$

and also the while operator would not be primitive anymore.