Chapter 2
The Language PCF

We will illustrate the various styles of semantics of programming languages with an example: the language PCF—*Programming language for computable functions*—, also called Mini-ML.

2.1 A Functional Language: PCF

2.1.1 Programs Are Functions

We observed in the previous chapter that a deterministic program computes a function, and from this observation we derived the principles of denotational semantics. This remark is also the basis of a class of programming languages: functional languages, such as Caml, Haskell or Lisp, which are traditionally used to begin the study of programming languages.

In these languages, the goal is to shorten the distance between the notion of a program and the notion of a mathematical function. In other words, the idea is to bring programs closer to their denotational semantics.

The basic constructions in the language PCF are the *explicit construction* of a function, written `fun x -> t`, and the *application* of a function to an argument, written `t u`.

PCF includes also a constant for each natural number, the operations `+`, `-`, `*`, `/`, and a test to detect zero `ifz t then u else v`. Addition and multiplication are defined for all natural numbers, and similarly for subtraction using the convention `n - m = 0` if `n < m`. Division is the standard Euclidean division, division by `0` produces an error.

2.1.2 Functions Are First-Class Objects

In many programming languages, it is possible to define a function that takes another function as argument, or that returns another function, but often this requires the use

G. Dowek, J.-J. Lévy, *Introduction to the Theory of Programming Languages*, Undergraduate Topics in Computer Science,
DOI 10.1007/978-0-85729-076-2_2, © Springer-Verlag London Limited 2011

of a syntax that is different from the syntax used for a standard argument such as an integer or a string. In a functional language, functions are defined in the same way whether they take numbers or functions as arguments.

For example, the composition of a function with itself is defined by `fun f -> fun x -> f (f x)`.

To highlight the fact that functions are not considered different, and thus they can be used as arguments or returned as results for other functions, we say that functions are *first class objects*.

2.1.3 Functions with Several Arguments

In PCF, there is no symbol to build a function with several arguments. These functions are built as functions with one argument, using the isomorphism `(A × B) -> C = A -> (B -> C)`. For instance, the function that associates to `x` and `y` the number `x * x + y * y` is defined as the function associating to `x` a function, which in turn associates to `y` the number `x * x + y * y`, that is, `fun x -> fun y -> x * x + y * y`.

Then, to apply the function `f` to the numbers `3` and `4` we need to apply it first to `3`, obtaining the term `f 3`, which represents the function that associates `3 * 3 + y * y` to `y`, and then to `4`, obtaining the term `(f 3) 4`. Since, by convention, application associates to the left, we will write this term simply as `f 3 4`.

2.1.4 No Assignments

In contrast with languages such as Caml or Java, the main feature of PCF is a total lack of *assignments*. There is no construction of the form `x := t` or `x = t` to assign a value to a "variable". We will describe, in Chap. 7, an extension of PCF with assignments.

2.1.5 Recursive Definitions

In Mathematics, some functions cannot be defined explicitly. For example, in a high-school textbook, the power function is often defined by

$$x,\ n \mapsto \underbrace{x \times \cdots \times x}_{n \text{ times}}$$

or through a definition by induction.

In programming languages, we use similar constructs: iterations and recursive definitions. PCF includes a special construct to define recursive functions.

It is often said that a function is recursive if the function is used in its own definition. This is absurd: in programming languages, as everywhere else, circular definitions are meaningless. We cannot "define" the function `fact` by `fun n -> ifz n then 1 else n * (fact (n - 1))`. In general, we cannot define a function `f` by a term `G` which contains an occurrence of `f`. However, we can define the function `f` as the fixed point of the function `fun f -> G`. For example, we can define the function `fact` as the fixed point of the function `fun f -> fun n -> ifz n then 1 else n * (f (n - 1))`.

Does this function have a fixed point? and if it does, is this fixed point unique? Otherwise, which fixed point are we referring to? We will leave these questions for a moment, and simply state that a recursive function is defined as a fixed point.

In PCF, the symbol `fix` binds a variable in its argument, and the term `fix f G` denotes the fixed point of the function `fun f -> G`. The function `fact` can then be defined by `fix f fun n -> ifz n then 1 else n * (f (n - 1))`.

Note, again, that using the symbol `fix` we can build the factorial function without necessarily giving it a name.

2.1.6 Definitions

We could, in theory, omit definitions and replace everywhere the defined symbols by their definitions. However, programs are simpler and clearer if we use definitions.

We add then a final construct in PCF, written `let x = t in u`. The occurrences of the variable `x` in `u` are bound, but those in `t` are not. The symbol `let` is a binary operator that binds a variable in its second argument.

2.1.7 The Language PCF

The language PCF contains

- a symbol `fun` with one argument, that binds a variable in its argument,
- a symbol α with two arguments, which does not bind any variables in its arguments,
- an infinite number of constants to represent the natural numbers,
- four symbols `+`, `-`, `*` and `/` with two arguments, which do not bind any variables in their arguments,
- a symbol `ifz` with three arguments, which does not bind any variables in its arguments,
- a symbol `fix` with one argument, which binds a variable in its argument,
- a symbol `let` with two arguments, which binds a variable in its second argument.

In other words, the syntax of PCF is inductively defined by

```
t = x
  | fun x -> t
  | t t
  | n
  | t + t | t - t | t * t | t / t
  | ifz t then t else t
  | fix x t
  | let x = t in t
```

Despite its small size, PCF is Turing complete, that is, all computable functions can be programmed in PCF.

Exercise 2.1 Write a PCF program that takes two natural numbers `n` and `p` as inputs and returns n^p.

Exercise 2.2 Write a PCF program that takes a natural number `n` as input and returns the number `1` if the input is a prime number, and `0` otherwise.

Exercise 2.3 (Polynomials in PCF) Write a PCF program that takes a natural number `q` as input, and returns the greatest natural number `u` such that `u (u + 1) / 2` $\leq$ `q`.

Cantor's function `K` is a function from $\mathbb{N}^2$ to $\mathbb{N}$ defined by `fun n -> fun p -> (n + p) (n + p + 1) / 2 + n`. Let `K'` be the function from $\mathbb{N}$ to $\mathbb{N}^2$ defined by `fun q -> (q - (u (u + 1) / 2), u - q + u (u + 1) / 2)` where `u` is the greatest natural number such that `u (u + 1) / 2` $\leq$ `q`.

Show that `K ∘ K' = id`. Let `n` and `p` be two natural numbers, show that the greatest natural number `u` such that `u (u + 1) / 2` $\leq$ `(n + p) (n + p + 1) / 2 + n` is `n + p`. Then deduce that `K' ∘ K = id`. From this fact, deduce that `K` is a bijection from $\mathbb{N}^2$ to $\mathbb{N}$.

Let `L` be the function `fun n -> fun p -> (K n p) + 1`. A polynomial with integer coefficients $a_0 + a_1 X + \cdots + a_i X^i + \cdots + a_n X^n$ can be represented by the integer `L` a_0 `(L` a_1 `(L` a_2 `... (L` a_n `0) ...))`.

Write a PCF program that takes two natural numbers as input and returns the value of the polynomial represented by the first number applied to the second.

2.2 Small-Step Operational Semantics for PCF

2.2.1 Rules

Let us apply the program `fun x -> 2 * x` to the constant `3`. We obtain the term `(fun x -> 2 * x) 3`. According to the principles of small-step operational semantics, let us try to evaluate this term step by step, to obtain a result: `6` if all

goes well. The first step in this simplification process is *parameter passing*, that is, the replacement of the formal argument `x` by the actual argument `3`. The initial term becomes, after a first small-step transformation, the term `2 * 3`. In the second step, the term `2 * 3` is evaluated, resulting in the number `6`. The first small step, parameter passing, can be performed each time we have a term of the form `(fun x -> t) u` where a function `fun x -> t` is applied to an argument `u`. As a consequence, we define the following rule, called *β-reduction rule*

$$\texttt{(fun x -> t) u} \longrightarrow \texttt{(u/x)t}$$

The relation `t` $\longrightarrow$ `u` should be read "`t` reduces—or rewrites—to `u`". The second step mentioned above can be generalised as follows

$$\texttt{p} \otimes \texttt{q} \longrightarrow \texttt{n} \,(\text{if } \texttt{p} \otimes \texttt{q} = \texttt{n})$$

where $\otimes$ is any of the four arithmetic operators included in PCF. We add similar rules for conditionals

$$\texttt{ifz 0 then t else u} \longrightarrow \texttt{t}$$

$$\texttt{ifz n then t else u} \longrightarrow \texttt{u} \,(\text{if } \texttt{n} \text{ is a number different from } \texttt{0})$$

a rule for fixed points

$$\texttt{fix x t} \longrightarrow \texttt{(fix x t/x)t}$$

and a rule for `let`

$$\texttt{let x = t in u} \longrightarrow \texttt{(t/x)u}$$

A *redex* is a term `t` that can be reduced. In other words, a term `t` is a redex if there exists a term `u` such that `t` $\longrightarrow$ `u`.

2.2.2 Numbers

It could be said, quite rightly, that the rule `p` $\otimes$ `q` $\longrightarrow$ `n` (if `p` $\otimes$ `q = n`), of which `2 * 3` $\longrightarrow$ `6` is an instance, does not really explain the semantics of the arithmetic operators, since it just replaces the multiplication in PCF by that of Mathematics. This choice is however motivated by the fact that we are not really interested in the semantics of arithmetic operators, instead, our goal is to highlight the semantics of the other constructs in the language.

To define the semantics of the arithmetic operators in PCF without referring to the mathematical operators, we should consider a variant of PCF without numeric constants, where we introduce just one constant for the number `0` and a symbol `S`—"successor"—with one argument. The number `3`, for instance, is represented by the term `S(S(S(0)))`. We then add small-step rules

`0 + u` $\longrightarrow$ `u`
`S(t) + u` $\longrightarrow$ `S(t + u)`
`0 - u` $\longrightarrow$ `0`

```
t - 0 ⟶ t
S(t) - S(u) ⟶ t - u
0 * u ⟶ 0
S(t) * u ⟶ t * u + u
t / S(u) ⟶ ifz t - u then 0 else S((t - S(u)) / S(u))
```

Note that, to be precise, we should add a rule for division by `0`, which should raise an exception: `error`.

Exercise 2.4 (Church numerals) Instead of introducing the symbols `0` and `S`, we can represent the number `n` by the term `fun z -> fun s -> s (s (... (s z)...))` rather than `S(S(...(0)...))`. Show that addition and multiplication can be programmed on these representations. Show that the function that checks whether a number is `0` can also be programmed.

Exercise 2.5 (Position numerals) It could be said that the representations of numbers using the symbols `0` and `S`, or using Church numerals, are not efficient, since the size of the term representing a number grows linearly with the number—as the representation in unary notation, where to write the number `n` we need `n` symbols—and not logarithmically, as it is the case with the usual position-based notation. An alternative could be to use a symbol `z` for the number `0` and two functions `O` and `I` to represent the functions `n` $\mapsto$ `2 * n` and `n` $\mapsto$ `2 * n + 1`. The number `26` would then be represented by the term `O(I(O(I(I(z)))))`, and reversing it we obtain `IIOIO`, the binary representation of this number.

Write a small-step operational semantics for the arithmetic operators in this language.

2.2.3 A Congruence

Using the rules of the small-step semantics we obtain

```
(fun x -> 2 * x) 3 ⟶ 2 * 3 ⟶ 6
```

Thus, denoting by $\longrightarrow^*$ the reflexive-transitive closure of $\longrightarrow$, we can write `(fun x -> 2 * x) 3` $\longrightarrow^*$ `6`.

However, with this definition, the term `(2 + 3) + 4` does not reduce to the term `9` according to $\longrightarrow^*$. Indeed, to reduce a term of the form `t + u` the terms `t` and `u` should be numeric constants, but our first term `2 + 3` is a sum, not a constant. The first step should then be the evaluation of `2 + 3`, which produces the number `5`. Then, a second step reduces `5 + 4` to `9`. The problem is that, with our definition, the term `2 + 3` reduces to `5`, but `(2 + 3) + 4` does not reduce to `5 + 4`.

We need to define another relation, where rules can be applied to any subterm of a term to be reduced. Let us define inductively the relation $\triangleright$ as follows

$$\frac{}{\texttt{t} \triangleright \texttt{u}} \text{ if } \texttt{t} \longrightarrow \texttt{u}$$

$$\frac{t \triangleright u}{t\ v \triangleright u\ v} \qquad \frac{t \triangleright u}{v\ t \triangleright v\ u}$$

$$\frac{t \triangleright u}{\texttt{fun x -> } t \triangleright \texttt{fun x -> } u} \qquad \frac{t \triangleright u}{t + v \triangleright u + v}$$

...

It is possible to show that a term is a redex with respect to the relation $\triangleright$ if and only if one of its subterms is a redex with respect to $\longrightarrow$.

2.2.4 An Example

To illustrate PCF's small-step semantic rules, let us compute the factorial of `3`.

```
(fix f fun n -> ifz n then 1 else n * (f (n - 1))) 3
▷ (fun n -> ifz n then 1 else n * ((fix f fun n ->
ifz n then 1 else n * (f (n - 1))) (n - 1))) 3
▷ ifz 3 then 1 else 3 * ((fix f fun n -> ifz n then 1
else n * (f (n - 1))) (3 - 1))
▷ 3 * ((fix f fun n -> ifz n then 1 else
n * (f (n - 1))) (3 - 1))
▷ 3 * ((fix f fun n -> ifz n then 1 else
n * (f (n - 1))) 2)
▷ 3 * ((fun n -> ifz n then 1 else n * ((fix f fun n ->
ifz n then 1 else n * (f (n - 1))) (n - 1))) 2)
▷ 3 * (ifz 2 then 1 else 2 * ((fix f fun n -> ifz n
then 1 else n * (f (n - 1))) (2 - 1)))
▷ 3 * (2 * ((fix f fun n -> ifz n then 1 else
n * (f (n - 1))) (2 - 1)))
▷ 3 * (2 * ((fix f fun n -> ifz n then 1 else
n * (f (n - 1))) 1))
▷ 3 * (2 * ((fun n -> ifz n then 1 else
n * ((fix f fun n -> ifz n then 1 else
n * (f (n - 1))) (n - 1))) 1))
▷ 3 * (2 * (ifz 1 then 1 else 1 * ((fix f fun n ->
ifz n then 1 else n * (f (n - 1))) (1 - 1))))
▷ 3 * (2 * (1 * ((fix f fun n -> ifz n then 1 else
n * (f (n - 1))) (1 - 1))))
```

```
▷ 3 * (2 * (1 * ((fix f fun n -> ifz n then 1
else n * (f (n - 1))) 0)))
▷ 3 * (2 * (1 * ((fun n -> ifz n then 1 else
n * ((fix f fun n -> ifz n then 1 else
n * (f (n - 1))) (n - 1))) 0)))
▷ 3 * (2 * (1 * ((ifz 0 then 1 else
0 * ((fix f fun n -> ifz n then 1 else
n * (f (n - 1))) (0 - 1))))))
▷ 3 * (2 * (1 * 1)) ▷ 3 * (2 * 1) ▷ 3 * 2 ▷ 6
```

2.2.5 Irreducible Closed Terms

A term t is *irreducible* if it cannot be reduced by ▷, that is, if there is no term u such that t ▷ u.

We can now define the relation "the term u is the result of the evaluation of term t", where t is a closed term, by: t ↪ u if and only if t $\triangleright^*$ u and u is irreducible. In this case, the term u must be closed. Finally, the relation "the program p with inputs e_1, ..., e_n produces the output s" is simply written p e_1 ... e_n ↪ s.

Exercise 2.6 (Classification of irreducible closed terms) Show that a term is irreducible and closed if and only if it is of one of the following forms

- fun x -> t where t is irreducible and does not contain any free variables except possibly x,
- n where n is a number,
- V_1 V_2, where V_1 and V_2 are irreducible closed terms and V_1 is not of the form fun x -> t,
- $V_1 \otimes V_2$, where V_1 and V_2 are irreducible closed terms and are not both numeric constants,
- ifz V_1 then V_2 else V_3 where V_1, V_2 and V_3 are irreducible closed terms and V_1 is not a number.

Numbers and irreducible closed terms of the form fun x -> t are called *values*. When the result of a computation is a value, we associate the value to the initial term, and we say that the term *evaluates* to this value.

Unfortunately, values are not the only possible results. For example, the term (fun x -> x) 1 2 can be reduced to the term 1 2, which is irreducible and closed, and thus the term 1 2 is the result of the computation of (fun x -> x) 1 2. This result is meaningless, because we cannot apply the object 1, which is not a function, to 2. An irreducible closed term that is not a value is said to be *stuck*. Stuck terms have the form V_1 V_2, where V_1 and V_2 are irreducible closed terms and V_1 is not a function fun x -> t (for example 1 2), $V_1 \otimes V_2$, where V_1 and V_2 are irreducible and closed and are not numbers (for example 1 +

`(fun x -> x)`), and `ifz` V_1 `then` V_2 `else` V_3 where V_1, V_2 and V_3 are irreducible and closed and V_1 is not a number (for example, `ifz (fun x -> x) then 1 else 2`).

Exercise 2.7 Which are the values associated to the terms

```
(fun x -> fun x -> x) 2 3
```

and

```
(fun x -> fun y -> ((fun x -> (x + y)) x)) 5 4
```

according to the small-step operational semantics of PCF?

Exercise 2.8 (Static binding) Does the small-step operational semantics of PCF associate the value `10` or the value `11` to the term

```
let x = 4 in let f = fun y -> y + x
in let x = 5 in f 6?
```

The first versions of the language Lisp produced the value `11` instead of `10` for this term. In this case, we say that the binding is *dynamic*.

2.2.6 Non-termination

It is easy to see that the relation $\hookrightarrow$ is not total, that is, there are terms `t` for which there is no term `u` such that `t` $\hookrightarrow$ `u`. For example, the term `b = fix x x` reduces to itself, and only to itself. It does not reduce to any irreducible term.

Exercise 2.9 Let b_1 `= (fix f (fun x -> (f x))) 0`. Show all the terms obtained by reducing this term. Does the computation produce a result in this case?

Exercise 2.10 (Curry's fixed point operator) Let `t` be a term and `u` be the term `(fun y -> (t (y y)))(fun y -> (t (y y)))`. Show that `u` reduces to `t u`.

Let `t` be a term and `v` be the term `(fun y -> ((fun x -> t) (y y)))(fun y -> ((fun x -> t) (y y)))`. Show that `v` reduces to `(v/x)t`.

Thus, we can deduce that the symbol `fix` is superfluous in PCF. However, it is not going to be superfluous later when we add types to PCF.

Write a term `u` without using the symbol `fix` and equivalent to `b = fix x x`. Describe the terms that can be obtained by reduction. Does the computation produce a result in this case?

2.2.7 Confluence

Is it possible for a closed term to produce several results? And, in general, can a term reduce to several different irreducible terms? The answer to these questions is negative. In fact, every PCF program is deterministic, but this is not a trivial property. Let us see why.

The term `(3 + 4) + (5 + 6)` has two subterms which are both redexes. We could then start by reducing `3 + 4` to `7` or `5 + 6` to `11`. Indeed, the term `(3 + 4) + (5 + 6)` reduces to both `7 + (5 + 6)` and `(3 + 4) + 11`. Fortunately, neither of these terms is irreducible, and if we continue the computation we reach in both cases the term `18`.

To prove that any term can be reduced to at most one irreducible term we need to prove that if two computations originating in the same term produce different terms, then they will eventually reach the same irreducible term.

This property is a consequence of another property of the relation $\triangleright$: *confluence*. A relation `R` is confluent if each time we have `a` $\mathtt{R}^*$ $\mathtt{b}_1$ and `a` $\mathtt{R}^*$ $\mathtt{b}_2$, there exists some `c` such that $\mathtt{b}_1$ $\mathtt{R}^*$ `c` and $\mathtt{b}_2$ $\mathtt{R}^*$ `c`.

It is not difficult to show that confluence implies that each term has at most one irreducible result. If the term `t` can be reduced to two irreducible terms $\mathtt{u}_1$ and $\mathtt{u}_2$, then we have `t` $\triangleright^*$ $\mathtt{u}_1$ and `t` $\triangleright^*$ $\mathtt{u}_2$. Since $\triangleright$ is confluent, there exists a term `v` such that $\mathtt{u}_1$ $\triangleright^*$ `v` and $\mathtt{u}_2$ $\triangleright^*$ `v`. Since $\mathtt{u}_1$ is irreducible, the only term `v` such that $\mathtt{u}_1$ $\triangleright^*$ `v` is $\mathtt{u}_1$ itself. Therefore, $\mathtt{u}_1$ `=` `v` and similarly $\mathtt{u}_2$ `=` `v`. We conclude that $\mathtt{u}_1$ `=` $\mathtt{u}_2$. In other words, `t` reduces to at most one irreducible term.

We will not give here the proof of confluence for the relation $\triangleright$. The idea is that when a term `t` contains two redexes $\mathtt{r}_1$ and $\mathtt{r}_2$, and $\mathtt{t}_1$ is obtained by reducing $\mathtt{r}_1$ and $\mathtt{t}_2$ is obtained by reducing $\mathtt{r}_2$, then we can find the residuals of $\mathtt{r}_2$ in $\mathtt{t}_1$ and reduce them. Similarly, we can reduce the residuals of $\mathtt{r}_1$ in $\mathtt{t}_2$, obtaining the same term. For example, by reducing `5 + 6` in `7 + (5 + 6)` and reducing `3 + 4` in `(3 + 4) + 11`, we obtain the same term: `7 + 11`.

2.3 Reduction Strategies

2.3.1 The Notion of a Strategy

Since in PCF each term has at most one result (due to the unicity property mentioned above), it does not matter in which order we reduce the redexes in a term: if we reach an irreducible term, it will always be the same. However, it may be the case that one sequence of reduction reaches an irreducible term whereas another one does not. For example, let `C` be the term `fun x -> 0` and let $\mathtt{b}_1$ be the term `(fix f (fun x -> (f x))) 0`. The term $\mathtt{b}_1$ reduces to $\mathtt{b}_2$ = `(fun x -> (fix f (fun x -> (f x)) x)) 0` and then again to $\mathtt{b}_1$. The term `C` $\mathtt{b}_1$ contains several redexes, and it can be reduced to `0` and to `C` $\mathtt{b}_2$ which in turn contains several redexes and can be reduced to `0` and `C` $\mathtt{b}_1$ (amongst

other terms). By reducing always the innermost redex, we can build an infinite reduction sequence $\texttt{C b}_1 \rhd \texttt{C b}_2 \rhd \texttt{C b}_1 \rhd \cdots$, whereas reducing the outermost redex produces the result `0`.

This example may seem an exception, because it contains a function `C` that does not use its argument; but note that the `ifz` construct is similar, and in the example of the factorial function, when computing the factorial of `3` for instance, we can observe the same behaviour: The term `ifz 0 then 1 else 0 * ((fix f fun n -> ifz n then 1 else n * (f (n - 1))) (0 - 1))` has several redexes. Outermost reduction produces the result `1` (the other redexes disappear), whereas reducing the redex `fix f fun n -> ifz n then 1 else n * (f (n - 1))` we get an infinite reduction sequence. In other words, the term `fact 3` can be reduced to `6`, but it can also generate reductions that go on forever.

Both $\texttt{C b}_1$ and `fact 3` produce a unique result, but not all reduction sequences reach a result.

Since the term $\texttt{C b}_1$ has the value `0` according to the PCF semantics, an *evaluator*, that is, a program that takes as input a PCF term and returns its value, should produce the result `0` when computing $\texttt{C b}_1$. Let us try to evaluate this term using some current compilers. In Caml, the program

```
let rec f x = f x in let g x = 0 in g (f 0)
```

does not terminate. In Java, we have the same problem with the program

```
class Omega {
 static int f (int x) {return f(x);}
 static int g (int x) {return 0;}
 static public void main (String [ ] args) {
  System.out.println(g(f(0)));}}
```

Only a small number of compilers, using *call by name* or *lazy* evaluation, such as Haskell, Lazy-ML or Gaml, produce a terminating program for this term.

This is because the small-step semantics of PCF does not correspond to the semantics of Caml or Java. In fact, it is too general and when a term has several redexes it does not specify which one should be reduced first. By default, it imposes termination of all programs that somehow can produce a result. An ingredient is missing in this semantic definition: the notion of a strategy, that specifies the order of reduction of redexes.

A *strategy* is a partial function that associates to each term in its domain one of its redex occurrences. Given a strategy `s`, we can define another semantics, replacing the relation $\rhd$ by a new relation $\rhd_{\texttt{s}}$ such that $\texttt{t} \rhd_{\texttt{s}} \texttt{u}$ if $\texttt{s t}$ is defined and `u` is obtained by reducing the redex $\texttt{s t}$ in `t`. Then, we define the relation $\rhd_{\texttt{s}}^*$ as the reflexive-transitive closure of $\rhd_{\texttt{s}}$, and the relation $\hookrightarrow_{\texttt{s}}$ as before.

Instead of defining a strategy, an alternative would be to weaken the reduction rules, in particular the congruence rules, so that only some specific reductions can be performed.

2.3.2 Weak Reduction

Before defining outermost or innermost strategies for the term `C b1`, let us give another example to show that the operational semantics defined above is too liberal, and to motivate the definition of strategies or weaker reduction rules. Let us apply the program `fun x -> x + (4 + 5)` to the constant `3`. We obtain the term `(fun x -> x + (4 + 5)) 3` that contains two redexes. We can then reduce it to `3 + (4 + 5)` or to `(fun x -> x + 9) 3`. The first reduction is part of the execution of the program, but not the second. Usually, if we execute a function before passing arguments to it, we say that we are *optimising* or *specialising* the program.

A *weak reduction strategy* never reduces a redex that is under a `fun`. Thus, weak reduction does not specialise programs, it just executes them. It follows that with a weak strategy all terms of the form `fun x -> t` are irreducible.

Alternatively, we can define weak reduction by weakening the reduction rules, more precisely, by discarding the congruence rule

$$\frac{\texttt{t} \rhd \texttt{u}}{\texttt{fun x -> t} \rhd \texttt{fun x -> u}}$$

Exercise 2.11 (Classification of weak irreducible closed terms) Show that, under weak reduction, a closed irreducible term must have one of the following forms:

- `fun x -> t`, where `t` has at most `x` free,
- `n` where `n` is a number,
- $\texttt{V}_1\ \texttt{V}_2$, where $\texttt{V}_1$ and $\texttt{V}_2$ are irreducible closed terms and $\texttt{V}_1$ is not a term of the form `fun x -> t`,
- $\texttt{V}_1 \otimes \texttt{V}_2$, where $\texttt{V}_1$ and $\texttt{V}_2$ are irreducible closed terms and are not both numbers,
- `ifz` $\texttt{V}_1$ `then` $\texttt{V}_2$ `else` $\texttt{V}_3$ where $\texttt{V}_1$, $\texttt{V}_2$ and $\texttt{V}_3$ are irreducible closed terms and $\texttt{V}_1$ is not a number.

What is the difference with Exercise 2.6?

Numbers and closed terms of the form `fun x -> t` are called *values*.

2.3.3 Call by Name

Let us analyse again the reductions available for the term `C b1`. We need to decide whether we should evaluate the arguments of the function `C` before they are passed to the function, or we should pass to the function the arguments without evaluating them.

The *call by name* strategy always reduces the leftmost redex first, and the weak call by name strategy always reduces the leftmost redex that is not under a `fun`. Thus, the term `C b1` reduces to `0`. This strategy is interesting due to the following

property, called standardisation: if a term can be reduced to an irreducible term, then the call by name strategy terminates. In other words, $\hookrightarrow_{\mathtt{n}} = \hookrightarrow$. Moreover, when we evaluate the term `(fun x -> 0) (fact 10)` using a call by name strategy, we do not need to compute the factorial of `10`. However, if we evaluate the term `(fun x -> x + x) (fact 10)`, using a call by name strategy, we will compute it twice, because this term reduces to `(fact 10) + (fact 10)`. Most call by name evaluators use sharing to avoid this duplication of computation, and in this case we call it *lazy* evaluation.

2.3.4 Call by Value

Call by value, in contrast, always evaluates the arguments of a function before passing them to the function. It is based on the following convention: we can only reduce a term of the form `(fun x -> t) u` if `u` is a value. Thus, when we evaluate the term `(fun x -> x + x) (fact 10)`, we start by reducing the argument to obtain `(fun x -> x + x) 3628800`, and then we reduce the leftmost redex. By doing this, we only compute the factorial of `10` once.

All the strategies that evaluate arguments before passing them are in this class. For instance, the strategy that reduces always the leftmost redex amongst those that are authorised. Thus, call by value is not a unique strategy, but a family of strategies.

This convention can also be defined by weakening the β-reduction rule: the term `(fun x -> t) u` is a redex only if the term `u` is a value.

A weak strategy is said to implement call by value if it reduces a term of the form `(fun x -> t) u` only when `u` is a value and is not under a `fun`.

2.3.5 A Bit of Laziness Is Needed

Even under a call by value strategy, a conditional construct `ifz` must be evaluated under call by name: in a term of the form `ifz t then u else v`, we should never evaluate the three arguments. Instead, we should first evaluate `t` and depending on the result, evaluate either `u` or `v`.

It is easy to see that if we evaluate the three arguments of an `ifz` then the evaluation of the term `fact 3` does not terminate.

Exercise 2.12 Characterise the irreducible closed terms under weak call by name, then characterise the irreducible closed terms under weak call by value.

2.4 Big-Step Operational Semantics for PCF

Instead of defining a strategy, or weakening the reduction rules of the small-step operational semantics, we can control the order in which redexes are reduced by defining a *big-step* operational semantics.

The big-step operational semantics of a programming language provides an inductive definition of the relation $\hookrightarrow$, without first defining $\longrightarrow$ and $\triangleright$.

2.4.1 Call by Name

Let us start by the call by name semantics for PCF. Consider a term of the form `t u` that is reduced under call by name to obtain an irreducible term `V`. We will start by reducing the redexes that occur in `t` until we obtain an irreducible term. If this term is of the form `fun x -> t'`, then the whole term reduces to `(fun x -> t') u` and the left-most redex is the term itself. It reduces to `(u/x)t'`, which in turn reduces to `V`. We can say that the term `t u` reduces under call by name to the irreducible term `V` if `t` reduces to `fun x -> t'` and `(u/x)t'` reduces to `V`.

This can be expressed as a rule

$$\frac{\texttt{t} \hookrightarrow \texttt{fun x -> t'} \quad \texttt{(u/x)t'} \hookrightarrow \texttt{V}}{\texttt{t u} \hookrightarrow \texttt{V}}$$

which will be part of the inductive definition of the relation $\hookrightarrow$ (without first defining $\longrightarrow$ and $\triangleright$).

Other rules state that the result of the computation for a term of the form `fun` is the term itself, that is, we are defining a weak reduction relation

$$\frac{}{\texttt{fun x -> t} \hookrightarrow \texttt{fun x -> t}}$$

and that the result of the computation of a term of the form `n` is the term itself

$$\frac{}{\texttt{n} \hookrightarrow \texttt{n}}$$

Also, there is a rule to give the semantics of arithmetic operators

$$\frac{\texttt{u} \hookrightarrow \texttt{q} \quad \texttt{t} \hookrightarrow \texttt{p}}{\texttt{t} \otimes \texttt{u} \hookrightarrow \texttt{n}} \text{ if } \texttt{p} \otimes \texttt{q} = \texttt{n}$$

two rules to define the semantics of the `ifz` construct

$$\frac{\texttt{t} \hookrightarrow \texttt{0} \quad \texttt{u} \hookrightarrow \texttt{V}}{\texttt{ifz t then u else v} \hookrightarrow \texttt{V}}$$

$$\frac{\texttt{t} \hookrightarrow \texttt{n} \quad \texttt{v} \hookrightarrow \texttt{V}}{\texttt{ifz t then u else v} \hookrightarrow \texttt{V}} \text{ if n is a number} \neq \texttt{0}$$

a rule to define the semantics of the fixed point operator

$$\frac{\texttt{(fix x t/x)t} \hookrightarrow \texttt{V}}{\texttt{fix x t} \hookrightarrow \texttt{V}}$$

and finally a rule to define the semantics of a `let`

$$\frac{(t/x)u \hookrightarrow V}{\texttt{let x = t in u} \hookrightarrow V}$$

We can prove by structural induction on the evaluation relation that the result of the computation of a term is always a value, that is, a number or a closed term of the form `fun`. There are no stuck terms. The computation of a term such as `((fun x -> x) 1) 2`, which gave rise to the term `1 2` (stuck) with the small-step semantics, does not produce a result with the big-step semantics, since none of the rules can be applied to this term. Indeed, there is no rule in the big-step semantics that explains how to evaluate an application where the left part evaluates to a number.

2.4.2 Call by Value

The rules defining the call by value semantics are similar, except for the application rule: we compute the value of the argument before passing it to the function

$$\frac{u \hookrightarrow W \quad t \hookrightarrow \texttt{fun x -> t'} \quad (W/x)t' \hookrightarrow V}{\texttt{t u} \hookrightarrow V}$$

and the `let` rule

$$\frac{t \hookrightarrow W \quad (W/x)u \hookrightarrow V}{\texttt{let x = t in u} \hookrightarrow V}$$

Summarising, we have the following rules

$$\frac{u \hookrightarrow W \quad t \hookrightarrow \texttt{fun x -> t'} \quad (W/x)t' \hookrightarrow V}{\texttt{t u} \hookrightarrow V}$$

$$\frac{}{\texttt{fun x -> t} \hookrightarrow \texttt{fun x -> t}}$$

$$\frac{}{\texttt{n} \hookrightarrow \texttt{n}}$$

$$\frac{u \hookrightarrow q \quad t \hookrightarrow p}{t \otimes u \hookrightarrow n} \text{ if } p \otimes q = n$$

$$\frac{t \hookrightarrow 0 \quad u \hookrightarrow V}{\texttt{ifz t then u else v} \hookrightarrow V}$$

$$\frac{t \hookrightarrow n \quad v \hookrightarrow V}{\texttt{ifz t then u else v} \hookrightarrow V} \text{ if n is a constant} \neq 0$$

$$\frac{\texttt{(fix x t/x)t} \hookrightarrow \texttt{V}}{\texttt{fix x t} \hookrightarrow \texttt{V}}$$

$$\frac{\texttt{t} \hookrightarrow \texttt{W} \quad \texttt{(W/x)u} \hookrightarrow \texttt{V}}{\texttt{let x = t in u} \hookrightarrow \texttt{V}}$$

Notice that, even under call by value, we keep the rules for the `ifz`

$$\frac{\texttt{t} \hookrightarrow \texttt{0} \quad \texttt{u} \hookrightarrow \texttt{V}}{\texttt{ifz t then u else v} \hookrightarrow \texttt{V}}$$

$$\frac{\texttt{t} \hookrightarrow \texttt{n} \quad \texttt{v} \hookrightarrow \texttt{V}}{\texttt{ifz t then u else v} \hookrightarrow \texttt{V}} \quad \text{if n is a constant} \neq \texttt{0}$$

that is, we do not evaluate the second and third arguments of an `ifz` until they are needed.

Note also that, even under call by value, we keep the rule

$$\frac{\texttt{(fix x t/x)t} \hookrightarrow \texttt{V}}{\texttt{fix x t} \hookrightarrow \texttt{V}}$$

We must resist the temptation to evaluate the term `fix x t` to a value `W` before substituting it in `t`, because the rule

$$\frac{\texttt{fix x t} \hookrightarrow \texttt{W} \quad \texttt{(W/x)t} \hookrightarrow \texttt{V}}{\texttt{fix x t} \hookrightarrow \texttt{V}}$$

requires, in order to evaluate `fix x t`, to start by evaluating `fix x t` which would create a loop and the term `fact 3` would never produce a value—its evaluation would give rise to an infinite computation.

Note finally that other rule combinations are possible. For example, some variants of the call by name semantics use call by value in the `let` rule.

Exercise 2.13 Which values do we obtain under big-step semantics for the terms

```
(fun x -> fun x -> x) 2 3
```

and

```
(fun x -> fun y -> ((fun x -> (x + y)) x)) 5 4?
```

Compare your answer with that of Exercise 2.7.

Exercise 2.14 Does the big-step semantics associate the value `10` or the value `11` to the term

```
let x = 4 in let f = fun y -> y + x
in let x = 5 in f 6?
```

Compare your answer with that of Exercise 2.8.

2.5 Evaluation of PCF Programs

A PCF evaluator is a program that takes a closed PCF term as input, and produces its value as output. When read in a bottom-up fashion, the rules in the big-step semantics can be seen as the kernel of such an evaluator: To evaluate an application `t u` one starts by evaluating `u` and `t`, ... this is easy to program in a language like Caml

```
let rec eval p = match p with
| App(t,u) -> let w = eval u
              in let v = eval t
              in ...
| ...
```

In the case of an application, the rules of the big-step semantics leave us the freedom to evaluate `u` first or `t` first—call by value is not a strategy, but a family of strategies—, but the term `(W/x)t'` must be the third to be evaluated, because it is built out of the results of the first two evaluations.

Exercise 2.15 Write a call by name evaluator for PCF, that is, a program that takes as input a closed term and computes its value. Write a call by value evaluator. Evaluate the term `fact 6` and the term `C` b_1 in both cases.

PCF's denotational semantics is more difficult to define. This may seem a paradox, since PCF is a functional language and it should be easy to interpret its programs as functions. However, in PCF, any object can be applied to any object, and nothing stops us writing for instance the term `fun x -> (x x)`. In contrast with mathematical functions, PCF functions do not have a domain. For this reasons, we will give a denotational semantics for PCF after we add types, in Chap. 5.