Roberto Bruni, Ugo Montanari

Models of Computation

– Monograph –

March 17, 2016

Springer

Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity.

Alan Turing[1]

[1] The purpose of ordinal logics (from Systems of Logic Based on Ordinals), Proceedings of the London Mathematical Society, series 2, vol. 45, 1939.

Contents

Acronyms

$\sim$	operational equivalence in IMP (see Definition 3.3)
$\equiv_{den}$	denotational equivalence in HOFL (see Definition 10.4)
$\equiv_{op}$	operational equivalence in HOFL (see Definition 10.3)
$\simeq$	CCS strong bisimilarity (see Definition 11.5)
$\approx$	CCS weak bisimilarity (see Definition 11.16)
$\cong$	CCS weak observational congruence (see Section 11.7.2)
$\approx_d$	CCS dynamic bisimilarity (see Definition 11.17)
$\overset{\circ}{\sim}_E$	π-calculus early bisimilarity (see Definition 13.3)
$\overset{\circ}{\sim}_L$	π-calculus late bisimilarity (see Definition 13.4)
$\sim_E$	π-calculus strong early full bisimilarity (see Section 13.5.3)
$\sim_L$	π-calculus strong late full bisimilarity (see Section 13.5.3)
$\overset{\bullet}{\approx}_E$	π-calculus weak early bisimilarity (see Section 13.5.4)
$\overset{\bullet}{\approx}_L$	π-calculus weak late bisimilarity (see Section 13.5.4)
$\mathscr{A}$	interpretation function for the denotational semantics of IMP arithmetic expressions (see Section 6.2.1)
ack	Ackermann function (see Example 4.18)
Aexp	set of IMP arithmetic expressions (see Chapter 3)
$\mathscr{B}$	interpretation function for the denotational semantics of IMP boolean expressions (see Section 6.2.2)
Bexp	set of IMP boolean expressions (see Chapter 3)
$\mathbb{B}$	set of booleans
$\mathscr{C}$	interpretation function for the denotational semantics of IMP commands (see Section 6.2.3)
CCS	Calculus of Communicating Systems (see Chapter 11)
Com	set of IMP commands (see Chapter 3)
CPO	Complete Partial Order (see Definition 5.11)
$\text{CPO}_\perp$	Complete Partial Order with bottom (see Definition 5.12)
CSP	Communicating Sequential Processes (see Section 16.2)
CTL	Computation Tree Logic (see Section 12.1.2)
CTMC	Continuous Time Markov Chain (see Definition 14.15)

DTMC	Discrete Time Markov Chain (see Definition 14.14)
Env	set of HOFL environments (see Chapter 9)
fix	(least) fixpoint (see Definition 5.2.2)
FIX	(greatest) fixpoint
gcd	greatest common divisor
HML	Hennessy-Milner modal Logic (see Section 11.5)
HM-Logic	Hennessy-Milner modal Logic (see Section 11.5)
HOFL	A Higher-Order Functional Language (see Chapter 7)
IMP	A simple IMPerative language (see Chapter 3)
int	integer type in HOFL (see Definition 7.2)
Loc	set of locations (see Chapter 3)
LTL	Linear Temporal Logic (see Section 12.1.1)
LTS	Labelled Transition System (see Definition 11.2)
lub	least upper bound (see Definition 5.7)
$\mathbb{N}$	set of natural numbers
$\mathscr{P}$	set of closed CCS processes (see Definition 11.1)
PEPA	Performance Evaluation Process Algebra (see Chapter 16)
Pf	set of partial functions on natural numbers (see Example 5.13)
PI	set of partial injective functions on natural numbers (see Problem 5.12)
PO	Partial Order (see Definition 5.1)
PTS	Probabilistic Transition System (see Section 14.3.2)
$\mathbb{R}$	set of real numbers
$\mathscr{T}$	set of HOFL types (see Definition 7.2)
Tf	set of total functions from $\mathbb{N}$ to $\mathbb{N}_\perp$ (see Example 5.14)
Var	set of HOFL variables (see Chapter 7)
$\mathbb{Z}$	set of integers

Chapter 6
Denotational Semantics of IMP

The point is that, mathematically speaking, functions are independent of their means of computation and hence are "simpler" than the explicitly generated, step-by-step evolved sequences of operations on representations. (Dana Scott)

Abstract In this chapter we give a more abstract, purely mathematical semantics to IMP, called *denotational semantics*. The operational semantics is close to the memory-based, executable machine-like view: given a program and a state, we derive the state obtained after the execution of that program. The denotational semantics takes a program and returns the transformation function over memories associated with that program: given an initial state as argument the final state is returned as a result. Since functions will be written in some fixed mathematical notation, i.e., they can also be regarded as "programs" of a suitable formalism, we can say that, to some extent, the operational semantics defines an "interpreter" of the language (given a program *and* the initial state it returns the final state obtained by executing the program), while the denotational semantics defines a "compiler" for the language (from programs to functions, i.e., programs written in a more abstract language). We conclude the chapter by reconciling the equivalences induced by the operational and the denotational semantics and by stating the principle of computational induction.

6.1 λ-Notation

In the following we shall rely on λ*-notation* as a *(meta-)language* for writing anonymous functions. When considering HOFL, then λ-notation will be used both at the level of the programming language and at the level of the denotational semantics, as meta-language.

The λ-calculus was introduced by Alonzo Church (1903-1995) in order to answer one of the questions posed by David Hilbert (1862–1943) in his program, known as Entscheidungsproblem (German for *decision problem*). Roughly, the problem consisted in the existence of an algorithm to decide whether a given statement of a first-order logic (possibly enriched with a finite number of axioms) is deducible or not from the axioms of logic. Alan Turing (1912-1954) proved that no effectively calculable algorithm can exist that solves the problem, where "calculable" meant

computable by a Turing machine. Independently, Alonzo Church answered negatively assuming that "calculable" meant a function expressible in the λ-calculus.

6.1.1 λ-Notation: Main Ideas

The λ-calculus is built around the idea of expressing a calculus of functions, where it is not necessary to assign names to functions, i.e., where functions can be expressed anonymously. Conceptually, this amounts to have the possibility of:

- forming (anonymous) functions by *abstraction* over names in an expression; and
- *applying* a function to an argument

Building on the two basic considerations above, Church developed a theory of functions based on rules for computation, as opposed to the classical set-theoretic view of functions as sets of pairs (argument, result).

Example 6.1. Let us start with a simple example from arithmetic. Take a polynomial such as

$$x^2 - 2x + 5.$$

What is the value of the above expression when x is replaced by 2? We compute the result by plugging in '2' for 'x' in the expression to get

$$2^2 - 2 \times 2 + 5 = 5.$$

In λ-notation, when we want to express that the value of an expression depends on some value to be plugged in, we use abstraction. Syntactically, this corresponds to prefix the expression by the special symbol λ and the name of the formal parameter, as, e.g., in:

$$\lambda x.\ (x^2 - 2x + 5)$$

The informal reading is:

> wait for a value v to replace x and then compute $v^2 - 2v + 5$.

We want to be able to pass some actual parameter to the function above, i.e., to apply the function to some value v. To this aim, we denote application by juxtaposition:

$$(\lambda x.\ (x^2 - 2x + 5))\ 2$$

means that the function $(\lambda x.\ (x^2 - 2x + 5))$ is applied to 2 (i.e., that the actual parameter 2 must replace the occurrences of the formal parameter x in $x^2 - 2x + 5$, to obtain $2^2 - 2 \times 2 + 5 = 5$.)

Note that:

- by writing $\lambda x.\ t$ we are declaring x as a formal parameter appearing in t;
- the symbol λ has no particular meaning (any other symbol could have been used);

- we say that λx 'binds' the (occurrences of the) variable x in t;
- the scope of the formal parameter x is just t; if x occurs also "outside" t, then it refers to another (homonymous) identifier.

Example 6.2. Let us consider another example:

$$(\lambda x.\ \lambda y.\ (x^2 - 2y + 5))\ 2$$

This time we have a function that is waiting for two arguments (first x, then y), but to which we pass one value (2). We have

$$(\lambda x.\ \lambda y.\ (x^2 - 2y + 5))\ 2 = \lambda y.\ (2^2 - 2y + 5) = \lambda y.\ (9 - 2y)$$

that is, the result of applying $\lambda x.\ \lambda y.\ (x^2 - 2y + 5)$ to 2 is still a function $(\lambda y.\ (9 - 2y))$.

In λ-calculus we can pass functions as arguments and return functions as results.

Example 6.3. Take the term $\lambda f.\ (f\ 2)$: it waits for a function f that will be applied to the value 2. If we pass the function $(\lambda x.\ \lambda y.\ (x^2 - 2y + 5))$ to $\lambda f.\ (f\ 2)$, written:

$$(\lambda f.\ (f\ 2))\ (\lambda x.\ \lambda y.\ (x^2 - 2y + 5))$$

then we get the function $\lambda y.\ (9 - 2y)$ as a result.

Definition 6.1 (Lambda terms). We define *lambda terms* as the terms generated by the grammar:

$$t \quad ::= \quad x \quad | \quad \lambda x.t \quad | \quad (t_0\ t_1) \quad | \quad t \to (t_0, t_1)$$

Where x is a variable.

As we can see the lambda notation is very simple, it has four constructs:

- x: is a simple *variable*.
- $\lambda x.t$: is the *lambda abstraction* which allows to define anonymous functions.
- $t_0\ t_1$: is the *application* of a function t_0 to its argument t_1.
- $t \to t_0, t_1$ is the conditional operator, i.e. the "if-then-else" construct in lambda notation.

Note that we omit some parentheses when no ambiguity can arise.

Lambda abstraction $\lambda x.t$ is the main feature. It allows to define functions, where x represents the parameter of the function and t is the lambda term which represents the body of the function. For example the term $\lambda x.x$ is the identity function.

Note that while we can have different terms t and t' that define the same function. Church proved that the problem of deciding whether $t = t'$ is undecidable.

Definition 6.2 (Conditional expressions). Let t, t_0 and t_1 be three lambda terms, we define:

$$t \to t_0, t_1 = \begin{cases} t_0 & \text{if } t = true \\ t_1 & \text{if } t = false \end{cases}$$

All the notions used in this definition, like "true" and "false" can be formalised in lambda notation only, by using lambda abstraction, as shown in Section 6.1.1.1 for the interested reader. In the following we will take the liberty to assume that data types such as integers and booleans are available in the lambda-notation as well as the usual operations on them.

Remark 6.1 (Associativity of abstraction and application). In the following, to limit the number of parentheses and keep the notation more readable, we assume that application is left-associative, and lambda-abstraction is right-associative, i.e.,

$$\begin{array}{ll} t_1\ t_2\ t_3\ t_4 & \text{is read as } (((t_1\ t_2)\ t_3)\ t_4) \\ \lambda x_1.\ \lambda x_2.\ \lambda x_3.\ \lambda x_4.\ t & \text{is read as } \lambda x_1.\ (\lambda x_2.\ (\lambda x_3.\ (\lambda x_4.\ t))) \end{array}$$

Remark 6.2 (Precedence of application). We will also assume that application has precedence over abstraction, i.e.:

$$\lambda x.\ t\ t' = \lambda x.\ (t\ t')$$

6.1.1.1 λ-Notation: Booleans and Church Numerals

In the above examples, we have enriched standard arithmetic expressions with abstraction and application. In general, it would be possible to encode booleans and numbers (and operations over them) just using abstraction and application.

For example, let us consider the following terms:

$$T \stackrel{\text{def}}{=} \lambda x.\ \lambda y.\ x$$
$$F \stackrel{\text{def}}{=} \lambda x.\ \lambda y.\ y$$

We can assume that T represents true and F represents false.

Under this convention, we can define the usual logical operations by letting:

$$\text{AND} \stackrel{\text{def}}{=} \lambda p.\ \lambda q.\ p\ q\ p$$
$$\text{OR} \stackrel{\text{def}}{=} \lambda p.\ \lambda q.\ p\ p\ q$$
$$\text{NOT} \stackrel{\text{def}}{=} \lambda p.\ \lambda x.\ \lambda y.\ p\ y\ x$$

Now suppose that P will reduce either to T or to F. The expression $P\ A\ B$ can be read as 'if P then A else B'.

For natural numbers, we can adopt the convention that the number n is represented by a function that takes a function f and an argument x and applies f to x for n times consecutively. For example:

$$
\begin{aligned}
0 &\stackrel{\text{def}}{=} \lambda f.\ \lambda x.\ x \\
1 &\stackrel{\text{def}}{=} \lambda f.\ \lambda x.\ f\ x \\
2 &\stackrel{\text{def}}{=} \lambda f.\ \lambda x.\ f\ (f\ x) \\
&\dots
\end{aligned}
$$

Then, the operations for successor, sum, multiplication can be defined by letting:

$$
\begin{aligned}
\text{SUCC} &\stackrel{\text{def}}{=} \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x) \\
\text{SUM} &\stackrel{\text{def}}{=} \lambda n.\ \lambda m.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x) \\
\text{MUL} &\stackrel{\text{def}}{=} \lambda n.\ \lambda m.\ \lambda f.\ n\ (m\ f)
\end{aligned}
$$

6.1.2 Alpha-Conversion, Beta-Rule and Capture-Avoiding Substitution

The names of the formal parameters we choose for a given function should not matter. Therefore, any two expressions that differ just for the particular choice of λ-abstracted variables and have the same structure otherwise, should be considered as equal.

For example, we do not want to distinguish between the terms

$$\lambda x.\ (x^2 - 2x + 5) \qquad \lambda y.\ (y^2 - 2y + 5)$$

On the other hand, the expressions

$$x^2 - 2x + 5 \qquad y^2 - 2y + 5$$

must be distinguished, because depending on the context where they are used, the symbols x and y could have a different meaning.

We say that two terms are α-convertible if one is obtained from the other by renaming some λ-abstracted variables. We call *free* the variables x whose occurrences are not under the scope of a λ binder.

Definition 6.3 (Free variables). The set of free variables occurring in a term is defined by structural recursion:

$$
\begin{aligned}
\text{fv}(x) &\stackrel{\text{def}}{=} \{x\} \\
\text{fv}(\lambda x.t) &\stackrel{\text{def}}{=} \text{fv}(t) \setminus \{x\} \\
\text{fv}(t_0\ t_1) &\stackrel{\text{def}}{=} \text{fv}(t_0) \cup \text{fv}(t_1) \\
\text{fv}(t \to t_0, t_1) &\stackrel{\text{def}}{=} \text{fv}(t) \cup \text{fv}(t_0) \cup \text{fv}(t_1)
\end{aligned}
$$

The second equation highlights that the lambda abstraction is a binding operator.

Definition 6.4 (Alpha-conversion). We define *α-conversion* as the equivalence induced by letting

$$\lambda x.\, t = \lambda y.\, (t[y/x]) \quad \text{if } y \notin \mathrm{fv}(t)$$

where $t[y/x]$ denotes the substitution of x with y applied to the term t.

Note the side condition $y \notin \mathrm{fv}(t)$, which is needed to avoid 'capturing' other free variables appearing in t.

For example:

$$\lambda z.\, z^2 - 2y + 5 = \lambda x.\, x^2 - 2y + 5 \neq \lambda y.\, y^2 - 2y + 5$$

We have now all ingredients to define the basic computational rule, called β-rule, which explains how to apply a function to an argument:

Definition 6.5 (Beta-rule). Let t, t' be two lambda terms we define:

$$(\lambda x.t')\ t = t'[t/x]$$

this axiom is called β-rule.

In defining alpha-conversion and the beta-rule we have used substitutions like $[y/x]$ and $[t/x]$. Let us now try to formalise the notion of substitution by structural recursion. What is wrong with the following naive attempt?

$$y[t/x] \stackrel{\text{def}}{=} \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases}$$

$$(\lambda y.t')[t/x] \stackrel{\text{def}}{=} \begin{cases} \lambda y.t' & \text{if } y = x \\ \lambda y.(t'[t/x]) & \text{if } y \neq x \end{cases}$$

$$(t_0\ t_1)[t/x] \stackrel{\text{def}}{=} (t_0[t/x])\ (t_1[t/x])$$

$$(t' \to t_0, t_1)[t/x] \stackrel{\text{def}}{=} (t'[t/x]) \to (t_0[t/x]), (t_1[t/x])$$

Example 6.4 (Substitution, without alpha-renaming). Consider the terms

$$t \stackrel{\text{def}}{=} \lambda x.\, \lambda y.\, (x^2 - 2y + 5) \qquad t' \stackrel{\text{def}}{=} y.$$

and apply t to t':

$$\begin{aligned} t\ t' &= (\lambda x.\, \lambda y.\, (x^2 - 2y + 5))\ y \\ &= (\lambda y.\, (x^2 - 2y + 5))[y/x] \\ &= \lambda y.\, ((x^2 - 2y + 5)[y/x]) \\ &= \lambda y.\, (y^2 - 2y + 5) \end{aligned}$$

It happens that the free variable $y \in \mathrm{fv}(t\ t')$ has been 'captured' by the lambda-abstraction λy. Instead, free variables occurring in t should remain free during the application of the substitution $[t/x]$.

Thus we need to correct the above version of substitution for the case related to $(\lambda y.t')[t/x]$ by applying first the alpha-conversion to $\lambda y.t'$ (to make sure that if $y \in \mathrm{fv}(t)$, then the free occurrences of y in t will not be captured by λy when replacing x in t') and then the substitution $[t/x]$. Formally, we let:

Definition 6.6 (Capture-avoiding substitution). Let t, t', t_0 and t_1 be four lambda terms, we define:

$$\begin{aligned} y[t/x] &= \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases} \\ (\lambda y.t')[t/x] &= \lambda z.((t'[z/y])[t/x]) \quad \text{if } z \notin \mathrm{fv}(\lambda y.\, t') \cup \mathrm{fv}(t) \cup \{x\} \\ (t_0\ t_1)[t/x] &= (t_0[t/x])\ (t_1[t/x]) \\ (t' \to t_0, t_1)[t/x] &= (t'[t/x]) \to (t_0[t/x]), (t_1[t/x]) \end{aligned}$$

Note that the matter of names is not so trivial. In the second equation we first rename y in t' with a fresh name z, then proceed with the substitution of x with t. As explained, this solution is motivated by the fact that y might not be free in t, but it introduces some non-determinism in the equations due to the arbitrary nature of the new name z. This non-determinism immediately disappear if we regard the terms up to the alpha-conversion equivalence, as previously introduced. Obviously α-conversion and substitution should be defined at the same time to avoid circularity. By using the α-conversion we can prove statements like $\lambda x.x = \lambda y.y$.

Example 6.5 (Application with alpha-renaming). Consider the terms t, t' from Example 6.4:

$$\begin{aligned} t\ t' &= (\lambda x.\ \lambda y.\ (x^2 - 2y + 5))\ y \\ &= (\lambda y.\ (x^2 - 2y + 5))[y/x] \\ &= \lambda z.\ ((x^2 - 2y + 5)[z/y][y/x]) \\ &= \lambda z.\ ((x^2 - 2z + 5)[y/x]) \\ &= \lambda z.\ (y^2 - 2z + 5) \end{aligned}$$

Finally we introduce some notational conventions for omitting parentheses when defining the domains and codomains of functions:

$$\begin{aligned} A \to B \times C &= A \to (B \times C) & \qquad A \times B \times C &= (A \times B) \times C \\ A \times B \to C &= (A \times B) \to C & \qquad A \to B \to C &= A \to (B \to C) \end{aligned}$$

6.2 Denotational Semantics of IMP

As we said we will use lambda notation as meta-language; this means that we will express the semantics of IMP by translating IMP syntax to lambda terms.

The denotational semantics of IMP consists of three separate *interpretation* functions, one for each syntax category ($Aexp, Bexp, Com$):

Aexp: each arithmetic expression is mapped to a function from states to integers:

$$\mathscr{A} : Aexp \to (\Sigma \to \mathbb{Z})$$

Bexp: each boolean expression is mapped to a function from states to booleans:

$$\mathscr{B} : Bexp \to (\Sigma \to \mathbb{B})$$

Com: each command is mapped to a (partial) function from states to states:

$$\mathscr{C} : Com \to (\Sigma \rightharpoonup \Sigma)$$

6.2.1 Denotational Semantics of Arithmetic Expressions: The Function $\mathscr{A}$

The denotational semantics of arithmetic expressions is defined as the function:

$$\mathscr{A} : Aexp \to \Sigma \to \mathbb{Z}$$

We shall define $\mathscr{A}$ by structural recursion over the syntax of arithmetic expressions. Let us fix some notation: We will rely on definitions of the form

$$\mathscr{A}[\![n]\!] \stackrel{\text{def}}{=} \lambda\sigma.n$$

with the following meaning:

- $\mathscr{A} : Aexp \to \Sigma \to \mathbb{Z}$ is the interpretation function,
- n is an arithmetic expression (i.e., a term in $Aexp$). The surrounding brackets $[\![$ and $]\!]$ emphasise that it is a piece of syntax rather then part of the metalanguage.
- the expression $\mathscr{A}[\![n]\!]$ is a function whose type is $\Sigma \to \mathbb{Z}$. Notice that also the right part of the equation must be of the same type $\Sigma \to \mathbb{Z}$.

We shall often define the interpretation function $\mathscr{A}$ by writing equalities such as:

$$\mathscr{A}[\![n]\!]\sigma \stackrel{\text{def}}{=} n$$

instead of

$$\mathscr{A}[\![n]\!] \stackrel{\text{def}}{=} \lambda\sigma.n$$

In this way, we simplify the notation in the right-hand side. Notice that both sides of the equation ($\mathscr{A}[\![n]\!]\sigma$ and n) have the type $\mathbb{Z}$.

Definition 6.7 (Denotational semantics of arithmetic expressions). The denotational semantics of arithmetic expressions is defined by structural recursion as:

$$\begin{aligned}
\mathscr{A}[\![n]\!]\sigma &\stackrel{\text{def}}{=} n \\
\mathscr{A}[\![x]\!]\sigma &\stackrel{\text{def}}{=} \sigma x \\
\mathscr{A}[\![a_0 + a_1]\!]\sigma &\stackrel{\text{def}}{=} (\mathscr{A}[\![a_0]\!]\sigma) + (\mathscr{A}[\![a_1]\!]\sigma) \\
\mathscr{A}[\![a_0 - a_1]\!]\sigma &\stackrel{\text{def}}{=} (\mathscr{A}[\![a_0]\!]\sigma) - (\mathscr{A}[\![a_1]\!]\sigma) \\
\mathscr{A}[\![a_0 \times a_1]\!]\sigma &\stackrel{\text{def}}{=} (\mathscr{A}[\![a_0]\!]\sigma) \times (\mathscr{A}[\![a_1]\!]\sigma)
\end{aligned}$$

Let us briefly comment on the above definitions.

Constants:	The denotational semantics of any constant n is just the constant function that always returns n for any σ.
Variables:	The denotational semantics of any variable x is the function that takes a memory σ and returns the value of x in σ.
Binary expressions:	The denotational semantics of any binary expression evaluates the arguments (with the same given σ) and combines the results by exploiting the corresponding arithmetic operation.

Note that the symbols $+$, $-$ and $\times$ are overloaded: in the left hand side they represent elements of the syntax, while in the right hand side they represent operators of the metalanguage. Similarly for the symbol n in the first equation.

6.2.2 Denotational Semantics of Boolean Expressions: The Function $\mathscr{B}$

The denotational semantics of boolean expression is given by a function $\mathscr{B}$ defined in a very similar way to $\mathscr{A}$. The only difference is that the values to be returned are elements of $\mathbb{B}$ and not of $\mathbb{Z}$ and that $\mathscr{B}$ is not always defined in terms of itself: some defining equations exploit the function $\mathscr{A}$.

Definition 6.8 (Denotational semantics of boolean expressions). The denotational semantics of boolean expressions is defined by structural recursion as follows:

$$\begin{aligned}
\mathscr{B}[\![v]\!]\sigma &\stackrel{\text{def}}{=} v \\
\mathscr{B}[\![a_0 = a_1]\!]\sigma &\stackrel{\text{def}}{=} (\mathscr{A}[\![a_0]\!]\sigma) = (\mathscr{A}[\![a_1]\!]\sigma) \\
\mathscr{B}[\![a_0 \leq a_1]\!]\sigma &\stackrel{\text{def}}{=} (\mathscr{A}[\![a_0]\!]\sigma) \leq (\mathscr{A}[\![a_1]\!]\sigma) \\
\mathscr{B}[\![\neg b_0]\!]\sigma &\stackrel{\text{def}}{=} \neg\, (\mathscr{B}[\![b]\!]\sigma) \\
\mathscr{B}[\![b_0 \vee b_1]\!]\sigma &\stackrel{\text{def}}{=} (\mathscr{B}[\![b_0]\!]\sigma) \vee (\mathscr{B}[\![b_1]\!]\sigma) \\
\mathscr{B}[\![b_0 \wedge b_1]\!]\sigma &\stackrel{\text{def}}{=} (\mathscr{B}[\![b_0]\!]\sigma) \wedge (\mathscr{B}[\![b_1]\!]\sigma)
\end{aligned}$$

6.2.3 Denotational Semantics of Commands: The Function $\mathscr{C}$

We are now ready to present the denotational semantics of commands. As one might expect, the interpretation function of commands is the most complex. It has the following type:

$$\mathscr{C} : Com \to (\Sigma \rightharpoonup \Sigma)$$

Since commands can diverge, the codomain of $\mathscr{C}$ is the set of partial functions from memories to memories. As we have discussed in Example 5.14, for each partial function we can define an equivalent total function. So we define:

$$\mathscr{C} : Com \to (\Sigma \to \Sigma_\perp)$$

This will simplify the notation.

Instead of presenting the whole, structurally recursive, definition of $\mathscr{C}$ and then commenting the defining equations, we give each rule separately accompanied by the necessary explanations.

We start from the simplest commands: skip and assignments.

$$\mathscr{C}[\![\textbf{skip}]\!]\,\sigma \stackrel{\text{def}}{=} \sigma \tag{6.1}$$

We see that $\mathscr{C}[\![\textbf{skip}]\!]$ is the identity function: **skip** does not modify the memory.

$$\mathscr{C}[\![x := a]\!]\,\sigma \stackrel{\text{def}}{=} \sigma[{}^{\mathscr{A}[\![a]\!]\sigma}/_{x}] \tag{6.2}$$

The denotational semantics of the assignment evaluates the arithmetic expression a via $\mathscr{A}$ and then modifies the memory assigning the corresponding value to the location x.

Let us now consider the sequential composition of two commands. In interpreting $c_0; c_1$ we first interpret c_0 in the starting memory and then c_1 in the state produced by c_0. The problem is that from the first application of $\mathscr{C}[\![c_0]\!]$ we obtain a value in $\Sigma_\perp$, not necessarily in Σ, so we can not apply $\mathscr{C}[\![c_1]\!]$. To work this problem out we introduce a *lifting* operator $(\cdot)^*$: it takes a function in $\Sigma \to \Sigma_\perp$ and returns a function in $\Sigma_\perp \to \Sigma_\perp$, i.e., its type is $(\Sigma \to \Sigma_\perp) \to (\Sigma_\perp \to \Sigma_\perp)$.

Definition 6.9 (Lifting). Let $f : \Sigma \to \Sigma_\perp$, we define a function $f^* : \Sigma_\perp \to \Sigma_\perp$ as follows:

$$f^*(x) = \begin{cases} \perp & \text{if } x = \perp \\ f(x) & \text{otherwise} \end{cases}$$

So the definition of the interpretation function for $c_0; c_1$ is:

$$\mathscr{C}[\![c_0; c_1]\!]\,\sigma \stackrel{\text{def}}{=} \mathscr{C}[\![c_1]\!]^*\,(\mathscr{C}[\![c_0]\!]\,\sigma) \tag{6.3}$$

Note that we apply the lifted version $\mathscr{C}[\![c_1]\!]^*$ of $\mathscr{C}[\![c_1]\!]$ to the argument $\mathscr{C}[\![c_0]\!]$.

Let us now consider the conditional command. Recall that the λ-calculus provides a conditional operator, then we have immediately:

$$\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma \stackrel{\text{def}}{=} \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c_0]\!]\sigma, \mathscr{C}[\![c_1]\!]\sigma \tag{6.4}$$

The definition of the denotational semantics of the while command is more intricate. We could think to define the interpretation simply as:

$$\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma \stackrel{\text{def}}{=} \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma$$

Obviously this definition is not a structural recursive, because the same expression $\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]$ whose meaning we want to define appears in the right-hand side of the defining equation. Indeed structural recursion allows only for the presence of subterms in the right-hand side, like $\mathscr{B}[\![b]\!]$ and $\mathscr{C}[\![c]\!]$. To solve this issue we will reduce the problem of defining the semantics of iteration to a fixpoint calculation. Let us define a function $\Gamma_{b,c} : (\Sigma \to \Sigma_\perp) \to \Sigma \to \Sigma_\perp$:

$$\Gamma_{b,c} \stackrel{\text{def}}{=} \lambda\varphi.\lambda\sigma.\underbrace{\underbrace{\underbrace{\mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{C}[\![c]\!]\sigma), \sigma}_{\Sigma_\perp}}_{\Sigma \to \Sigma_\perp}}_{(\Sigma \to \Sigma_\perp) \to \Sigma \to \Sigma_\perp}$$

The function $\Gamma_{b,c}$ takes a function $\varphi : \Sigma \to \Sigma_\perp$, and returns the function

$$\lambda\sigma.\mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{C}[\![c]\!]\sigma), \sigma$$

of type is $\Sigma \to \Sigma_\perp$, which given a memory σ evaluates $\mathscr{B}[\![b]\!]\sigma$ and depending on the outcome returns either $\varphi^*(\mathscr{C}[\![c]\!]\sigma)$ or σ. Note that the definition of $\Gamma_{b,c}$ refers only to subterms of the command **while** b **do** c. Clearly we require that $\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]$ is a fixpoint of $\Gamma_{b,c}$, i.e., that:

$$\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!] = \Gamma_{b,c}\,\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]$$

As there can be several fixpoints for $\Gamma_{b,c}$, we define $\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]$ as the least one. Next we show that $\Gamma_{b,c}$ is a monotone and continuous function, so that we can prove that $\Gamma_{b,c}$ has a least fixpoint and that by the fixpoint Theorem 5.6:

$$\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!] \stackrel{\text{def}}{=} \text{fix } \Gamma_{b,c} = \bigsqcup_{n\in\mathbb{N}} \Gamma^n_{b,c}(\perp_{\Sigma\to\Sigma_\perp}) \tag{6.5}$$

To prove continuity we will consider $\Gamma_{b,c}$ as operating on partial functions:

$$\Gamma_{b,c} : (\Sigma \rightharpoonup \Sigma) \longrightarrow (\Sigma \rightharpoonup \Sigma).$$

Partial functions in $\Sigma \rightharpoonup \Sigma$ can be represented as sets of pairs (σ, σ') that we write as formulas $\sigma \longrightarrow \sigma'$. Then the effect of $\Gamma_{b,c}$ can be represented by the immediate consequence operators for the following set of rules.

$$R_{\Gamma_{b,c}} \stackrel{\text{def}}{=} \left\{ \frac{\mathscr{B}[\![b]\!]\sigma \quad \mathscr{C}[\![c]\!]\sigma = \sigma'' \quad \sigma'' \to \sigma'}{\sigma \to \sigma'}, \qquad \frac{\neg\mathscr{B}[\![b]\!]\sigma}{\sigma \to \sigma} \right\}$$

Note that there are infinitely many instance of the rules, but each rule has only a finite number of premises and that

$$\widehat{R}_{\Gamma_{b,c}} = \Gamma_{b,c}.$$

The only formulas appearing in the rules are $\sigma'' \longrightarrow \sigma'$ (as a premise of the first rule), $\sigma \longrightarrow \sigma'$ and $\sigma \longrightarrow \sigma$ (as conclusions); the other formulas express side-conditions: $\mathscr{B}[\![b]\!]\sigma \wedge \mathscr{C}[\![c]\!]\sigma = \sigma''$ for the first rule and $\neg\mathscr{B}[\![b]\!]\sigma$ for the second rule. An instance of the first rule schema is obtained by picking up two memories σ and σ'' such that $\mathscr{B}[\![b]\!]\sigma$ is true and $\mathscr{C}[\![c]\!]\sigma = \sigma''$. Then for every σ' such that $\sigma'' \to \sigma'$ we can derive $\sigma \to \sigma'$. The second rule schema is an axiom expressing that $\sigma \to \sigma$ whenever $\neg\mathscr{B}[\![b]\!]\sigma$.

Since all the rules obtained in this way have a finite number of premises (actually one or none), we can apply Theorem 5.8, which ensures the continuity of $\widehat{R}_{\Gamma_{b,c}}$. Now by using Theorem 5.10 we have:

$$\text{fix } \Gamma_{b,c} = \text{fix } \widehat{R}_{\Gamma_{b,c}} = I_{R_{\Gamma_{b,c}}}$$

Let us conclude this section with three examples which explain how to use the definitions we have given.

Example 6.6. Let us consider the command:

$$w = \textbf{while true do skip}$$

now we will see how to calculate its semantics. We have $\mathscr{C}[\![w]\!] \stackrel{\text{def}}{=} \text{fix } \Gamma_{\textbf{true},\textbf{skip}}$ where

$$\begin{aligned}\Gamma_{\textbf{true},\textbf{skip}}\varphi\sigma &= \mathscr{B}[\![\textbf{true}]\!]\sigma \to \varphi^*(\mathscr{C}[\![\textbf{skip}]\!]\sigma), \sigma \\ &= \varphi^*(\mathscr{C}[\![\textbf{skip}]\!]\sigma) \\ &= \varphi^*\sigma \\ &= \varphi\sigma\end{aligned}$$

So we have $\Gamma_{\textbf{true},\textbf{skip}}\varphi = \varphi$, that is $\Gamma_{\textbf{true},\textbf{skip}}$ is the identity function. Then each function φ is a fixpoint of $\Gamma_{\textbf{true},\textbf{skip}}$, but we are looking for the least fixpoint. This means that the sought solution is the least function in the $CPO_\perp$ of functions $\Sigma \to \Sigma_\perp$. Then we have

$$\text{fix } \Gamma_{\textbf{true},\textbf{skip}} = \lambda\sigma.\perp_{\Sigma_\perp}.$$

In the following we will often write just Γ when the subscripts b and c are obvious from the context.

Example 6.7. Let us consider the commands

$$w \stackrel{\text{def}}{=} \textbf{while } b \textbf{ do } c$$
$$c' \stackrel{\text{def}}{=} \textbf{if } b \textbf{ then } (c\,;\,w) \textbf{ else skip}$$

Now we show the denotational equivalence between w and c' for any b and c. Since $\mathscr{C}[\![w]\!]$ is a fixpoint we have:

$$\mathscr{C}[\![w]\!] = \Gamma(\mathscr{C}[\![w]\!]) = \lambda\sigma.\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![w]\!]^*(\mathscr{C}[\![c]\!]\sigma),\sigma$$

For c' we have:

$$\begin{aligned}\mathscr{C}[\![\textbf{if } b \textbf{ then } (c;w) \textbf{ else skip}]\!] &= \lambda\sigma.\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c;w]\!]\sigma, \mathscr{C}[\![\textbf{skip}]\!]\sigma \\ &= \lambda\sigma.\mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![w]\!]^*(\mathscr{C}[\![c]\!]\sigma),\sigma\end{aligned}$$

Hence $\mathscr{C}[\![w]\!] = \mathscr{C}[\![c']\!]$.

Example 6.8. Let us consider the command:

$$c \stackrel{\text{def}}{=} \textbf{while } x \neq 0 \textbf{ do } x := x-1$$

we have $\mathscr{C}[\![c]\!] \stackrel{\text{def}}{=} \text{fix } \Gamma$ where:[1]

$$\begin{aligned}\Gamma\varphi &\stackrel{\text{def}}{=} \lambda\sigma.\ \mathscr{B}[\![x \neq 0]\!] \to \varphi^*(\mathscr{C}[\![x := x-1]\!]\sigma),\sigma \\ &= \lambda\ \sigma.\ \sigma x \neq 0 \to \varphi^*\sigma[^{\sigma x-1}/_x],\sigma \\ &= \lambda\sigma.\sigma x \neq 0 \to \varphi\ \sigma[^{\sigma x-1}/_x],\sigma\end{aligned}$$

Let us see some calculation for approximating the fixpoint:

$$\begin{aligned}\varphi_0 &= \Gamma^0\ \bot_{\Sigma\to\Sigma_\bot} = \bot_{\Sigma\to\Sigma_\bot} = \lambda\ \sigma.\bot_{\Sigma_\bot} \\ \varphi_1 &= \Gamma\ \varphi_0 \\ &= \lambda\ \sigma.\sigma x \neq 0 \to \underbrace{\bot_{\Sigma\to\Sigma_\bot}}_{\varphi_0}\ \sigma[^{\sigma x-1}/_x],\sigma \\ &= \lambda\ \sigma.\sigma x \neq 0 \to \bot_{\Sigma_\bot},\sigma \\ \varphi_2 &= \Gamma\ \varphi_1 \\ &= \lambda\sigma.\ \sigma x \neq 0 \to\ \underbrace{(\lambda\sigma'.\ \sigma' x \neq 0 \to \bot_{\Sigma_\bot},\sigma')}_{\varphi_1}\ \sigma[^{\sigma x-1}/_x],\sigma\end{aligned}$$

Now we have the following possibilities for computing $\varphi_2\sigma$:

$\sigma x < 0$) Then $\sigma x \neq 0$ and $\sigma[^{\sigma x-1}/_x]x \neq 0$ and thus $\varphi_2\sigma = \bot_{\Sigma_\bot}$.
$\sigma x = 0$) Then $\sigma x \neq 0$ is false and thus $\varphi_2\sigma = \sigma = \sigma[^0/_x]$
$\sigma x = 1$) Then $\sigma x \neq 0$ and $\sigma[^{\sigma x-1}/_x]x = 0$ and thus $\varphi_2\sigma = \sigma[^{\sigma x-1}/_x] = \sigma[^0/_x]$.

[1] Note that in the last step we can remove the lifting operation from φ^* because $\sigma[^{\sigma x-1}/_x] \neq \bot$.

$\sigma x > 1$) Then $\sigma x \neq 0$ and $\sigma[^{\sigma x-1}/_x]x \neq 0$ and thus $\varphi_2\sigma = \bot_{\Sigma_\bot}$.

Summarising:

$$\frac{\sigma x < 0}{\varphi_2\sigma = \bot} \qquad \frac{\sigma x = 0}{\varphi_2\sigma = \sigma[^0/_x]} \qquad \frac{\sigma x = 1}{\varphi_2\sigma = \sigma[^0/_x]} \qquad \frac{\sigma x > 1}{\varphi_2\sigma = \bot}$$

So we have:

$$\varphi_2 = \lambda\sigma.\ \sigma x < 0 \to \bot\ ,\ (\sigma x < 2 \to \sigma[^0/_x], \bot)$$

We can conjecture that $\forall n \in \mathbb{N}.\ P(n)$, where:

$$P(n) \stackrel{\text{def}}{=} (\ \varphi_n = \lambda\sigma.\ \sigma x < 0 \to \bot\ ,\ (\sigma x < n \to \sigma[^0/_x], \bot)\ )$$

We are now ready to prove our conjecture by mathematical induction on n.

Base case: The base case is trivial, indeed we know $\varphi_0 = \lambda\sigma.\ \bot$ and

$$\begin{aligned}
&\lambda\sigma.\ \sigma x < 0 \to \bot\ ,\ (\sigma x < 0 \to \sigma[^0/_x], \bot)\\
&\quad = \lambda\sigma.\ \sigma x < 0 \to \bot\ ,\ (0 \leq \sigma x < 0 \to \sigma[^0/_x], \bot)\\
&\quad = \lambda\sigma.\ \sigma x < 0 \to \bot\ ,\ (\mathbf{false} \to \sigma[^0/_x], \bot)\\
&\quad = \lambda\sigma.\ \sigma x < 0 \to \bot\ ,\ \bot\\
&\quad = \bot
\end{aligned}$$

Inductive case: For the inductive case, let us assume

$$P(n) \stackrel{\text{def}}{=} \left(\varphi_n = \lambda\sigma.\ \sigma x < 0 \to \bot\ ,\ (\sigma x < n \to \sigma[^0/_x], \bot)\right).$$

We want to prove:

$$P(n+1) \stackrel{\text{def}}{=} \left(\varphi_{n+1} = \lambda\sigma.\ \sigma x < 0 \to \bot\ ,\ (\sigma x < n+1 \to \sigma[^0/_x], \bot)\right)$$

By definition:

$$\varphi_{n+1} = \Gamma\ \varphi_n = \lambda\sigma.\ \sigma x \neq 0 \to \varphi_n(\sigma[^{\sigma x-1}/_x]), \sigma$$

Letting $\sigma' \stackrel{\text{def}}{=} \sigma[^{\sigma x-1}/_x]$ and the inductive hypothesis, we have:

$$\begin{aligned}
\varphi_n\sigma' &= \sigma' x < 0 \to \bot\ ,\ (\sigma' x < n \to \sigma'[^0/_x], \bot)\\
&= \sigma x - 1 < 0 \to \bot\ ,\ (\sigma x - 1 < n \to \sigma[^0/_x], \bot)\\
&= \sigma x < 1 \to \bot\ ,\ (\sigma x < n+1 \to \sigma[^0/_x], \bot)
\end{aligned}$$

Thus:

$$\varphi_{n+1}\sigma = \sigma x \neq 0 \to (\sigma x < 1 \to \bot\ ,\ (\sigma x < n+1 \to \sigma[^0/_x], \bot)\ )\ ,\ \sigma$$

Now we have the following possibilities for computing $\varphi_{n+1}\sigma$:

$\sigma x < 0$)	Then $\sigma x \neq 0$ and $\sigma x < 1$, thus $\varphi_{n+1}\sigma = \bot$.
$\sigma x = 0$)	Then $\sigma x \neq 0$ is false and thus $\varphi_{n+1}\sigma = \sigma = \sigma[^0/_x]$
$1 \leq \sigma x < n+1$)	Then $\sigma x \neq 0$, $\sigma x < 1$ is false and $\sigma x < n+1$, thus $\varphi_{n+1}\sigma = \sigma[^0/_x]$.
$\sigma x \geq n+1$)	Then $\sigma x \neq 0$, $\sigma x < 1$ is false and so is $\sigma x < n+1$, thus $\varphi_{n+1}\sigma = \bot$.

Summarising:

$$\frac{\sigma x < 0}{\varphi_{n+1}\sigma = \bot} \quad \frac{\sigma x = 0}{\varphi_{n+1}\sigma = \sigma[^0/_x]} \quad \frac{1 \leq \sigma x < n+1}{\varphi_{n+1}\sigma = \sigma[^0/_x]} \quad \frac{\sigma x \geq n+1}{\varphi_{n+1}\sigma = \bot}$$

Then:

$$\varphi_{n+1} = \lambda\sigma.\ \sigma x < 0 \to \bot\ ,\ (\sigma x < n+1 \to \sigma[0/x], \bot)$$

which proves $P(n+1)$.

Finally we have:

$$\mathscr{C}[\![c]\!] = \text{fix}\ \Gamma = \bigsqcup_{n\in\mathbb{N}} \Gamma^n \bot = \bigsqcup_{n\in\mathbb{N}} \varphi_n = \lambda\sigma.\sigma x < 0 \to \bot, \sigma[0/x]$$

6.3 Equivalence Between Operational and Denotational Semantics

This section deals with the issue of equivalence between the two semantics of IMP introduced up to now. As we will show, the denotational and operational semantics agree. As usual we will handle first arithmetic and boolean expressions, then assuming the proved equivalences we will show that operational and denotational semantics agree also on commands.

6.3.1 Equivalence Proofs For Expressions

We start by considering arithmetic expressions. We want to prove that the operational and denotational semantics coincide, that is, the results of evaluating an arithmetic expression both by operational and denotational semantics are the same. If we regard the operational semantics as an interpreter and the denotational semantics as a compiler we are proving that interpreting an IMP program and executing its compiled version starting from the same memory leads to the same result.

Theorem 6.1. *For all arithmetic expressions $a \in Aexp$, the predicate $P(a)$ holds, where:*

$$P(a) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \langle a, \sigma \rangle \to \mathscr{A}[\![a]\!]\sigma$$

Proof. The proof is by structural induction on arithmetic expressions.

Const: $P(n) \stackrel{\text{def}}{=} \forall \sigma.\ \langle n, \sigma \rangle \to \mathscr{A}[\![n]\!]\sigma$ holds because, given a generic σ, we have $\langle n, \sigma \rangle \to n$ and $\mathscr{A}[\![n]\!]\sigma = n$.

Vars: $P(x) \stackrel{\text{def}}{=} \forall \sigma.\ \langle x, \sigma \rangle \to \mathscr{A}[\![x]\!]\sigma$ holds because, given a generic σ, we have $\langle x, \sigma \rangle \to \sigma x$ and $\mathscr{A}[\![x]\!]\sigma = \sigma x$.

Ops: Let us generalize the proof for the binary operations of arithmetic expressions. Consider two arithmetic expressions a_0 and a_1 and a binary operator $\odot \in \{+, -, \times\}$ of IMP, whose corresponding semantic operator is $\cdot$. We assume:

$$P(a_0) \stackrel{\text{def}}{=} \langle a_0, \sigma \rangle \to \mathscr{A}[\![a_0]\!]\sigma$$
$$P(a_1) \stackrel{\text{def}}{=} \langle a_1, \sigma \rangle \to \mathscr{A}[\![a_1]\!]\sigma$$

and we want to prove

$$P(a_0 \odot a_1) \stackrel{\text{def}}{=} \langle a_0 \odot a_1, \sigma \rangle \to \mathscr{A}[\![a_0 \odot a_1]\!]\sigma.$$

By using the inductive hypothesis we derive:

$$\langle a_0 \odot a_1, \sigma \rangle \to \mathscr{A}[\![a_0]\!]\sigma \cdot \mathscr{A}[\![a_1]\!]\sigma$$

Finally, by definition of $\mathscr{A}$

$$\mathscr{A}[\![a_0]\!]\sigma \cdot \mathscr{A}[\![a_1]\!]\sigma = \mathscr{A}[\![a_0 \odot a_1]\!]\sigma.$$

□

The case of boolean expressions is completely similar to that of arithmetic expressions, so we leave the proof as an exercise (see Problem 6.2).

Theorem 6.2. *For all boolean expressions $b \in Bexp$, the predicate $P(b)$ holds, where:*

$$P(b) \stackrel{\text{def}}{=} \forall \sigma \in \Sigma.\ \langle b, \sigma \rangle \to \mathscr{B}[\![b]\!]\sigma$$

From now on we will assume the equivalence between denotational and operational semantics for boolean and arithmetic expressions.

6.3.2 Equivalence Proof for Commands

Central to the proof of equivalence between denotational and operational semantics is the case of commands. Operational and denotational semantics are defined in very different formalisms: on the one hand we have an inference rule system which

allows to calculate the execution of each command, on the other hand we have a function which associates to each command its functional meaning. So to show the equivalence between the two semantics we will prove the following property:

Theorem 6.3. $\forall c \in Com.\ \forall \sigma, \sigma' \in \Sigma.\ \langle c, \sigma\rangle \to \sigma' \Longleftrightarrow \mathscr{C}[\![c]\!]\,\sigma = \sigma'$.

As usual we divide the proof in two parts:

Completeness: $\forall c \in Com,\ \forall \sigma, \sigma' \in \Sigma$ we prove

$$P(\langle c, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\,\sigma = \sigma'.$$

Correctness: $\forall c \in Com$ we prove

$$P(c) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \in \Sigma.\ \mathscr{C}[\![c]\!]\,\sigma = \sigma' \Rightarrow \langle c, \sigma\rangle \to \sigma'.$$

Notice that in this way the non defined cases are also handled for the equivalence: for instance we have

$$\langle c, \sigma\rangle \not\to \quad \Rightarrow \quad \mathscr{C}[\![c]\!]\,\sigma = \perp_{\Sigma_\perp}$$

since otherwise, assuming $\mathscr{C}[\![c]\!]\,\sigma = \sigma'$ for some $\sigma' \in \Sigma$, it would follow that $\langle c, \sigma\rangle \to \sigma'$. Similarly in the opposite direction.

$$\mathscr{C}[\![c]\!]\,\sigma = \perp_{\Sigma_\perp} \quad \Rightarrow \quad \langle c, \sigma\rangle \not\to$$

6.3.2.1 Completeness of the Denotational Semantics

Let us prove the first part of Theorem 6.3. We let

$$P\left(\langle c, \sigma\rangle \to \sigma'\right) \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\,\sigma = \sigma'$$

and prove that $P(\langle c, \sigma\rangle \to \sigma')$ holds for any $c \in Com$ and $\sigma, \sigma' \in \Sigma$.

We proceed by rule induction. So for each rule we will assume the property holds for the premises and we will prove the property holds for the conclusion.

skip: Let us consider the operational rule for the **skip**

$$\frac{}{\langle \mathbf{skip}, \sigma\rangle \to \sigma}$$

We want to prove:

$$P(\langle \mathbf{skip}, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \mathscr{C}[\![\mathbf{skip}]\!]\,\sigma = \sigma$$

Obviously the proposition is true by definition of denotational semantic.

assign: Let us consider the rule for the assignment command:

$$\frac{\langle a, \sigma \rangle \to m}{\langle x := a, \sigma \rangle \to \sigma\,[^{m}/_{x}]}$$

We can assume $\langle a, \sigma \rangle \to m$ and therefore $\mathscr{A}\,[\![a]\!]\,\sigma = m$ by the correspondence between the operational and denotational semantics of arithmetic expressions.
We want to prove:

$$P(\langle x := a, \sigma \rangle \to \sigma\,[^{m}/_{x}]) \stackrel{\text{def}}{=} \mathscr{C}\,[\![x := a]\!]\,\sigma = \sigma\,[^{m}/_{x}]$$

By the definition of denotational semantics:

$$\mathscr{C}\,[\![x := a]\!]\,\sigma = \sigma[^{\mathscr{A}[\![a]\!]\sigma}/_{x}] = \sigma[^{m}/_{x}]$$

seq: Let us consider the concatenation rule:

$$\frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'}$$

We assume:

$$P(\langle c_0, \sigma \rangle \to \sigma'') \stackrel{\text{def}}{=} \mathscr{C}\,[\![c_0]\!]\,\sigma = \sigma''$$
$$P(\langle c_1, \sigma'' \rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}\,[\![c_1]\!]\,\sigma'' = \sigma'$$

We want to prove:

$$P(\langle c_0; c_1, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}\,[\![c_0; c_1]\!]\,\sigma = \sigma'$$

By denotational semantics definition and inductive hypotheses:

$$\mathscr{C}\,[\![c_0; c_1]\!]\,\sigma = \mathscr{C}\,[\![c_1]\!]^{*}\,(\mathscr{C}\,[\![c_0]\!]\,\sigma) = \mathscr{C}\,[\![c_1]\!]^{*}\,\sigma'' = \mathscr{C}\,[\![c_1]\!]\,\sigma'' = \sigma'$$

Note that the lifting operator can be removed because $\sigma'' \neq \perp$ by inductive hypothesis.

iftt: Let us consider the rule:

$$\frac{\langle b, \sigma \rangle \to \textbf{true} \quad \langle c_0, \sigma \rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma \rangle \to \sigma'}$$

We assume:

- $\langle b, \sigma \rangle \to \textbf{true}$ and therefore $\mathscr{B}\,[\![b]\!]\,\sigma = \textbf{true}$ by the correspondence between operational and denotational semantics for boolean expressions;
- $P(\langle c_0, \sigma \rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}\,[\![c_0]\!]\,\sigma = \sigma'$.

We want to prove:

$$P(\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \sigma'$$

In fact, we have:

$$\begin{aligned}\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma &= \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c_0]\!]\sigma, \mathscr{C}[\![c_1]\!]\sigma \\ &= \textbf{true} \to \sigma', \mathscr{C}[\![c_1]\!]\sigma \\ &= \sigma'\end{aligned}$$

ifff: The proof for the second rule of the conditional coommand is completely analogous to the previous one and thus omitted.

whff: Let us consider the rule:

$$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma}$$

We assume $\langle b, \sigma\rangle \to \textbf{false}$ and therefore $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$. We want to prove:

$$P(\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma) \stackrel{\text{def}}{=} \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \sigma$$

By the fixpoint property of the denotational semantics:

$$\begin{aligned}\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma &= \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma \\ &= \textbf{false} \to \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma \\ &= \sigma\end{aligned}$$

whtt: At last we consider the second rule of the while command

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'}$$

We assume:

- $\langle b, \sigma\rangle \to \textbf{true}$ and therefore $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$
- $P(\langle c, \sigma\rangle \to \sigma'') \stackrel{\text{def}}{=} \mathscr{C}[\![c]\!]\sigma = \sigma''$
- $P(\langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma'' = \sigma'$

We want to prove:

$$P(\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma') \stackrel{\text{def}}{=} \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \sigma'$$

By the definition of the denotational semantics and inductive hypotheses:

$$\begin{aligned}\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma &= \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*(\mathscr{C}[\![c]\!]\sigma), \sigma \\ &= \textbf{true} \to \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*\sigma'', \sigma \\ &= \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]^*\sigma'' \\ &= \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma'' \\ &= \sigma'\end{aligned}$$

Note that the lifting operator can be removed since $\sigma'' \neq \bot$.

6.3.2.2 Correctness of the Denotational Semantics

Let us conclude the proof of Theorem 6.3 by showing the correctness of the denotational semantics. We need to prove, for all $c \in Com$:

$$P(c) \stackrel{\text{def}}{=} \forall \sigma, \sigma' \in \Sigma.\ \mathscr{C}[\![c]\!]\sigma = \sigma' \Rightarrow \langle c, \sigma\rangle \to \sigma'$$

Since the denotational semantics is given by structural recursion we will proceed by induction on the structure of commands.

skip: We need to prove:

$$P(\textbf{skip}) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \mathscr{C}[\![\textbf{skip}]\!]\sigma = \sigma' \Rightarrow \langle \textbf{skip}, \sigma\rangle \to \sigma'$$

By definition we have $\mathscr{C}[\![\textbf{skip}]\!]\sigma = \sigma$ and $\langle \textbf{skip}, \sigma\rangle \to \sigma$ is an axiom of the operational semantics.

assign: We need to prove:

$$P(x := a) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \mathscr{C}[\![x := a]\!]\sigma = \sigma' \Rightarrow \langle x := a, \sigma\rangle \to \sigma'$$

By denotational semantics definition we have $\sigma' = \sigma[{}^{\mathscr{A}[\![a]\!]\sigma}/_{x}]$ and by the equivalence between operational and denotational semantics for expressions we have $\langle a, \sigma\rangle \to \mathscr{A}[\![a]\!]\sigma$, thus we can apply the rule (assign) to conclude

$$\langle x := a, \sigma\rangle \to \sigma[{}^{\mathscr{A}[\![a]\!]\sigma}/_{x}].$$

seq: We assume:

- $P(c_0) \stackrel{\text{def}}{=} \forall \sigma, \sigma''.\ \mathscr{C}[\![c_0]\!]\sigma = \sigma'' \Rightarrow \langle c_0, \sigma\rangle \to \sigma''$
- $P(c_1) \stackrel{\text{def}}{=} \forall \sigma'', \sigma'.\ \mathscr{C}[\![c_1]\!]\sigma'' = \sigma' \Rightarrow \langle c_1, \sigma''\rangle \to \sigma'$

We want to prove:

$$P(c_0; c_1) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \mathscr{C}[\![c_0; c_1]\!]\sigma = \sigma' \Rightarrow \langle c_0; c_1, \sigma\rangle \to \sigma'$$

We assume $\mathscr{C}[\![c_0; c_1]\!]\sigma = \sigma'$ and prove $\langle c_0; c_1, \sigma\rangle \to \sigma'$. We have

$$\mathscr{C}[\![c_0;c_1]\!]\sigma = \mathscr{C}[\![c_1]\!]^*(\mathscr{C}[\![c_0]\!]\sigma) = \sigma'.$$

Since $\sigma' \neq \bot$, it must be $\mathscr{C}[\![c_0]\!]\sigma \neq \bot$, i.e., we can assume the termination of c_0 and can omit the lifting operator:

$$\mathscr{C}[\![c_0;c_1]\!]\sigma = \mathscr{C}[\![c_1]\!](\mathscr{C}[\![c_0]\!]\sigma) = \sigma'$$

Let $\mathscr{C}[\![c_0]\!]\sigma = \sigma''$. We have $\mathscr{C}[\![c_1]\!]\sigma'' = \sigma'$. Then we can apply modus ponens to the inductive assumptions $P(c_0)$ and $P(c_1)$, to get $\langle c_0,\sigma\rangle \to \sigma''$ and $\langle c_1,\sigma''\rangle \to \sigma'$. Thus we can apply the inference rule:

$$\frac{\langle c_0,\sigma\rangle \to \sigma'' \qquad \langle c_1,\sigma''\rangle \to \sigma'}{\langle c_0;c_1,\sigma\rangle \to \sigma'}$$

to conclude $\langle c_0;c_1,\sigma\rangle \to \sigma'$.

if: We assume:

- $P(c_0) \stackrel{\text{def}}{=} \forall\sigma,\sigma'.\, \mathscr{C}[\![c_0]\!]\sigma = \sigma' \Rightarrow \langle c_0,\sigma\rangle \to \sigma'$
- $P(c_1) \stackrel{\text{def}}{=} \forall\sigma,\sigma'.\, \mathscr{C}[\![c_1]\!]\sigma = \sigma' \Rightarrow \langle c_1,\sigma\rangle \to \sigma'$

We need to prove:

$$P(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1) \stackrel{\text{def}}{=} \forall\sigma,\sigma'.\, \mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \sigma' \Rightarrow \langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1,\sigma\rangle \to \sigma'$$

Let us assume the premise $\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \sigma'$ and prove that $\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1,\sigma\rangle \to \sigma'$. By definition:

$$\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c_0]\!]\sigma, \mathscr{C}[\![c_1]\!]\sigma$$

Now, either $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$ or $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$.
If $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$, we have also $\langle b,\sigma\rangle \to \textbf{false}$. Then:

$$\mathscr{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma = \mathscr{C}[\![c_1]\!]\sigma = \sigma'$$

By modus ponens on the inductive hypothesis $P(c_1)$ we have $\langle c_1,\sigma\rangle \to \sigma'$. Thus we can apply the rule:

$$\frac{\langle b,\sigma\rangle \to \textbf{false} \qquad \langle c_1,\sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1,\sigma\rangle \to \sigma'}$$

to conclude $\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1,\sigma\rangle \to \sigma'$.
The case where $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$ is completely analogous and thus omitted.

while: We assume:

$$P(c) \stackrel{\text{def}}{=} \forall\sigma,\sigma''.\, \mathscr{C}[\![c]\!]\sigma = \sigma'' \Rightarrow \langle c,\sigma\rangle \to \sigma''$$

We need to prove:

$$P(\textbf{while } b \textbf{ do } c) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$

By definition $\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \text{fix } \Gamma_{b,c}\,\sigma = \left(\bigsqcup_{n\in\mathbb{N}} \Gamma^n_{b,c}\bot\right)\sigma$ so:

$$\begin{gathered}\mathscr{C}[\![\textbf{while } b \textbf{ do } c]\!]\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma' \\ \Leftrightarrow \\ \left(\bigsqcup_{n\in\mathbb{N}} \Gamma^n_{b,c}\bot\right)\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma' \\ \Leftrightarrow \\ \forall n \in \mathbb{N}.\ \left(\Gamma^n_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'\right)\end{gathered}$$

Notice that the last two properties are equivalent. In fact, if there is a pair $\sigma \to \sigma'$ in the limit, it must also occur in $\Gamma^k_{b,c}\bot$ for some k. Vice versa, if it belongs to $\Gamma^n_{b,c}\bot$ for some n then it belongs also to the limit. Let

$$A(n) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \Gamma^n_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$

We prove that $\forall n.\ A(n)$ by mathematical induction.

Base case: We have to prove $A(0)$, namely:

$$\forall \sigma, \sigma'.\ \Gamma^0_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$$

Since $\Gamma^0_{b,c}\bot\sigma = \bot\sigma = \bot$ and $\sigma' \neq \bot$ the premise is false and hence the implication is true.

Ind. case: Let us assume

$$A(n) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \Gamma^n_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'.$$

We want to show that

$$A(n+1) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\ \Gamma^{n+1}_{b,c}\bot\sigma = \sigma' \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'.$$

We assume $\Gamma^{n+1}_{b,c}\bot\sigma = \Gamma_{b,c}\left(\Gamma^n_{b,c}\bot\right)\sigma = \sigma'$, that is

$$\mathscr{B}[\![b]\!]\sigma \to \left(\Gamma^n_{b,c}\bot\right)^*\left(\mathscr{C}[\![c]\!]\sigma\right), \sigma = \sigma'$$

Now either $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$ or $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$.

- If $\mathscr{B}[\![b]\!]\sigma = \textbf{false}$, we have $\langle b, \sigma\rangle \to \textbf{false}$ and $\sigma' = \sigma$. Now by using the rule (whff):

$$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma}$$

we conclude $\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma$.

- if $\mathscr{B}[\![b]\!]\sigma = \textbf{true}$ we have $\langle b,\sigma\rangle \to \textbf{true}$ and

$$\left(\Gamma^n_{b,c}\bot\right)^*(\mathscr{C}[\![c]\!]\sigma) = \sigma'.$$

Since $\sigma' \neq \bot$ there must exists some $\sigma'' \neq \bot$ with $\mathscr{C}[\![c]\!]\sigma = \sigma''$ and by structural induction $\langle c,\sigma\rangle \to \sigma''$. Since $\left(\Gamma^n_{b,c}\bot\right)^*(\mathscr{C}[\![c]\!]\sigma) = \left(\Gamma^n_{b,c}\bot\right)\sigma'' = \sigma'$ we have by the mathematical induction hypothesis $A(n)$ that

$$\left\langle \textbf{while } b \textbf{ do } c, \sigma''\right\rangle \to \sigma'.$$

Finally by using the rule (whtt):

$$\frac{\langle b,\sigma\rangle \to \textbf{true} \quad \langle c,\sigma\rangle \to \sigma'' \quad \left\langle \textbf{while } b \textbf{ do } c, \sigma''\right\rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'}$$

we conclude $\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'$.

6.4 Computational Induction

How are we able to prove properties on fixpoints? To fill this gap we introduce *computational induction*, which applies to a class of properties corresponding to inclusive sets.

Definition 6.10 (Inclusive property). Let $(D,\sqsubseteq)$ be a CPO, let $P \subseteq D$ be a set, we say that P is an *inclusive* set if and only if:

$$(\forall n \in \mathbb{N}, d_n \in P) \Rightarrow \bigsqcup_{n\in\mathbb{N}} d_n \in P$$

A property is *inclusive* if the set of values on which it holds is inclusive.

Intuitively, a set P is inclusive if whenever we form a chain out of elements in P, then the limit of the chain is also in P, i.e., P is inclusive if and only if it is a CPO.

Example 6.9 (Non inclusive property). Let $(\{a,b\}^* \cup \{a,b\}^\infty, \sqsubseteq)$ be a CPO where $x \sqsubseteq y \Leftrightarrow \exists z.\ y = xz$. So the elements of the CPO are sequences of a and b and $x \sqsubseteq y$ iff x is a finite prefix of y. Let us now define the following property:

- $x \in \{a,b\}^* \cup \{a,b\}^\infty$ is fair iff $\nexists y \in \{a,b\}^*.\ x = ya^\infty \ \vee\ x = yb^\infty$

Fairness is the property of an arbiter which does not favor one of two competitors all the times from some point on. Fairness is not inclusive, indeed,

- the sequence a^n is finite and thus fair for any $n \in \mathbb{N}$;

- $\bigsqcup_{n\in\mathbb{N}} a^n = a^\infty$;
- a^∞ is obviously not fair.

Theorem 6.4 (Computational Induction). *Let P be a property, $(D,\sqsubseteq)$ a $CPO_\perp$ and f a monotone and continuous function on it. Then the inference rule:*

$$\frac{P \text{ inclusive} \quad \bot \in P \quad \forall d \in D.\ (d \in P \Rightarrow f(d) \in P)}{\operatorname{fix} f \in P}$$

is sound.

Proof. Given the second and the third premises, it is easy to prove by mathematical induction that $\forall n.\ f^n(\bot) \in P$. Then also $\bigsqcup_{n\in\mathbb{N}} f^n(\bot) \in P$ since P is inclusive and $\operatorname{fix}(f) = \bigsqcup_{n\in\mathbb{N}} f^n(\bot)$. □

Example 6.10 (Computational induction). Let us consider the command

$$w \stackrel{\text{def}}{=} \textbf{while } x \neq 0 \textbf{ do } x := x-1$$

from Example 6.8. We want to prove the property

$$\mathscr{C}[\![\textbf{while } x \neq 0 \textbf{ do } x := x-1]\!]\,\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]$$

By definition:

$$\mathscr{C}[\![w]\!] = \operatorname{fix}\Gamma \quad \text{where} \quad \Gamma \stackrel{\text{def}}{=} \lambda\varphi.\ \lambda\sigma.\ \sigma x \neq 0 \rightarrow \varphi\sigma[^{\sigma x-1}/_x], \sigma$$

Let us define the property:

$$P(\varphi) \stackrel{\text{def}}{=} \forall\sigma,\sigma'.\ (\varphi\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x])$$

we will show that the property is inclusive, that is, taken a chain $\{\varphi_i\}_{i\in\mathbb{N}}$ we have:

$$(\forall i \in \mathbb{N}.\ P(\varphi_i)) \Rightarrow P(\bigsqcup_{i\in\mathbb{N}} \varphi_i)$$

Let us assume $\forall i \in \mathbb{N}.\ P(\varphi_i)$, namely that:

$$\forall i,\sigma,\sigma'.\ \left(\varphi_i\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]\right)$$

We want to prove that

$$\forall\sigma,\sigma'.\left((\bigsqcup_{i\in\mathbb{N}} \varphi_i)\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]\right)$$

Suppose $(\bigsqcup_{i\in\mathbb{N}} \varphi_i)\sigma = \sigma'$. We are left to prove that $\sigma x \geq 0 \wedge \sigma' = \sigma[^0/_x]$. By $(\bigsqcup_{i\in\mathbb{N}} \varphi_i)\sigma = \sigma'$ we have that $\exists k \in \mathbb{N}.\ \varphi_k\sigma = \sigma'$. Then we can conclude the thesis by $P(\varphi_k)$.

We can now use the computational induction:

$$\frac{P \text{ inclusive} \quad P(\bot) \quad \forall \varphi.\, P(\varphi) \Rightarrow P(\Gamma\varphi)}{P(\operatorname{fix}\Gamma)}$$

as $P(\operatorname{fix}\Gamma) = P(\mathscr{C}[\![w]\!])$.

P inclusive:	It has been proved above.
$P(\bot)$:	It is obvious, since $\bot\sigma = \sigma'$ is always false.
$\forall\varphi.\, P(\varphi) \Rightarrow P(\Gamma\varphi)$:	We assume

$$P(\varphi) \stackrel{\text{def}}{=} \forall\sigma,\sigma'.\, (\varphi\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[{}^{0}/_{x}])$$

and we want to prove

$$P(\Gamma\varphi) = \forall\sigma,\sigma'.\, (\Gamma\varphi\sigma = \sigma' \Rightarrow \sigma x \geq 0 \wedge \sigma' = \sigma[{}^{0}/_{x}])$$

We assume the premise

$$\Gamma\varphi\sigma = \big(\sigma x \neq 0 \to \varphi\sigma[{}^{\sigma x-1}/_{x}], \sigma\big) = \sigma'$$

and need to prove that $\sigma x \geq 0 \wedge \sigma' = \sigma[{}^{0}/_{x}]$. There are two cases to consider:

- If $\sigma x = 0$, we have

$$(\sigma x \neq 0 \to \varphi\sigma[{}^{\sigma x-1}/_{x}], \sigma) = \sigma$$

therefore $\sigma' = \sigma$ and trivially

$$\sigma x = 0 \geq 0 \qquad \sigma' = \sigma = \sigma[{}^{0}/_{x}].$$

- If $\sigma x \neq 0$, we have

$$(\sigma x \neq 0 \to \varphi\sigma[{}^{\sigma x-1}/_{x}], \sigma) = \varphi\sigma[{}^{\sigma x-1}/_{x}].$$

Let $\sigma'' = \sigma[{}^{\sigma x-1}/_{x}]$. We exploit $P(\varphi)$ over σ'', σ':

$$\varphi\underbrace{\sigma[{}^{\sigma x-1}/_{x}]}_{\sigma''} = \sigma' \Rightarrow \sigma'' x \geq 0 \wedge \sigma' = \sigma''[{}^{0}/_{x}]$$

we have:

$$\sigma'' x \geq 0 \Leftrightarrow \sigma[{}^{\sigma x-1}/_{x}]x \geq 0 \Leftrightarrow \sigma x \geq 1 \Rightarrow \sigma x \geq 0$$

$$\sigma' = \sigma''[{}^{0}/_{x}] = \sigma[{}^{\sigma x-1}/_{x}][{}^{0}/_{x}] = \sigma[{}^{0}/_{x}]$$

Finally, we can conclude by computational induction that the property P holds for the fixpoint $\mathrm{fix}\,\Gamma$ and thus for the semantics of the command w as $\mathscr{C}[\![w]\!] = \mathrm{fix}\,\Gamma$.

Problems

6.1. The following problems serve to get acquainted with the use of variables in the lambda-notation.

1. Is $\lambda x.\ \lambda x.\ x$ α-convertible to one or more of the following expressions?
   a. $\lambda y.\ \lambda x.\ x$
   b. $\lambda y.\ \lambda x.\ y$
   c. $\lambda y.\ \lambda y.\ y$
   d. $\lambda x.\ \lambda y.\ x$
   e. $\lambda z.\ \lambda w.\ w$
2. Is $((\lambda x.\ \lambda y.\ x)\ y)$ equivalent to one or more of the following expressions?
   a. $\lambda y.\ \lambda y.\ y$
   b. $\lambda y.\ y$
   c. $\lambda y.\ z$
   d. $\lambda z.\ y$
   e. $\lambda x.\ y$

6.2. Prove Theorem 6.2.

6.3. Prove that the commands

$$c \stackrel{\text{def}}{=} x := 0;\ \textbf{if}\ x = 0\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \qquad\qquad c' \stackrel{\text{def}}{=} x := 0;\ c_1$$

are semantically equivalent for any other commands c_1, c_2. Carry out the proof using both the operational semantics and the denotational semantics.

6.4. Prove that the two commands

$$w \stackrel{\text{def}}{=} \textbf{while}\ b\ \textbf{do}\ c \qquad\qquad w' \stackrel{\text{def}}{=} \textbf{while}\ b\ \textbf{do}\ (\textbf{if}\ b\ \textbf{then}\ c\ \textbf{else skip})$$

are equivalent for any b and c using the denotational semantics.

6.5. Prove that $\mathscr{C}[\![\textbf{while}\ \mathit{true}\ \textbf{do skip}]\!] = \mathscr{C}[\![\textbf{while}\ \mathit{true}\ \textbf{do}\ x := x+1]\!]$.

6.6. Prove that $\mathscr{C}[\![\textbf{while}\ x \neq 0\ \textbf{do}\ x := 0]\!] = \mathscr{C}[\![x := 0]\!]$.

6.7. Prove that

$$\mathscr{C}[\![\textbf{while}\ x = 0\ \textbf{do skip}]\!] = \mathscr{C}[\![\textbf{if}\ x = 0\ \textbf{then}\ (\textbf{while true do}\ x := 0)\ \textbf{else skip}]\!].$$

6.8. Introduce in **IMP** the command

$$\textbf{repeat } n \textbf{ times } c$$

with n natural number, instead of the command **while**. Its denotational semantics is

$$\mathscr{C}[\![\textbf{repeat } n \textbf{ times } c]\!]\,\sigma = (\mathscr{C}[\![c]\!])^n \sigma$$

1. Define the operational semantics of the new command.
2. Extend the proof of equivalence of the operational and denotational semantics of IMP to take into account the new command.
3. Prove that the execution of every command terminates.

6.9. Add to IMP the command

$$\textbf{reset } x \textbf{ in } c$$

with the following informal meaning: execute the command c in the state where x is reset to 0, then after the execution of c reassign to location x its original value.

1. Define the operational semantics of the new command.
2. Define the denotational semantics of the new command.
3. Extend the proof of equivalence of the operational and denotational semantics of IMP to take into account the new command.

6.10. Add to **IMP** the command

$$\textbf{do } c \textbf{ undoif } b$$

with the following informal meaning: execute c; if after the execution of c the boolean expression b is satisfied, then go back to the state before the execution of c.

1. Define the operational semantics of the new command.
2. Define the denotational semantics of the new command.
3. Extend the proof of equivalence of the operational and denotational semantics of IMP to take into account the new command.

6.11. Extend IMP with the command

$$\textbf{try } c_1 = c_2 \textbf{ else } c_3$$

that returns the store obtained by computing c_1 if it coincides with the one obtained by computing c_2; if they differ returns the store obtained by computing c_3; it diverges otherwise.

1. Define the operational semantics of the new command.
2. Define the denotational semantics of the new command.
3. Extend the proof of correspondence between the operational and the denotational semantics.

6.12. Consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } y > 0 \textbf{ do } (r := r \times x \,;\, y := y - 1)$$

Compute the denotational semantics $\mathscr{C}[\![w]\!] = \text{fix } \Gamma$.
Hint: Prove that letting $\varphi_n \stackrel{\text{def}}{=} \Gamma^n \bot_{\Sigma \to \Sigma_\bot}$ it holds $\forall n \geq 1$

$$\varphi_n = \lambda\sigma.\ (\sigma y > 0) \to (\ (\sigma y \geq n) \to \bot_{\Sigma_\bot}\ ,\ \sigma[\sigma r \times (\sigma x)^{\sigma y}/r, 0/y]\ )\ ,\ \sigma$$

6.13. Consider the IMP command

$$w \stackrel{\text{def}}{=} \textbf{while } x \neq 0 \textbf{ do } (x := x - 1 \,;\, y := y + 1)$$

Prove, using Scott computational induction, that for all σ, σ' we have:

$$\mathscr{C}[\![w]\!]\,\sigma = \sigma' \quad \Rightarrow \quad \sigma(x) \geq 0 \wedge \sigma' = \sigma[\sigma(x) + \sigma(y)/y, 0/x]$$