Type-theoretic functional semantics

Yves Bertot, Venanzio Capretta, and Kuntal Das Barman

Project LEMME, INRIA Sophia Antipolis,
Yves.Bertot,Venanzio.Capretta,Kuntal.Das_Barman@sophia.inria.fr

Abstract. We describe the operational and denotational semantics of a small imperative language in type theory with inductive and recursive definitions. The operational semantics is given by natural inference rules, implemented as an inductive relation. The realization of the denotational semantics is more delicate: The nature of the language imposes a few difficulties on us. First, the language is Turing-complete, and therefore the interpretation function we consider is necessarily partial. Second, the language contains strict sequential operators, and therefore the function necessarily exhibits nested recursion. Our solution combines and extends recent work by the authors and others on the treatment of general recursive functions and partial and nested recursive functions. The first new result is a technique to encode the approach of Bove and Capretta for partial and nested recursive functions in type theories that do not provide simultaneous induction-recursion. A second result is a clear understanding of the characterization of the definition domain for general recursive functions, a key aspect in the approach by iteration of Balaa and Bertot. In this respect, the work on operational semantics is a meaningful example, but the applicability of the technique should extend to other circumstances where complex recursive functions need to be described formally.

1 Introduction

There are two main kinds of semantics for programming languages.

Operational semantics consists in describing the steps of the computation of a program by giving formal rules to derive judgments of the form $\langle p, a\rangle \rightsquigarrow r$, to be read as "the program p, when applied to the input a, terminates and produces the output r".

Denotational semantics consists in giving a mathematical meaning to data and programs, specifically interpreting data (input and output) as elements of certain domains and programs as functions on those domains; then the fact that the program p applied to the input a gives r as result is expressed by the equality $[\![p]\!]([\![a]\!]) = [\![r]\!]$, where $[\![-]\!]$ is the interpretation.

Our main goal is to develop operational and denotational semantics inside type theory, to implement them in the proof-assistant **Coq** [11], and to prove their main properties formally. The most important result in this respect is a soundness and completeness theorem stating that operational and denotational semantics agree.

The implementation of operational semantics is straightforward: The derivation system is formalized as an inductive relation whose constructors are direct rewording of the derivation rules.

The implementation of denotational semantics is much more delicate. Traditionally, programs are interpreted as partial functions, since they may diverge on certain inputs. However, all function of type theory are total. The problem of representing partial functions in a total setting has been the topic of recent work by several authors [7, 5, 12, 4, 13]. A standard way of solving it is to restrict the domain to those elements that are interpretations of inputs on which the program terminates and then interpret the program as a total function on the restricted domain. There are different approaches to the characterization of the restricted domain. Another approach is to lift the co-domain by adding a bottom element, this approach is not applicable here because the expressive power of the programming language imposes a limit to computable functions.

Since the domain depends on the definition of the function, a direct formalization needs to define domain and function simultaneously. This is not possible in standard type theory, but can be achieved if we extend it with Dybjer's simultaneous induction-recursion [6]. This is the approach adopted in [4].

An alternative way, adopted by Balaa and Bertot in [1], sees the partial function as a fixed point of an operator F that maps total functions to total functions. It can be approximated by a finite number of iterations of F on an arbitrary base function. The domain can be defined as the set of those elements for which the iteration of F stabilizes after a finite number of steps independently of the base function.

The drawback of the approach of [4] is that it is not viable in standard type theories (that is, without Dybjer's schema). The drawback of the approach of [1] is that the defined domain is the domain of a fixed point of F that is not in general the least fixed point. This maybe correct for lazy functional languages (call by name), but is incorrect for strict functional languages (call by value), where we need the least fixed point. The interpretation of an imperative programming language is essentially strict and therefore the domain is too large: The function is defined for values on which the program does not terminate.

Here we combine the two approaches of [4] and [1] by defining the domain in a way similar to that of [4], but disentangling the mutual dependence of domain and function by using the iteration of the functional F with a variable index in place of the yet undefined function.

We claim two main results. First, we develop denotational semantics in type theory. Second, we model the accessibility method in a weaker system, that is, without using simultaneous induction-recursion.

Here is the structure of the paper.

In Section 2 we define the simple imperative programming language IMP. We give an informal description of its operational and denotational semantics. We formalize the operational semantics by an inductive relation. We explain the difficulties related to the implementation of the denotational semantics.

In Section 3 we describe the iteration method. We point out the difficulty in characterizing the domain of the interpretation function by the convergence of the iterations.

In Section 4 we give the denotational semantics using the accessibility method. We combine it with the iteration technique to formalize nested recursion without the use of simultaneous induction-recursion.

All the definitions have been implemented in **Coq** and all the results proved formally in it. We use here an informal mathematical notation, rather than giving **Coq** code. There is a direct correspondence between this notation and the **Coq** formalization. Using the **PCoq** graphical interface (available on the web at the location `http://www-sop.inria.fr/lemme/pcoq/index.html`), we also implemented some of this more intuitive notation. The **Coq** files of the development are on the web at `http://www-sop.inria.fr/lemme/Kuntal.Das_Barman/imp/`.

2 IMP and its semantics

Winskel [14] presents a small programming language IMP with *while* loops. IMP is a simple imperative language with integers, truth values true and false, memory locations to store the integers, arithmetic expressions, boolean expressions and commands. The formation rules are

arithmetic expressions: $a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1$;
boolean expressions: $b ::= \mathsf{true} \mid \mathsf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \vee b_1 \mid b_0 \wedge b_1$;
commands: $c ::= \mathsf{skip} \mid X \leftarrow a \mid c_0; c_1 \mid \mathsf{if}\ b\ \mathsf{then}\ c_0\ \mathsf{else}\ c_1 \mid \mathsf{while}\ b\ \mathsf{do}\ c$

where n ranges over integers, X ranges over locations, a ranges over arithmetic expressions, b ranges over boolean expressions and c ranges over commands.

We formalize it in **Coq** by three inductive types AExp, BExp, and Command.

For simplicity, we work with natural numbers instead of integers. We do so, as it has no significant importance in the semantics of IMP. Locations are also represented by natural numbers. One should not confuse the natural number denoting a location with the natural number contained in the location. Therefore, in the definition of AExp, we denote the constant value n by $\mathsf{Num}(n)$ and the memory location with address v by $\mathsf{Loc}(v)$

We see commands as state transformers, where a state is a map from memory locations to natural numbers. The map is in general partial, indeed it is defined only on a finite number of locations. Therefore, we can represent a state as a list of bindings between memory locations and values. If the same memory location is bound twice in the same state, the most recent binding, that is, the leftmost one, is the valid one.

$$\begin{array}{l}\mathsf{State}\colon \mathbf{Set}\\ []\colon \mathsf{State}\\ [\cdot \mapsto \cdot, \cdot]\colon \mathbb{N} \to \mathbb{N} \to \mathsf{State} \to \mathsf{State}\end{array}$$

The state $[v \mapsto n, s]$ is the state s with the content of the location v replaced by n.

Operational semantics consists in three relations giving meaning to arithmetic expressions, boolean expressions, and commands. Each relation has three arguments: The expression or command, the state in which the expression is evaluated or the command executed, and the result of the evaluation or execution.

$$(\langle\cdot,\cdot\rangle_{\mathsf{A}} \rightsquigarrow \cdot)\colon \mathsf{AExp} \to \mathsf{State} \to \mathbb{N} \to \mathbf{Prop}$$
$$(\langle\cdot,\cdot\rangle_{\mathsf{B}} \rightsquigarrow \cdot)\colon \mathsf{BExp} \to \mathsf{State} \to \mathbb{B} \to \mathbf{Prop}$$
$$(\langle\cdot,\cdot\rangle_{\mathsf{C}} \rightsquigarrow \cdot)\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State} \to \mathbf{Prop}$$

For arithmetic expressions we have that constants are interpreted in themselves, that is, we have axioms of the form

$$\langle \mathsf{Num}(n), \sigma\rangle_{\mathsf{A}} \rightsquigarrow n$$

for every $n\colon\mathbb{N}$ and $\sigma\colon\mathsf{State}$. Memory locations are interpreted by looking up their values in the state. Consistently with the spirit of operational semantics, we define the lookup operation by derivation rules rather than by a function.

$$\frac{(\mathsf{value\_ind}\ \sigma\ v\ n)}{\langle \mathsf{Loc}(v), \sigma\rangle_{\mathsf{A}} \rightsquigarrow n}$$

where

$\mathsf{value\_ind}\colon \mathsf{State} \to \mathbb{N} \to \mathbb{N} \to \mathbf{Prop}$
$\mathsf{no\_such\_location}\colon (v\colon\mathbb{N})(\mathsf{value\_ind}\ []\ v\ 0)$
$\mathsf{first\_location}\colon (v, n\colon\mathbb{N}; \sigma\colon\mathsf{State})(\mathsf{value\_ind}\ [v \mapsto n, \sigma]\ v\ n)$
$\mathsf{rest\_locations}\colon (v, v', n, n'\colon\mathbb{N}; \sigma\colon\mathsf{State})$
$\qquad v \neq v' \to (\mathsf{value\_ind}\ \sigma\ v\ n) \to (\mathsf{value\_ind}\ [v' \mapsto n', \sigma]\ v\ n)$

Notice that we assign the value 0 to empty locations, rather that leaving them undefined. This corresponds to giving a default value to uninitialized variables rather than raising an exception.

The operations are interpreted in the obvious way, for example,

$$\frac{\langle a_0, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_0 \quad \langle a_1, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_1}{\langle a_0 + a_1, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_0 + n_1}$$

where the symbol $+$ is overloaded: $a_0 + a_1$ denotes the arithmetic expression obtained by applying the symbol $+$ to the expressions a_0 and a_1, $n_0 + n_1$ denotes the sum of the natural numbers n_0 and n_1.

In short, the operational semantics of arithmetic expressions is defined by the inductive relation

$(\langle\cdot,\cdot\rangle_{\mathsf{A}} \rightsquigarrow \cdot)\colon \mathsf{AExp} \to \mathsf{State} \to \mathbb{N} \to \mathbf{Prop}$
$\mathsf{eval\_Num}\colon (n\colon\mathbb{N}; \sigma\colon\mathsf{State})(\langle \mathsf{Num}(n), \sigma\rangle_{\mathsf{A}} \rightsquigarrow n)$
$\mathsf{eval\_Loc}\colon (v, n\colon\mathbb{N}; \sigma\colon\mathsf{State})(\mathsf{value\_ind}\ \sigma\ v\ n) \to (\langle \mathsf{Loc}(v), \sigma\rangle_{\mathsf{A}} \rightsquigarrow n)$
$\mathsf{eval\_Plus}\colon (a_0, a_1\colon\mathsf{AExp}; n_0, n_1\colon\mathbb{N}; \sigma\colon\mathsf{State})$
$\qquad (\langle a_0, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_0) \to (\langle a_1, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_0) \to$
$\qquad (\langle a_0 + a_1, \sigma\rangle_{\mathsf{A}} \rightsquigarrow n_0 + n_1)$
$\mathsf{eval\_Minus}\colon \cdots$
$\mathsf{eval\_Mult}\colon \cdots$

For the subtraction case the cutoff difference is used, that is, $n - m = 0$ if $n \leq m$.

The definition of the operational semantics of boolean expressions is similar and we omit it.

The operational semantics of commands specifies how a command maps states to states. skip is the command that does nothing, therefore it leaves the state unchanged.

$$\frac{}{\langle \mathsf{skip}, \sigma \rangle_{\mathsf{C}} \rightsquigarrow \sigma}$$

The assignment $X \leftarrow a$ evaluates the expression a and then updates the contents of the location X to the value of a.

$$\frac{\langle a, \sigma \rangle_{\mathsf{A}} \rightsquigarrow n \qquad \sigma_{[X \mapsto n]} \rightsquigarrow \sigma'}{\langle X \leftarrow a, \sigma \rangle_{\mathsf{C}} \rightsquigarrow \sigma'}$$

where $\sigma_{[X \mapsto n]} \rightsquigarrow \sigma'$ asserts that σ' is the state obtained by changing the contents of the location X to n in σ. It could be realized by simply $\sigma' = [X \mapsto n, \sigma]$. This solution is not efficient, since it duplicates assignments of existing locations and it would produce huge states during computation. A better solution is to look for the value of X in σ and change it.

$(\cdot_{[\cdot \mapsto \cdot]} \rightsquigarrow \cdot)\colon \mathsf{State} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathsf{State} \rightarrow \mathbf{Prop}$
$\mathsf{update\_no\_location}\colon (v, n\colon \mathbb{N})([\,]_{[v \mapsto n]} \rightsquigarrow [\,])$
$\mathsf{update\_first}\colon (v, n_1, n_2\colon \mathbb{N}; \sigma\colon \mathsf{State})([v \mapsto n_1, \sigma]_{[v \mapsto n_2]} \rightsquigarrow [v \mapsto n_2, \sigma])$
$\mathsf{update\_rest}\colon (v_1, v_2, n_1, n_2\colon \mathbb{N}; \sigma_1, \sigma_2\colon \mathbb{N}) v_1 \neq v_2 \rightarrow$
$\qquad (\sigma_{1[v_2 \mapsto n_2]} \rightsquigarrow \sigma_2) \rightarrow ([v_1 \mapsto n_1, \sigma_1]_{[v_2 \mapsto n_2]} \rightsquigarrow [v_1 \mapsto n_1, \sigma_2])$

Notice that we require a location to be already defined in the state to update it. If we try to update a location not present in the state, we leave the state unchanged. This corresponds to requiring that all variables are explicitly initialized before the execution of the program. If we use an uninitialized variable in the program, we do not get an error message, but an anomalous behaviour: The value of the variable is always zero.

Evaluating a sequential composition $c_1; c_2$ on a state σ consists in evaluating c_1 on σ, obtaining a new state σ_1, and then evaluating c_2 on σ_1 to obtain the final state σ_2.

$$\frac{\langle c_1, \sigma \rangle_{\mathsf{C}} \rightsquigarrow \sigma_1 \quad \langle c_2, \sigma_1 \rangle_{\mathsf{C}} \rightsquigarrow \sigma_2}{\langle c_1; c_2, \sigma \rangle_{\mathsf{C}} \rightsquigarrow \sigma_2}$$

Evaluating conditionals uses two rules. In both rules, we evaluate the boolean expression b, but they differ on the value returned by this step and the sub-instruction that is executed.

$$\frac{\langle b, \sigma \rangle_{\mathsf{B}} \rightsquigarrow \mathsf{true} \quad \langle c_1, \sigma \rangle_{\mathsf{C}} \rightsquigarrow \sigma_1}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma \rangle_{\mathsf{C}} \rightsquigarrow \sigma_1} \qquad \frac{\langle b, \sigma \rangle_{\mathsf{B}} \rightsquigarrow \mathsf{false} \quad \langle c_2, \sigma \rangle_{\mathsf{C}} \rightsquigarrow \sigma_2}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma \rangle_{\mathsf{C}} \rightsquigarrow \sigma_2}$$

As for conditionals, we have two rules for *while* loops. If b evaluates to true, c is evaluated on σ to produce a new state σ', on which the loop is evaluated recursively. If b evaluates to false, we exit the loop leaving the state unchanged.

$$\frac{\langle b, \sigma \rangle_{\mathsf{B}} \rightsquigarrow \mathsf{true} \quad \langle c, \sigma \rangle_{\mathsf{C}} \rightsquigarrow \sigma' \quad \langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma' \rangle_{\mathsf{C}} \rightsquigarrow \sigma''}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma \rangle_{\mathsf{C}} \rightsquigarrow \sigma''} \qquad \frac{\langle b, \sigma \rangle_{\mathsf{B}} \rightsquigarrow \mathsf{false}}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma \rangle_{\mathsf{C}} \rightsquigarrow \sigma}$$

The above rules can be formalized in **Coq** in a straightforward way by an inductive relation.

$$
\begin{array}{l}
\langle\cdot,\cdot\rangle_{\mathsf{C}} \rightsquigarrow \cdot\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State} \to \mathbf{Prop}\\
\mathsf{eval\_skip}\colon (\sigma\colon \mathsf{State})(\langle \mathsf{skip}, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma)\\
\mathsf{eval\_assign}\colon (\sigma,\sigma'\colon \mathsf{State}; v,n\colon \mathbb{N}; a\colon \mathsf{AExp})\\
\qquad (\langle a,\sigma\rangle_{\mathsf{A}} \rightsquigarrow n) \to (\sigma_{[v\mapsto n]} \rightsquigarrow \sigma') \to (\langle v \leftarrow a, \sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma')\\
\mathsf{eval\_scolon}\colon (\sigma,\sigma_1,\sigma_2\colon \mathsf{State}; c_1,c_2\colon \mathsf{Command})\\
\qquad (\langle c_1,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_1) \to (\langle c_2,\sigma_1\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2) \to (\langle c_1;c_2,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2)\\
\mathsf{eval\_if\_true}\colon (b\colon \mathsf{BExp}; \sigma,\sigma_1\colon \mathsf{State}; c_1,c_2\colon \mathsf{Command})\\
\qquad (\langle b,\sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{true}) \to (\langle c_1,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_1) \to\\
\qquad (\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_1)\\
\mathsf{eval\_if\_false}\colon (b\colon \mathsf{BExp}; \sigma,\sigma_2\colon \mathsf{State}; c_1,c_2\colon \mathsf{Command})\\
\qquad (\langle b,\sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{false}) \to (\langle c_2,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2) \to\\
\qquad (\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma_2)\\
\mathsf{eval\_while\_true}\colon (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma,\sigma',\sigma''\colon \mathsf{State})\\
\qquad (\langle b,\sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{true}) \to (\langle c,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma') \to\\
\qquad (\langle \mathsf{while}\ b\ \mathsf{do}\ c,\sigma'\rangle_{\mathsf{C}} \rightsquigarrow \sigma'') \to (\langle \mathsf{while}\ b\ \mathsf{do}\ c,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma'')\\
\mathsf{eval\_while\_false}\colon (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma\colon \mathsf{State})\\
\qquad (\langle b,\sigma\rangle_{\mathsf{B}} \rightsquigarrow \mathsf{false}) \to (\langle \mathsf{while}\ b\ \mathsf{do}\ c,\sigma\rangle_{\mathsf{C}} \rightsquigarrow \sigma)
\end{array}
$$

For the rest of the paper we leave out the subscripts A, B, and C in $\langle\cdot,\cdot\rangle \rightsquigarrow \cdot$.

3 Functional interpretation

Denotational semantics consists in interpreting program evaluation as a function rather than as a relation. We start by giving a functional interpretation to expression evaluation and state update. This is quite straightforward, since we can use structural recursion on expressions and states. For example, the interpretation function on arithmetic expressions is defined as

$$
\begin{array}{l}
[\![\cdot]\!]\colon \mathsf{AExp} \to \mathsf{State} \to \mathbb{N}\\
[\![\mathsf{Num}(n)]\!]_\sigma := n\\
[\![\mathsf{Loc}(v)]\!]_\sigma := \mathsf{value\_rec}(\sigma, v)\\
[\![a_0 + a_1]\!]_\sigma := [\![a_0]\!]_\sigma + [\![a_1]\!]_\sigma\\
[\![a_0 - a_1]\!]_\sigma := [\![a_0]\!]_\sigma - [\![a_1]\!]_\sigma\\
[\![a_0 * a_1]\!]_\sigma := [\![a_0]\!]_\sigma \cdot [\![a_1]\!]_\sigma
\end{array}
$$

where $\mathsf{value\_rec}(\cdot,\cdot)$ is the function giving the contents of a location in a state, defined by recursion on the structure of the state. It differs from value_ind because it is a function, not a relation; value_ind is its graph. We can now prove that this interpretation function agrees with the operational semantics given by the inductive relation $\langle\cdot,\cdot\rangle \rightsquigarrow \cdot$ (all the lemmas and theorems given below have been checked in a computer-assisted proof).

Lemma 1. $\forall\sigma\colon \mathsf{State}.\forall a\colon \mathsf{AExp}.\forall n\colon \mathbb{N}.\langle\sigma, a\rangle \rightsquigarrow n \Leftrightarrow [\![a]\!]_\sigma = n$.

In the same way, we define the interpretation of boolean expressions

$$[\![\cdot]\!]\colon \mathsf{BExp} \to \mathsf{State} \to \mathbb{B}$$

and prove that it agrees with the operational semantics.

Lemma 2. $\forall \sigma\colon \mathsf{State}.\forall b\colon \mathsf{BExp}.\forall t\colon \mathbb{B}.\langle \sigma, b\rangle \rightsquigarrow t \Leftrightarrow [\![a]\!]_\sigma = t.$

We overload the Scott brackets $[\![\cdot]\!]$ to denote the interpretation function both on arithmetic and boolean expressions (and later on commands).

Similarly, we define the update function

$$\cdot[\cdot/\cdot]\colon \mathsf{State} \to \mathbb{N} \to \mathbb{N} \to \mathsf{State}$$

and prove that it agrees with the update relation

Lemma 3. $\forall \sigma, \sigma'\colon \mathsf{State}.\forall v, n\colon \mathbb{N}.\sigma_{[v\mapsto n]} \rightsquigarrow \sigma' \Leftrightarrow \sigma[n/v] = \sigma'.$

The next step is to define the interpretation function $[\![\cdot]\!]$ on commands. Unfortunately, this cannot be done by structural recursion, as for the cases of arithmetic and boolean expressions. Indeed we should have

$$\begin{array}{l}
[\![\cdot]\!]\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State} \\
[\![\mathsf{skip}]\!]_\sigma := \sigma \\
[\![X \leftarrow a]\!]_\sigma := \sigma[[\![a]\!]_\sigma/X] \\
[\![c_1; c_2]\!]_\sigma := [\![c_1]\!]_{[\![c_2]\!]_\sigma} \\
[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]_\sigma := \begin{cases} [\![c_1]\!]_\sigma \ \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{true} \\ [\![c_2]\!]_\sigma \ \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{false} \end{cases} \\
[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_\sigma := \begin{cases} [\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_{[\![c]\!]_\sigma} & \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{true} \\ \sigma & \mathsf{if}\ [\![b]\!]_\sigma = \mathsf{false} \end{cases}
\end{array}$$

but in the clause for *while* loops the interpretation function is called on the same argument if the boolean expression evaluates to true. Therefore, the argument of the recursive call is not structurally smaller than the original argument.

So, it is not possible to associate a structural recursive function to the instruction execution relation as we did for the lookup, update, and expression evaluation relations. The execution of *while* loops does not respect the pattern of structural recursion and termination cannot be ensured: for good reasons too, since the language is Turing complete. However, we describe now a way to work around this problem.

3.1 The iteration technique

A function representation of the computation can be provided in a way that respects typing and termination if we don't try to describe the execution function itself but the *second order function of which the execution function is the least* *fixed point*. This function can be defined in type theory by cases on the structure of the command.

$$\begin{array}{l}
\mathsf{F}\colon (\mathsf{Command} \to \mathsf{State} \to \mathsf{State}) \to \mathsf{Command} \to \mathsf{State} \to \mathsf{State}\\
(\mathsf{F}\ f\ \mathsf{skip}\ \sigma) := \sigma\\
(\mathsf{F}\ f\ (X \leftarrow a)\ \sigma) := \sigma[[\![a]\!]_\sigma/X]\\
(\mathsf{F}\ f\ (c_1; c_2)\ \sigma) := (f\ c_2\ (f\ c_1\ \sigma))\\
(\mathsf{F}\ f\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \sigma) := \begin{cases} (f\ c_1\ \sigma)\ \text{if}\ [\![b]\!]_\sigma = \mathsf{true}\\ (f\ c_2\ \sigma)\ \text{if}\ [\![b]\!]_\sigma = \mathsf{false}\end{cases}\\
(\mathsf{F}\ f\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ \sigma) := \begin{cases} (f\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ (f\ c\ \sigma)) & \text{if}\ [\![b]\!]_\sigma = \mathsf{true}\\ \sigma & \text{if}\ [\![b]\!]_\sigma = \mathsf{false}\end{cases}
\end{array}$$

Intuitively, writing the function F is exactly the same as writing the recursive execution function, except that the function being defined is simply replaced by a bound variable (here f). In other words, we replace recursive calls with calls to the function given in the bound variable f.

The function F describes the computations that are performed at each iteration of the execution function and the execution function performs the same computation as the function F when the latter is repeated *as many times as needed*. We can express this with the following theorem.

Theorem 1 (eval_com_ind_to_rec)**.**

$$\begin{array}{l}
\forall c\colon \mathsf{Command}.\forall \sigma_1, \sigma_2\colon \mathsf{State}.\\
\langle c, \sigma_1\rangle \rightsquigarrow \sigma_2 \Rightarrow \exists k\colon \mathbb{N}.\forall g\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}.(\mathsf{F}^k\ g\ c\ \sigma_1) = \sigma_2
\end{array}$$

where we used the following notation

$$\mathsf{F}^k = (\mathsf{iter}\ (\mathsf{Command} \to \mathsf{State} \to \mathsf{State})\ \mathsf{F}\ k) = \lambda g.\underbrace{(\mathsf{F}\ (\mathsf{F}\ \cdots\ (\mathsf{F}}_{k\ times}\ g)\ \cdots\ ))$$

definable by recursion on k,

$$\begin{array}{l}
\mathsf{iter}\colon (A\colon \mathbf{Set})(A \to A) \to \mathbb{N} \to A \to A\\
(\mathsf{iter}\ A\ f\ 0\ a) := a\\
(\mathsf{iter}\ A\ f\ (\mathsf{S}\ k)\ a) := (f\ (\mathsf{iter}\ A\ f\ k\ a)).
\end{array}$$

Proof. Easily proved using the theorems described in the previous section and an induction on the derivation of $\langle c, \sigma_1\rangle \rightsquigarrow \sigma_2$: This kind of induction is also called *rule induction* in [14]. □

3.2 Extracting an interpreter

The **Coq** system provides an *extraction* facility [10], which makes it possible to produce a version of any function defined in type theory that is written in a functional programming language's syntax, usually the **OCaml** implementation of ML. In general, the extraction facility performs some complicated program manipulations, to ensure that arguments of functions that have only a logical content are not present anymore in the extracted code. For instance, a division function is a 3-argument function inside type theory: The first argument is the number to be divided, the second is the divisor, and the third is a proof that the second is non-zero. In the extracted code, the function takes only two arguments: The extra argument does not interfere with the computation and its presence cannot help ensuring typing, since the programming language's type system is too weak to express this kind of details.

The second order function F and the other recursive functions can also be extracted to ML programs using this facility. However, the extraction process is a simple translation process in this case, because none of the various function actually takes proof arguments.

To perform complete execution of programs, using the ML translation of F, we have the possibility to compute using the extracted version of the iter function. However, we need to guess the right value for the k argument. One way to cope with this is to create an artificial "infinite" natural number, that will always appear to be big enough, using the following recursive data definition:

$$\mathsf{letrec}\ \omega = (\mathsf{S}\ \omega).$$

This definition does not correspond to any natural number that can be manipulated inside type theory: It is an infinite tree composed only of S constructors. In memory, it corresponds to an S construct whose only field points to the whole construct: It is a loop.

Using the extracted iter with ω is not very productive. Since ML evaluates expressions with a call-by-value strategy, evaluating

$$(\mathsf{iter}\ \mathsf{F}\ g\ \omega\ c\ \sigma)$$

imposes that one evaluates

$$(\mathsf{F}\ (\mathsf{iter}\ \mathsf{F}\ g\ \omega)\ c\ \sigma)$$

which in turn imposes that one evaluates

$$(\mathsf{F}\ (\mathsf{F}\ (\mathsf{iter}\ \mathsf{F}\ g\ \omega))\ c\ \sigma)$$

and so on. Recursion unravels unchecked and this inevitably ends with a stack overflow error. However, it is possible to use a variant of the iteration function that avoids this infinite looping, even for a call-by-value evaluation strategy. The trick is to η-expand the expression that provokes the infinite loop, to force the evaluator to stop until an extra value is provided, before continuing to evaluate the iterator. The expression to define this variant is as follows:

$$\begin{array}{l}
\mathsf{iter}': (A, B\colon \mathbf{Set})((A \to B) \to A \to B) \to \mathbb{N} \to (A \to B) \to A \to B \\
(\mathsf{iter}'\ A\ B\ G\ 0\ f) := f \\
(\mathsf{iter}'\ A\ B\ G\ (\mathsf{S}\ k)\ f) := (G\ \lambda a\colon A.(\mathsf{iter}'\ A\ B\ G\ k\ f\ a))
\end{array}$$

Obviously, the expression $\lambda a\colon A.(\mathsf{iter}'\ A\ B\ G\ k\ f\ a)$ is η-equivalent to the expression $(\mathsf{iter}'\ A\ B\ G\ k\ f)$. However, for call-by-value evaluation the two expression are not equivalent, since the λ-expression in the former stops the evaluation process that would lead to unchecked recursion in the latter.

With the combination of iter' and ω we can now execute any terminating program without needing to compute in advance the number of iterations of F that will be needed. In fact, ω simply acts as a *natural number that is big enough*. We obtain a functional interpreter for the language we are studying, that is (almost) proved correct with respect to the inductive definition $\langle\cdot,\cdot\rangle \leadsto \cdot$.

Still, the use of ω as a natural number looks rather like a dirty trick: This piece of data cannot be represented in type theory, and we are taking advantage of important differences between type theory and ML's memory and computation models: How can we be sure that what we proved in type theory is valid for what we execute in ML? A first important difference is that, while executions of iter or iter' are sure to terminate in type theory, $(\mathsf{iter}'\ \mathsf{F}\ \omega\ g)$ will loop if the program passed as argument is a looping program.

The purpose of using ω and iter' is to make sure that F will be called as many times as needed when executing an arbitrary program, with the risk of non-termination when the studied program does not terminate. This can be done more easily by using a *fixpoint* function that simply returns the fixpoint of F. This fixpoint function is defined in ML by

$$\mathsf{letrec}\ (\mathsf{fix}\ f) = f(\lambda x.\mathsf{fix}\ f\ x).$$

Obviously, we have again used the trick of η-expansion to avoid looping in the presence of a call-by-value strategy. With this fix function, the interpreter function is

$$\begin{array}{l}\mathsf{interp}\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}\\ \mathsf{interp} := \mathsf{fix}\ \mathsf{F}.\end{array}$$

To obtain a usable interpreter, it is then only required to provide a parser and printing functions to display the results of evaluation. This shows how we can build an interpreter for IMP in ML. But we realized it by using some tricks of functional programming that are not available in type theory. If we want to define an interpreter for IMP in type theory, we have to find a better solution to the problem of partiality.

3.3 Characterizing terminating programs

Theorem 1 gives one direction of the correspondence between operational semantics and functional interpretation through the iteration method. To complete the task of formalizing denotational semantics, we need to define a function in type theory that interprets each command. As we already remarked, this function cannot be total, therefore we must first restrict its domain to the terminating commands. This is done by defining a predicate D over commands and states, and then defining the interpretation function $[\![\cdot]\!]$ on the domain restricted by this predicate. Theorem 1 suggests the following definition:

$$\begin{array}{l}D\colon \mathsf{Command} \to \mathsf{State} \to \mathbf{Prop}\\ (D\ c\ \sigma) := \exists k\colon \mathbb{N}.\forall g_1, g_2\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}.\\ \qquad\qquad (\mathsf{F}^k\ g_1\ c\ \sigma) = (\mathsf{F}^k\ g_2\ c\ \sigma).\end{array}$$

Unfortunately, this definition is too weak. In general, such an approach cannot be used to characterize terminating "nested" iteration. This is hard to see in the case of the IMP language, but it would appear plainly if one added an exception instruction with the following semantics:

$$\langle \mathsf{exception}, \sigma\rangle \rightsquigarrow [].$$

Intuitively, the programmer could use this instruction to express that an exceptional situation has been detected, but all information about the execution state would be destroyed when this instruction is executed.

With this new instruction, there are some commands and states for which the predicate D is satisfied, but whose computation does not terminate.

$$c := \mathsf{while\ true\ do\ skip; exception}.$$

It is easy to see that for any state σ the computation of c on σ does not terminate. In terms of operational semantics, for no state σ' is the judgment $\langle c, \sigma\rangle \rightsquigarrow \sigma'$ derivable.

However, $(D\ c\ \sigma)$ is provable, because $(\mathsf{F}^k\ g\ c\ \sigma) = []$ for any $k > 1$.

In the next section we work out a stronger characterization of the domain of commands, that turn out to be the correct one in which to interpret the operational semantics.

4 The Accessibility predicate

A common way to represent partial functions in type theory is to restrict their domain to those arguments on which they terminate. A partial function $f: A \rightharpoonup B$ is then represented by first defining a predicate $D_f: A \to \mathbf{Prop}$ that characterizes the domain of f, that is, the elements of A on which f is defined; and then formalizing the function itself as $f: (\Sigma x: A.(D_f\ x)) \to B$, where $\Sigma x: A.(D_f\ x)$ is the type of pairs $\langle x, h\rangle$ with $x: A$ and $h: (D_f\ x)$.

The predicate D_f cannot be defined simply by saying that it is the domain of definition of f, since, in type theory, we need to define it before we can define f. Therefore, D_f must be given before and independently from f. One way to do it is to characterize D_f as the predicate satisfied by those elements of A for which the iteration technique converges to the same value for every initial function. This is a good definition when we try to model lazy functional programming languages, but, when interpreting strict programming languages or imperative languages, we find that this predicate would be too weak, being satisfied by elements for which the associated program diverges, as we have seen at the end of the previous section.

Sometimes the domain of definition of a function can be characterized independently of the function by an inductive predicate called *accessibility* [?,7,5, 3]. This simply states that an element of a can be proved to be in the domain if the application of f on a calls f recursively on elements that have already been proved to be in the domain. For example, if in the recursive definition of f there is a clause of the form

$$f(e) := \cdots f(e_1) \cdots f(e_2) \cdots$$

and a matches e, that is, there is a substitution of variables ρ such that $a = \rho(e)$; then we add a clause to the inductive definition of Acc of type

$$\mathsf{Acc}(e_1) \to \mathsf{Acc}(e_2) \to \mathsf{Acc}(e).$$

This means that to prove that a is in the domain of f, we must first prove that $\rho(e_1)$ and $\rho(e_2)$ are in the domain.

This definition does not always work. In the case of nested recursive calls of the function, we cannot eliminate the reference to f in the clauses of the inductive definition Acc. If, for example, the recursive definition of f contains a clause of the form

$$f(e) := \cdots f(f(e')) \cdots$$

then the corresponding clause in the definition of Acc should be

$$\mathsf{Acc}(e') \to \mathsf{Acc}(f(e')) \to \mathsf{Acc}(e)$$

because we must require that all arguments of the recursive calls of f satisfy Acc to deduce that also e does. But this definition is incorrect because we haven't defined the function f yet and so we cannot use it in the definition of Acc. Besides, we need Acc to define f, therefore we are locked in a vicious circle.

In our case, we have two instances of nested recursive clauses, for the sequential composition and *while* loops. When trying to give a semantics of the commands, we come to the definition

$$[\![c_1; c_2]\!]_\sigma := [\![c_2]\!]_{[\![c_1]\!]_\sigma}$$

for sequential composition and

$$[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_\sigma := [\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_{[\![c]\!]_\sigma}$$

for a *while* loop, if the interpretation of b in state σ is true.

Both cases contain a nested occurrence of the interpretation function $[\![-]\!]$.

An alternative solution, presented in [4], exploits the extension of type theory with simultaneous induction-recursion [6]. In this extension, an inductive type or inductive family can be defined simultaneously with a function on it. For the example above we would have

$$\begin{array}{l}
\mathsf{Acc}\colon A \to \mathbf{Prop} \\
f\colon (x\colon A)(\mathsf{Acc}\ x) \to B \\
\vdots \\
\mathsf{acc}_n\colon (h'\colon (\mathsf{Acc}\ e'))(\mathsf{Acc}\ (f\ e'\ h')) \to (\mathsf{Acc}\ e) \\
\vdots \\
(f\ e\ (\mathsf{acc}_n\ h'\ h)) := \cdots (f\ (f\ e'\ h)\ h) \cdots \\
\vdots
\end{array}$$

This method leads to the following definition of the accessibility predicate and interpretation function for the imperative programming language IMP:

$$
\begin{array}{l}
\mathsf{comAcc}\colon \mathsf{Command} \to \mathsf{State} \to \mathbf{Prop}\\
[\![]\!]\colon (c\colon \mathsf{Command}; \sigma\colon \mathsf{State})(\mathsf{comAcc}\ c\ \sigma) \to \mathsf{State}\\
\\
\mathsf{accSkip}\colon (\sigma\colon \mathsf{State})(\mathsf{comAcc}\ \mathsf{skip}\ \sigma)\\
\mathsf{accAssign}\colon (v\colon \mathbb{N}; a\colon \mathsf{AExp}; \sigma\colon \mathsf{State})(\mathsf{comAcc}\ (v \leftarrow a)\ \sigma)\\
\mathsf{accScolon}\colon (c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State}; h_1\colon (\mathsf{comAcc}\ c_1\ \sigma))(\mathsf{comAcc}\ c_2\ [\![c_1]\!]_\sigma^{h_1})\\
\qquad \to (\mathsf{comAcc}\ (c_1; c_2)\ \sigma)\\
\mathsf{accIf\_true}\colon (b\colon \mathsf{BExp}; c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State})[\![b]\!]_\sigma = \mathsf{true} \to (\mathsf{comAcc}\ c_1\ \sigma)\\
\qquad \to (\mathsf{comAcc}\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \sigma)\\
\mathsf{accIf\_false}\colon (b\colon \mathsf{BExp}; c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State})[\![b]\!]_\sigma = \mathsf{false} \to (\mathsf{comAcc}\ c_2\ \sigma)\\
\qquad \to (\mathsf{comAcc}\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \sigma)\\
\mathsf{accWhile\_true}\colon (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma\colon \mathsf{State})[\![b]\!] = \mathsf{true}\\
\qquad \to (h\colon (\mathsf{comAcc}\ c\ \sigma))(\mathsf{comAcc}\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ [\![c]\!]_\sigma^h)\\
\qquad \to (\mathsf{comAcc}(\mathsf{while}\ b\ \mathsf{do}\ c)\ \sigma)\\
\mathsf{accWhile\_false}\colon (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma\colon \mathsf{State})[\![b]\!] = \mathsf{false}\\
\qquad \to (\mathsf{comAcc}\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ \sigma)
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{skip}]\!]_\sigma^{(\mathsf{accSkip}\ \sigma)} := \sigma\\
[\![(v := a)]\!]_\sigma^{(\mathsf{accAssign}\ v\ a\ \sigma)} := \sigma[a/v]\\
[\![(c_1; c_2)]\!]_\sigma^{(\mathsf{accScolon}\ c_1\ c_2\ \sigma\ h_1\ h_2)} := [\![c_2]\!]^{h_2}_{[\![c_1]\!]_\sigma^{h_1}}\\
[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]_\sigma^{(\mathsf{accIf\_true}\ b\ c_1\ c_2\ \sigma\ p\ h_1)} := [\![c_1]\!]_\sigma^{h_1}\\
[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]_\sigma^{(\mathsf{accIf\_false}\ b\ c_1\ c_2\ \sigma\ q\ h_2)} := [\![c_2]\!]_\sigma^{h_2}\\
[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_\sigma^{(\mathsf{accWhile\_true}\ b\ c\ \sigma\ p\ h\ h')} := [\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]^{h'}_{[\![c]\!]_\sigma^h}\\
[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]_\sigma^{(\mathsf{accWhile\_false}\ b\ c\ \sigma\ q)} := \sigma
\end{array}
$$

This definition is admissible in systems that implement Dybjer's schema for simultaneous induction-recursion. But on systems that do not provide such schema, for example **Coq**, this definition is not valid.

We must disentangle the definition of the accessibility predicate from the definition of the evaluation function. As we have seen before, the evaluation function can be seen as the limit of the iteration of the functional F on an arbitrary base function $f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}$. Whenever the evaluation of a command c is defined on a state σ, we have that $[\![c]\!]_\sigma$ is equal to $(F_f^k\ c\ \sigma)$ for a sufficiently large number of iterations k. Therefore, we consider the functions F_f^k as approximations to the interpretation function being defined. We can formulate the accessibility predicate by using such approximations in place of the explicit occurrences of the evaluation function. Since the iteration approximation has two extra parameters, the number of iterations k and the base function f, we must also add them as new arguments of comAcc. The resulting inductive definition is

$$
\begin{array}{l}
\mathsf{comAcc}\colon \mathsf{Command} \to \mathsf{State} \to \mathbb{N} \to (\mathsf{Command} \to \mathsf{State} \to \mathsf{State}) \to \mathbf{Prop} \\
\mathsf{accSkip}\colon (\sigma\colon \mathsf{State}; k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State})(\mathsf{comAcc}\ \mathsf{skip}\ \sigma\ k+1\ f) \\
\mathsf{accAssign}\colon (v\colon \mathbb{N}; a\colon \mathsf{AExp}; \sigma\colon \mathsf{State}; k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}) \\
\qquad\qquad (\mathsf{comAcc}\ (v \leftarrow a)\ \sigma\ k+1\ f) \\
\mathsf{accScolon}\colon (c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State}; k\colon \mathbb{N}; f\colon (\mathsf{Command} \to \mathsf{State} \to \mathsf{State})) \\
\qquad\qquad (\mathsf{comAcc}\ c_1\ \sigma\ k\ f) \to (\mathsf{comAcc}\ c_2\ (F_f^k\ c_1\ \sigma)\ k\ f) \\
\qquad\qquad \to (\mathsf{comAcc}\ (c_1; c_2)\ \sigma\ k+1\ f) \\
\mathsf{accIf\_true}\colon (b\colon \mathsf{BExp}; c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State}; \\
\qquad\qquad k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State})(\langle b, \sigma\rangle \rightsquigarrow \mathsf{true}) \\
\qquad\qquad \to (\mathsf{comAcc}\ c_1\ \sigma\ k\ f) \to (\mathsf{comAcc}\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \sigma\ k+1\ f) \\
\mathsf{accIf\_false}\colon (b\colon \mathsf{BExp}; c_1, c_2\colon \mathsf{Command}; \sigma\colon \mathsf{State}; \\
\qquad\qquad k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State})(\langle b, \sigma\rangle \rightsquigarrow \mathsf{false}) \\
\qquad\qquad \to (\mathsf{comAcc}\ c_2\ \sigma\ k\ f) \to (\mathsf{comAcc}\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \sigma\ k+1\ f) \\
\mathsf{accWhile\_true}\colon (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma\colon \mathsf{State}; \\
\qquad\qquad\quad k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State})(\langle b, \sigma\rangle \rightsquigarrow \mathsf{true}) \\
\qquad\qquad\quad \to (\mathsf{comAcc}\ c\ \sigma\ k\ f) \to (\mathsf{comAcc}\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ (F_f^k\ c\ \sigma)) \\
\qquad\qquad\quad \to (\mathsf{comAcc}(\mathsf{while}\ b\ \mathsf{do}\ c)\ \sigma\ k+1\ f) \\
\mathsf{accWhile\_false}\colon (b\colon \mathsf{BExp}; c\colon \mathsf{Command}; \sigma\colon \mathsf{State}; \\
\qquad\qquad\quad k\colon \mathbb{N}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State})(\langle b, \sigma\rangle \rightsquigarrow \mathsf{false}) \\
\qquad\qquad\quad \to (\mathsf{comAcc}\ (\mathsf{while}\ b\ \mathsf{do}\ c)\ \sigma\ k+1\ f).
\end{array}
$$

This accessibility predicate characterizes the points in the domain of the program parametrically on the arguments k and f. To obtain an independent definition of the domain of the evaluation function we need to quantify on them. We quantify existentially on k, because if a command c and a state σ are accessible in k steps, then they will still be accessible in a higher number of steps. We quantify universally on f because we do not want the result of the computation to depend on the choice of the base function.

$$
\begin{array}{l}
\mathsf{comDom}\colon \mathsf{Command} \to \mathsf{State} \to \mathbf{Set} \\
(\mathsf{comDom}\ c\ \sigma) = \Sigma k\colon \mathbb{N}.\forall f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}.(\mathsf{comAcc}\ c\ \sigma\ k\ f)
\end{array}
$$

The reason why the sort of the predicate comDom is **Set** and not **Prop** is that we need to extract the natural number k from the proof to be able to compute the following evaluation function:

$$
\begin{array}{l}
[\![]\!]\colon (c\colon \mathsf{Command}; \sigma\colon \mathsf{State}; f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}) \\
\quad (\mathsf{comDom}\ c\ \sigma) \to \mathsf{State} \\
[\![c]\!]_{\sigma,f}^{\langle k,h\rangle} = (F_f^k\ c\ \sigma)
\end{array}
$$

To illustrate the meaning of these definitions, let us see how the interpretation of a sequential composition of two commands is defined. The interpretation of the command $(c_1; c_2)$ on the state σ is $[\![c_1; c_2]\!]_\sigma^H$, where H is a proof of $(\mathsf{comDom}\ (c_1; c_2)\ \sigma)$. Therefore H must be in the form $\langle k, h\rangle$, where $k\colon \mathbb{N}$ and $h\colon \forall f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}.(\mathsf{comAcc}\ (c_1; c_2)\ \sigma\ k\ f)$. To see how

h can be constructed, let us assume that $f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}$ and prove $(\mathsf{comAcc}\ (c_1; c_2)\ \sigma\ k\ f)$. This can be done only by using the constructor accScolon. We see that it must be $k = k' + 1$ for some k' and we must have proofs $h_1\colon (\mathsf{comAcc}\ c_1\ \sigma\ k'\ f)$ and $h_2\colon (\mathsf{comAcc}\ c_2\ (F_f^{k'}\ c_1\ \sigma)\ k'\ f)$. Notice that in h_2 we don't need to refer to the evaluation function $[\![]\!]$ anymore, and therefore the definitions of comAcc does not depend on the evaluation function anymore. We have now that $(h\ f) := (\mathsf{accScolon}\ c_1\ c_2\ \sigma\ k'\ f\ h_1\ h_2)$. The definition of $[\![c_1; c_2]\!]_\sigma^H$ is also not recursive anymore, but consists just in iterating F k times, where k is obtained from the proof H.

We can now prove an exact correspondence between operational semantics and denotational semantics given by the interpretation operator $[\![\cdot]\!]$.

Theorem 2.

$$\begin{array}{l}\forall c\colon \mathsf{Command}.\forall \sigma, \sigma'\colon \mathsf{State}.\\ \langle c, \sigma\rangle \rightsquigarrow \sigma' \Leftrightarrow \exists H\colon (\mathsf{comDom}\ c\ \sigma).\forall f\colon \mathsf{Command} \to \mathsf{State} \to \mathsf{State}.[\![c]\!]_{\sigma,f}^H = \sigma'.\end{array}$$

Proof. From left to right, it is proved by rule induction on the derivation of $\langle c, \sigma\rangle \rightsquigarrow \sigma'$. The number of iterations k is the depth of the proof and the proof of the comAcc predicate is a translation step by step of it. From right to left, it is proved by induction on the proof of comAcc.

5 Conclusions

The combination of the iteration technique and the accessibility predicate has, in our opinion, a vast potential that goes beyond its application to denotational semantics. Not only does it provide a path to the implementation and reasoning about partial and nested recursive functions that does not require simultaneous induction-recursion; but it gives a finer analysis of convergence of recursive operators. As we pointed out in Section 3, it supplies not just any fixed point of an operator, but the least fixed point.

We were not the first to formalize parts of Winskel's book in a proof system. Nipkow [9] formalized the first 100 pages of it in ISABELLE/HOL. The main difference between our work and his, is that he does not represent the denotation as a function but as a subset of $\mathsf{State} \times \mathsf{State}$ that happens to be the graph of a function. Working on a well developed library on sets, he has no problem in using a least-fixpoint operator to define the subset associated to a *while* loop: But this approach stays further removed from functional programming than an approach based directly on the functions provided by the prover. In this respect, our work is the first to reconcile a theorem proving framework with total functions with denotational semantics. One of the gains is directly executable code (through extraction or ι-reduction). The specifications provided by Nipkow are only executable in the sense that they all belong to the subset of inductive properties that can be translated to PROLOG programs. In fact, the reverse process has been used and those specifications had all been obtained by a translation from a variant of PROLOG to a theorem prover [2]. However, the prover's function had not been used to represent the semantics.

Our method tries to maximize the potential for automation: Given a recursive definition, the functional operator F, the iterator, the accessibility predicate, the domain, and the evaluation function can all be generated automatically. Moreover, it is possible to automate the proof of the accessibility predicate, since there is only one possible proof step for any given argument; and the obtained evaluation function is computable inside type theory.

We expect this method to be widely used in the future in several areas of formalization of mathematics in type theory.

References

1. Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In Harrison and Aagaard [8], pages 1–16.
2. Yves Bertot and Ranan Fraer. Reasoning with executable specifications. In *International Joint Conference of Theory and Practice of Software Development (TAPSOFT/FASE'95)*, volume 915 of *LNCS*. Springer-Verlag, 1995.
3. A. Bove. Simple general recursion in type theory. *Nordic Journal of Computing*, 8(1):22–42, Spring 2001.
4. Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, 2001.
5. Catherine Dubois and Véronique Viguié Donzeau-Gouge. A step towards the mechanization of partial functions: domains as inductive predicates. Presented at CADE-15, Workshop on Mechanization of Partial Functions, 1998.
6. Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.
7. Simon Finn, Michael Fourman, and John Longley. Partial functions in a total setting. *Journal of Automated Reasoning*, 18(1):85–104, February 1997.
8. J. Harrison and M. Aagaard, editors. *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
9. Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *LNCS*, pages 180–192. Springer, 1996.
10. Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.
11. The Coq Development Team. LogiCal Project. *The Coq Proof Assistant. Reference Manual. Version 7.2.* INRIA, 2001.
12. K. Slind. Another look at nested recursion. In Harrison and Aagaard [8], pages 498–518.
13. Freek Wiedijk and Jan Zwanenburg. First order logic with domain conditions. Available at `http://www.cs.kun.nl/~freek/notes/partial.ps.gz`, 2002.
14. Glynn Winskel. *The Formal Semantics of Programming Languages, an introduction.* Foundations of Computing. The MIT Press, 1993.