CS4215—Programming Language Implementation

Martin Henz

Thursday 2 February, 2012

Chapter 7

Denotational Semantics of simPL

7.1 Introduction

The dynamic semantics that we have seen so far relies on the idea of contraction. An expression was evaluated by contracting subexpressions, until no further contraction was possible. This evaluation process constituted the "meaning" of programs. We treated the evaluation of expressions as a mere transformation of expressions to expressions. That means that we never left the syntactic realm. Evaluation of expressions was the game of transforming expressions. A minor nuisance was that we had infinitely many rules for the game. We list some major disadvantages of this approach to defining the semantics of a programming language.

- Contraction relies on the idea of substitution, which is the syntactic replacement of expressions for variables. Substitution is mathematically rather complicated and far away from what happens when real programs are executed. We would like to have a simpler mechanism for variable binding.
- Primitive operations that are not total functions, such as division, can make the evaluation process get stuck. This means that evaluation fails to find a value. We would like to have a more explicit way of handling such a runtime error.
- Dynamic semantics cannot be extended easily to other language paradigms such as imperative programming. In Chapter 10, we shall define the semantics of a simple imperative language.

As a result of these difficulties, dynamic semantics are rarely used for describing the meaning of computer programs. We will abandon this approach in this and the following chapters in favor of so-called denotational semantics.

The idea of denotational semantics is to directly assign a mathematical value as a meaning to an expression. Compared to dynamic semantics, there are two main advantages of this approach. Firstly, we can employ known mathematical concepts such as integers, booleans, functions etc to describe the meaning of programs. Compare this option with the awkward construction of an infinite number of rules for defining simple arithmetic operators! Secondly, denotational semantics avoids the clumsy construction of evaluation as the transitive closure of one-step evaluation, which forced us to define erroneous programs as programs whose evaluation gets "stuck".

We follow the approach of [Sch88], and define a denotational semantics as consisting of three parts:

- A description of the syntax of the language in question,
- a collection of semantic domains with associated algebraic operations and properties, and
- a collection of semantic functions which together describe the meaning of programs.

7.2 Decimal Numerals

Before we start with the denotational semantics of the language simPL, we shall concentrate on a small aspect of simPL, namely decimal numerals representing non-negative integers in simPL programs. The language **N** of decimal numerals contains non-empty strings of decimal digits. We can describe the language using the following twenty rules:

$$\frac{}{\texttt{0}} \quad \cdots \quad \frac{}{\texttt{9}} \qquad \frac{n}{n\texttt{0}} \quad \cdots \quad \frac{n}{n\texttt{9}}$$

For example, the sequence of digits `12` and `987654321` are elements of the language **N** of decimal numerals. Such numerals occur in simPL programs such as `12 + 987654321`. The syntax rules for simPL refer to such numerals by the symbol n.

As semantic domain, we choose the integers, denoted by **Int**, taking for granted the ring properties of **Int** with respect to the operations of addition and multiplication.

Our semantic function

$$\rightarrowtail_{\mathbf{N}} : \mathbf{N} \to \mathbf{Int}$$

describes the meaning of decimal numerals as their corresponding integer value. We use the usual notation of rules to describe $\rightarrowtail_{\mathbf{N}}$ as a relation.

$$\frac{}{\texttt{0} \rightarrowtail_{\mathbf{N}} 0}$$

Note that the $\mathtt{0}$ on the left hand side denotes an element of our language of decimal numerals, whereas the 0 on the right hand side denotes the integer value 0, the neutral element for addition in the ring of integers.

The other nineteen rules for $\rightarrowtail_{\mathbf{N}}$ are:

$$\frac{}{\mathtt{1} \rightarrowtail_{\mathbf{N}} 1} \qquad \cdots \qquad \frac{}{\mathtt{9} \rightarrowtail_{\mathbf{N}} 9}$$

$$\frac{n \rightarrowtail_{\mathbf{N}} i}{n\mathtt{0} \rightarrowtail_{\mathbf{N}} 10 \cdot i} \qquad \frac{n \rightarrowtail_{\mathbf{N}} i}{n\mathtt{1} \rightarrowtail_{\mathbf{N}} 10 \cdot i + 1} \qquad \cdots \qquad \frac{n \rightarrowtail_{\mathbf{N}} i}{n\mathtt{9} \rightarrowtail_{\mathbf{N}} 10 \cdot i + 9}$$

Again, note the difference between the left and right hand side of $\rightarrowtail_{\mathbf{N}}$. In the last ten rules, the n on the left hand side denote elements of our language of decimal numerals, whereas the i on the right hand side denote integer values. The symbols $+$ and $\cdot$ denote addition and multiplication in the ring of integers.

The left hand sides of $\rightarrowtail_{\mathbf{N}}$ in the bottom of all four rules are mutually distinct. It is therefore easy to see that the relation defined by the rules is indeed a function. This will be the case for all relations described in this chapter.

Furthermore, it is not difficult to see that $\rightarrowtail_{\mathbf{N}}$ is a total function, since the rules defining $\rightarrowtail_{\mathbf{N}}$ cover all rules defining $\mathbf{N}$.

To demonstrate the usefulness of denotational semantics for proving properties of languages, let us prove that the "successor" operation on decimal numerals coincides with the successor function on integers.

We define the successor function $'$ on decimal numerals as follows:

$$\frac{}{\mathtt{0}' = \mathtt{1}} \qquad \cdots \qquad \frac{}{\mathtt{8}' = \mathtt{9}} \qquad \frac{}{\mathtt{9}' = \mathtt{10}}$$

$$\frac{}{n\mathtt{0}' = n\mathtt{1}} \qquad \cdots \qquad \frac{}{n\mathtt{8}' = n\mathtt{9}} \qquad \frac{n' = m}{n\mathtt{9}' = m\mathtt{0}}$$

Proposition 7.1 *For all $n \in \mathbf{N}$, if $n \rightarrowtail_{\mathbf{N}} i$, and $n' \rightarrowtail_{\mathbf{N}} j$, then $j = i + 1$.*

Proof: The cases for $\mathtt{0}, \ldots, \mathtt{9}$, and $n\mathtt{0}, \ldots, n\mathtt{8}$ are immediate. We prove by induction on the rules of $\rightarrowtail_{\mathbf{N}}$ that if $n\mathtt{9} \rightarrowtail_{\mathbf{N}} i$, and $n\mathtt{9}' \rightarrowtail_{\mathbf{N}} j$, then $j = i + 1$. For numerals of the form $n\mathtt{9}$, we have $n\mathtt{9}' = n'\mathtt{0}$ according to the definition of $'$. From the definition of $\rightarrowtail_{\mathbf{N}}$, we have $n'\mathtt{0} \rightarrowtail_{\mathbf{N}} 10 \cdot k$, where $n' \rightarrowtail_{\mathbf{N}} k$. From the induction hypothesis, we have: if $n \rightarrowtail_{\mathbf{N}} h$, then $k = h + 1$. Therefore, $10 \cdot k = 10 \cdot (h + 1) = 10 \cdot h + 9 + 1$. From the definition of $\rightarrowtail_{\mathbf{N}}$ and since $n \rightarrowtail_{\mathbf{N}} h$, we have $10 \cdot h + 9 + 1 = i + 1$, and thus $n\mathtt{9}' \rightarrowtail_{\mathbf{N}} j$, where $j = i + 1$. $\square$

7.3 Outline

In order to reach a denotational semantics for simPL, we are going to introduce the denotational semantics of various ever more complex sub-languages of simPL.

- We start out with the language **simPL0**, a language which is only able to evaluate integer and boolean expressions, where division is not allowed.
- We extend **simPL0** to **simPL1** by adding `let` and `if`.
- **simPL2** adds division and thus the proper treatment of errors values.
- **simPL3** adds functions, and
- **simPL4** adds recursive functions and thus is the same as **simPL**.

7.4 Denotational Semantics for simPL0

simPL0 is a calculator language with arithmetic and boolean operators (no division), defined by the following rules.

$$\frac{}{n} \qquad \frac{}{\texttt{true}} \qquad \frac{}{\texttt{false}}$$

$$\frac{E_1 \quad E_2}{p[E_1, E_2]} \text{ where } p \in \{\texttt{|},\texttt{\&},\texttt{+},\ \texttt{-},\texttt{*},\texttt{=},\texttt{>},\texttt{<}\} \qquad \frac{E}{p[E]} \text{ where } p \in \{\backslash\}$$

The following semantic domains are suitable for this language.

Semantic Domain	Definition	Explanation
Bool	$\{true, false\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	**Bool** + **Int**	expressible values

The ring of integers **Int** is already introduced in the previous section. The ring of booleans is the ring formed by the set $\{true, false\}$ with the operators disjunction, denoted by $\vee$, and conjunction, denoted by $\wedge$.

The symbol $+$ that we are using in the last line denotes *disjoint union*. Informally, disjoint union is a kind of union that preserves the origin of the values. That means from an element of **Int** + **Bool** we can find out whether it came from **Int** or **Bool**, regardless of how integers and booleans are represented, i.e. even if boolean values are represented by integers such as 0 and 1.

Formally, disjoint union can be defined as follows:

$$S_1 + S_2 = \{(1, x_1) \mid x_1 \in S_1\} \cup \{(2, x_2) \mid x_2 \in S_2\}$$

We canonically extend the operations and properties of the component domains **Bool** and **Int** to the set $\mathbf{Int} + \mathbf{Bool}$. We choose the name **EV** (expressible values) for this set to indicate that its elements are the possible results of evaluating **simPL** expressions.

The semantic function

$$\cdot \rightarrowtail \cdot : \mathbf{simPL0} \to \mathbf{EV}$$

defined by the following rules, expresses the meaning of elements of **simPL0**, by defining the value of each element.

$$\frac{}{\texttt{true} \rightarrowtail \mathit{true}} \qquad \frac{}{\texttt{false} \rightarrowtail \mathit{false}} \qquad \frac{n \rightarrowtail_{\mathbf{N}} i}{n \rightarrowtail i}$$

Note that the last rule employs the denotational semantics of decimal numerals described in the previous section.

On the right hand sides of $\rightarrowtail$ in the following rules, we are making use of the operations of addition, subtraction and multiplication in the ring of integers.

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{+}E_2 \rightarrowtail v_1 + v_2} \qquad \frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{-}E_2 \rightarrowtail v_1 - v_2}$$

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{*}E_2 \rightarrowtail v_1 \cdot v_2}$$

The following three rules make use of disjunction, conjunction and negation in the ring of booleans.

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{\&}E_2 \rightarrowtail v_1 \wedge v_2} \qquad \frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\,\texttt{|}\,E_2 \rightarrowtail v_1 \vee v_2} \qquad \frac{E \rightarrowtail v}{\backslash E \rightarrowtail \neg v}$$

The operation $\equiv$ in the following rule reifies the identity on integers to a boolean value. For example, $1 \equiv 2 = \mathit{false}$ and $3 \equiv 3 = \mathit{true}$.

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{=}E_2 \rightarrowtail v_1 \equiv v_2}$$

The operations $>$ and $<$ in the final two rules reflect the less-than and greater-than operations using the usual total ordering on integers.

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{>}E_2 \rightarrowtail v_1 > v_2} \qquad \frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{<}E_2 \rightarrowtail v_1 < v_2}$$

Example 7.1 `1 + 2 > 3` $\rightarrowtail$ *false holds because* `1 + 2` $\rightarrowtail 3$ *and* $3 > 3$ *is false.*

7.5 Denotational Semantics for simPL1

We add conditionals, identifiers and the `let` construct. Note that we are not reducing the `let` construct to function definition and application here. The reason is that we want to show that `let` can be defined in a language without functions. Furthermore, this approach allows us to introduce the concept of environments that will play an important role later.

The language **simPL1** is defined by adding the following rules to the rules defining **simPL0**.

$$\frac{E \quad E_1 \quad E_2}{\texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ end}}$$

$$\frac{}{x} \qquad \frac{E \quad E_1 \quad \ldots \quad E_n}{\texttt{let } x_1 \texttt{ = } E_1 \ \cdots x_n \texttt{ = } E_n \texttt{ in } E \texttt{ end}}$$

In order to define $\rightarrowtail$ for **simPL1**, we need to introduce environments that allow us to keep track of the binding of identifiers. These environments map identifiers to *denotable values*.

Semantic domain	Definition	Explanation
Bool	$\{true, false\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	**Bool** + **Int**	expressible values
DV	**Bool** + **Int**	denotable values
Id	alphanumeric strings	identifiers
Env	**Id** $\rightsquigarrow$ **DV**	environments

The set **Id** is the set of symbols that can occur as identifiers in **simPL** expressions. The term **Id** $\rightsquigarrow$ **DV** denotes the set of all partial functions from **Id** to **DV**. Denotable values **DV** are values that can be referred to by identifiers. For **simPL1**, **DV** = **EV**, but we will see in the next section that this is not always the case.

For environments Δ, we introduce an operation $\Delta[x \leftarrow v]$, which denotes an environment that works like Δ, except that $\Delta[x \leftarrow v](x) = v$. The semantic function $\cdot \rightarrowtail \cdot$ now needs to be defined using an auxiliary semantic function $\cdot \Vdash \cdot \rightarrowtail \cdot$ that gets an environment as additional argument.

$$\cdot \rightarrowtail \cdot : \textbf{simPL1} \rightarrow \textbf{EV}$$

$$\frac{\emptyset \Vdash E \rightarrowtail v}{E \rightarrowtail v}$$

Here $\emptyset$ stands for the empty environment. The semantic function $\cdot \Vdash \cdot \rightarrowtail \cdot$ is defined as a ternary relation (binary partial function):

$$\cdot \Vdash \cdot \rightarrowtail \cdot : \mathbf{Env} * \text{simPL1} \rightarrow \mathbf{EV}$$

The following rules define $\cdot \Vdash \cdot \rightarrowtail \cdot$:

$$\frac{}{\Delta \Vdash \texttt{true} \rightarrowtail \mathit{true}} \qquad \frac{}{\Delta \Vdash \texttt{false} \rightarrowtail \mathit{false}} \qquad \frac{n \rightarrowtail_{\mathbf{N}} i}{\Delta \Vdash n \rightarrowtail i}$$

The meaning of an identifier in the following rule is given by the environment. If a given environment Δ is undefined on the identifier, then the identifier has no meaning with respect to Δ.

$$\frac{}{\Delta \Vdash x \rightarrowtail \Delta(x)}$$

The rules for the primitive operations are similar to the corresponding rules for **simPL0**.

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1\texttt{+}E_2 \rightarrowtail v_1 + v_2} \qquad \frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1\texttt{-}E_2 \rightarrowtail v_1 - v_2}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1\texttt{*}E_2 \rightarrowtail v_1 \cdot v_2}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1\texttt{\&}E_2 \rightarrowtail v_1 \wedge v_2} \qquad \frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 \texttt{|} E_2 \rightarrowtail v_1 \vee v_2}$$

$$\frac{\Delta \Vdash E \rightarrowtail v}{\Delta \Vdash \backslash E \rightarrowtail \neg v} \qquad \frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1\texttt{=}E_2 \rightarrowtail v_1 \equiv v_2}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1\texttt{>}E_2 \rightarrowtail v_1 > v_2} \qquad \frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1\texttt{<}E_2 \rightarrowtail v_1 < v_2}$$

The meaning of a conditional is the meaning of its then-part, if the meaning of the condition is *true*, and it is the meaning of the else-part, if the meaning of the condition is *false*.

$$\frac{\Delta \Vdash E \rightarrowtail \mathit{true} \quad \Delta \Vdash E_1 \rightarrowtail v_1}{\Delta \Vdash \texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2\ \texttt{end} \rightarrowtail v_1} \qquad \frac{\Delta \Vdash E \rightarrowtail \mathit{false} \quad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash \texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2\ \texttt{end} \rightarrowtail v_2}$$

Note that in this way, we are explaining the **simPL** `if...then...else...end` statement by employing the if-then-else statement in English (or mathematics). We are not going to further explore the philosophical shortcomings of this approach.

Finally, and most interestingly, the meaning of the `let` construct is given by the meaning of its components in the following way.

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \quad \cdots \quad \Delta \Vdash E_n \rightarrowtail v_n \quad \Delta[x_1 \leftarrow v_1] \cdots [x_n \leftarrow v_n] \Vdash E \rightarrowtail v}{\Delta \Vdash \texttt{let}\ x_1\ \texttt{=}\ E_1 \cdots x_n\ \texttt{=}\ E_n\ \texttt{in}\ E\ \texttt{end} \rightarrowtail v}$$

The given environment Δ is extended by bindings of the local variables to their meaning. We are making use of the fact that every expressible value is also a denotable value. The meaning of the `let` expression is then defined as the meaning of its body with respect to the extended environment.

Example 7.2

$$\cfrac{\cfrac{\cfrac{\cdots}{\emptyset[\mathit{AboutPi} \leftarrow 3] \Vdash \mathit{AboutPi}\ \texttt{+}\ 2 \rightarrowtail 5}}{\emptyset \Vdash \texttt{let}\ \mathit{AboutPi}\ \texttt{=}\ 3\ \texttt{in}\ \mathit{AboutPi}\ \texttt{+}\ 2 \rightarrowtail 5}}{\texttt{let}\ \mathit{AboutPi}\ \texttt{=}\ 3\ \texttt{in}\ \mathit{AboutPi}\ \texttt{+}\ 2\ \texttt{end} \rightarrowtail 5}$$

7.6 Denotational Semantics for simPL2

The language **simPL2** adds division to **simPL1**.

$$\frac{E_1 \quad E_2}{E_1 / E_2}$$

The difficulty lies in the fact that division on integers is a partial function, not being defined for 0 as second argument. In this chapter, we are more ambitious than in the previous one, and want to give meaning to programs, even if division

by 0 occurs. For this purpose, we extend the definitions of semantic domains and functions as follows.

Semantic domain	Definition	Explanation
Bool	$\{true, false\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	$\mathbf{Bool} + \mathbf{Int} + \{\bot\}$	expressible values
DV	$\mathbf{Bool} + \mathbf{Int}$	denotable values
Id	alphanumeric string	identifiers
Env	$\mathbf{Id} \rightsquigarrow \mathbf{DV}$	environments

Note that we add the symbol $\bot$ to the set of expressible values. The meaning of expressions that execute a division by 0 will be $\bot$. The semantic function $\cdot \Vdash \cdot \rightarrowtail \cdot$ is modified to take the occurrence of the error value $\bot$ into account.

$$\frac{}{\Delta \Vdash \texttt{true} \rightarrowtail true} \qquad \frac{}{\Delta \Vdash \texttt{false} \rightarrowtail false}$$

$$\frac{n \rightarrowtail_{\mathbf{N}} i}{\Delta \Vdash n \rightarrowtail i} \qquad \frac{}{\Delta \Vdash x \rightarrowtail \Delta(x)}$$

Instead of having one single rule for each primitive operator, we now have three rules for each of the binary operators `+`, `-`, and `*`. The two additional rules in each case express that the meaning of an expression is $\bot$ if the meaning of one of the component expressions is $\bot$.

$$\frac{\Delta \Vdash E_1 \rightarrowtail \bot}{\Delta \Vdash E_1\texttt{+}E_2 \rightarrowtail \bot} \qquad \frac{\Delta \Vdash E_2 \rightarrowtail \bot}{\Delta \Vdash E_1\texttt{+}E_2 \rightarrowtail \bot}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1\texttt{+}E_2 \rightarrowtail v_1 + v_2} \text{ if } v_1, v_2 \neq \bot$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail \bot}{\Delta \Vdash E_1\texttt{-}E_2 \rightarrowtail \bot} \qquad \frac{\Delta \Vdash E_2 \rightarrowtail \bot}{\Delta \Vdash E_1\texttt{-}E_2 \rightarrowtail \bot}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1\texttt{-}E_2 \rightarrowtail v_1 - v_2} \text{ if } v_1, v_2 \neq \bot$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail \bot}{\Delta \Vdash E_1 * E_2 \rightarrowtail \bot} \qquad \frac{\Delta \Vdash E_2 \rightarrowtail \bot}{\Delta \Vdash E_1 * E_2 \rightarrowtail \bot}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 * E_2 \rightarrowtail v_1 \cdot v_2} \text{ if } v_1, v_2 \neq \bot$$

The first three rules for division are similar. In the third rule / stands for integer division (with rounding towards 0).

$$\frac{\Delta \Vdash E_1 \rightarrowtail \bot}{\Delta \Vdash E_1 / E_2 \rightarrowtail \bot} \qquad \frac{\Delta \Vdash E_2 \rightarrowtail \bot}{\Delta \Vdash E_1 / E_2 \rightarrowtail \bot}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 / E_2 \rightarrowtail v_1 / v_2} \text{ if } v_1, v_2 \neq \bot \text{ and } v_2 \neq 0$$

The last rule for division covers the case that the meaning of the second argument of division is 0. Since division by 0 is not defined, the meaning of the entire expression is $\bot$.

$$\frac{\Delta \Vdash E_2 \rightarrowtail 0}{\Delta \Vdash E_1 / E_2 \rightarrowtail \bot}$$

Equipped with this scheme of handling the error value, the remaining rules for **simPL2** are not surprising.

$$\frac{\Delta \Vdash E_1 \rightarrowtail \bot}{\Delta \Vdash E_1 \,\&\, E_2 \rightarrowtail \bot} \qquad \frac{\Delta \Vdash E_2 \rightarrowtail \bot}{\Delta \Vdash E_1 \,\&\, E_2 \rightarrowtail \bot}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 \,\&\, E_2 \rightarrowtail v_1 \wedge v_2} \text{ if } v_1, v_2 \neq \bot$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail \bot}{\Delta \Vdash E_1 \mid E_2 \rightarrowtail \bot} \qquad \frac{\Delta \Vdash E_2 \rightarrowtail \bot}{\Delta \Vdash E_1 \mid E_2 \rightarrowtail \bot}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 \,|\, E_2 \rightarrowtail v_1 \vee v_2} \text{ if } v_1, v_2 \neq \bot$$

$$\frac{\Delta \Vdash E \rightarrowtail \bot}{\Delta \Vdash \backslash E \rightarrowtail \bot} \qquad \frac{\Delta \Vdash E \rightarrowtail v}{\Delta \Vdash \backslash E \rightarrowtail \neg v} \text{ if } v \neq \bot$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail \bot}{\Delta \Vdash E_1 \texttt{=} E_2 \rightarrowtail \bot} \qquad \frac{\Delta \Vdash E_2 \rightarrowtail \bot}{\Delta \Vdash E_1 \texttt{=} E_2 \rightarrowtail \bot}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 \texttt{=} E_2 \rightarrowtail v_1 \equiv v_2} \text{ if } v_1, v_2 \neq \bot$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail \bot}{\Delta \Vdash E_1 \texttt{>} E_2 \rightarrowtail \bot} \qquad \frac{\Delta \Vdash E_2 \rightarrowtail \bot}{\Delta \Vdash E_1 \texttt{>} E_2 \rightarrowtail \bot}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 \texttt{>} E_2 \rightarrowtail v_1 > v_2} \text{ if } v_1, v_2 \neq \bot$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail \bot}{\Delta \Vdash E_1 \texttt{<} E_2 \rightarrowtail \bot} \qquad \frac{\Delta \Vdash E_2 \rightarrowtail \bot}{\Delta \Vdash E_1 \texttt{<} E_2 \rightarrowtail \bot}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 \texttt{<} E_2 \rightarrowtail v_1 < v_2} \text{ if } v_1, v_2 \neq \bot$$

$$\frac{\Delta \Vdash E \rightarrowtail \bot}{\Delta \Vdash \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ end} \rightarrowtail \bot}$$

$$\frac{\Delta \Vdash E \rightarrowtail \mathit{true} \quad \Delta \Vdash E_1 \rightarrowtail v_1}{\Delta \Vdash \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ end} \rightarrowtail v_1}$$

$$\frac{\Delta \Vdash E \rightarrowtail \mathit{false} \quad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash \texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2\ \texttt{end} \rightarrowtail v_2}$$

$$\frac{\Delta \Vdash E_i \rightarrowtail \bot}{\Delta \Vdash \texttt{let}\ x_1\ \texttt{=}\ E_1 \cdots x_n\ \texttt{=}\ E_n\ \texttt{in}\ E\ \texttt{end} \rightarrowtail \bot}\ \text{for } i, 1 \leq i \leq n$$

$$\frac{\Delta[x_1 \leftarrow v_1] \cdots [x_n \leftarrow v_n] \Vdash E \rightarrowtail v \quad \Delta \Vdash E_1 \rightarrowtail v_1 \cdots \Delta \Vdash E_n \rightarrowtail v_n}{\Delta \Vdash \texttt{let}\ x_1\ \texttt{=}\ E_1 \cdots x_n\ \texttt{=}\ E_n\ \texttt{in}\ E\ \texttt{end} \rightarrowtail v}\ \text{otherwise}$$

Note that by introducing the error value, we achieve that $\rightarrowtail$ is still a total function although its component function / is not. Also note that we achieve excluding the error values from denotable values by letting the `let` construct return error when the expression between `=` and `in` evaluates to error instead of binding the variable to the error value.

Example 7.3 *For any environment* Δ, $\Delta \Vdash$`5+(3/0)` $\rightarrowtail \bot$, *since* $\Delta \Vdash$`3/0` $\rightarrowtail \bot$.

Semantic rules that properly treat error values tend to be complex. (They took us more than three pages.) In the following, we are therefore omitting the treatment of the error value for simplicity.

7.7 Denotational Semantics for simPL3

The next step is to add (non-recursive) function definition and application.

$$\frac{E}{\texttt{fun}\ \{\cdot\}\ x_1 \ldots x_n\ \texttt{->}\ E\ \texttt{end}} \qquad \frac{E \quad E_1 \ \cdots \ E_n}{\texttt{(}E\ E_1\ \ldots\ E_n\texttt{)}}$$

We add functions to our denotable and expressible values, resulting in the following semantic domains.

Semantic domain	Definition	Explanation
Bool	$\{\mathit{true}, \mathit{false}\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	$\mathbf{Bool} + \mathbf{Int} + \{\bot\} + \mathbf{Fun}$	expressible values
DV	$\mathbf{Bool} + \mathbf{Int} + \mathbf{Fun}$	denotable values
Id	alphanumeric string	identifiers
Env	$\mathbf{Id} \rightsquigarrow \mathbf{DV}$	environments
Fun	$\mathbf{DV} * \cdots * \mathbf{DV} \rightsquigarrow \mathbf{EV}$	function values

We need to add rules for function definition and application to $\cdot \Vdash \cdot \rightarrowtail \cdot$. The meaning of a function definition is a function (in the mathematical sense) that takes as many denoted values as argument as the function definition has formal parameters.

$$\frac{}{\Delta \Vdash \texttt{fun}\ \{\ \cdot\ \}\ x_1 \ldots x_n \texttt{->} E\ \texttt{end} \rightarrowtail f}$$

where f is a function such that $f(y_1, \ldots, y_n) = v$, where $\Delta[x_1 \leftarrow y_1] \cdots [x_n \leftarrow y_n] \Vdash E \rightarrowtail v$

$$\frac{\Delta \Vdash E \rightarrowtail f \quad \Delta \Vdash E_1 \rightarrowtail v_1 \ \cdots\ \Delta \Vdash E_n \rightarrowtail v_n}{\Delta \Vdash \texttt{(}E\ E_1\ \ldots\ E_n\texttt{)}\ \rightarrowtail f(v_1, \ldots, v_n)}$$

How do we know that such a function $\rightarrowtail$ exists? The meaning of an expression is still defined in terms of the meaning of its component expressions. Therefore, rule induction still serves as an effective proof technique.

7.8 Denotational Semantics for simPL4

The last (and from a theoretical point of view most challenging) step is to add recursive functions.

$$\frac{E}{\texttt{recfun}\ f\ \{\ \cdot\ \}\ x_1 \ldots x_n\ \texttt{->}\ E\ \texttt{end}}$$

We would like to add the following rule to our definition of $\rightarrowtail$.

$$\frac{}{\Delta \Vdash \texttt{recfun}\ g\ \{\ \cdot\ \}\ x_1 \ldots x_n\ \texttt{->}\ E\ \texttt{end} \rightarrowtail f}$$

where f is a function such that $f(y_1, \ldots, y_n) = v$, where $\Delta[x_1 \leftarrow y_1] \cdots [x_n \leftarrow y_n]$ $[g \leftarrow f] \Vdash E \rightarrowtail v$

Such a definition is not sound, however, because the function f occurs in its own definition. The question whether a function f exists that has the property described above becomes non-trivial.

Example 7.4 *The meaning of the function*

```
recfun fac {int -> int} n ->
   if n < 2 then 1
   else n * (fac n-1)
   end
end
```

is uniquely defined by the equation above as the following function:

$$f(v) = v! \ \text{ for all } v \geq 2 \text{ and } 1 \text{ for all } v < 2$$

Example 7.5 *The meaning of the function*

```
recfun f {int -> int} n -> (f n) end
```

is not uniquely defined. Both of the following two functions fulfill the required equation:

$$f(v) = 0, \textit{ for all } v \textit{ in } \boldsymbol{DV}$$

$$f(v) = 1, \textit{ for all } v \textit{ in } \boldsymbol{DV}$$

The theory of fix-points, beyond the scope of this course, answers the question how to uniquely identify the right function as the meaning of recursive function definitions. An introduction to this field is given in [Ten91] and [Sch88].

Bibliography

[Sch88] David A. Schmidt. *Denotational Semantics—A Methodology for Language Development.* Wm. C. Brown Publishers, Dubuque, Iowa, 1988.

[Ten91] R. D. Tennent. *Semantics of Programming Languages.* Prentice Hall, Hertfordshire, UK, 1991.