# CS4215—Programming Language Implementation

Martin Henz and Chin Wei Ngan

Sunday 8th January, 2017

# Chapter 1

# Static Semantics of simPL

Similar to ePL, not all expressions in simPL make sense. For example,

```
if fun {int -> int} x -> x end then 1 else 0 end
```

does not make sense, because `fun {int -> int} x -> x end` is a function, whereas the conditional test expects a boolean value as first component. We say that the expression is *ill-typed*, because a typing condition is not met. Expressions that meet these conditions are called *well-typed*. Section 1.1 uses a typing relation to identify well-typed simPL expressions. What properties do well-typed expressions have? Section 1.2 answers this question by showing that the evaluation of well-typed expressions enjoys specific properties.

## 1.1 Well-Typedness of simPL Programs

For simPL, well-typedness of an expression depends on the context in which the expression appears. The expression `x + 3` may or may not be well-typed, depending on the type of `x`. Thus in order to formalize the notion of a context, we define a *type environment*, denoted by $\Gamma$, that keeps track of the type of identifiers appearing in the expression. More formally, the partial function $\Gamma$ from identifiers to types expresses a context, in which an identifier $x$ is associated with type $\Gamma(x)$.

We define a relation $\Gamma[x \leftarrow t]\Gamma'$ on type environments $\Gamma$, identifiers $x$, types $t$, and type environments $\Gamma'$, which constructs a type environment that behaves like the given one, except that the type of $x$ is $t$. More formally, if $\Gamma[x \leftarrow t]\Gamma'$, then $\Gamma'(y)$ is $t$, if $y = x$ and $\Gamma(y)$ otherwise. Obviously, this uniquely identifies $\Gamma'$ for a given $\Gamma$, $x$, and $t$, and thus the type environment extension relation is functional in its first three arguments.

The set of identifiers, on which a type environment $\Gamma$ is defined, is called the domain of $\Gamma$, denoted by $dom(\Gamma)$.

Note that we used a slightly different notation $\cdot\,[\cdot \leftarrow\!\!\cdot]\rightsquigarrow\cdot$ to denote substitution in Section 6.2. Even if they were written the same way, it will always be clear from the context, which operation is meant.

**Example 1.1** *The empty typing relation* $\Gamma = \emptyset$ *is not defined for any identifier. We can extend the empty environment* $\emptyset$ *with type bindings by* $\emptyset[\texttt{AboutPi} \leftarrow \texttt{int}]\Gamma'$*, where* $\Gamma'$ *is an environment that can be applied only to the identifier* `AboutPi`; the result of $\Gamma'$(`AboutPi`) is the type `int`. *Similarly, we can define* $\Gamma''$ *by* $\Gamma'[\texttt{Square} \leftarrow \texttt{int -> int}]\Gamma''$. *The type environment* $\Gamma''$ *may be applied to either the identifier* `AboutPi`*, or to the identifier* `Square`. *Thus,* $dom(\Gamma'') = \{\texttt{AboutPi}, \texttt{Square}\}$.

The set of well-typed expressions is defined by the ternary *typing relation*, written $\Gamma \vdash E : t$, where $\Gamma$ is a type environment such that $E \bowtie X$ and $X \subseteq dom(\Gamma)$. This relation can be read as "the expression $E$ has type $t$, under the assumption that its free identifiers have the types given by $\Gamma$". When $E$ has no free identifiers (we say $E$ is *closed*), we can write $E : t$ instead of $\emptyset \vdash E : t$.

**Example 1.2** *Continuing Example 1.1, we will define the typing relation such that the following expressions hold:*

- $\Gamma' \vdash$ `AboutPi` $* 2 :$ `int`
- $\Gamma'' \vdash$ `fun{int -> int} x -> AboutPi * (Square 2) end` : `int -> int`

*but:*

- $\Gamma' \vdash$ `fun {int -> int} x -> AboutPi * (Square 2) end` : `int -> int`
  *does not hold, because* `Square` *occurs free in the expression, but the type environment* $\Gamma'$ *to the left of the* $\vdash$ *symbol is not defined for* `Square`.
- $\Gamma \vdash$ `true + 1` $: t$
  *does not hold for any type environment* $\Gamma$ *or type* $t$*, because in the expression, integer addition is applied to a boolean value.*
- $\Gamma \vdash$ `3 + 1 * 5` : `bool`
  *does not hold for any type environment* $\Gamma$*, because the expression has type* `int`*, whereas* `bool` *is given after the* : *symbol.*

We define the typing relation inductively as follows.

The type of an identifier needs to be provided by the type environment.

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ [\textbf{VarT}]$$

If $\Gamma(x)$ is not defined, then this rule is not applicable. In this case, we say that there is no type for $x$ derivable from the assumptions $\Gamma$.

Constants get their obvious type. For any type environment $\Gamma$ and any integer $n$, the following rules hold:

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ [\textbf{NumT}] \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ [\textbf{TrueT}]$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\,[\textbf{FalseT}]$$

For each primitive operation in simPL, we have exactly one rule, as follows:

$$\frac{\Gamma \vdash E : \texttt{bool}}{\Gamma \vdash \backslash[E] : \texttt{bool}}\,[\textbf{Prim}_1] \qquad \frac{\Gamma \vdash E : \texttt{int}}{\Gamma \vdash \sim[E] : \texttt{int}}\,[\textbf{Prim}_2]$$

For each binary primitive operation $p_2$, we have a rule of the following form:

$$\frac{\Gamma \vdash E_1 : t_1 \qquad \Gamma \vdash E_2 : t_2}{\Gamma \vdash p_2[E_1, E_2] : t}\,[\textbf{PrimT}]$$

where the types $t_1, t_2, t$ are given by the following table.

| $p$ | $t_1$ | $t_2$ | $t$ |
|---|---|---|---|
| + | int | int | int |
| - | int | int | int |
| * | int | int | int |
| / | int | int | int |
| & | bool | bool | bool |
| \| | bool | bool | bool |
| = | int | int | bool |
| < | int | int | bool |
| > | int | int | bool |

Important for typing conditionals is that the "then" and the "else" clauses get the same type.

$$\frac{\Gamma \vdash E : \texttt{bool} \qquad \Gamma \vdash E_1 : t \qquad \Gamma \vdash E_2 : t}{\Gamma \vdash \texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2\ \texttt{end} : t}\,[\textbf{IfT}]$$

For function definition, we introduce the following rules.

$$\frac{\Gamma_1[x_1 \leftarrow t_1]\Gamma_2\ \cdots \Gamma_n[x_n \leftarrow t_n]\Gamma_{n+1} \qquad \Gamma_{n+1} \vdash E : t}{\Gamma_1 \vdash \texttt{fun}\ \{t_1\texttt{->}\cdots\texttt{->}t_n\ \texttt{->}\ t\}\ x_1\ \ldots x_n\ \texttt{->}\ E\ \texttt{end} : t_1\texttt{->}\cdots\texttt{->}t_n\ \texttt{->}\ t}\,[\textbf{FunT}]$$

Thus for a function definition to be well-typed under the assumptions given by type environment $\Gamma_1$, the body of the function needs to be well-typed under the assumptions given by an extended environment $\Gamma_{n+1}$, where $\Gamma_{n+1}$ extends $\Gamma_1$ with bindings of the function's formal parameters to its declared types. Furthermore, the type of the body needs to coincide with the declared return type

of the function.

**Example 1.3** *For the environment* $\Gamma'$ *given in Example 1.1, the following holds:*
$\Gamma' \vdash$ `fun {int -> bool} x -> AboutPi > x end : int -> bool`
*since*
$\Gamma_{ext} \vdash$ `AboutPi > x : bool`
*holds, where* $\Gamma_{ext}$ *extends* $\Gamma'$ *with a binding of* `x` *to the declared type* `int`*:*

$$\Gamma'[\boldsymbol{x} \leftarrow \boldsymbol{int}]\Gamma_{ext}$$

*Furthermore, the type of the body* `bool` *coincides with the declared return type of the function.*

Similarly, we have the following typing rule for recursive function definition.

$$\frac{\Gamma[f \leftarrow (t_1\texttt{->}\cdots\texttt{->}t_n\texttt{->}t)]\Gamma_1\ \Gamma_1[x_1 \leftarrow t_1]\Gamma_2\ \cdots\Gamma_n[x_n \leftarrow t_n]\Gamma_{n+1} \quad \Gamma_{n+1} \vdash E : t}{\Gamma \vdash \texttt{recfun}\ f\ \{t_1\texttt{->}\cdots\texttt{->}t_n\texttt{->}t\}\ x_1 \ldots x_n\ \texttt{->}\ E\ \texttt{end} : t_1\texttt{->}\cdots\texttt{->}t_n\texttt{->}t}\ \textbf{[RecFunT]}$$

Here, we find a type $t$ for the body of the function under the assumption that the function identifier has the type that is declared for the function.

Finally, we have the following rule for function application. One way of writing this rule is the following:

$$\frac{\Gamma \vdash E : t_1\texttt{->}\cdots\texttt{->}t_n\ \texttt{->}\ t \qquad \Gamma \vdash E_1 : t_1 \quad \cdots \quad \Gamma \vdash E_n : t_n}{\Gamma \vdash (E\ E_1 \cdots E_n) : t}\ \textbf{[ApplT]}$$

The type of the operator needs to be a function type with the right number of parameters, and the type of every argument needs to coincide with the corresponding parameter type of the function type. If all these conditions are met, the type of the function application is the same as the return type of the function type that is the type of the operator.

However, since we may support curried functions, this typing rule is a little restrictive since it did not allow partially-applied functions. To allow partially-applied applications which will return functions that expects the rest of the arguments, we shall be using the following more general binary application type rule instead:

$$\frac{\Gamma \vdash E : t_1\texttt{->}t \qquad \Gamma \vdash E_1 : t_1}{\Gamma \vdash (E\ E_1) : t}\ \textbf{[BinApplT]}$$

This completes the definition of the typing relation. Now we can define what it means for an expression to be well-typed.

**Definition 1.1** *An expression* $E$ *is well-typed, if there is a type* $t$ *such that* $E : t$*.*

Note that this definition of well-typedness requires that a well-typed expression has no free identifiers.

**Example 1.4** *The following proof shows that the typing relation holds for the expression* $\emptyset \vdash 2 * 3 > 7 : \texttt{bool}$.

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \texttt{2} : \texttt{int}} \quad \dfrac{}{\emptyset \vdash \texttt{3} : \texttt{int}}}{\emptyset \vdash \texttt{2->3} : \texttt{int}} \quad \dfrac{}{\emptyset \vdash \texttt{7} : \texttt{int}}}{\emptyset \vdash \texttt{2->3>7} : \texttt{bool}}$$

**Example 1.5** *The following proof shows that the typing relation holds for the expression*

$$\emptyset \vdash \ \textit{(fun int -> int x -> x+1 end 2)} : \textit{int}$$

*The reader may annotate each rule application with the name of the applied rule as in the previous example.*

$$\dfrac{\dfrac{\dfrac{}{\emptyset[\texttt{x} \leftarrow \texttt{int}]\Gamma} \quad \dfrac{\dfrac{}{\Gamma \vdash \texttt{x} : \texttt{int}} \quad \dfrac{}{\Gamma \vdash \texttt{1} : \texttt{int}}}{\Gamma \vdash \texttt{x+1} : \texttt{int}}}{\emptyset \vdash \texttt{fun } \{\texttt{int -> int}\} \texttt{ x -> x+1 end} : \texttt{int -> int}} \quad \dfrac{}{\emptyset \vdash \texttt{2} : \texttt{int}}}{\emptyset \vdash \texttt{(fun } \{\texttt{int -> int}\} \texttt{ x -> x+1 end 2)} : \texttt{int}}$$

**Lemma 1.1** *For every expression* $E$ *and every type assignment* $\Gamma$*, there exists at most one type* $t$ *such that* $\Gamma \vdash E : t$.

**Proof:** We prove this statement using structural induction over the given expression $E$. That means we consider the following property $P$ of simPL expressions $E$:

> For every type assignment $\Gamma$, there exists at most one type $t$ such that $\Gamma \vdash E : t$ holds.

If we are able to show that this property (taken as a set) meets all rules given for simPL expressions $E$, we know that simPL $\subseteq P$, which means that every element of simPL has the property $P$. So let us look at the rules defining simPL.

- $\dfrac{}{x}$

  The only typing rule that applies in this case is rule VarT (page 4). Since type environments are functions, it is immediately clear that for every type environment $\Gamma$, there can be at most one type for $x$, namely $\Gamma(x)$.

- $\dfrac{}{n}$, $\dfrac{}{\texttt{true}}$, $\dfrac{}{\texttt{false}}$

  The only typing rules that apply in these cases are the respective rules for typing of constants, NumT, TrueT and FalseT (page 4). They assign a unique type (`int` for numbers, `bool` for `true` and `false`) to the constant expressions.

- $\dfrac{E}{p_1[E]}$, $\dfrac{E_1 \quad E_2}{p_2[E_1, E_2]}$

  We need to show that our property $P$ meets the rules for simPL primitive operations. For our unary operation \, we need to show:

  > If for every type assignment $\Gamma$, there exists at most one type $t$ such that $\Gamma \vdash E : t$ holds, then for every type assignment $\Gamma'$, there exists at most one type $t'$ such that $\Gamma' \vdash \backslash[E] : t'$ holds.

  The only typing rule that applies in this case is the rule $\text{Prim}_1$. The only possible type for $\backslash[E]$ according to this rule is `bool`.

  The arguments for the other unary $\sim$ operator and the binary primitive operations are similar.

- $\dfrac{E \qquad E_1 \qquad E_2}{\texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ end}}$

  The only typing rule that applies here is the rule IfT.

$$\frac{\Gamma \vdash E : \texttt{bool} \qquad \Gamma \vdash E_1 : t \qquad \Gamma \vdash E_2 : t}{\Gamma \vdash \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ end} : t}\ \textbf{[IfT]}$$

It is clear from this rule that if there is at most one type $t$ for $E_1$, then there is at most one type for the entire conditional `if` $E$ `then` $E_1$ `else` $E_2$ `end`, namely the same type $t$.

- $$\frac{E \qquad E_1}{\texttt{(}E\ E_1\texttt{)}}$$

  The only rule that applies here is the rule BinApplT:

  $$\frac{\Gamma \vdash E : t_1\texttt{->}\ t \qquad \Gamma \vdash E_1 : t_1}{\Gamma \vdash \texttt{(}E\ E_1\texttt{)} : t}$$

  This rule applies only if $E$ has a type of the form $t_1$`->` $t$. It is clear from this rule that if there is only one such type $t_1$`->` $t$ for $E$ for any $\Gamma$, then there is at most one type for the entire application, namely $t$.

- $$\frac{E}{\texttt{fun}\ \{t_1\texttt{->}\cdots\texttt{->}t_n\ \texttt{->}\ t\}\ x_1 \cdots x_n\ \texttt{->}\ E\ \texttt{end}}$$

  The only rule that applies in this case is the rule FunT (page 5), which states that the type of a function definition can only be its declared type. Thus, our property $P$ meets the rule. Note that the do not even need to use the assumption that the body $E$ has property $P$.

- $$\frac{E}{\texttt{recfun}\ f\ \{t_1\texttt{->}\cdots\texttt{->}t_n\ \texttt{->}\ t\}\ x_1 \cdots x_n\ \texttt{->}\ E\ \texttt{end}}$$

  Similar to the case of function definition; the only rule that applies is RecFunT, which assigns the declared type to the recursive function definition.

□

Since for each expression, there is at most one rule that applies, we can invert the rules and state the following theorem.

**Theorem 1.1**

1. If $\Gamma \vdash x : t$, then $\Gamma(x) = t$.
2. If $\Gamma \vdash n : t$, then $t =$ `int`, for any integer $n$, and similarly for `true` and `false`.

3. If $\Gamma \vdash$ `if` $E$ `then` $E_1$ `else` $E_2$ `end` $: t$, then $\Gamma \vdash E :$ `bool`, $\Gamma \vdash E_1 : t$, and $\Gamma \vdash E_2 : t$.

4. If $\Gamma_1 \vdash$ `fun` $\{t_1$`->`$\cdots$`->`$t_n$ `->` $t\}$ $x_1 \ldots x_n$ `->` $E$ `end` $: t_1$`->`$\cdots$`->`$t_n$ `->` $t$, then there exist $\Gamma_2 \ldots \Gamma_{n+1}$ such that $\Gamma[x_1 \leftarrow t_1]\Gamma_2 \cdots \Gamma_n[x_n \leftarrow t_n]\Gamma_{n+1}$ and $\Gamma_{n+1} \vdash E : t$.

5. If $\Gamma \vdash$ `recfun` $f$ $\{t_1$`->`$\cdots$`->`$t_n$ `->` $t\}$ $x_1 \ldots x_n$ `->` $E$ `end` $: t_1$`->`$\cdots$`->`$t_n$ `->` $t$, then there exist $\Gamma_1 \ldots \Gamma_{n+1}$ such that $\Gamma[f \leftarrow t_1$`->`$\cdots$`->`$t_n$ `->` $t]\Gamma_1$, $\Gamma_1[x_1 \leftarrow t_1]\Gamma_2 \cdots \Gamma_n[x_n \leftarrow t_n]\Gamma_{n+1}$, and $\Gamma_{n+1} \vdash E : t$.

6. If $\Gamma \vdash$ `(`$E$ $E_1 \ldots E_n$`)` $: t$, then there exist types $t_1, \ldots, t_n$ such that $\Gamma \vdash E :$ $t_1$`->`$\cdots$`->`$t_n$ `->` $t$ and $\Gamma \vdash E_1 : t_1, \ldots \Gamma \vdash E_n : t_n$.

This theorem means that we can often infer the type of a given expression by looking at the form of the expression. Some programming languages exploit this fact by avoiding (most) type declarations for the user. The programming system carries out type inference and calculates the required type declarations. Type checking for such languages is done at the same time as type inference.

The following properties of the typing relation are useful for reasoning on types.

**Lemma 1.2** *Typing is not affected by "junk" in the type assignment. If* $\Gamma \vdash E : t$, *and* $\Gamma \subset \Gamma'$, *then* $\Gamma' \vdash E : t$.

**Lemma 1.3** *Substituting an identifier by an expression of the same type does not affect typing. If* $\Gamma[x \leftarrow t']\Gamma'$, $\Gamma' \vdash E : t$, *and* $\Gamma \vdash E' : t'$, *then* $\Gamma \vdash E'' : t$, *where* $E[x \leftarrow E']E''$.

## 1.2 Type Safety of simPL

Type safety is a property of a given language with a given static and dynamic semantics. It says that if a program of the languge is well-typed, certain problems are guaranteed not to occur at runtime.

What do we consider as "problems"? One kind of problem is that we would get stuck in the process of evaluation. That is the case when no evaluation rule applies to an expression, but the expression is not a value. We would like to be able to guarantee to make *progress* in evaluation, and that we will never get stuck because of type error. A second kind of problem is that the type changes as evaluation proceeds. For example, if the user declares that the result of a program should be of type `int`, then the evaluation cannot return a result of type `bool`. This property is called *preservation*.

The notion of *type safety* formalizes these two properties.

**Definition 1.2** *A programming language with a given typing relation* $\cdots \vdash \cdots : \cdots$ *and one-step evaluation* $\mapsto$ *is called type-safe, if the following two conditions hold:*

1. **Preservation.** *If $E$ is a well-typed program with respect to $\cdots \vdash \cdots : \cdots$ and $E \mapsto E'$, then $E'$ is also a well-typed program with respect to $\vdash$.*

2. **Progress.** *If $E$ is a well-typed program, then either $E$ is a value or there exists a program $E'$ such that $E \mapsto E'$.*

Is simPL type-safe? Neither preservation nor progress can hold without some assumptions on the primitive operations of the given language. For preservation, we must assume that if the result of applying an operation $p$ to arguments $v_1, \ldots, v_n$ is $v$ and $p[v_1, \ldots, v_n] : t$ then $v : t$. Fortunately, this is the case for all operators of the language simPL.

**Theorem 1.2 (Preservation)** *If for a simPL expression $E$ and some type $t$ holds $E : t$ and if $E \mapsto_{simPL} E'$, then $E' : t$.*

**Proof:** The proof is by structural induction on the rules defining simPL. □

**Lemma 1.4 (Canonical Forms)** *Suppose that the simPL expression $v$ is a closed, well-typed value and $v : t$.*

1. If $t =$ `bool`, then either $v =$ `true` or $v =$ `false`.

2. If $t =$ `int`, then $v = n$ for some $n$.

3. If $t = t_1$`->`$\cdots$`->`$t_n$ `->` $t'$, then
   $v =$ `fun` $\{t_1$`->`$\cdots$`->`$t_n$ `->` $t'\}$ $x_1 \ldots x_n$ `->` $E$ `end`,
   for some $x_1, \ldots, x_n$ and $E$, or
   $v =$ `recfun` $f$ $\{t_1$`->`$\cdots$`->`$t_n$ `->` $t'\}$ $x_1 \ldots x_n$ `->` $E$ `end`,
   for some $x_1, \ldots, x_n$ and $E$ and $f$.

**Proof:** The proof is by inspection of the typing rules. For example for the first statement, we look at all rules that assign types to values (TrueT, FalseT, NumT, FunT and RecFunT), and find that the only cases where the type is `bool` are TrueT and FalseT. □

For progress, we must assume that if $p[v_1, \ldots, v_n]$ is well-typed, then there exists a value $v$ such that $v$ is the result of applying $p$ to the arguments $v_1, \ldots, v_n$. This means that primitive operations are not allowed to be undefined on some arguments. Unfortunately, this is not the case for all operators of simPL. Integer division is not defined on 0 as first argument. So, let simPL' be the result of restricting simPL by excluding integer division from the set of primitive operators.

**Theorem 1.3 (Progress)** *If for a simPL' expression $E$ holds $E : t$ for some type $t$, then either $E$ is a value, or there exists an expression $E'$ such that $E \mapsto_{simPL'} E'$.*

**Proof:** The proof is by induction on the rules defining simPL'. □

The type safety of simPL' ensures that evaluation of a well-typed simPL' expression "behaves properly", which means does not get stuck (due to type error). Can we say the reverse by claiming that any expression for which the dynamic semantics produces a value is well-typed? If this was the case, the type system for simPL' would do a perfect job by statically identifying exactly those simPL' expressions that get stuck. Unfortunately, this is not the case. A simple counter-example is the expression

```
if true then 1 else false end
```

This expression evaluates to `1`, but is not well-typed.

## 1.3 Implementing Type System

The type rules we have provided are general, and can be used for type-checking, as well as type-inference to ensure that the entire program is type-safe. The type rules are of the following generic form:

$$\Gamma \vdash E : t$$

In the case of type-checking, we would have to implement a version of the above rule where all three items, namely $\Gamma$, $E$ and $y$ are given. The main task of type-checking is therefore to return either true or false, depending on whether the code is type-safe or otherwise. An implementation of such a pure type-checking procedure in OCaml would have the following type signature:

```
let type_check (env:env_type) (e:sPL_expr)
      (t:sPL_type) : bool = ...
```

However, since function applications are not fully annotated in our program, we would have to first infer the type of each function prior to checking such an application. To help in this process, we shall also provide a version of the type-checking procedure that would also infer the type of its expression. Such a function would minimally have the following signature:

```
let type_infer (env:env_type) (e:sPL_expr)
      : sPL_type option = ...
```

This inference procedure performs type-checking, and would also infer the type of its expression. If type-checking succeeds, we will return the type inferred/expected via Some constructor. If type-checking fails, due to some inconsistency, we simply return None as its result to indicate the presence of type-error.

This is the minimum we can do. However, it is often a good thing to return a new abstract syntax tree that is fully type-annotated after type inference itself. To help support this process, we could provide an implementation of type inference with the following signature:

```
let type_infer (env:env_type) (e:sPL_expr)
    : sPL_type option * sPL_expr = ...
```

Here, we also return a new expression that may have extra type annotations inserted (particularly for function application) if the type-checking process succeeded. If type-checking fails, we will return None together with the original expression instead. Returning a fully typeannotated expression is a useful thing, since it could help us with some extra consistency checking tasks. For example:

- It can be used to facilitate type-preserving program transformation.
- It allow us to check the outcome of either type-inference or type-preserving transformation. In case of bugs, this is often manifested by some type inconsistency in the resulting code.

# Chapter 7

# Denotational Semantics of simPL

## 7.1 Introduction

The small-step dynamic semantics that we have seen so far relies on the idea of contraction. An expression was evaluated by contracting subexpressions, which is repeatedly applied 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. We list some 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 is rather inefficient, as it relies on the concept of repeated checking for reducible expression.

As a result of these difficulties, dynamic semantics are not always the desired way for 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, which is closer in spirit to big-step operational semantics (or natural semantics). Operational semantics are semantics that are described in terms of

abstract machine configuration, and would have a set of reduction rules that mimic the execution of program code. Small-step operational semantics are closely related to dynamic semantics and would mirror the execution traces of program code. In contrast, big-step operational semantics are closer to denotational semantics as we attempt to assign some final equivalent value for each program code.

The idea of denotational semantics is to directly assign a mathematical value as a meaning to an expression. Compared to small-step 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 (and inefficient) 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 [**?**], 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 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.

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

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:

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

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

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:

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

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

**Proposition 7.1** *For all $n \in \boldsymbol{N}$, if $n \rightarrowtail_{\boldsymbol{N}} i$, and $n' \rightarrowtail_{\boldsymbol{N}} j$, 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 $n'\texttt{0} \rightarrowtail_{\mathbf{N}} 10 \cdot k$, 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 $n\texttt{9}' \rightarrowtail_{\mathbf{N}} j$, where $j = i + 1$. □

## 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.

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

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

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 **Int + 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.

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

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{*}E_2 \rightarrowtail v_1 \cdot v_2} \qquad \frac{E \rightarrowtail v}{{\sim}E \rightarrowtail 0{-}v}$$

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**.

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

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

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 $\Delta$, we introduce an operation $\Delta[x \leftarrow v]$, which denotes an environment that works like $\Delta$, 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.

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

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

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

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

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 $\Delta$ is undefined on the identifier, then the identifier has no meaning with respect to $\Delta$.

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

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

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

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 \texttt{*} E_2 \rightarrowtail v_1 \cdot v_2} \qquad \frac{\Delta \Vdash E \rightarrowtail v}{\Delta \Vdash {\sim}E \rightarrowtail 0 - v}$$

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

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

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

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 $\Delta$ 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**

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

## 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.

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

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

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 $\bot$.

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

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

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

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

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

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

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

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

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

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 $\bot$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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* $\Delta$, $\Delta \Vdash$`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.

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

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

| Semantic domain | Definition | Explanation |
|---|---|---|
| **Bool** | $\{true, false\}$ | ring of booleans |
| **Int** | $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ | ring of integers |
| **EV** | $\mathbf{Bool} + \mathbf{Int} + \{\bot\} + \mathbf{Fun}$ | 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:*

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

**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 [**?**] and [**?**].