CS4215—Programming Language Implementation

Martin Henz and Chin Wei Ngan

Sunday 8th January, 2017

Chapter 4

The Language simPL

In this chapter, we are extending the language ePL in order to provide a more powerful programming language. In particular, we extend ePL by the following features.

- conditional expressions,
- function definition and application, and
- recursive function definitions.

The language simPL allows us to study various styles of language implementation, and allows us to introduce the important notion of type safety in detail.

4.1 The Syntax of simPL

We divide the syntax of simPL into two categories, *types* and *expressions*. The set of types is the least set that satisfies the following rules.

$$\frac{}{\texttt{int}} \qquad \frac{}{\texttt{bool}} \qquad \frac{t_1 \quad t_2}{t_1\ \texttt{->}\ t_2}$$

The set of expressions is the least set that satisfies the following rules, where x ranges over a countably infinite set of identifiers V, n ranges over the integers, p_1 ranges over the set of unary primitive operations $P_1 = \{\texttt{\textbackslash},\texttt{\~{}}\}$, and p_2 ranges over the set of binary primitive operations $P_2 = \{\texttt{|},\texttt{\&},\texttt{+},\ \texttt{-},\texttt{*},\texttt{/},\ \texttt{=},\texttt{>},\texttt{<}\}$.

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

$$\frac{E}{p_1[E]} \qquad \frac{E_1 \quad E_2}{p_2[E_1, E_2]}$$

$$\frac{E \quad E_1 \quad E_2}{\texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ end}} \qquad \frac{E \quad E_1 \quad \cdots \quad E_n}{\texttt{(}E\ E_1 \cdots E_n\texttt{)}}$$

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

if $t_1, \ldots, t_n$ and t are types, $n \geq 1$. The identifiers $x_1, \ldots, x_n$ must be pairwise distinct.

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

Similarly, for the definition of recursive methods. If $t_1, \ldots, t_n$ and t are types, $n \geq 1$. The identifiers $f, x_1, \cdots, x_n$ must be pairwise distinct.
We shall maintain the meaning given to the primitive operators in ePL for simPL.

4.2 Syntactic Conventions

Similar to ePL, we introduce syntactic conventions that can be used in actual simPL programs:

- We can use parentheses in order to group expressions and types together.
- We use the usual infix and prefix notation for operators. The binary operators are left-associative and the usual precedence rules apply such that

  ```
  x + x * y > 10 - x
  ```

  stands for

  ```
  >[+[x,*[x,y]],-[10,x]]
  ```

- The type constructor `->` is right-associative, so that the type

  ```
  int -> int -> int
  ```

  is equivalent to

  ```
  int -> (int -> int)
  ```

Thus, the function

```
fun {int -> int -> int} x ->
   fun {int -> int} y -> x + y end
end
```

takes an integer `x` as argument and returns a function, whereas the function

```
fun {(int -> int) -> int} f -> (f 2) end
```

takes a function `f` as argument and returns an integer.

Based on the curried notation, the former can equivalently be expressed in a more succinct manner as:

```
fun {int -> int -> int} x y -> x + y end
```

4.3 Let As Abbreviation (or Syntactic Sugar)

We introduce the following convenient notation to allow for the introduction of local identifiers.

$$\texttt{let}\ \{t_1\}\ x_1 = E_1 \cdots \{t_n\}\ x_n = E_n\ \texttt{in}\ \{t\}\ E\ \texttt{end}$$

This would stand for:

$$\texttt{(fun}\ \{t_1\texttt{->}\cdots\texttt{->}t_n\ \texttt{->}\ t\}\ x_1 \cdots x_n\ \texttt{->}\ E\ \texttt{end}\ E_1 \cdots E_n\texttt{)}$$

Example 4.1 *Let us say we want to use the identifier* `AboutPi`*, which should stand for the integer 3, and the identifier* `Square`*, which should be the square function, inside an expression. For example, to calculate the surface of the earth in square kilometers, the average radius of the earth being 6371 km, we would like to write* `4 * AboutPi * {Square 6371}`*. Using* `let`*, we can do so, as shown below.*

```
let {int} AboutPi = 3
    {int -> int} Square =
         fun {int -> int} x -> x * x end
in  {int} 4 * AboutPi * (Square 6371)
end
```

According to the definition of `let`*, this expression is an abbreviation for*

```
(fun {int -> (int -> int) -> int}
    AboutPi Square
    ->
    4 * AboutPi * (Square 6371)
 end
 3
 fun {int -> int} x -> x * x end)
```

Exercise 4.1 *Translate the following Java functions into simPL with* `let`*.*

```
class TimesFour {
   public static int timestwo(int x) {
      return x + x;
   }
   public static int timesfour(int x) {
      return timestwo(timestwo(x));
   }
   public static void main(String[] args) {
      System.out.println(timesfour(10));
   }
}
```

Translate the result to simPL without `let`*, using the translation scheme above.*

Exercise 4.2 *The simPL language here can be viewed as a functional subset of OCaml. As an exercise, manually translate Example 4.1 into its equivalent OCaml code with corresponding let construct.*

In the rest of this chapter, we shall be freely making use of `let` in examples, knowing that it is just convenient syntax for function definition and application. We do not need to cover `let` expressions in our formal treatment of simPL.

4.4 Some simPL Programming

Example 4.2 (Power function) *In simPL, the power function, which raises a given integer x to the power of the integer y, can be defined as follows:*

```
recfun power {int -> int -> int}
   x y ->
   if y = 0
   then 1
   else x * (power x y - 1)
   end
end
```

Using the `let` *construct, we can use the power function inside an expression E as follows:*

```
let {int -> int -> int}
    power = recfun power {int -> int -> int}
               x y ->
               if y = 0
               then 1
               else x * (power x y - 1)
               end
            end
```

```
in  {int}
    (power 17 3)
end
```

Note the need to declare the identifier `power` *twice, which is a syntactic quirk of simPL. We could use a different identifier in the* `recfun` *expression without changing the meaning of the program:*

```
let {int -> int -> int}
    power = recfun mypower {int -> int -> int}
               x y ->
               if y = 0
               then 1
               else x * (mypower x y - 1)
               end
            end
in  {int}
    (power 17 3)
end
```

Exercise 4.3 *As an exercise in understanding simPL, rewrite the* `power` *method into its equivalent program in OCaml. You may place type ascriptions at similar places to that given in the Example 4.2.*

Our simPL language supports higher-order programming. We can provide a more general method for which the method `power` is just a special instance, by taking functions as parameters, as shown in our next example.

Example 4.3 (General iteration) *We can generalize the recursion over one integer by passing the function to be applied at each step and the value for 0 as arguments.*

```
let {int -> int -> (int -> int -> int) -> int -> int}
    recurse
    = recfun recurse
          {int -> int -> (int -> int -> int) -> int -> int}
          x y operation initvalue
       -> if y = 0 then initvalue
          else (operation x
                (recurse x (y - 1) operation initvalue))
          end
      end
in {int}
...
(recurse 2 3 fun {int -> int -> int}
                x z -> x * z
             end
```

```
         1)
...
(recurse 2 3 fun {int -> int -> int}
                x z -> x + z end
         0)
...
(recurse 2 3 fun {int -> int -> int}
                x z -> z / x end
         128)
...
end
```

The three applications of `recurse` *compute the numbers 8, 6 and 16, respectively.*

Exercise 4.4 *Which of the above uses of* `recurse` *is a special instantiation of the* `power` *method? Can you similarly rewrite this example into OCaml?*

4.4.1 Implementation OCaml

Now that we have seen the various features of simPL, let us now look at how they can be implemented in OCaml. Firstly, types for sPL can be implemented, as follows.

```
type sPL_type =
  | BoolType
  | IntType
  | Arrow of sPL_type * sPL_type
```

Note that we have only two basic types, IntType and BoolType, and a rather expressive function type that is denoted by the Arrow constructor. This type supports higher-order programming since it may take values of function-type as arguments and also return results that are of the function-type.

The various language features of simPL may now be implemented as the following abstract syntax tree data type below:

```
type sPL_expr =
  | BoolConst of bool
  | IntConst of int
  | Var of id
  | UnaryPrimApp of op_id * sPL_expr
  | BinaryPrimApp of op_id * sPL_expr * sPL_expr
  | Cond of sPL_expr * sPL_expr * sPL_expr
  | Func of sPL_type * (id list) * sPL_expr
  | RecFunc of sPL_type * id * (id list) * sPL_expr
  | Appln of sPL_expr * sPL_type option
           * (sPL_expr list)
  | Let of ((sPL_type * id * sPL_expr) list)
           * sPL_type * sPL_expr
```

Most of the features are taken directly from the concrete syntax. Compared to the earlier ePL language, we have now added variable Var, conditional constructs Cond, two ways of building functions Func and RecFunc, a more general way of applying functions to arguments Appln, and also the let construct Let. We have kept our earlier formulations for unary and binary primitive operators, though these could have been merged into the more general notation of function application.

One interesting thing to note carefully is the presence of types in the abstract syntax tree. Their presence facilitate the type-checking and type-safe program transformation process. We give a type for each function constructed, and this is sufficient to help us determine the types of arguments of our functions. We also give types for let bindings and its results, and this is sufficient to help us transform each let construct into its corresponding function application.

In the case of function application, it is typically written as (e e1 .. en) without the need for any types to be given. However, during type checking, we would need to first figure out the type for e, from which we can then determine the types of e1 .. en and the type of the application itself. To facilitate the process, we have used an option type (sPL_type option) to capture the type of function e. At the start, this type would be None since programmers are not required to supply it. However, after a type inference process, we would be able to figure out the type and could replace it with the inferred type into our abstract syntax tree. Hence, the need for option type here for the type of function in general application.

Chapter 5

Dynamic Semantics of simPL

In order to define how programs are executed, we use an approach similar to evaluation of ePL expressions. We are going to define a contraction, and based on this contraction, we define inductively a relation that tells us how to evaluate simPL programs. However, the one-step evaluation we shall introduce for simPL is going to be more restricted than evaluation in ePL, because we would like it to be *deterministic*, which means there has to be at most one way to perform each evaluation step. We will then define evaluation as the reflexive transitive closure of one-step evaluation.

5.1 Values

The goal of evaluating an expression is to reach a *value*, an expression that cannot be further evaluated. In simPL, a value is either an integer, or a boolean value or a function (`fun`$\cdots$`->`$\cdots$`end` or `recfun`$\cdots$`->`$\cdots$`end`). In the following rules defining the contraction relation $>_{\text{simPL}}$ for simPL, we denote values by v. That means any rule in which v appears is restricted to values in the place of v.

Note that function values can have executable expressions in their body. For example,

```
fun {int -> int} x -> 3 * 4 end
```

is a value although its body `3 * 4` is not a value. So in contrast to ePL, where contraction can be applied to any subexpression, the dynamic semantics of simPL prevents evaluation within the bodies of function definitions. This notion of values conforms with the intuition that the body of a function gets executed only when the function is applied.

5.2 Contraction

As for ePL, we define contraction rules for each primitive operation p and each set of values v_1, v_2 such that the result of applying p to v_1 and v_2 is a value v.

$$\frac{}{p_1[v_1] >_{\text{simPL}} v}\text{[OpVals]} \qquad \frac{}{p_2[v_1, v_2] >_{\text{simPL}} v}\text{[OpVals]}$$

Contraction of conditionals distinguishes the cases that the condition is `true` or `false`.

$$\frac{}{\texttt{if true then } E_1 \texttt{ else } E_2 \texttt{ end} \quad >_{\text{simPL}} \quad E_1}\text{[IfTrue]}$$

$$\frac{}{\texttt{if false then } E_1 \texttt{ else } E_2 \texttt{ end} \quad >_{\text{simPL}} \quad E_2}\text{[IfFalse]}$$

These two contractions require the conditional test to be values, while the branches can be expressions that are yet to be evaluated.

Lastly, in order to define the contraction of function application, we need two further definitions; *free identifiers* and *substitution*.

Free Identifiers

We need to be able to find out what identifiers in a given simPL expression are bound by enclosing function definitions, and what identifiers are free, e.g. not bound. For example, the identifier `square` occurs free in

```
(fun {int -> int} x -> 4 * (square x) end 3)
```

because it is not declared by any surrounding `fun` or `recfun` expression, whereas the identifier `x` is bound by the surrounding `fun` expression.

Formally, we are looking for a relation

$$\bowtie: \text{simPL} \times 2^V$$

that defines the set of free identifiers of a given expression. For example, `4 * (square x)` $\bowtie$ {`square`,`x`}, which we read as "the set of free identifiers of the expression `4 * (square x)` is {`square`,`x`}.

The relation $\bowtie$ is defined by the following rules:

$$\frac{}{x \bowtie \{x\}} \qquad \frac{}{n \bowtie \emptyset} \qquad \frac{}{\texttt{true} \bowtie \emptyset} \qquad \frac{}{\texttt{false} \bowtie \emptyset}$$

$$\frac{E \bowtie X}{p_1[E] \bowtie X} \qquad \frac{E_1 \bowtie X_1 \qquad E_2 \bowtie X_2}{p_2[E_1, E_2] \bowtie X_1 \cup X_2}$$

$$\frac{E_1 \bowtie X_1 \qquad E_2 \bowtie X_2 \qquad E_3 \bowtie X_3}{\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \texttt{ end} \bowtie X_1 \cup X_2 \cup X_3}$$

$$\frac{E \bowtie X}{\texttt{fun } \{\cdot\}\ x_1 \cdots x_n \texttt{ -> } E \texttt{ end} \bowtie X - \{x_1, \ldots, x_n\}}$$

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

Exercise 5.1 *Prove that the relation $\bowtie$ is a total function, i.e. for every simPL expression E, there is exactly one set of identifiers X such that $E \bowtie X$.*

Substitution

In order to carry out a function application, we need to replace all free occurrences of the formal parameters in the function body by the actual arguments. For example, in order to contract the expression

```
(fun {int -> int} x -> x * x end 4)
```

we need to replace every free occurrence of `x` in the body of the function `x * x` by the actual parameter `4`, leading to the expression `4 * 4`.

Formally, substitution is defined by the substitution relation

$$\cdot\,[\cdot{\leftarrow}\cdot] \rightsquigarrow \cdot : \text{simPL} \times V \times \text{simPL} \times \text{simPL}$$

such that `x * x`[`x` $\leftarrow$ `4`]`4 * 4` holds.

In order to define the relation $\cdot\,[\cdot{\leftarrow}\cdot] \rightsquigarrow \cdot$ we employ as usual an inductive definition using the following rules.

$$\frac{}{v\,[v{\leftarrow}E_1] \rightsquigarrow E_1}\text{for any identifier } v$$

$$\frac{}{x\,[v{\leftarrow}E_1] \rightsquigarrow x}\text{for any identifier } x \neq v$$

The rules for primitive applications, function applications, and conditionals apply the substitution to all components. For example, the rule for a one-argument function application looks like this:

$$\frac{E_1\,[v{\leftarrow}E] \rightsquigarrow E_1' \qquad E_2\,[v{\leftarrow}E] \rightsquigarrow E_2'}{\texttt{(}E_1\ E_2\texttt{)}\,[v{\leftarrow}E] \rightsquigarrow \texttt{(}E_1'\ E_2'\texttt{)}}$$

The rules for multiple-argument applications, primitive applications and conditionals are left as an exercise.

Exercise 5.2 *Give the rule that defines substitution for conditionals.*

Let us now look at the substitution rule for unary functions. It is sufficient to focus on unary function since each multiple-argument function can easily be converted to a series of unary functions. The rules are as follows.

$$\frac{}{\texttt{fun } \{\cdot\}\, v \texttt{ -> } E \texttt{ end}\,[v{\leftarrow}E_1] \rightsquigarrow \texttt{fun } \{\cdot\}\, v \texttt{ -> } E \texttt{ end}}$$

$$\frac{E\,[v{\leftarrow}E_1] \rightsquigarrow E' \qquad x \neq v \qquad E_1 \bowtie X_1 \qquad x \notin X_1}{\texttt{fun } \{\,\cdot\,\}\ x \texttt{->} E \texttt{ end}\,[v{\leftarrow}E_1] \rightsquigarrow \texttt{fun } \{\,\cdot\,\}\ x \texttt{ -> } E' \texttt{ end}}$$

$$\frac{E_1 \bowtie X_1 \quad x \in X_1 \quad E \bowtie X \quad E[x \leftarrow z]E' \quad E'[v \leftarrow E_1]E'' \quad x \neq v}{\texttt{fun } \{\cdot\}\, x \texttt{ -> } E \texttt{ end}\,[v{\leftarrow}E_1] \rightsquigarrow \texttt{fun } \{\cdot\}\, z \texttt{ -> } E'' \texttt{ end}}$$

where we choose z such that $z \notin X_1 \cup X$.

Example 5.1 *The following substitutions hold:*

- `fun {int -> int} factor -> factor * 4 * y end`
  `[factor`$\leftarrow$ `x + 1]` $\rightsquigarrow$
  `fun {int -> int} factor -> factor * 4 * y end`

- `fun {int -> int} factor -> factor * 4 * y end`
  [y← `x + 1`] $\rightsquigarrow$
  `fun {int -> int} factor -> factor * 4 * (x + 1) end`
- `fun {int -> int} factor -> factor * 4 * y end`
  [y← `factor + 1`] $\rightsquigarrow$
  ```
  fun {int -> int} newfactor ->
     newfactor * 4 * (factor + 1) end
  end
  ```

Exercise 5.3 *Give the rule for substitution of ternary recursive function definition.*

Exercise 5.4 *Is substitution a mathematical function on its first three arguments, i.e. for any given expressions E_1 and E_2, and identifier x, is there is exactly one expression E_3 such that $E_1[x \leftarrow E_2] \rightsquigarrow E_3$ holds?*

Contraction of Function Application

We define function application of unary (non-recursive) functions as follows.

$$\frac{E\,[x{\leftarrow}v] \rightsquigarrow E'}{(\texttt{fun}\ \{\,\cdot\,\}\ x\ \texttt{->}\ E\ \texttt{end}\quad v\texttt{)} >_{\text{simPL}} E'}\,[\text{CallFun}]$$

Note that the arguments of a function application must be values (denoted by the letter v), before the function is applied.

In order to define contraction of an application of a recursive function, we need to make sure that the recursive function is used, when the body is evaluated. We do this by replacing every free occurrence of the function identifier by the definition of the function.

$$\frac{E\,[f{\leftarrow}\texttt{recfun}\ \{\,\cdot\,\}\ f\ x\ \texttt{->}\ E\ \texttt{end}] \rightsquigarrow E' \quad E'\,[x{\leftarrow}v] \rightsquigarrow E''}{(\texttt{recfun}\ f\ x\ \texttt{->}\ E\ \texttt{end}\quad v\texttt{)} >_{\text{simPL}} E''}\,[\text{RF}]$$

Exercise 5.5 *Note that this rule first replaces f by the recursive function definition, and then x by the argument v. Consider a version of the rule that does it the other way around. Would you always get the same result?*

Application of Functions with Multiple Parameters

The rules presented so far only handle functions (and recursive functions) with single arguments. A simple way to handle multiple argument functions is by treating them as an abbreviation for single-argument functions, in the following way:

$$\frac{\begin{array}{l}(\cdots((\texttt{fun}\ \{t_1\ \texttt{->}\ t_2\ \texttt{->}\cdots\texttt{->}\ t_n\ \texttt{->}\ t\ \}\ x_1\ \texttt{->}\\ \qquad \texttt{fun}\ \{t_2\ \texttt{->}\cdots\texttt{->}\ t_n\ \texttt{->}\ t\ \}\ x_2\ \texttt{->}\ \cdots\texttt{->}\\ \qquad\qquad \texttt{fun}\ \{t_n\ \texttt{->}\ t\}\ x_n\ \texttt{->}\\ \qquad\qquad\qquad E\\ \qquad\qquad \texttt{end}\\ \qquad\qquad \cdots\\ \qquad \texttt{end}\\ \qquad \texttt{end}\\ \qquad v_1\texttt{)}\\ \qquad v_2\texttt{)}\\ \vdots\\ v_n\texttt{)}\\ >_{\text{simPL}}\ E'\end{array}}{\texttt{(fun}\ \{\ t_1\ \texttt{->}\ \cdots\ \texttt{->}\ t_n\ \texttt{->}\ t\ \}\ x_1\cdots x_n\ \texttt{->}\ E\ \texttt{end}\quad v_1\cdots v_n\texttt{)}\ >_{\text{simPL}}\ E'}\text{[MP]}$$

Another approach is to accept handle multiple-arguments function directly, but to consider a function application $fE_1..E_m$ to be reducible only when it has the full set of arguments that are expected from f. If a function application has only some (but not all) of its expected arguments, it will be treated as a (irreducible) value.

5.3 One-Step Evaluation

We define one-step evaluation $\mapsto_{\text{simPL}}$ inductively by the following rules.

The base case for evaluation is contraction.

$$\frac{E >_{\text{simPL}} E'}{E \mapsto_{\text{simPL}} E'}\text{[Contraction]}$$

The application of primitive operations is evaluated using the following rules.

$$\frac{E \mapsto_{\text{simPL}} E'}{p_1[E] \mapsto_{\text{simPL}} p_1[E']}\text{[OpArg}_1\text{]} \qquad \frac{E_1 \mapsto_{\text{simPL}} E_1'}{p_2[E_1, E_2] \mapsto_{\text{simPL}} p_2[E_1', E_2]}\text{[OpArg}_2\text{]}$$

$$\frac{E_2 \mapsto_{\text{simPL}} E_2'}{p_2[v_1, E_2] \mapsto_{\text{simPL}} p_2[v_1, E_2']}\text{[OpArg}_3\text{]}$$

Note that for the second argument of a primitive application to be evaluated, the first argument must be a value. That means that applications are evaluated from left to right. In conditionals, only the condition can be evaluated.

$$\frac{E \mapsto_{\text{simPL}} E'}{\texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{end} \mapsto_{\text{simPL}} \texttt{if } E' \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ end}} \text{[IfTest]}$$

Eventually, the condition will evaluate to *true* or *false*, at which point one of the two Contraction rules applies; see Section 5.2.

Function applications are also evaluated from left to right, i.e. first the function position and then the argument positions from left to right.

$$\frac{E \mapsto_{\text{simPL}} E'}{\texttt{(}E\ E_1 \ldots E_n\texttt{)} \mapsto_{\text{simPL}} \texttt{(}E'\ E_1 \ldots E_n\texttt{)}} \text{[AppFun]}$$

$$\frac{E_i \mapsto_{\text{simPL}} E_i'}{\texttt{(}v\ v_1 \ldots v_{i-1}\ E_i \ldots E_n\texttt{)} \mapsto_{\text{simPL}} \texttt{(}v\ v_1 \ldots v_{i-1}\ E_i' \ldots E_n\texttt{)}} \text{[AppArg]}$$

This completes the definition of one-step evaluation. Note that there are no rules for function definitions. The extremal clause for one-step evaluation implies that no contractions can be carried out in function bodies.

5.4 Evaluation

As in the dynamic semantics of ePL, the evaluation relation $\mapsto_{\text{simPL}}{}^*$ is defined as the reflexive transitive closure of one-step evaluation $\mapsto_{\text{simPL}}$.

Lemma 5.1 *For every closed expression E, there exists at most one expression E' such that $E \mapsto_{simPL} E'$, up to renaming of bound identifiers.*

Proof: By induction over the structure of E. □

Lemma 5.1 states that $\mapsto_{\text{simPL}}$ is a partial function. This means that evaluation is deterministic; there exists only one place in any expression, where evaluation can apply a contraction.

Lemma 5.2 *For every closed expression E, there exists at most one value v such that $E \mapsto_{simPL}{}^* v$, up to renaming of bound identifiers.*

Proof: Let us assume that there are expressions v_1 and v_2 such that for a given expression E, we have $E{\mapsto_{\text{simPL}}}^*v_1$ and $E{\mapsto_{\text{simPL}}}^*v_2$. From Lemma 1 and the definition of reflexive transitive closure follows that $v_1 {\mapsto_{\mathrm{simPL}}}^* v_2$ (or $v_2 {\mapsto_{\mathrm{simPL}}}^* v_1$, in which case the following argument is similar). According to the definition of values, the expression v_1 could be an integer, a boolean or a (possibly recursive) function. In none of these cases, there is a rule in the definition of $\mapsto_{\mathrm{simPL}}$ with which to evaluate v_1 in one step. According to the definition of $\mapsto^*_{\mathrm{simPL}}$, we must therefore have $v_1 = v_2$. $\square$

Chapter 6

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 6.1 uses a typing relation to identify well-typed simPL expressions. What properties do well-typed expressions have? Section 6.2 answers this question by showing that the evaluation of well-typed expressions enjoys specific properties.

6.1 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 Γ, that keeps track of the type of identifiers appearing in the expression. More formally, the partial function Γ 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 Γ, identifiers x, types t, and type environments Γ', 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 Γ' for a given Γ, 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 Γ is defined, is called the domain of Γ, 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 6.1 *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* Γ' *is an environment that can be applied only to the identifier* `AboutPi`; the result of $\Gamma'(\texttt{AboutPi})$ is the type `int`. *Similarly, we can define* Γ'' *by* $\Gamma'[\texttt{Square} \leftarrow \texttt{int -> int}]\Gamma''$*. The type environment* Γ'' *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 Γ 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 Γ". When E has no free identifiers (we say E is *closed*), we can write $E : t$ instead of $\emptyset \vdash E : t$.

Example 6.2 *Continuing Example 6.1, we will define the typing relation such that the following expressions hold:*

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

but:

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

Constants get their obvious type. For any type environment Γ 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 Γ_1, the body of the function needs to be well-typed under the assumptions given by an extended environment Γ_{n+1}, where Γ_{n+1} extends Γ_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 6.3 *For the environment Γ' given in Example 6.1, the following holds:*
$\Gamma' \vdash$ `fun {int -> bool} x -> AboutPi > x end : int -> bool`
since
$\Gamma_{ext} \vdash$ `AboutPi > x : bool`
holds, where Γ_{ext} extends Γ' 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:

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

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:

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

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

Definition 6.1 *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 6.4 *The following proof shows that the typing relation holds for the expression* $\emptyset \vdash 2 * 3 > 7 : \texttt{bool}$*.*

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

Example 6.5 *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.

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

Lemma 6.1 *For every expression E and every type assignment Γ, 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 Γ, 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.

- $$\frac{}{x}$$

  The only typing rule that applies in this case is rule VarT (page 20). Since type environments are functions, it is immediately clear that for every type environment Γ, there can be at most one type for x, namely $\Gamma(x)$.

- $$\frac{}{n}, \quad \frac{}{\texttt{true}}, \quad \frac{}{\texttt{false}}$$

  The only typing rules that apply in these cases are the respective rules for typing of constants, NumT, TrueT and FalseT (page 20). They assign a unique type (`int` for numbers, `bool` for `true` and `false`) to the constant expressions.

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

  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 Γ, there exists at most one type t such that $\Gamma \vdash E : t$ holds, then for every type assignment Γ', 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 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.

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

  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 Γ, 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 21), 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 6.1

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 `->` $t\}$ $x_1 \dots x_n$ `->` E `end` $: t_1$`->`$\cdots$`->`t_n `->` t, then there exist $\Gamma_2 \dots \Gamma_{n+1}$ 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 `->` $t\}$ $x_1 \dots x_n$ `->` E `end` $: t_1$`->`$\cdots$`->`t_n `->` t, then there exist $\Gamma_1 \dots \Gamma_{n+1}$ 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 $\Gamma \vdash$ `(`$E\ E_1 \dots E_n$`)` $: t$, then there exist types $t_1, \dots, t_n$ such that $\Gamma \vdash E : t_1$`->`$\cdots$`->`t_n `->` t and $\Gamma \vdash E_1 : t_1, \dots \Gamma \vdash E_n : t_n$.

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 6.2 *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 6.3 *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* $E[x \leftarrow E']E''$.

6.2 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 6.2 *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 6.2 (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. $\square$

Lemma 6.4 (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. $\square$

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 6.3 (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.

6.3 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 Γ, 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.