CS4215—Programming Language Implementation

Martin Henz and Chin Wei Ngan

Sunday 8th January, 2017

Chapter 10

imPL: A Simple Imperative Language

10.1 Introduction

So far, we considered only languages, in which an identifier refers to a value. Once the value is computed, it does not change. Pass-by-need makes use of this property by avoiding repeated evaluation; the resulting value would be the same, anyway. A given identifier in a given environment always denotes the same value. A given expression is always evaluated to the same value in a given environment. Languages that have this property are called *referentially transparent*. This property makes it particularly easy to formally define language semantics and to reason about programs such as prove their correctness and termination. The languages ePL, simPL and dPL are referentially transparent.

However, it is often convenient to deviate from referential transparency. Certain algorithms can be formulated more naturally and certainly more efficiently in a language, in which some mutable identifiers refer to locations of a random-access memory. An operation called *assignment* allows to change the value stored in the memory location associated with such a mutable identifier. Languages with such a construct are called *imperative* languages.

In this chapter, we examine the semantics and implementation of imperative languages. We start with a conservative extension of simPL with typical imperative constructs such as assignment and loops, resulting in imPL0. In order to investigate parameter passing techniques, we extend imPL0 by data constructor with mutable fields, resulting in imPL1. We will examine pass-by-value and pass-by-reference parameter passing for identifiers and pass-by-copy parameter passing for mutable data constructors.

10.2 imPL0: Imperative Programming

We extend the syntax of simPL by assignments, sequences and while-loops, resulting in imPL0.

Assignment. An assignment expression allows to change the value of an identifier to the result of evaluating an expression:

$$\frac{E}{x\ \texttt{:=}\ E}$$

Sequence. A sequence expression allows to evaluate first on expression to change the value of identifiers, and then evaluate another expression with the changes in effect:

$$\frac{E_1 \quad E_2}{E_1\ \texttt{;}\ E_2}$$

While-loop. A while loop allows to repeatedly evaluate an expression as long as a boolean expression evaluates to *true*:

$$\frac{E_1 \quad E_2}{\texttt{while}\ E_1\ \texttt{do}\ E_2\ \texttt{end}}$$

10.3 Examples

Example 10.1 *We classify local variables declared in a let construct into two categories. By default, each variable is immutable, but it could be explicitly declared as mutable with a* `mut` *qualifier. The body of the following let expression may repeatedly change the value of a mutable identifier* `x`*, but not an immutable identifier* `y`*. The results of evaluation of intermediate values of mutable identifiers are ignored. They are only executed for their "side effect", namely the changing of the value of mutable identifier* `x`*.*

```
let mut x = 0
    y = 3
in
   x := 1;
   x := x + 2;
   x := x + y;
   x
end
```

Example 10.2 *We now have the possibility to write interesting programs without using recursion. An alternative function for computing the factorial function is:*

```
fun x ->
   let mut i = 1
       mut f = 1 in
         while i <= x do
            f := f * i;
            i := i + 1
         end;
         f
      end
   end
end
```

Example 10.3 *The GCD program can now be written as a loop, with the help of pass-by-reference parameters, expressed with the mutable qualifier:*

```
let gcd = fun mut a mut b ->
             while (a = b) do
                if a > b
                then a := a - b
                else b := b - a
                end
             end;
             a
          end
    mut c = 6
    mut d = 10
in
   (gcd c d)
```

10.4 Denotational Semantics of imPL0

Of course, the language that we use to describe our denotational semantics, namely mathematical notation, enjoys referential transparency. So the question arises how to describe assignment in our denotational semantics.

Example 10.4 `let mut x = 0`

```
    y = 3
in
   x := 1;
   x := x + 2;
   x := x + y;
   x
end
```

This program can only be understood in terms of its effect on the stored value of mutable identifier `x`. In order to make the intuitive notion of "stored value" explicit in our semantic framework, we introduce a new semantic domain, called the *store*. In that way we can say that the first assignment makes a binding of `x` to 1 in a given store Σ, resulting in a new store Σ'. This new store is then used by the second assignment, resulting in yet another store Σ'' and so on. By introducing the store, on which a program operates, we can describe the meaning of programs in a referentially transparent way.

We are changing the semantic domains such that now, identifiers always denote *locations*. These locations are used to access a store, which holds storable values.

Domain name	Definition	Explanation
EV	$\mathbf{Int} + \mathbf{Bool} + \mathbf{Fun} + \perp(\mathbf{Int})$	expressible values
SV	$\mathbf{Int} + \mathbf{Bool} + \mathbf{Fun}$	storable values
DV	$\mathbf{SV} + \mathbf{Loc}$	denotable values
Fun	$\mathbf{DV} * \cdots * \mathbf{DV} * \mathbf{Store} \leadsto (\mathbf{EV},\ \mathbf{Store})$	function values
Store	$\mathbf{Loc} \leadsto \mathbf{SV}$	stores
Env	$\mathbf{Id} \leadsto \mathbf{DV}$	environments

Environments now refer to either values or locations. The locations are passed to stores in order to access the current value of the identifier with respect to the store.

For stores Σ we introduce an operation $\Sigma[l \leftarrow v]$, which denotes a store that works like Σ, except that $\Sigma(l) = v$ (similar to the corresponding operation on environments). The symbol $\emptyset_{\mathbf{Store}}$ stands for the empty store, and similarly $\emptyset_{\mathbf{Env}}$ stands for the empty environment.

Example 10.5 *Let us say we have a store with the value 1 at location l*

$$\Sigma = \emptyset_{\mathbf{Store}}[l \leftarrow 1]$$

and an environment that carries the location l at identifier x

$$\Delta = \emptyset_{\mathbf{Env}}[x \leftarrow l]$$

Then we can access the value of x in the store as follows:

$$\Sigma(\Delta(x)) = 1$$

By introducing the store on which a program operates, we can describe the meaning of programs in a referentially transparent way.

The semantic function $\cdot \rightarrowtail \cdot$ now needs to be defined using a semantic function $\cdot \mid \cdot \Vdash \cdot \rightarrowtail \cdot$ that gets a store as additional argument.

$$\cdot \rightarrowtail \cdot : \textbf{imPL0} \to \textbf{EV}$$

$$\frac{\emptyset_{\textbf{Store}} \mid \emptyset_{\textbf{Env}} \Vdash E \rightarrowtail (v, \Sigma)}{E \rightarrowtail v}$$

The semantic function $\cdot \mid \cdot \Vdash \cdot \rightarrowtail \cdot$ is defined as a four-argument relation (ternary partial function):

$$\cdot \mid \cdot \Vdash \cdot \rightarrowtail \cdot : \textbf{Store} * \textbf{Env} * \textbf{imPL0} \to \textbf{EV} * \textbf{Store}$$

The evaluation of `let` expressions works as follows (for clarity, we limit ourselves to the case with only one definition). In the case of immutable identifiers, we have:

$$\frac{\begin{array}{c}\Sigma \mid \Delta \Vdash E_1 \rightarrowtail (v_1, \Sigma') \\ \Sigma' \mid \Delta[x_1 \leftarrow v_1] \Vdash E \rightarrowtail (v, \Sigma'')\end{array}}{\Sigma \mid \Delta \Vdash \texttt{ let } \{\cdot\}\ x_1 = E_1 \texttt{ in } E \texttt{ end} \rightarrowtail (v, \Sigma'')}$$

In the case of mutable identifiers, we have:

$$\frac{\begin{array}{c}\textit{fresh } l_1 \qquad \Sigma \mid \Delta \Vdash E_1 \rightarrowtail (v_1, \Sigma') \\ \Sigma'[l_1 \leftarrow v_1] \mid \Delta[x_1 \leftarrow l_1] \Vdash E \rightarrowtail (v, \Sigma'')\end{array}}{\Sigma \mid \Delta \Vdash \texttt{ let } \textit{mut } \{\cdot\}\ x_1 = E_1 \texttt{ in } E \texttt{ end} \rightarrowtail (v, \Sigma'')}$$

Take note that l_1 is a new location which means $\Sigma'(l_1)$ is undefined. Furthemore, as it is tied closely to the identifier, we can remove it from the store once the scope of the mutable identifier has ended.

Correspondingly, the evaluation of mutable identifiers needs to access the store.

$$\frac{\Delta(x) \in \textit{dom}(\Sigma)}{\Sigma \mid \Delta \Vdash x \rightarrowtail (\Sigma(\Delta(x)), \Sigma)}$$

On the other hand, the evaluation of immutable identifiers need only access the environment, as follows:

$$\frac{\Delta(x) \notin dom(\Sigma)}{\Sigma \mid \Delta \Vdash x \rightarrowtail (\Delta(x), \Sigma)}$$

The semantic functions for assignment returns an updated store.

$$\frac{\Delta(x) \in dom(\Sigma) \qquad \Sigma \mid \Delta \Vdash E \rightarrowtail (v, \Sigma')}{\Sigma \mid \Delta \Vdash x \texttt{ := } E \rightarrowtail (v, \Sigma'[\Delta(x) \leftarrow v])}$$

Example 10.6 $\emptyset_{\textit{\textbf{Store}}}[l \leftarrow 1] \mid \emptyset_{\textit{\textbf{Env}}}[\texttt{a}\leftarrow l] \Vdash \texttt{a := 2} \rightarrowtail (2, \emptyset_{\textit{\textbf{Store}}}[l \leftarrow 1][l \leftarrow 2])$
The resulting store $\emptyset_{\textit{\textbf{Store}}}[l \leftarrow 1][l \leftarrow 2])$ is of course the same as $\emptyset_{\textit{\textbf{Store}}}[l \leftarrow 2]$. The original binding of l to 1 is overwritten by the new value 2.

Note that assignment is defined to always evaluate to the value to which its right hand side is evaluated. This choice is somewhat arbitrary, since typically, assignments are carried out for their "side effect", only, which means for the effect they have on the store.

In an imperative language, the question arises what parameter passing means when identifiers occur in argument position. Is the denotable value passed directly to the function, or do we create a new location, and place the value of the argument in the store at this new location? The first possibility is called "pass-by-reference" and the second "pass-by-value". We decide to use pass-by-value for identifiers, following most modern imperative languages. The following equations describe a general parameter passing. For simplicity, we only treat single-argument functions here.

$$\frac{}{\Sigma \mid \Delta \Vdash \texttt{fun } x \texttt{ -> } E \texttt{ end} \rightarrowtail (f, \Sigma)} \quad \begin{array}{l}\text{where } f(v, \Sigma') = (v', \Sigma''),\\ \text{where } \Sigma' \mid \Delta[x \leftarrow v] \Vdash E \rightarrowtail (v', \Sigma'')\end{array}$$

Note that v may either be a value or a location. By default a value will be passed into the parameter. However, if the parameter is declared to be mutable, a location with be passed into the method instead. With this, we will then be using the pass-by-reference mechanism. Correspondingly, function application (using pass-by-value mechanism) is defined as follows:

$$\frac{imm(param(f)) \qquad \Sigma \mid \Delta \Vdash E_1 \rightarrowtail (f, \Sigma') \qquad \Sigma' \mid \Delta \Vdash E_2 \rightarrowtail (v_2, \Sigma'')}{\Sigma \mid \Delta \Vdash \texttt{(}E_1\ E_2\texttt{)} \rightarrowtail f(v_2, \Sigma'')}$$

Here, we are assuming that the parameter used by the function is immutable.

Note the different treatment of environments and stores in function definition and application. Function values keep the environment of the function definition, but drop the store of the function definition. Applications use the environment of the function value, but the store resulting from the previous expression.

The meaning of sequences is given by the following rule.

$$\frac{\Sigma \mid \Delta \Vdash E_1 \rightarrowtail (v_1, \Sigma') \qquad \Sigma' \mid \Delta \Vdash E_2 \rightarrowtail (v_2, \Sigma'')}{\Sigma \mid \Delta \Vdash E_1\texttt{;}E_2 \rightarrowtail (v_2, \Sigma'')}$$

Note that the result v_1 of evaluating the first component E_1 of a sequence is ignored. The result of the sequence is the result v_2 of evaluating the second component E_2.

With while loops, we face the same problem as with `recfun` in simPL, namely circularity in rules. We give the following specification for the meaning of loops. A proper definition is beyond the scope of this course.

$$\frac{\Sigma \mid \Delta \Vdash E_1 \rightarrowtail (\mathit{false}, \Sigma')}{\Sigma \mid \Delta \Vdash \texttt{while } E_1 \texttt{ do } E_2 \texttt{ end} \rightarrowtail (\mathit{true}, \Sigma')}$$

The choice of *true* as the result of the expression is arbitrary; while loops are executed for their effect on the store, and not for obtaining their “value”.

$$\frac{\begin{array}{c}\Sigma \mid \Delta \Vdash E_1 \rightarrowtail (\mathit{true}, \Sigma') \\ \Sigma' \mid \Delta \Vdash E_2 \rightarrowtail (v_2, \Sigma'') \\ \Sigma'' \mid \Delta \Vdash \texttt{while } E_1 \texttt{ do } E_2 \texttt{ end} \rightarrowtail (v, \Sigma''')\end{array}}{\Sigma \mid \Delta \Vdash \texttt{while } E_1 \texttt{ do } E_2 \texttt{ end} \rightarrowtail (v, \Sigma''')}$$

This rule is circular, since the condition on the right hand side assumes a semantics of `while`; as with recursive functions in simPL, a thorough discussion of how to interpret such a circular rule is beyond the scope of this investigation. None the less, the rule will serve us as a specification for implementing while loops.

10.5 imPL1: Mutable Constructor

Aggregate data values such as dPL’s constructor provide the opportunity to further increase the expressive power of an imperative language. The idea is to add the possibility of changing the value of an individual field of a constructor.

To achieve this aim, we provide a way to distinguish immutable from mutable fields of data constructors. As a simple example, consider a pair of object with two fields. Let us assume that the first field is immutable, but the second field is mutable, as follows:

```
type pair 'a 'b = Pair 'a (mut 'b)
```

With this, we may access both fields of a given pair; but may selectively update the value of the second pair. An example of how the second field is updated is illustrated below:

```
let p = Pair 1 2
in match p with
     Pair x y -> y:=y+1
   end
end
```

During pattern-matching, we would assign the variables associated with the mutable field, such as `y`, with a mutable status. Furthermore, the memory location of this mutable value will be placed into the mutable store which may be directly updated. Thus, while the `Pair 1 2` object is placed in an immutable heap, the location of the second field, containing the value `2` will be placed into the mutable store instead which could be updated.

In order to capture the semantics of data constructor with mutable fields, we need to extend the semantic domains as follows:

Domain name	Definition
EV	$\mathbf{Int} + \mathbf{Bool} + \mathbf{Fun} + \mathbf{Dat} + \{\bot\}$
SV	$\mathbf{Int} + \mathbf{Bool} + \mathbf{Fun} + \mathbf{Dat}$
DV	$\mathbf{SV} + \mathbf{Loc}$
Dat	$\mathbf{c}\ \mathbf{DV} \cdots \mathbf{DV}$

Note that constructors are represented as data objects whose fields may either be immutable values (as represented by **EV**) or locations to store values (as represented by **SV**). The store locations may be mutated using the assignment statements.

Thus, when we constructed an object, we clearly distinguish immutable values from mutable locations, by the following evaluation rule:

$$\frac{\begin{array}{ll}\Sigma \mid \Delta \Vdash E_1 \rightarrowtail (v_1, \Sigma_1) \quad \cdots & \Sigma_{n-1} \mid \Delta \Vdash E_n \rightarrowtail (v_n, \Sigma_n) \\ C\ t_1 \ldots t_n \in t & \Sigma_n, [v_1 : t_1; ..; v_n : t_n] \ \Rightarrow\ \Sigma', [w_1; ..; w_n]\end{array}}{\Sigma \mid \Delta \Vdash (C\ E_1 \ldots E_n) : t \rightarrowtail (C\ w_1 \ldots w_n, \Sigma')}$$

Note that Σ captures the store where mutation are allowed, while Δ captures the environment register where immutable values are stored. We split mutable and immutable values between the store and the environment, as shown below, with the help of mutable field type declaration, as follows:

$$\frac{\mathit{immutable}(t) \qquad \Sigma, \mathit{rest} \;\Rightarrow\; \Sigma', \mathit{rest}'}{\Sigma, (v : t) :: \mathit{rest} \;\Rightarrow\; \Sigma', v :: \mathit{rest}'}$$

$$\frac{\mathit{mutable}(t) \qquad \mathit{fresh}\ l \qquad \Sigma, \mathit{rest} \;\Rightarrow\; \Sigma', \mathit{rest}'}{\Sigma, (v : t) :: \mathit{rest} \;\Rightarrow\; \Sigma'[l \mapsto v], l :: \mathit{rest}'}$$

10.6 Pass-by-Value Parameter Passing

Similar to passing of identifiers, the question arises here how constructors are passed to functions. Let us see what happens if a constructor is passed as argument to a function. According to the definition of function application, we allocate a new location in the environment for each constructor. However, the locations *in* the constructor still refer to the original locations. Thus, assignments to mutable fields remain visible after returning from the function (with pass-by-value), whereas assignments to the constructor itself are not visible after returning from the function (with pass-by-copy).

Example 10.7 *We have a similar behavior in Java, when passing objects to functions:*

```
void f(MyClass myobject) {
   myobject.myfield = 1;
   myobject = new MyClass();
   myobject.myfield = 2;
}
```

After returning from a call `f(obj)`*, the identifier* `obj` *still refers to the same object (in the sense of object identity), whereas the field* `Myfield` *has changed to 1.*

Example 10.8 *Constructors in imPL are passed similar to objects in Java. The expression*

```
let a = Pair 1 2
in
    (fun b ->
        match b with
         Pair f s -> s:=f+s
        end
```

```
     end
     a);
    match a with Pair _ s -> s end
end
```

evaluates to the integer 3.

10.7 Pass-by-Reference Parameter Passing

In Section 10.4, we decided to pass arguments "by-value" to functions. In an application `(f x)`, the body of function `f` cannot change the value in the store to which `x` refers. Instead, the function call creates a new location into which the value is copied.

Thus, pass-by-value parameter passing can be implemented by the following evaluation rule:

$$\frac{\mathit{imm}(\mathit{param}(f)) \qquad \Sigma \mid \Delta \Vdash E_1 \rightarrowtail (f, \Sigma') \qquad \Sigma' \mid \Delta \Vdash E_2 \rightarrowtail (v_2, \Sigma'')}{\Sigma \mid \Delta \Vdash \texttt{(}E_1\ E_2\texttt{)} \rightarrowtail f(v_2, \Sigma'')}$$

Note how we simply passed a copy of the value to the function, even if we should have a mutable location.

In this section, we explore an alternative to this scheme, called pass-by-reference, that can be used to support mutable values. When mutable variables appear as function arguments, we may choose to pass their location directly to the function, instead of copying their value to a new location. Thus, we pass the reference (or mutable location) to the function, instead the value. The semantic rule for function definition remains unchanged. The semantic rules for function application are as follows.

$$\frac{\mathit{mut}(\mathit{param}(f)) \qquad \Delta(x) \in \mathit{dom}(\Sigma) \qquad \Sigma \mid \Delta \Vdash E_1 \rightarrowtail (f, \Sigma')}{\Sigma \mid \Delta \Vdash \texttt{(}E_1\ x\texttt{)} \rightarrowtail f(\Delta(x), \Sigma')}$$

Note that if x is a mutable location, the access $\Delta(x)$ would simply pass the location of the mutable store to the function. Thus, any update on this location will be effected through the mutable store.
In the case where the argument is not an identifier, the old rule for application applies, where a value is passed into a new temporary location that can be mutated.

$$\frac{mut(param(f)) \qquad \Sigma \mid \Delta \Vdash E_1 \rightarrowtail (f, \Sigma') \qquad \Sigma' \mid \Delta \Vdash E_2 \rightarrowtail (v_2, \Sigma'')}{\Sigma \mid \Delta \Vdash \texttt{(}E_1\ E_2\texttt{)} \rightarrowtail f(l, \Sigma''[l \leftarrow v_2])}$$

if E_2 is not an identifier, where l is a new location in Σ''.

10.8 Pass-by-Copy Parameter Passing

In Section 10.5 we saw that a strange mix of pass-by-value and pass-by-reference occurs when constructors are passed as arguments to functions. The argument itself is passed by-value, in a sense that constructor parameter cannot be updated. On the other hand, the components of the constructor are passed by-reference; they can be changed in the body and these changes are visible in the constructor after the function has returned.

In an alternative meaning of passing constructor, we could pass the constructor's components by-value. Since this semantics involves copying the components during function application, this parameter passing technique is called pass-by-copy.

10.9 Imperative Programming and Exception Handling

The usual implementation of error handling in imperative programming is that as soon as an error occurs, the current store is returned along with the error value. For example, division by zero gets the following meaning.

$$\frac{\Sigma \mid \Delta \Vdash E_1 \rightarrowtail (v_1, \Sigma') \qquad \Sigma' \mid \Delta \Vdash E_2 \rightarrowtail (0, \Sigma'')}{\Delta \Vdash E_1 \texttt{/} E_2 \rightarrowtail (e, \Sigma'')}$$

if $v_1 \notin \mathbf{Exc}$ and where e = `[divisionByZero:true]`, and $e \in \mathbf{Exc}$

In the rules for the `try...catch...with...end` expression, the store returned by the try part is used in the `with` part.

$$\frac{\Sigma \mid \Delta \Vdash E_1 \rightarrowtail (v, \Sigma')}{\Sigma \mid \Delta \Vdash \texttt{try}\ E_1\ \texttt{catch}\ x\ \texttt{with}\ E_2\ \texttt{end} \rightarrowtail (v, \Sigma')}$$

if $v \notin \mathbf{Exc}$

$$\frac{\Sigma \mid \Delta \Vdash E_1 \rightarrowtail (v_1, \Sigma') \quad \Sigma'[l \leftarrow v_1] \mid \Delta[x \leftarrow l] \Vdash E_2 \rightarrowtail (v_2, \Sigma'')}{\Sigma \mid \Delta \Vdash \texttt{try}\ E_1\ \texttt{catch}\ x\ \texttt{with}\ E_2\ \texttt{end} \rightarrowtail (v_2, \Sigma'')}$$

if $v_1 \in \mathbf{Exc}$
l new loc.

Thus, the changes to the store made in the `try` part are visible in the `with` part although an exception has occurred. The reason for this design choice is efficiency of implementation.

Arguably semantically more sound and intuitive would be a semantics that uses the incoming store of the `try...catch...with...end` expression for evaluating the `with` part as shown in the following rule.

$$\frac{\Sigma \mid \Delta \Vdash E_1 \rightarrowtail (v_1, \Sigma') \quad \Sigma[l \leftarrow v_1] \mid \Delta[x \leftarrow l] \Vdash E_2 \rightarrowtail (v_2, \Sigma'')}{\Sigma \mid \Delta \Vdash \texttt{try } E_1 \texttt{ catch } x \texttt{ with } E_2 \texttt{ end} \rightarrowtail (v_2, \Sigma'')} \quad \begin{array}{l}\text{if } v_1 \in \mathbf{Exc} \\ l \text{ new loc.}\end{array}$$

This would mean that a copy of the store would have to be saved for every `try...catch...with...end` expression. In programming language practice, this is infeasible. However, in databases, such a semantics is sometimes desirable. The corresponding technique is called "roll-back" and allows for recovery from database inconsistencies.

10.10 A Virtual Machine for imPL

The semantics of imperative constructs is designed to allow for an efficient implementation. The store is threaded through the entire run of the program. We can prove for the semantics presented in this chapter that after a rule has constructed a new store, the old store will never be used again.[1] In an actual implementation, there is no need for constructing a new store in each rule. We only need one copy of the store, and the operations in the store can be destructive.

In Chapter 9, we saw already a technique for realistically representing the data structures used in the machine. In the presented framework a heap allowed us to explicitly manipulate the objects that are created at runtime. It turns out that we can reuse the heap to provide a realistic implementation of imPL. We can view the targets of edges in the heap as locations. Following this view, the assignment expression simply changes the target node in the environment. Chapter 9 already introduced an *update* operation in order to efficiently manipulate stacks and other data structures on the heap. To implement assignment in the heap, we shall reuse this *update* operation.

Let us first translate assignment expressions using a new instruction `ASSIGNS` (Assign Symbolic) as follows.

$$\frac{E \hookrightarrow s}{x \texttt{ := } E \hookrightarrow s.\texttt{ASSIGNS } x}$$

Thus the instruction `ASSIGNS` carries with it the identifier, whose location gets a new value in the store, and finds that new value on the operand stack. Thus

[1]Note that this property would be violated by a "roll-back" semantics for `try...catch...with...` expressions.

its execution needs to update the heap such that the identifier x refers to the new value.

$$\frac{s(pc) = \texttt{ASSIGNS}\ x}{(v.os, pc, e, rs, h) \rightrightarrows_s (os, pc+1, e, rs, h[e[x] \leftarrow v])}$$

The compilation of sequences introduces the new instruction `POP`.

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1 ; E_2 \hookrightarrow s_1.\texttt{POP}.s_2}$$

The instruction `POP` simply pops the top entry from the operand stack and ignores it. This corresponds to the denotational semantics of sequences, which ignores the value of the first component.

$$\frac{s(pc) = \texttt{POP}}{(v.os, pc, e, rs, h) \rightrightarrows_s (os, pc+1, e, rs, h)}$$

The compilation of `while` expressions reuses the `JOFR`, `POP`, `GOTOR` and `LDCB` instructions and thus does not require any new instructions.

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{\begin{array}{c}\texttt{while}\ E_1\ \texttt{do}\ E_2 \\ \hookrightarrow \\ \texttt{LABEL}\ l_1 : s_1.\texttt{JOF}\ l_2.s_2.\texttt{POP}. \\ \texttt{GOTO}\ l_1.\texttt{LABEL}\ l_2 : \texttt{LDCB}\ \mathit{true}\end{array}}$$

Note that the `GOTO` instruction jumps to the beginning of the code for E_1, which means that the condition will be re-evaluated in each iteration through the loop. The result of the body of the `while` expression is ignored using `POP`, and after the execution of the loop, the boolean value *true* is pushed as required by the denotational semantics of `while`.