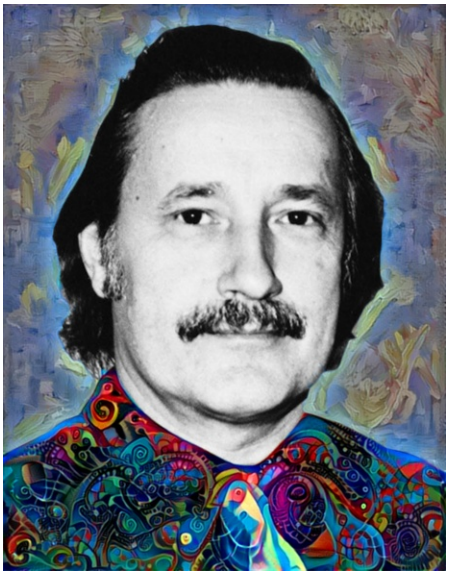
Chapter 11
An Ode to Compositionality



Christopher Strachey.[1]

Abstract In this chapter, we complement the earlier development of operational semantics with another approach to defining semantics, namely the higher-order functional approach of denotational semantics. We focus here on compositionality, which is a structuring principle for interpreters, analyses, and yet other functionality for languages. We discuss two styles of denotational semantics: the simpler "direct" style and the more versatile "continuation" style capable of dealing with, for example, nonbasic control flow constructs. Denotational semantics can be implemented easily as interpreters, for example, in Haskell, as we will demonstrate.

[1] Twenty-five years after his death, two papers by Christoper Strachey appeared [13, 14]: one on his lectures on programming language semantics and another (coauthored with Christopher P. Wadsworth) on continuations. Domain theory would probably not exist without Strachey [11]. My supervisor's generation would have known the work of Strachey (and Scott) through Joseph E. Stoy's textbook [12] and Peter D. Mosses' thesis [5]. I would fall in love with denotational style also, thanks to its applications to parallel and logic programming [6, 2]. Every software language engineer, in fact, every software engineer, should understand and leverage "compositionality" [1].

R. Lämmel, *Software Languages*,
https://doi.org/10.1007/978-3-319-90800-7_11

11.1 Compositionality

Denotational semantics [12, 3, 15] is not too popular in today's language definition culture, but the notion of compositionality is. Therefore, we will skip over the mathematical details of denotational semantics here and simply focus on the notion of compositionality. That is, we speak of a compositional semantics when it is defined as a mapping from syntax to semantics with cases for each syntactic pattern such that the meaning of a compound construct is obtained directly and only from the meanings of its constituent phrases ("subterms").

Consider the following two inference rules of a big-step operational semantics for statement sequences and while-loops in an imperative programming language, as discussed earlier (Section 8.1.6.1):

$$\frac{m_0 \vdash s_1 \Rightarrow m_1 \quad m_1 \vdash s_2 \Rightarrow m_2}{m_0 \vdash \mathsf{seq}(s_1, s_2) \Rightarrow m_2} \qquad [\mathsf{SEQ}]$$

$$\frac{m \vdash \mathsf{if}(e, \mathsf{seq}(s, \mathsf{while}(e, s)), \mathsf{skip}) \Rightarrow m'}{m \vdash \mathsf{while}(e, s) \Rightarrow m'} \qquad [\mathsf{WHILE}]$$

Rule [SEQ] agrees with the principle of compositionality, while rule [WHILE] does not. That is, the rule [SEQ] for statement sequences applies the judgment for statement execution simply to the two constituents s_1 and s_2 of the sequence $\mathsf{seq}(s_1, s_2)$. By contrast, the rule [WHILE] for while-loops carries a premise with *a newly composed syntactic pattern*. Thus, the meaning of a while-loop $\mathsf{while}(e, s)$ is *not* simply composed from the meanings of its immediate constituents e and s.

Simply speaking, compositionality is good in the same sense as primitive recursion is better understood and better controllable than general recursion. Compositionality simplifies reasoning about the semantics without relying on stepwise computation. Compositionality also helps in separating syntax and semantics in a language definition. Compositionality is not always straightforward to achieve for all language constructs, as we will demonstrate below with a fixed-point semantics of while-loops. To this end, we leverage a different approach to semantics definition: denotational semantics. Within the framework of operational semantics, it appears to be hard to define a compositional semantics of while-loops.

11.2 Direct Style

We develop the basic approach to denotational semantics on the basis of so-called "direct style," which suffices for many programming language constructs; this includes structured programming (sequence, iteration, selection) in imperative programming languages.

11.2.1 Semantic Domains

Denotational semantics assumes that each syntactic category is associated with a semantic domain. We have also used this term earlier in the context of ad hoc interpreters (Section 5.1.1) and operational semantics (Section 8.1.5.1). The difference is that the typical semantic domain of a denotational semantics is a domain of *functions*. Elements of domains directly represent meanings of program phrases; there is no reference to stepwise computation. In the case of the imperative programming language BIPL, we need these domains to be associated with the syntactic categories of statements and expressions:

$$\begin{aligned} \mathit{storeT} &= \mathit{store} \nrightarrow \mathit{store} \quad \text{// Type of store transformation} \\ \mathit{storeO} &= \mathit{store} \nrightarrow \mathit{value} \quad \text{// Type of store observation} \end{aligned}$$

In these type definitions, we assume the same definitions for *store* and *value* as in the earlier operational semantics (Section 8.1.6.1). That is, *value* denotes the domain of integer and Boolean values, whereas *store* denotes the domain of collections of variable name-value pairs. We can read these definitions as follows. The meaning of a statement is a store transformer, i.e., a function on stores, thereby describing the effect of a given statement on a given store. The meaning of an expression is a store observer, i.e., a function that takes a store and returns a value, where the store may need to be consulted owing to the variable expression form. We assume here that expressions do not modify the store. We deal with partial functions, as denoted by "$\nrightarrow$" above. The partiality is due to the possibility of nontermination, ill-typed expressions, and undefined variables.

11.2.2 Semantic Functions

Denotational semantics leverages function definitions as opposed to inference rules. The functions assign meanings (compositionally) to the different syntactic categories. In the case of BIPL's imperative programs, we need these functions:

$$\begin{aligned} \mathscr{S} &: \mathit{stmt} \to \mathit{storeT} \quad \text{// Semantics of statements} \\ \mathscr{E} &: \mathit{expr} \to \mathit{storeO} \quad \text{// Semantics of expressions} \end{aligned}$$

That is, the function for statements, $\mathscr{S}$, maps statements to store transformers. The function for expressions, $\mathscr{E}$, maps expressions to store observers.

Compositionality is implied by the following style for defining the semantic functions; in fact, we use a somewhat extreme style for clarity:

$$
\begin{array}{ll}
\mathscr{S}\,[\![\mathsf{skip}]\!] & = \underline{\mathit{skip}} \\
\mathscr{S}\,[\![\mathsf{assign}(x,e)]\!] & = \underline{\mathit{assign}}\ x\ (\mathscr{E}\,[\![e]\!]) \\
\mathscr{S}\,[\![\mathsf{seq}(s_1,s_2)]\!] & = \underline{\mathit{seq}}\ (\mathscr{S}\,[\![s_1]\!])\ (\mathscr{S}\,[\![s_2]\!]) \\
\mathscr{S}\,[\![\mathsf{if}(e,s_1,s_2)]\!] & = \underline{\mathit{if}}\ (\mathscr{E}\,[\![e]\!])\ (\mathscr{S}\,[\![s_1]\!])\ (\mathscr{S}\,[\![s_2]\!]) \\
\mathscr{S}\,[\![\mathsf{while}(e,s)]\!] & = \underline{\mathit{while}}\ (\mathscr{E}\,[\![e]\!])\ (\mathscr{S}\,[\![s]\!]) \\
 & \\
\mathscr{E}\,[\![\mathsf{intconst}(i)]\!] & = \underline{\mathit{intconst}}\ i \\
\mathscr{E}\,[\![\mathsf{var}(x)]\!] & = \underline{\mathit{var}}\ x \\
\mathscr{E}\,[\![\mathsf{unary}(o,e)]\!] & = \underline{\mathit{unary}}\ o\ (\mathscr{E}\,[\![e]\!]) \\
\mathscr{E}\,[\![\mathsf{binary}(o,e_1,e_2)]\!] & = \underline{\mathit{binary}}\ o\ (\mathscr{E}\,[\![e_1]\!])\ (\mathscr{E}\,[\![e_2]\!])
\end{array}
$$

That is:

- Applications of the semantic functions $\mathscr{S}$ and $\mathscr{E}$ to syntactical patterns or components thereof are surrounded by the so-called Oxford brackets $[\![\cdots]\!]$. One can trivially check that, in the right-hand sides of equations, the functions are really just applied to components that have been matched on the left-hand sides.
- The intermediate meanings determined for the components are composed by function combinators $\underline{\mathit{skip}}$, $\underline{\mathit{assign}}$, etc. The combinator names are the underlined names of the constructs.
- Some primitive constituents are not mapped. That is, variable names (see the equations for the phrases $\mathsf{assign}(x,e)$ and $\mathsf{var}(x)$) and operator symbols (see the equations for the phrases $\mathsf{unary}(o,e)$ and $\mathsf{binary}(o,e_1,e_2)$) are directly passed to the corresponding combinators, but no other syntax is passed on or constructed otherwise.
- We apply "curried" notation for the combinators, i.e., function arguments are lined up by juxtaposition as opposed to enclosing them in parentheses, for example, $f\ x\ y$ as opposed to $f(x,y)$.

11.2.3 Semantic Combinators

It remains to define the combinators $\underline{\mathit{skip}}$, $\underline{\mathit{assign}}$, etc. Let us capture their types first, as they are implied by the use of the combinators in the compositional scheme:

$$
\begin{array}{ll}
\underline{\mathit{skip}} & : \mathit{storeT} \\
\underline{\mathit{assign}} & : \mathit{string} \to \mathit{storeO} \to \mathit{storeT} \\
\underline{\mathit{seq}} & : \mathit{storeT} \to \mathit{storeT} \to \mathit{storeT} \\
\underline{\mathit{if}} & : \mathit{storeO} \to \mathit{storeT} \to \mathit{storeT} \to \mathit{storeT} \\
\underline{\mathit{while}} & : \mathit{storeO} \to \mathit{storeT} \to \mathit{storeT} \\
 & \\
\underline{\mathit{intconst}} & : \mathit{int} \to \mathit{storeO} \\
\underline{\mathit{var}} & : \mathit{string} \to \mathit{storeO} \\
\underline{\mathit{unary}} & : \mathit{uo} \to \mathit{storeO} \to \mathit{storeO} \\
\underline{\mathit{binary}} & : \mathit{bo} \to \mathit{storeO} \to \mathit{storeO} \to \mathit{storeO}
\end{array}
$$

Let us define the combinators in a semiformal, intuitive functional notation here. A rigorous development of formal notation for denotational semantics [12, 3, 15] is beyond the scope of this book.

// The identity function for type *store*
$\underline{skip}\ m\ =\ m$

// Pointwise store update
$\underline{assign}\ x\ f\ m\ =\ m[x \mapsto (f\ m)]$ if $f\ m$ is defined

// Function composition for type *storeT*
$\underline{seq}\ f\ g\ m\ =\ g\ (f\ m)$

// Select either branch for Boolean value
$$\underline{if}\ f\ g\ h\ m\ = \begin{cases} g\ m \text{ if } f\ m = \mathsf{true} \\ h\ m \text{ if } f\ m = \mathsf{false} \\ \mathit{undefined} \text{ otherwise} \end{cases}$$

We have left out the definition of *while* because it requires some extra effort, as discussed below. For brevity, we have omitted the definition of the combinators needed for $\mathscr{E}$ because the earlier operational semantics of expressions (Section 8.1.6.1) is essentially compositional.

Exercise 11.1 (Denotational semantics of expressions) [Basic level]
Define the combinators needed for $\mathscr{E}$.

11.2.4 Fixed-Point Semantics

The compositional semantics of while-loops involves a fixed-point construction, as we will clarify now. That is, we aim at a definition of $\underline{while}\ f\ g$ with f as the meaning of the condition and g as the meaning of the loop's body. Let us assume, just for the moment, that we already know the meaning of the while-loop; let us refer to it as t. If so, then it is easy to see that the following equivalence should hold:

$$t\ \equiv\ \underline{if}\ f\ (\underline{seq}\ g\ t)\ \underline{skip}$$

That is, by the choice of *if*, we test the loop's condition; if the condition evaluates to false, we use the state transformer *skip*; otherwise, we sequentially compose the meaning g of the loop's body and the assumed meaning t of the loop itself. Thus, we explicitly construct the meaning of a while-loop, the body of which is executed zero or one times, and we resort to t for repetitions past the first one. It is crucial to understand that we do not use any syntax in this equivalence. Instead, we simply compose meanings.

Alas, we do not yet know t. Let us capture the right-hand side expression of the equivalence as h and parameterize it in t:

$$h\,t \;=\; \underline{\mathit{if}}\;f\;(\underline{\mathit{seq}}\;g\;t)\;\underline{\mathit{skip}}$$

Now consider the following progression of applications of h:

$$\begin{array}{c} h\;\mathit{undefined} \\ h\;(h\;\mathit{undefined}) \\ h\;(h\;(h\;\mathit{undefined})) \\ \vdots \end{array}$$

Here, *undefined* denotes the completely undefined store transformation, which, given any store m returns a store which maps all variable names to the undefined value. Note that the elements in this progression correspond to approximations to the meaning t of the while-loop that agree with t in terms of the resulting store for the cases of 0, 1, ... required repetitions of the body. Thus, if we can express an unbounded number of applications of h to *undefined*, then we have indeed defined t. This is essentially achieved by taking the fixed point of h. Thus:

$$t \;\equiv\; \mathit{fix}\;h$$

One way to think of *fix* is as being defined "computationally" according to the fixed-point property, as discussed earlier in the context of the lambda calculus (Section 10.1.5). Thus:

$$\mathit{fix}\;k \;=\; k\;(\mathit{fix}\;k)$$

That is, we assume that the fixed point of k is computed by applying k to the computation of the fixed point. Another way to think of *fix* is as being defined as the least upper bound of the elements in the infinite progression described above. The least upper bound is defined here essentially in terms of being more "defined", i.e., returning a less undefined or possibly fully defined store transformer.

To conclude, we define the meaning of a while-loop in terms of the *while* combinator as a fixed point as follows:

$$\begin{array}{l} \underline{\mathit{while}}\;f\;g = \mathit{fix}\;h \\ \qquad\qquad \text{where} \\ \qquad\qquad h\;t = \underline{\mathit{if}}\;f\;(\underline{\mathit{seq}}\;g\;t)\;\underline{\mathit{skip}} \end{array}$$

Our discussion of fixed points has been very superficial here, and we point to the literature on denotational semantics [12, 3, 15] and on domain theory specifically [11]. In particular, semantic domains and combinators over them must satisfy a number of fundamental properties for such a fixed-point semantics to be well defined in that the fixed point is uniquely defined. To this end, the domains are more than just sets; they are equipped with a partial order to deal with undefinedness and

approximation. Also, the combinators need to be monotone and continuous in a specific sense to facilitate fixed-point computation by taking least upper bounds with respect to the said partial orders.

Exercise 11.2 (Existence and uniqueness of fixed points) [Basic level]
This exercise hints at the challenge of making sure that fixed points exist and are uniquely defined. Define functions a, b, and c on natural numbers such that a has no fixed point, b has exactly one fixed point, and c has an infinite number of fixed points. Use the fixed-point property to check whether a given natural number is indeed a fixed point of a given function. That is, x_0 is a fixed point of f if $f\ x_0 = x_0$.

Regardless of the informality of the development, it is "computationally effective," as a discussion of denotational interpreters shows below.

Exercise 11.3 (Expression-oriented imperative language) [Intermediate level]
Define the denotational semantics of an imperative language such that the syntactic category for expressions incorporates all statement forms. There were similar assignments for big- and small-step operational semantics in Chapter 8 (Exercises 8.4 and 8.9).

11.2.5 Direct-Style Interpreters

Arguably, semantic domains, functions, and combinators are easily encoded as interpreters in functional programming. Such an implementation benefits from the fact that denotational semantics is clearly a functional approach to defining semantics. That is, domains are types of functions; semantic functions are functions anyway. Semantic combinators are (higher-order) function combinators. The actual details of a systematic and well-defined encoding are nontrivial [8, 9, 10], as there may be some mismatch between the mathematical view of a metanotation for semantics and the actual semantics of the functional metalanguage, but we skip over such details here. We encode denotational semantics in Haskell.

Here is how we expect to use the interpreter for executing the sample program for Euclidean division; we apply values for the variables x and y and execution computes the variables q and r as the quotient and remainder of dividing x by y:

Interactive Haskell session:

```
▶ execute euclideanDiv (fromList [("x", Left 14), ("y", Left 4)])
fromList [("q", Left 3), ("r", Left 2), ("x", Left 14), ("y", Left 4)]
```

Let us start the implementation of an interpreter with a Haskell encoding of the semantic domains as shown below.

Illustration 11.1 (Semantic domains for imperative programs)

Haskell module Language.BIPL.DS.Domains

```
-- Results of expression evaluation
type Value = Either Int Bool
-- Stores as maps from variable ids to values
type Store = Map String Value
-- Store transformers (semantics of statements)
type StoreT = Store → Store
-- Store observers (semantics of expressions)
type StoreO = Store → Value
```

The definitions are straightforward. The definition of Store exhibits an element of choice. We could also model stores more directly as functions of type String → Value, but we opt for Haskell's library type Map to model stores as maps (say, dictionaries) from variable names to values because the underlying representation is more convenient to use for testing and debugging, as dictionaries are "observable" as a whole whereas genuine functions can only be "queried" at specific points.

Let us continue the implementation of an interpreter with a Haskell encoding of the compositional mapping over statements as shown below.

Illustration 11.2 (Compositional mapping)

Haskell module Language.BIPL.DS.Interpreter

```
execute :: Stmt → StoreT
execute Skip = skip'
execute (Assign x e) = assign' x (evaluate e)
execute (Seq s1 s2) = seq' (execute s1) (execute s2)
execute (If e s1 s2) = if' (evaluate e) (execute s1) (execute s2)
execute (While e s) = while' (evaluate e) (execute s)

evaluate :: Expr → StoreO
evaluate (IntConst i) = intconst' i
evaluate (Var x) = var' x
evaluate (Unary o e) = unary' o (evaluate e)
evaluate (Binary o e1 e2) = binary' o (evaluate e1) (evaluate e2)
```

That is, the semantic functions $\mathscr{S}$ and $\mathscr{E}$ are called execute and evaluate for clarity, and the underlined combinators of the semiformal development are modeled as primed functions in Haskell; see, for example, skip' instead of *skip*. (By priming, we also avoid clashes. For instance, if is readily taken in Haskell.) There is one equation per language construct. On the left-hand side of an equation, the construct is matched to provide access to the constituents of the construct. On the right-hand side of an equation, the meanings of the constituents are determined by recursive occurrences of the interpreter functions and they are combined by the corresponding semantic combinator.

We complete the implementation of an interpreter with a Haskell implementation of the semantic combinators definitions as shown below.

Illustration 11.3 (Combinators of semantic meanings)

Haskell module Language.BIPL.DS.Combinators

```
skip' :: StoreT
skip' = id
assign' :: String → StoreO → StoreT
assign' x f m = insert x (f m) m
seq' :: StoreT → StoreT → StoreT
seq' = flip (.)
if' :: StoreO → StoreT → StoreT → StoreT
if' f g h m = let Right v = f m in if v then g m else h m
while' :: StoreO → StoreT → StoreT
while' f g = fix h where h t = if' f (seq' g t) skip'
intconst' :: Int → StoreO
intconst' i _ = Left i
var' :: String → StoreO
var' x m = m!x
unary' :: UOp → StoreO → StoreO
unary' Negate f m = let Left i = f m in Left (negate i)
unary' Not f m = let Right b = f m in Right (not b)
binary' :: BOp → StoreO → StoreO → StoreO
...
```

In the code shown above, we make reasonable use of functional programming idioms in Haskell. In the definition of while', we use a polymorphic fixed-point combinator that is readily defined in the Haskell library like this:

```
fix :: (a → a) → a
fix f = f (fix f)
```

Exercise 11.4 (Interpretation without throwing) [Basic level]
*The interpreter may "throw" for different reasons, for example, in the case of applying Boolean negation (*Not*) to an integer constant. Identify all such reasons and revise the interpreter so that statement execution and expression evaluation do not simply throw, but* Nothing *of Haskell's* Maybe *type is returned instead.*

The present section can be summarized by means of a recipe.

Recipe 11.1 (Compositional interpretation).

Abstract syntax *Implement abstract syntax, as discussed previously (Recipe 4.1).*

Semantic domains *Implement the semantic domains; these are often function types. For instance, the semantic domain for expressions in a language with variables maps variable identifiers to values.*

Semantic combinators *There is one combinator per language construct with as many arguments as there are constituent phrases ("subterms"), with the argument types equaling the semantic domains for the constituents and the result type equaling the semantic domain for the construct's category.*

Compositional mapping *Implement functions from the syntactic to the semantic domains. There is one function per syntactic category. There is one equation per construct. In each equation, apply the semantic functions to constituents and combine the results with the combinator for the construct.*

11.3 Continuation Style

We now turn from the direct to the more advanced continuation style of denotational semantics. The main idea, when one is applying the style to imperative programs, is to parameterize meanings in the "rest" of the program so that each meaning can freely choose to deviate from the default continuation, whenever it may be necessary for control-flow constructs such as throws of exceptions or gotos. In functional programming, there also exists a related style, the *continuation-passing style* (CPS), which helps with adding error handling to programs and with structuring functional programs, for example, in the context of implementing web applications [4].

11.3.1 Continuations

In direct style, as assumed so far, control flow is quite limited. To see this, let us recall that the semantic combinator for sequential composition was defined as follows:

$$\begin{aligned} &\underline{seq} \;:\; storeT \to storeT \to storeT \\ &\underline{seq}\; f\; g\; m \;=\; g\;(f\; m) \end{aligned}$$

Thus, f applies the given store m and passes on the resulting store to g. Now suppose that f corresponds to a phrase with a goto or a throw of an exception in which case g should be ignored. Within the bounds of the semantic domains at hand, there is no reasonable definition for $\underline{seq}$ such that g could be ignored if necessary.

In continuation style, we use more advanced semantic domains; we do not use "store transformers" but we rather use "store transformer transformers" defined as follow:

$$storeTT = storeT \to storeT$$

The idea is that any meaning is parameterized by a store transformer corresponding to what should "normally" be executed next. We refer to such parameters as continuations. The type and definition of the semantic combinator *seq* are revised as follows:

$$\begin{aligned} &\underline{seq} \;:\; storeTT \to storeTT \to storeTT \\ &\underline{seq}\; f\; g\; c \;=\; f\; (g\; c) \end{aligned}$$

That is, the sequential composition is parameterized by a continuation c for whatever follows the statement sequence. The order of functionally composing the arguments of *seq* is reversed compared with direct style. This makes sense because we are not composing store transformers; instead, we pass store transformers as arguments.

11.3.2 Continuation-Style Interpreters

We will work out any more details of continuation style in a semiformal notation here. Instead, we will explain details directly by means of interpreters. For now, we just convert the earlier interpreter into continuation style – without yet leveraging the added expressiveness. In the next section, we add gotos to leverage continuation style proper.

We implement the new semantic domain as follows.

Illustration 11.4 (Store transformer transformers)

Haskell module Language.BIPL.CS.Domains

```
type StoreTT = StoreT → StoreT
```

The compositional mapping does not change significantly, as shown below:

Illustration 11.5 (Compositional mapping with continuations)

Haskell module Language.BIPL.CS.Interpreter

```
execute :: Stmt → StoreT
execute s = execute' s id
  where
    execute' :: Stmt → StoreTT
    execute' Skip = skip'
    execute' (Assign x e) = assign' x (evaluate e)
    execute' (Seq s1 s2) = seq' (execute' s1) (execute' s2)
    execute' (If e s1 s2) = if' (evaluate e) (execute' s1) (execute' s2)
    execute' (While e s) = while' (evaluate e) (execute' s)
```

In the code shown above, the top-level function execute maps statements to store transformers and uses the locally defined function execute' to map statements to store transformer transformers starting from the "empty" continuation id.

The semantic combinators have to be changed as follows.

Illustration 11.6 (Combinators of semantic meanings)

Haskell module Language.BIPL.CS.Combinators

```
skip' :: StoreTT
skip' = id
assign' :: String → StoreO → StoreTT
assign' x f c sto = c (insert x (f sto) sto)
seq' :: StoreTT → StoreTT → StoreTT
seq' = (.)
if' :: StoreO → StoreTT → StoreTT → StoreTT
if' f g h c = DS.if'  f (g c) (h c)
while' :: StoreO → StoreTT → StoreTT
while' f g = fix h where h t = if' f (seq' g t) skip'
```

The combinators differ from direct style as follows:

- skip': The identity function is applied here to store transformers as opposed to stores. The definition models that the current continuation is simply applied.
- assign': The store is transformed, just as in the case of direct style, and then passed to the continuation received.
- seq': The definition models that (the meaning of) the second statement, once applied to the given continuation, acts as a continuation of (the meaning of) the first statement.
- if' : The meaning of an if-statement is the same as in direct style, except that we need to pass the continuation to both branches. We reuse the combinator $DS.if'$ of direct style.
- while': The meaning of a while-loop is defined similarly to direct style, except that there is an extra argument for the continuation (suppressed by currying).

11.3.3 Semantics of Gotos

As a simple exercise in leveraging continuation style, we consider an imperative language without while-loops, but with general gotos instead. To this end, we use the following syntax.

Illustration 11.7 (Syntax of imperative statements with gotos)

Haskell module Language.BIPL.Goto.Syntax

```
data Stmt
  = Skip
  | Assign String Expr
  | Seq Stmt Stmt
  | If Expr Stmt Stmt
  | Label String
  | Goto String
```

A sample program follows.

Illustration 11.8 (Euclidean division with goto instead of while)

Haskell module Language.BIPL.Goto.Sample

```
euclideanDiv :: Stmt
euclideanDiv =

  -- Sample operands for Euclidean division
   Seq (Assign "x" (IntConst 14))
  (Seq (Assign "y" (IntConst 4))

  -- Compute quotient q=3 and remainder r=2
  (Seq (Assign "q" (IntConst 0))
  (Seq (Assign "r" (Var "x"))
  (Seq (Label "a")
       (If (Binary Geq (Var "r") (Var "y"))
          (Seq (Assign "r" (Binary Sub (Var "r") (Var "y")))
          (Seq (Assign "q" (Binary Add (Var "q") (IntConst 1)))
               (Goto "a")))
          Skip)))))
```

The denotational semantics of imperative programs with gotos relies on an extra argument for the "goto table" in which to look up the meaning of a label upon encountering a goto. Thus, the semantic domain for meanings of statements evolves as follows.

Illustration 11.9 (Goto tables)

Haskell module Language.BIPL.Goto.Domains

```
type Gotos = [(String, StoreT)] -- Goto tables
type StoreTT' = (StoreT, Gotos) → (StoreT, Gotos) -- Transformation with gotos
```

The compositional mapping is adapted to deal with goto tables, as shown below.

Illustration 11.10 (Compositional mapping with gotos)

Haskell module Language.BIPL.Goto.Interpreter

```
execute :: Stmt → StoreT
execute s = let (c, g) = execute' s (id, g) in c
  where
    execute' :: Stmt → StoreTT'
    execute' Skip = skip'
    execute' (Assign x e) = assign' x (evaluate e)
    execute' (Seq s1 s2) = seq' (execute' s1) (execute' s2)
    execute' (If e s1 s2) = if' (evaluate e) (execute' s1) (execute' s2)
    execute' (Label l) = label' l
    execute' (Goto l) = goto' l
```

The top-level function execute maps statements to store transformers and uses the locally defined function execute' which takes goto tables into account. In fact, as is evident from the definition of StoreTT', the goto table is both received as an argument and returned as part of the result. This may be surprising at first, but in fact the mapping needs to add to the goto table (see the combinator label' in the following illustration) and to read from the goto table (see the combinator goto' in the following illustration).

Illustration 11.11 (Combinators of semantic meanings with gotos)

Haskell module Language.BIPL.Goto.Combinators

```
skip' :: StoreTT'
skip' (c, t) = (c, [])
assign' :: String → StoreO → StoreTT'
assign' x f (c, t) = (λ m → c (insert x (f m) m), [])
seq' :: StoreTT' → StoreTT' → StoreTT'
seq' f g (c, t) = let (c', t') = g (c, t) in let (c'', t'') = f (c', t) in (c'', t'++t'')
if' :: StoreO → StoreTT' → StoreTT' → StoreTT'
if' f g h (c, t) = let ((c1, t1), (c2, t2)) = (g (c, t), h (c, t)) in (DS.if'  f c1 c2, t1++t2)
label' :: String → StoreTT'
label' l (c, t) = (c, [(l, c)])
goto' :: String → StoreTT'
goto' l (c, t) = (fromJust (lookup l t), [])
```

The combinators are explained one by one as follows:

- skip': The given continuation is simply preserved. The received goto table is not consulted. The returned goto table is empty ([]).
- assign': The store is transformed and then passed to the received continuation. The received goto table is not consulted. The returned goto table is empty ([]).

- seq': The two meanings are essentially composed by function composition, except that the given goto table is passed to both operands and the individually returned goto tables are combined (i.e., appended with (++)) to serve as the resulting goto table for the statement sequence.
- if' : The given goto table is passed to both the then- and the else-branch. The goto tables for the then- and else-branches are combined as the resulting goto table. Other than that, we reuse the semantic combinator of direct style.
- label': The current continuation is captured and associated with the label at hand to form a goto table.
- goto': The current continuation is ignored; in fact, it is replaced by the continuation associated with the given label according to the given goto table.

Exercise 11.5 (Exceptions) [Intermediate level]
Add the following two statement forms:

- *Throwing an exception* throw(x)*: A string x. An exception terminates the regular (sequential) control-flow and propagates through the compound statement until it is handled by a* trycatch *statement (see below) or to the top, where it terminates the program irregularly.*
- *Catching an exception* trycatch(s,x,s')*: While the statement s is being executed, any exception x is caught and s' would be executed. If no exception occurs within s, then the statement behaves just like s. If an exception other than x occurs within s, then the exception is propagated as described above.*

These statement forms should be added by extending the Haskell-based interpreter while leveraging continuation style. This form of exception is developed in some detail in [7].

Exercise 11.6 (Fixed points with gotos) [Intermediate level]
A recursive let is used in Illustration 11.11, to tie the recursive knot needed for passing the goto table returned by the mapping back into the same mapping. Thus, the semantics is arguably not (obviously) compositional. Revise the semantics so that the recursive let is replaced by some explicit fixed-point construction.

Summary and Outline

We have described the denotational (functional) approach to defining semantics. In its full beauty, denotational semantics is a mathematically elegant approach. We focused here, though, on the key principle of the approach: compositionality, i.e., defining meanings of compound constructs solely in terms of recursively determined meanings of constituent phrases, thereby achieving a full separation of syntax and

semantics. We have also touched upon continuation style, which is a sophisticated pattern for structuring semantics definitions (and declarative programs).

In the remaining (technical) chapter, we will discuss a few nontrivial metaprogramming techniques – some of which are also informed by programming language theory. In one case, we will also discuss how denotational semantics can be used to specify program analyses by replacing the semantic algebra for composing meanings by another interpretation geared towards computing program properties that may be useful, for example, for program optimization.

References

1. Blikle, A.: Denotational engineering. Sci. Comput. Program. **12**(3), 207–253 (1989)
2. Brogi, A., Lamma, E., Mello, P.: Compositional model-theoretic semantics for logic programs. New Generation Comput. **11**(1), 1–21 (1992)
3. Gunter, C.: Semantics of Programming Languages: Structures and Techniques. MIT Press (1992)
4. Krishnamurthi, S., Hopkins, P.W., McCarthy, J.A., Graunke, P.T., Pettyjohn, G., Felleisen, M.: Implementation and use of the PLT scheme web server. Higher Order Symbol. Comput. **20**(4), 431–460 (2007)
5. Mosses, P.D.: Mathematical semantics and compiler generation. Ph.D. thesis, University of Oxford, UK (1975)
6. Nielson, F., Nielson, H.R.: Code generation from two-level denotational meta-languages. In: Proc. Programs as Data Objects 1985, *LNCS*, vol. 217, pp. 192–205. Springer (1986)
7. Nielson, H.R., Nielson, F.: Semantics with Applications: An Appetizer. Undergraduate Topics in Computer Science. Springer (2007)
8. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proc. ACM Annual Conference – Volume 2, ACM '72, pp. 717–740. ACM (1972)
9. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. Higher Order Symbol. Comput. **11**(4), 363–397 (1998)
10. Reynolds, J.C.: Definitional interpreters revisited. Higher Order Symbol. Comput. **11**(4), 355–361 (1998)
11. Stoltenberg-Hansen, V., Lindström, I., Griffor, E.R.: Mathematical Theory of Domains. Cambridge University Press (1994)
12. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics. MIT Press (1977)
13. Strachey, C.: Fundamental concepts in programming languages. Higher Order Symbol. Comput. **13**(1/2), 11–49 (2000)
14. Strachey, C., Wadsworth, C.P.: Continuations: A mathematical semantics for handling full jumps. Higher Order Symbol. Comput. **13**(1/2), 135–152 (2000)
15. Tennent, R.D.: Denotational semantics. In: Handbook of logic in computer science, vol. 3, pp. 169—-322. Oxford University Press (1994)