10 $\mathcal{SOOL}$, a Simple Object-Oriented Language

$\mathcal{SOOL}$

In the last two chapters we introduced the formal notation necessary in order to specify programming languages. Now we are ready to present the formal specification of a simple object-oriented programming language, $\mathcal{SOOL}$, in terms of the formal notation of $\Lambda^{P}_{<:}$. The language $\mathcal{SOOL}$ is intended to share the core features of most statically typed class-based object-oriented programming languages. It is essentially the same language used in the examples in the earlier chapters of this book. It does not include the extensions discussed in Chapter 4. These will be added in Part IV, starting with Chapter 15.

Analyzing this language and its extensions will be the focus of most of the remaining chapters of this text. We will first carefully give formal specifications of the syntax, type-checking rules, and semantics of the $\mathcal{SOOL}$. With that specification complete we will be able to prove that the type system is sound.

In order to make it simpler to write down semantics and type-checking rules for our language, the syntax for the language will be somewhat more cluttered than most common object-oriented languages. However, we will also introduce abbreviations which provide a more friendly and familiar programming style. One can imagine the early steps of a compiler expanding the code from the language with abbreviations to the full language.

10.1 Informal description and example

An early example of the programming notation introduced in Chapter 2 was the program repeated in Figure 10.1. That will be a valid class definition in our language with abbreviations (as will the version that drops all of the "`self.`" subexpressions), but the formal language specification will define

```
class CellClass {
   x:Integer = 0;

   function get(): Integer is
   { return self.x }

   function set(nuVal:Integer): Void is
   { self.x := nuVal }

   function bump(): Void is
   { self ⇐ set(self ⇐ get()+1) }
}
```

Figure 10.1 Cell class.

a more verbose version of the language which allows function and classes to be defined as "anonymous" expressions. That is, similar to the way the lambda calculus allows one to define unnamed function expressions using the lambda notation, we will be able to write down unnamed expressions that represent functions and classes.

In Figure 10.2, `CellClass` is shown as written in the fully expanded language. We explain the changes in notation between the two versions below.

In the fully expanded language we will adopt some of the notation of the lambda calculus with references described in Section 5.1.3. Types of the form `Ref T` are reference types, denoting the types of variables holding values of type `T`. An expression with type `Integer` (*e.g.*, `3+7`) denotes an integer value, while an expression `x` of type `Ref Integer` denotes an updatable variable holding values of type `Integer`.

An expression of the form `ref e` denotes a reference or location holding the value of `e`. These expressions are used to initialize variables. If `x` is declared to be a variable of type `T` with initial value `e` in the abbreviated language, then `x` will be defined as a constant with type `Ref T` and value `ref e` in the expanded language.

We will use the `val` qualifier to distinguish between the value of a variable as a location (*e.g.*, as used on the left side of assignment statements) and the value stored in the variable (*e.g.*, as used on the right side of expressions). Thus if `x` is an integer variable (identifier with type `Ref Integer`), we will

```
CellClass = class({|
         x:Ref Integer = ref 0
    |}, {|
         get:Void → Integer =
                function(v:Void): Integer is {
             return val self.x };

         set:Integer → Command =
                function(nuVal:Integer): Command is {
             self.x := nuVal;
             return nop };

         bump:Void → Command =
                function(v:Void): Command is {
             self ⇐ set(self ⇐ get()+1);
             return nop }
    |})
```

Figure 10.2 CellClass in the fully expanded language.

write

```
x := (val x) + 1;
```

rather than the more usual `x := x + 1`.

We will generally not bother to include the `Ref` types, `ref` expressions, and `val` qualifiers in example code, but we will use them in formal specifications in order to distinguish variables of type `T` from values of type `T` and the two kinds of values of variables. Having these distinctions show up in the language will make it significantly easier to write type-checking rules and semantics for $\mathcal{SOOL}$.

In order to avoid having to distinguish between functions and procedures in the semantics, the expanded language will represent procedures as functions which return a value `nop` of type `Command`, the type of commands. The expression `nop` stands for a command which does nothing. In the abbreviated language, we will allow `return nop` to be omitted.

Because of this convention, the return type of procedures will also be given as `Command`, the type of all other statements, rather than the `Void` we have been using in the abbreviated language. While this differs from the conventions used in C-style syntax, it is helpful to distinguish between a function which takes a default value `()` of type `Void`, and a procedure whose return type indicates that it is executed to change the state of the computer.

Invoking a parameterless method involves applying it to `()`. In order to make method definitions and invocations more uniform, parameterless methods will be considered to be abbreviations for methods which take a new (and unused) parameter of type `Void`. Because `()` has type `Void`, this will work smoothly with our type-checking rules.

Other differences between the original and the non-abbreviated version of the notation for class definitions are that names of classes and methods are separated from the expressions that define them, and method names are provided with their types. One reason for specifying this somewhat more verbose version of the language is that it will allow us to have anonymous class and function expressions.

The method `get` in Figure 10.2 illustrates these differences. The method name with its signature (typing) occurs before the equals sign. After the equals sign is the definition of the anonymous function represented by `get`.

Another change is that the collections of instance variables and methods are grouped together using the symbols “$\{\!|$” and “$|\!\}$”. These symbols are chosen to be distinct from the curly brackets used to enclose sequences of statements. As we will see in the formal specification, records are not considered expressions or types of $\mathcal{SOOL}$. However, notation representing classes as being composed of a record of instance variable declarations and another record of method definitions will make it easier to specify type-checking rules.

These changes are included to make the formal syntax more uniform and allow us to write fewer formal rules for grammar, type checking rules, and semantics. It should be clear that it is easy to automatically translate a program in the abbreviated language into one of the expanded language. Again, all formal specification will be of the expanded language.

10.2 Syntax and type-checking rules

In this section we will provide formal specifications of the syntax and type-checking rules for our simple language, $\mathcal{SOOL}$. Later on we will add more constructs to the language to increase its expressiveness. The expressions of $\mathcal{SOOL}$ are limited to those which type check correctly. As with the typed lambda calculus, we will introduce these expressions in two steps. In this section we provide the syntactic specification of pre-expressions of the language, some examples, and then the type checking rules. The legal expressions of the language will be comprised of those pre-expressions which can be assigned a type via the type-checking rules.

10.2.1 Syntax of types and expressions

The syntax of $\mathcal{SOOL}$ types and expressions will be given by a context-free grammar. Because expressions contain type expressions, we begin by specifying the syntax of types.

Syntax of types

The definition of type expressions will be built from a collection of built-in types (or type constants), $\mathcal{TC}$. We assume that $\mathcal{TC}$ contains at least the type constants `Integer` for the set of integers, `Real` for the real numbers, `Boolean` for the booleans, `Character` for the characters, `String` for the strings, `Void` for a type with only a single value, written (), and `Command` for the type of statements or commands. When convenient, we will assume $\mathcal{TC}$ has other built-in types.

Definition 10.2.1 *The set, $\mathcal{TYPE}_{\mathcal{SOOL}}(\mathcal{TC}, \mathcal{L}, \mathcal{TI})$, of type expressions of $\mathcal{SOOL}$ over a set $\mathcal{TC}$ of type constants, a set $\mathcal{L}$ of record labels, and a set $\mathcal{TI}$ of type identifiers is given by the following context-free grammar (where we assume $\mathtt{t} \in \mathcal{TI}$, $\mathtt{C} \in \mathcal{TC}$, $\mathtt{l}_i \in \mathcal{L}$, and $\mathtt{T}, \mathtt{T}_i \in \mathcal{TYPE}_{\mathcal{SOOL}}(\mathcal{TC}, \mathcal{L}, \mathcal{TI})$):*

$$
\begin{array}{rcl}
\mathtt{T} \in \mathtt{Type} & ::= & \mathtt{C} \mid \mathtt{t} \mid \mathtt{T}_1 \times \ldots \times \mathtt{T}_n \rightarrow \mathtt{T}_{n+1} \mid \mathtt{Ref\ T} \mid \\
 & & \mathtt{ObjectType\ RT} \mid \mathtt{VisObjectType(RT_i, RT_m)} \mid \\
 & & \mathtt{ClassType(RT_i, RT_m)} \\
\mathtt{RT} \in \mathtt{RType} & ::= & \{\!|\ \mathtt{l}_1 \colon \mathtt{T}_1; \ldots; \mathtt{l}_n \colon \mathtt{T}_n\ |\!\}
\end{array}
$$

Items generated by `RType` are not themselves considered legal types of $\mathcal{SOOL}$. Instead these record types are used only to build up legal object and class types. Record types consist of a list of labels and their associated types.

Types of the form $\mathtt{T}_1 \times \ldots \times \mathtt{T}_n \rightarrow \mathtt{T}_{n+1}$ are function types, denoting functions which take *n* arguments of types $\mathtt{T}_1$ through $\mathtt{T}_n$ and return a value of type $\mathtt{T}_{n+1}$. Object types are provided with a record type that specifies the names and types of methods.

As discussed earlier, types of the form `Ref T` are reference types, denoting the types of variables holding values of type `T`. An expression with type `In-`

`teger` (*e.g.*, `3+7`) denotes an integer value, while an expression `x` of type `Ref Integer` denotes an updatable variable holding values of type `Integer`.

Types of the form `ObjectType M` are the types of objects with method names and types given by the record type `M`. (We often use identifiers `M` and `IV` to represent record types, where `M` is used as the type of the record of methods in a class or object, while `IV` is used as the type of the record of instance variables.) The names and types of instance variables are not visible in object types.

The keyword `self` provides access to both the instance variables and methods of the object. As a result, we need a type for `self` that includes the names and types of instance variables, reflecting the fact that they are visible or accessible via `self`. Thus the type of `self` will be a "visible" object type rather than `ObjectType M`. Types of the form `VisObjectType(`$\texttt{IV}^{ref}$`, M)` are introduced explicitly to serve as the types of `self`. In definitions of this form, $\texttt{IV}^{ref}$ will represent the type of the record of instance variables (all of which will be references), whereas `M` will represent the type of the record of methods (all of which will be functions).

We have so far only encountered class types briefly in Section 5.3, where we discussed the lack of subtyping between class types. Class types are normally not explicitly written in programs of $\mathcal{SOOL}$. However, because we will support class expressions, it is possible to have variables that hold class values or to pass class expressions as parameters to functions.

While objects hide their record of instance variables, the instance variables will be visible in classes, as they may be used in subclasses. (Thus our instance variables behave like protected members of Java or C++, and are visible in subclasses.) Thus the types of instance variables must be specified in class types.

In a class type of the form `ClassType(IV, M)`, `IV` is the record type specifying the names and types of the initial values of instance variables, while `M` specifies the names and types of the methods. Notice the difference between the types in visible object types and class types. In the former, $\texttt{IV}^{ref}$ represents the type of the record of instance *variables*. That is, it is a record of reference types. In class types, `IV` represents the type of a record of *initial values* of instance variables. Typically these will not be explicit references.

Syntax of expressions

The syntax of $\mathcal{SOOL}$ programs is given by a context-free grammar. Rather than specifying all of the built-in constants and pre-defined functions of the language, we will simply assume that there is a collection, $\mathcal{EC}$, of such expressions. This will allow us to introduce such built-in expressions as needed rather than give a complete specification now. We also assume a collection, $\mathcal{L}$, of labels (used for method and instance variable names), and a set, $\mathcal{EI}$, of expression identifiers.

Definition 10.2.2 *The pre-expressions, $\mathcal{PEXP}_{\mathcal{SOOL}}(\mathcal{EC}, \mathcal{L}, \mathcal{EI})$, of $\mathcal{SOOL}$ over a set $\mathcal{EC}$ of expression constants, a set $\mathcal{L}$ of labels, and a set $\mathcal{EI}$ of identifier names, is given by the following context-free grammar (where we assume* $\texttt{id} \in \mathcal{EI}$, $\texttt{t} \in \mathcal{TI}$, $\texttt{c} \in \mathcal{EC}$, $\texttt{l}_i, \texttt{m}_i \in \mathcal{L}$, *and* $\texttt{T}, \texttt{T}_i \in \mathcal{TYPE}_{\mathcal{SOOL}}(\mathcal{TC}, \mathcal{L}, \mathcal{TI})$ *):*

$$\begin{array}{rcl}
\texttt{Prog} & ::= & \texttt{Program id; Blk.} \\
\texttt{Blk} \in \texttt{Block} & ::= & \texttt{TDs CDs \{ S return E \}} \\
\texttt{TDs} \in \texttt{TDefs} & ::= & \epsilon \mid \texttt{type TDL} \\
\texttt{TDL} \in \texttt{TDefLst} & ::= & \texttt{t = T} \mid \texttt{t = T; TDL} \\
\texttt{CDs} \in \texttt{CDefs} & ::= & \epsilon \mid \texttt{const CDL} \\
\texttt{CDL} \in \texttt{CDefLst} & ::= & \texttt{id: T = E} \mid \texttt{id: T = E; CDL} \\
\texttt{E} \in \texttt{Exp} & ::= & \texttt{id} \mid \texttt{c} \mid \texttt{nil} \mid \texttt{()} \mid \texttt{val E} \mid \texttt{ref E} \mid \texttt{E(E}_1, \ldots, \texttt{E}_n\texttt{)} \mid \\
 & & \texttt{function(id}_1\texttt{: T}_1, \ldots, \texttt{id}_n\texttt{: T}_n\texttt{): T}_{n+1} \texttt{ is Blk} \mid \\
 & & \texttt{class(R}_\texttt{i}\texttt{, R}_\texttt{m}\texttt{)} \mid \texttt{new E} \mid \texttt{E} \Leftarrow \texttt{m} \mid \texttt{E.l} \mid \\
 & & \texttt{class inherits E modifies l}_{j_1}, \ldots, \texttt{l}_{j_m} \texttt{ (R}_i\texttt{, R}_m\texttt{)} \\
\texttt{R} \in \texttt{Rec} & ::= & \{\!|\ \texttt{l}_1\texttt{: T}_1 \texttt{ = E}_1; \ldots; \texttt{l}_n\texttt{: T}_n \texttt{ = E}_n \ |\!\} \\
\texttt{S} \in \texttt{Stmt} & ::= & \texttt{nop} \mid \texttt{id := E} \mid \texttt{if E then \{ S}_1 \texttt{ \} else \{ S}_2 \texttt{ \}} \mid \\
 & & \texttt{while E do \{ S \}} \mid \texttt{S}_1\texttt{; S}_2
\end{array}$$

A program in $\mathcal{SOOL}$ consists of a `Program` statement with the name of the program, followed by a block. The block consists of a list of type and constant definitions (which include definitions of functions and classes), and finally by a statement and return expression surrounded by curly brackets. For programs, the block will be the main program and will return the default value `nop`. In function bodies, the return value can be any expression of the appropriate type.

The symbol ϵ indicates that a construct may be replaced by an empty string. In other words, the ϵ in type and constant definitions indicates that these constructs are optional.

Expressions in $\mathcal{SOOL}$ consist of identifiers, constants, the keyword `nil`, `val` expressions, `ref` expressions, function applications, function expressions, `class` expressions, `new` expressions (for creating new objects), message sends, extracting a labeled field, and subclass expressions. The keyword `nil` represents an uninitialized object.

We have already discussed `ref` and `val` expressions. As stated earlier, if `x` is declared to be a variable of type `T` in the abbreviated language, then `x` will be defined as a constant with type `Ref T` in the expanded language. It will be initialized with an appropriate default value for the type. If `T` is an object type then it will be initialized with `nil`. Variables of number types will be initialized to the zero value of the type, characters to the "`NUL`" character, and strings to the empty string. Functions will be initialized to functions returning the default value of the result type.

For example, a variable declaration of the form

```
x: Integer;
```

will be transformed into the constant definition

```
x: Ref Integer = ref 0;
```

Thus `x` is initialized to be a reference to a location that initially holds `0`.

Assignment statements can update the value held (referred to) by `x`, but do not change the reference (location) itself. Hence it may be included in with the other constant definitions.

Function applications consist of a function expression followed by the list of actual parameters. Function expressions contain a list of formal parameters and their types, the return type (which is `Command` for procedures), and the function body.

Class expressions contain a record of initial values of instance variables and a record of method definitions. Subclass expressions differ from class expressions by including the superclass and a list of methods that are to be overridden by the subclass. "`New`" expressions contain the class that is used as a template to generate a new object.

Message sends include the target object and the name of the message. Recall that we use $\Leftarrow$ to distinguish message sends from accesses to instance variables, which use the "." notation. Sending a message to `nil` should result in a run-time error. Labelled field projection normally appears only in the bodies of method expressions. It typically appears in the form `self.x`, where `x` is the name of an instance variable of the class.

The statements of the language include `nop`, assignment statements, if-then-else expressions, while loops, and sequencing of statements. The expression `nop` represents a statement that has no effect (a "no op" or "skip").

Two keywords of the language do not appear in the grammar above. They are `self`, a name for the object currently executing a method, and `close`, a function used to convert the type of `self` in order to hide the instance variables. We do not introduce them as constants because they may appear only inside method definitions, and cannot be assigned types independently of those contexts. In particular their meanings will vary depending on the context. To be consistent with our type-checking rules presented below, these keywords will instead be introduced as "bound" identifiers. The type of `self` will always be the "visible" object type generated by the class definition that contains it. Recall this is necessary so that access is allowed to instance variables.

We have not yet discussed the `close` keyword. Its sole purpose is to convert objects of type `VisObjectType`(`IV`^{ref},`M`) to type `ObjectType M` once we no longer need access to instance variables. If the value of `self` is provided outside of the class definition, either by returning it from a method or using it as a parameter in another method, the instance variables should no longer be accessible. In particular, we do not wish to have either the return type or the parameter type of a method be a visible object type that includes the types of instance variables.

Thus, before passing `self` as a parameter or returning it from a method, we will apply `close` in order to convert it to an object type. As with `val`, the abbreviated form of the language will not require the programmer to insert applications of `close`, instead they will automatically be inserted where needed.

10.2.2 Examples

Figure 10.3 contains a very simple example of a complete program in $\mathcal{SOOL}$. It defines a `Point` class, creates an object from `Point`, and moves it. It uses the syntax abbreviations that allow the program to look more like it was written in a typical object-oriented language. Notice that we do not require the section headings `type` and `const`, and variable declarations are written in a form similar to those found in most languages.

If we remove all of the abbreviations, we get the program in Figure 10.4. For this program, the syntax is defined exactly by the context-free grammar given in the previous section.

We have not yet discussed constructors for classes. We will omit constructors in $\mathcal{SOOL}$, replacing them by functions that return classes. In the language with abbreviations we will write these as parameterized classes, but these can be replaced with functions returning classes.

In Figure 10.5, a variation of the program `PointExample` from Figure

```
Program PointExample;

   PointType = ObjectType {
      move: Integer × Integer → Command;
      getx: Void → Integer;
      gety: Void → Integer
   };

   class Point {
      x: Integer = 0;
      y: Integer = 0;
      function move(dx: Integer,dy: Integer): Command is
      {
         x := x + dx;
         y := y + dy }
      function getx(): Integer is { return x }
      function gety(): Integer is { return y }
   };

   pt: PointType;

{
   pt := new Point;
   pt ⇐ move(3,2)
}
```

Figure 10.3 `PointExample` program in language with abbreviations.

10.3 illustrates the use of parameterized classes. The class `PPoint` takes two integers as parameters. The two integer parameters are used to initialize the instance variables `x` and `y`.

In Figure 10.6 we remove the abbreviations to reveal the parameterized class to be a function returning a class. In the assignment to `pt` in the main program, the application of `PPoint` to the arguments `2` and `7` has higher priority than the `new` operator.

```
Program PointExample;
type
   PointType = ObjectType {|
      move:Integer × Integer → Command;
      getx:Void → Integer;
      gety:Void → Integer
   |};
   PtClassType = ClassType ( {|
      x:Integer;
      y:Integer
   |}, {|
      move:Integer × Integer → Command;
      getx:Void → Integer;
      gety:Void → Integer
   |});
const
   Point:PtClassType = class ( {|
      x:Integer = 0;
      y:Integer = 0;
   |}, {|
      move:Integer × Integer → Command =
        function(dx:Integer, dy:Integer): Command is {
         self.x := (val self.x) + dx;
         self.y := (val self.y) + dy;
         return nop };
      getx:Void → Integer = function(): Integer is {
         return val self.x };
      gety:Void → Integer = function(): Integer is {
         return val self.y };
   |});
   pt: Ref PointType = ref nil;
{
   pt := new Point;
   pt ⇐ move(3,2);
   return nop
}
```

Figure 10.4 `PointExample` program in language without abbreviations.

```
Program PPointExample;

   ...

   class PPoint(x0: Integer, y0: Integer) {
      x: Integer = x0;
      y: Integer = y0;
      ...
   };

   pt: PointType;

{
   pt := new PPoint(2,7);
   pt ⇐ move(3,2)
}
```

Figure 10.5 Parameterized Point class in language with abbreviations.

10.2.3 Type-checking rules

Most programming languages do not provide explicit formal rules for statically type-checking programs to the programmer. Instead they are typically implicit in the language definition and then in the code of compilers. However, these rules are not difficult to formulate. As in the typed lambda calculus, the type-checking rules can be given inductively based on the productions of the context-free grammar that generates the language. In this section we provide the formal type-checking rules for $\mathcal{SOOL}$.

As in the typed lambda calculus, the type checking rules require information about the types of free identifiers in an expression and information about the meaning of type identifiers. The static type environment $\mathcal{E}$ associates expression identifiers with types.

Definition 10.2.3 *A* static type environment*, $\mathcal{E}$, is a finite set of associations between identifiers and type expressions of the form* $\mathtt{x}\colon \mathtt{T}$*, where each* $\mathtt{x}$ *is unique in* $\mathcal{E}$ *and* $\mathtt{T}$ *is a type. If the relation* $\mathtt{x}\colon \mathtt{T} \in \mathcal{E}$*, then we sometimes write* $\mathcal{E}(\mathtt{x}) = \mathtt{T}$.

An important difference between $\mathcal{SOOL}$ and the typed lambda calculus is that $\mathcal{SOOL}$ can include type definitions. These definitions are recorded

```
Program PPointExample;

type
   ...

const

   PPoint: Integer × Integer → PtClassType =
     function(x0: Integer,y0: Integer): PtClassType is {
        return class ({|
                        x: Integer = x0;
                        y: Integer = y0;
                      |}, {|
                        ...
                      |})}

   pt: Ref PointType = ref nil;

{
   pt := new PPoint(2,7);
   pt ⇐ move(3,2);
   return nop
}
```

Figure 10.6 Parameterized Point class in language without abbreviations.

for type checking purposes in a type constraint system that associates type identifiers with their definitions. That way the identifiers can be replaced by their definitions wherever necessary.

TYPE CONSTRAINT SYSTEM

Definition 10.2.4 *Relations of the form* `t = T`*, where* `t` *is a type identifier and* `T` *is a type expression, are said to be* type definitions*. A* type constraint system*, $\mathcal{C}$, is defined as follows:*

1. *The empty set, $\emptyset$, is a type constraint system.*

2. *If $\mathcal{C}$ is a type constraint system and* `t` *is a type identifier that does not appear in $\mathcal{C}$ or* `T`*, then $\mathcal{C} \cup \{\texttt{t} = \texttt{T}\}$ is a type constraint system.*

For example, in type checking the body of the program `PointExample`, $\mathcal{C}$ would contain type definitions for both `PointType` and `PtClassType`.

Recall that a type constraint system for the polymorphic lambda calculus kept track of subtyping assumptions rather than type definitions. We retain the same name here because we will add subtyping definitions to type constraint systems when we add polymorphism to the language.

The function $\mathcal{C}(\mathtt{T})$ returns the type expression formed by replacing all type identifiers in `T` by their definitions in $\mathcal{C}$ (recursively, if necessary).

$\mathcal{C}(\mathtt{T})$

Definition 10.2.5 *$\mathcal{C}(\mathtt{T})$, for `T` a type expression, is defined inductively as follows:*

1. *If `t` is a type identifier, then $\mathcal{C}(\mathtt{t}) = \mathcal{C}(\mathtt{T})$ if $(\mathtt{t} = \mathtt{T}) \in \mathcal{C}$, otherwise $\mathcal{C}(\mathtt{t}) = \mathtt{t}$.*
2. *If `C` is a type constant, then $\mathcal{C}(\mathtt{C}) = \mathtt{C}$.*
3. $\mathcal{C}(\mathtt{T}_1 \times \ldots \times \mathtt{T}_n \to \mathtt{T}_{n+1}) = \mathcal{C}(\mathtt{T}_1) \times \ldots \times \mathcal{C}(\mathtt{T}_n) \to \mathcal{C}(\mathtt{T}_{n+1})$.
4. $\mathcal{C}(\mathtt{Ref\,T}) = \mathtt{Ref}\,\mathcal{C}(\mathtt{T})$.
5. $\mathcal{C}(\mathtt{ObjectType\ RT}) = \mathtt{ObjectType}\,\mathcal{C}(\mathtt{RT})$.
6. $\mathcal{C}(\mathtt{VisObjectType(RT_i, RT_m)}) = \mathtt{VisObjectType}(\mathcal{C}(\mathtt{RT_i}), \mathcal{C}(\mathtt{RT_m}))$.
7. $\mathcal{C}(\mathtt{ClassType(RT_i, RT_m)}) = \mathtt{ClassType}(\mathcal{C}(\mathtt{RT_i}), \mathcal{C}(\mathtt{RT_m}))$.
8. $\mathcal{C}(\{\!|\ \mathtt{l}_1\colon \mathtt{T}_1; \ldots; \mathtt{l}_n\colon \mathtt{T}_n\ |\!\}) = \{\!|\ \mathtt{l}_1\colon \mathcal{C}(\mathtt{T}_1); \ldots; \mathtt{l}_n\colon \mathcal{C}(\mathtt{T}_n)\ |\!\}$

The function $\mathcal{C}(\mathtt{T})$ is guaranteed to terminate because the restrictions in clause (2) of the definition of $\mathcal{C}$ rule out recursive (or mutually recursive) type definitions. If all free identifiers of `T` are contained in the domain of $\mathcal{C}$, then $\mathcal{C}(\mathtt{T})$ will be a type expression with no free identifiers. A recursive definition is necessary because type definitions occurring later in a program may use type identifiers defined earlier.

We begin by introducing rules that process declarations. Processing a type or constant definition or variable declaration results in adding type information about the new identifiers introduced into the type constraint system, $\mathcal{C}$, or type environment, $\mathcal{E}$.

The symbol $\diamond$ is used to separate a declaration in $\mathcal{SOOL}$ from the resulting constraint system and type environment. A *judgement* of the form $\mathcal{C}, \mathcal{E} \vdash \mathtt{Dec} \diamond \mathcal{C}', \mathcal{E}'$ asserts that, under the assumptions in $\mathcal{C}$ and $\mathcal{E}$, processing the declaration `Dec` results in an enhanced type constraint system, $\mathcal{C}'$, and type

JUDGEMENT

environment, $\mathcal{E}'$. This notation makes it clear that the result of processing a declaration is changes to the type constraint system and type environment.

The processing rules for declarations are written similarly to the type-checking rules for the lambda calculus given in Chapter 8:

$$\textit{AxiomName} \qquad \mathcal{C}, \mathcal{E} \vdash \texttt{Dec} \diamond \mathcal{C}', \mathcal{E}'$$

if the rule is not dependent on hypotheses, or

$$\textit{RuleName} \qquad \frac{\mathcal{C}, \mathcal{E} \vdash \texttt{Dec}_1 \diamond \mathcal{C}_1, \mathcal{E}_1, \ \ldots, \ \mathcal{C}_{n-1}, \mathcal{E}_{n-1} \vdash \texttt{Dec}_n \diamond \mathcal{C}_n, \mathcal{E}_n}{\mathcal{C}, \mathcal{E} \vdash \texttt{Dec} \diamond \mathcal{C}_n, \mathcal{E}_n},$$

where the conclusion is written below the line, and the hypotheses are above the line. Typically, the expressions appearing in judgements above the line in rules are subexpressions of the expressions appearing in judgements below the line.

We read the rules starting from the bottom and then proceeding from left to right across the top. For example, the sample rule can be read as stating, "from the type assumptions in $\mathcal{C}$ and $\mathcal{E}$, processing `Dec` results in a richer collection of type assumptions $\mathcal{C}_n, \mathcal{E}_n$, if processing $\texttt{Dec}_1$ from $\mathcal{C}$ and $\mathcal{E}$ results in $\mathcal{C}_1, \mathcal{E}_1, \ldots$, and processing $\texttt{Dec}_n$ from $\mathcal{C}_{n-1}, \mathcal{E}_{n-1}$ results in $\mathcal{C}_n, \mathcal{E}_n$."

The type-checking rules for declarations can be found in Figure 10.7. They are relatively straightforward. By rule *TypeDef*, processing a type definition results in adding the definition to $\mathcal{C}$, but no changes to the type environment, $\mathcal{E}$. In contrast, rule *ConstDef* indicates that processing a constant declaration results in adding the constant name and its type to the type environment, but no change to the type constraint system, $\mathcal{C}$. A list of type or constant declarations is processed by processing the first declaration to get an enriched type environment, and then processing the rest of the declarations recursively to get the final type environment.

Type-checking rules for non-declarations are written similarly. A judgement of the form $\mathcal{C}, \mathcal{E} \vdash \texttt{exp}\colon \texttt{T}$ asserts that, under the type assumptions in $\mathcal{C}, \mathcal{E}$, the expression `exp` has type `T`. We introduce a special notation to make writing type-checking rules easier. If `RType` is a record type of the form $\{\!|\texttt{x}_1\colon \texttt{T}_1; \ldots; \texttt{x}_n\colon \texttt{T}_n|\!\}$, we let $\texttt{RType}^{\textit{ref}} = \{\!|\texttt{x}_1\colon \texttt{Ref}\ \texttt{T}_1; \ldots; \texttt{x}_n\colon \texttt{Ref}\ \texttt{T}_n|\!\}$.

We will give type-checking rules for each of the other syntactic categories of $\mathcal{SOOL}$. We start by providing the type-checking rules for expressions in Figure 10.8.

Most of these rules are straightforward. By rule *Identifier*, an identifier can be deduced to have the type that is specified for it in the type environment, *TypeSec* $$\frac{\mathcal{C},\mathcal{E}\vdash \texttt{TDefLst} \diamond \mathcal{C}',\mathcal{E}'}{\mathcal{C},\mathcal{E}\vdash \texttt{type TDefLst} \diamond \mathcal{C}',\mathcal{E}'}$$

TypeDefLst $$\frac{\mathcal{C},\mathcal{E}\vdash \texttt{t} = \texttt{T} \diamond \mathcal{C}_1,\mathcal{E}_1 \quad \mathcal{C}_1,\mathcal{E}_1\vdash \texttt{TDefLst} \diamond \mathcal{C}_2,\mathcal{E}_2}{\mathcal{C},\mathcal{E}\vdash \texttt{t} = \texttt{T};\ \texttt{TDefLst} \diamond \mathcal{C}_2,\mathcal{E}_2}$$

TypeDef $$\mathcal{C},\mathcal{E}\vdash \texttt{t} = \texttt{T} \diamond \mathcal{C}\cup\{\texttt{t} = \texttt{T}\},\mathcal{E} \quad \textit{if}\ \texttt{t}\notin \textit{dom}(\mathcal{C})$$

ConstSec $$\frac{\mathcal{C},\mathcal{E}\vdash \texttt{CDefLst} \diamond \mathcal{C}',\mathcal{E}'}{\mathcal{C},\mathcal{E}\vdash \texttt{const CDefLst} \diamond \mathcal{C}',\mathcal{E}'}$$

ConstDefList $$\frac{\mathcal{C},\mathcal{E}\vdash \texttt{id: T} = \texttt{E} \diamond \mathcal{C},\mathcal{E}_1 \qquad \mathcal{C},\mathcal{E}_1\vdash \texttt{CDefLst} \diamond \mathcal{C},\mathcal{E}_2}{\mathcal{C},\mathcal{E}\vdash \texttt{id: T} = \texttt{E};\ \texttt{CDefLst} \diamond \mathcal{C},\mathcal{E}_2}$$

ConstDef $$\frac{\mathcal{C},\mathcal{E}\vdash \texttt{E: T} \quad \texttt{id}\notin \textit{dom}(\mathcal{E})}{\mathcal{C},\mathcal{E}\vdash \texttt{id: T} = \texttt{E} \diamond \mathcal{C},\mathcal{E}\cup\{\texttt{id: T}\}}$$

Figure 10.7 Type checking rules for declarations.

$\mathcal{E}$. In reporting the type of the identifier, all of the type identifiers in the type are replaced by their definitions in $\mathcal{C}$. This is done by writing the resulting type as $\mathcal{C}(\texttt{T})$. Because all user-introduced names will be replaced by their definitions during type-checking, $\mathcal{SOOL}$ uses structural rather than name equivalence of types [Lou93].

By rule *Const*, built-in constants are assigned the type that is provided in the language definition. The expression `()` is the only value of type `Void`. The expression `nil` is unusual in that it can have any object type. Which type it is assigned will depend on context. If `E` has type `T`, then `ref E` is a reference to `E`, and hence has type `Ref T`. If `E` has type `Ref T`, *i.e.*, is a variable holding values of type `T`, then `val E` has type `T`.

Type checking function definitions is only a bit more complicated. By the *Function* rule, a function has type $\texttt{T}_1 \times \ldots \times \texttt{T}_n \rightarrow \texttt{T}_{n+1}$ iff the body has type $\texttt{T}_{n+1}$ under the added assumptions (in $\mathcal{E}$) that the parameters have types $\texttt{T}_1$ through $\texttt{T}_n$. The assumption on the types of formal parameters is necessary since the function body may involve the formal parameters. Similarly if a function has type $\texttt{T}_1 \times \ldots \times \texttt{T}_n \rightarrow \texttt{T}_{n+1}$, and the actual parameters have types $\texttt{T}_1$ through $\texttt{T}_n$, the *Application* rule indicates that the result of applying the function to the actual parameters has type $\texttt{T}_{n+1}$.

Identifier $$\mathcal{C}, \mathcal{E} \cup \{\texttt{id}\colon \texttt{T}\} \vdash \texttt{id}\colon \mathcal{C}(\texttt{T})$$

Constant $$\mathcal{C}, \mathcal{E} \vdash \texttt{c}\colon \texttt{C}$$

where `C` is the pre-assigned type for built-in constant `c`.

Void $$\mathcal{C}, \mathcal{E} \vdash ()\colon \texttt{Void}$$

Nil $$\mathcal{C}, \mathcal{E} \vdash \texttt{nil}\colon \texttt{ObjectType}\ \texttt{M}$$

Value $$\frac{\mathcal{C}, \mathcal{E} \vdash \texttt{E}\colon \texttt{Ref}\ \texttt{T}}{\mathcal{C}, \mathcal{E} \vdash \texttt{val}\ \texttt{E}\colon \texttt{T}}$$

Function $$\frac{\mathcal{C}, \mathcal{E} \cup \{\texttt{id}_1\colon \texttt{T}_1, \ldots, \texttt{id}_n\colon \texttt{T}_n\} \vdash \texttt{Block}\colon \texttt{T}_{n+1}}{\begin{array}{r}\mathcal{C}, \mathcal{E} \vdash \texttt{function}(\texttt{id}_1\colon \texttt{T}_1, \ldots, \texttt{id}_n\colon \texttt{T}_n)\colon \texttt{T}_{n+1}\ \texttt{is Block}\colon \\ \texttt{T}_1 \times \ldots \times \texttt{T}_n \rightarrow \texttt{T}_{n+1}\end{array}}$$

Application $$\frac{\mathcal{C}, \mathcal{E} \vdash \texttt{E}\colon \texttt{T}_1 \times \ldots \times \texttt{T}_n \rightarrow \texttt{T}_{n+1},\ \ \mathcal{C}, \mathcal{E} \vdash \texttt{E}_1\colon \texttt{T}_1,\ \ldots,\ \mathcal{C}, \mathcal{E} \vdash \texttt{E}_n\colon \texttt{T}_n}{\mathcal{C}, \mathcal{E} \vdash \texttt{E}(\texttt{E}_1, \ldots, \texttt{E}_n)\colon \texttt{T}_{n+1}}$$

Reference $$\frac{\mathcal{C}, \mathcal{E} \vdash \texttt{E}\colon \texttt{T}}{\mathcal{C}, \mathcal{E} \vdash \texttt{ref}\ \texttt{E}\colon \texttt{Ref}\ \texttt{T}}$$

Class $$\frac{\mathcal{C}, \mathcal{E} \vdash \texttt{inst}\colon \texttt{IV}, \qquad \mathcal{C}', \mathcal{E}' \vdash \texttt{meth}\colon \texttt{M}}{\mathcal{C}, \mathcal{E} \vdash \texttt{class}\,(\texttt{inst}, \texttt{meth})\colon \texttt{ClassType}\,(\texttt{IV}, \texttt{M})}$$

where

- $\mathcal{C}' = \mathcal{C} \cup \{\texttt{SelfType} = \texttt{VisObjectType}\,(\texttt{IV}^{ref}, \texttt{M})\}$,
- $\mathcal{E}' = \mathcal{E} \cup \{\texttt{self}\colon \texttt{SelfType}, \texttt{close}\colon \texttt{SelfType} \rightarrow \texttt{ObjectType M}\}$, and
- `SelfType` does not occur in `IV` or `M`.

New $$\frac{\mathcal{C}, \mathcal{E} \vdash \texttt{E}\colon \texttt{ClassType}\,(\texttt{IV}, \texttt{M})}{\mathcal{C}, \mathcal{E} \vdash \texttt{new}\ \texttt{E}\colon \texttt{ObjectType M}}$$

Message $$\frac{\mathcal{C}, \mathcal{E} \vdash \texttt{E}\colon \texttt{ObjectType}\ \{\!|\texttt{m}_1\colon \texttt{T}_1, \ldots, \texttt{m}_n\colon \texttt{T}_n|\!\}}{\mathcal{C}, \mathcal{E} \vdash \texttt{E} \Leftarrow \texttt{m}_i\colon \texttt{T}_i}$$

Figure 10.8 Type-checking rules for expressions of $\mathcal{SOOL}$, part 1.

The type checking of classes is more challenging. A class of the form `class`(`inst`, `meth`) has type `ClassType(IV, M)`, if two conditions hold. First, the record of instance variables, `inst`, must have type `IV`. Second, the record of methods, `meth`, must have type `M`.

There is a complication, however. Because method bodies may contain occurrences of `self`, we need to type check methods in a context that allows us to assign a type to all expressions involving `self`. We assign the type `SelfType` to `self`, where `SelfType = VisObjectType(IV`^{ref}`, M)`. Recall that the type expression `IV`^{ref} represents a record type formed from `IV` by adding `Ref` before the type of each field type. That is, if `IV` = $\{\mathtt{l}_1\!:\mathtt{T}_1; \ldots; \mathtt{l}_n\!:\mathtt{T}_n\}$, then `IV`^{ref} = $\{\mathtt{l}_1\!:\mathtt{Ref}\ \mathtt{T}_1; \ldots; \mathtt{l}_n\!:\mathtt{Ref}\ \mathtt{T}_n\}$.

While `inst` with type `IV` is the record of initial *values* of instance variables, the type `SelfType` of `self` includes the types of the instance *variables* themselves. That is, if `x` is an instance variable and the class contains an initial value of `x` of type `T`, then the corresponding type in the visible object type is `Ref T`.

Because `self` will be assigned type `SelfType` in $\mathcal{E}$, we will be able to type check expressions involving sending messages to `self` and extracting instance values from `self` inside the bodies of methods. (The type checking rules for these operations will be introduced in the next figure.) As discussed earlier, `close` is introduced as a function to convert values (like `self`) from `SelfType` to `ObjectType M`. Its typing is inserted into $\mathcal{E}'$ in order to type check the bodies of methods.

Applying rule *Class* to type check a class definition requires that we know the types of the instance variables and methods *before* we begin type checking, because this information must be added to $\mathcal{C}$ and $\mathcal{E}$ when type checking methods of the class. Our syntax for classes ensures that each instance variable and method definition inside a class does appear with its type information, so no difficulties arise with having to guess what these types might be if `inst` and `meth` are written as record expressions. However it is possible that `inst` or `meth` could be provided in some other way, *e.g.*, as a formal parameter or the value of a variable, so that we only know a type, rather than the most explicit type.

For example, we might only have partial information on the instance variables or methods in type-checking an expression. Suppose `m` is a method that is never referenced in any of the other methods of the class, and let `M'` be the type of the record consisting of all of the methods of the class aside from `m`. We could then type check the methods under the assumption that `SelfType= VisObjectType(IV`^{ref}`, M')` rather than `VisObjectType(IV`^{ref}`, M)`.

If the type checking succeeds, then the resulting type of the class would be `ClassType(IV, M′)` rather than `ClassType(IV, M)`.

Of course the type of objects generated from the class with this typing would be `ObjectType M′` $<:$ `ObjectType M`. Thus there is little advantage to obtaining a weaker type for the class. As a result, a type-checking algorithm will normally deduce the most specific type possible for the class. Most languages do not allow a programmer to pass a record of methods or even an individual method into a class. Thus this issue rarely arises.

Finally, note the restriction that `SelfType` may not occur in the types of either instance variables or methods. "Visible object types" are not really part of $\mathcal{SOOL}$, they are simply introduced as an aid in type-checking programs. As a result it may not appear explicitly anywhere in a program, though it is used when type checking expressions involving `self`.

Expressions of the form `new E` generate new objects from a class. The new object has an object type with methods corresponding to those given in the class type of `E`. By rules *Message*, a message send has the type of the corresponding method of the object type.

The type-checking rules for subclasses, records, and blocks are given in Figure 10.9.

The type-checking rule for subclasses is longer than any we have seen so far, but it is simpler than it may first appear. Recall from Chapter 6 that a subclass may add new instance variables and methods, but that the types of instance variables may not be changed, and that the types of overridden methods may only be replaced by subtypes. Thus parameters may only change in a contravariant way, and return types in a covariant way.

As a result, to type-check the expression

$$\texttt{class inherits E modifies } \texttt{l}_{i_1}, \ldots, \texttt{l}_{i_m}(\texttt{inst}, \texttt{meth}),$$

we must first determine the type of superclass, `E`, the new instance variables in `inst`, and the new methods in `meth`. We also must ensure that the names of the new instance variables do not overlap with those of `E`, and that if any of the new method names overlap with those of `E`, then the new types in the subclass will be subtypes of those for the superclass `E`. (We do not bother to check whether instance variable and method identifiers overlap since they are referred to in different ways.)

If the type of the subclass is expected to be `ClassType(IV, M)`, then we determine the types of the new methods under the assumption that the type of `self` is `SelfType` = `VisObjectType` $(\texttt{IV}^{ref}, \texttt{M})$, and that the type of `close` is `SelfType` $\rightarrow$ `ObjectType M`, as with the type checking rule for classes.

Subclass
$$\frac{\begin{array}{c}\mathcal{C},\mathcal{E} \vdash \texttt{E}\colon \texttt{ClassType}\,(\texttt{IV}_{sup}, \texttt{M}_{sup}\,),\\ \mathcal{C},\mathcal{E} \vdash \texttt{inst}\colon \texttt{IV}_{sub}, \qquad \mathcal{C}',\mathcal{E}' \vdash \texttt{meth}\colon \texttt{M}_{sub}\end{array}}{\begin{array}{l}\mathcal{C},\mathcal{E} \vdash \texttt{class inherits E modifies } \texttt{l}_{i_1},\ldots,\texttt{l}_{i_m}\\ \qquad\qquad (\texttt{inst},\texttt{meth})\colon \texttt{ClassType}\,(\texttt{IV},\texttt{M})\end{array}}$$

where

- $\texttt{IV} = \texttt{IV}_{sup} \oplus \texttt{IV}_{sub}$ and $\texttt{M} = \texttt{M}_{sup} \oplus \texttt{M}_{sub}$,
- there is no overlap in the labels occurring in $\texttt{IV}_{sup}$ and $\texttt{IV}_{sub}$,
- the overlapping labels in $\texttt{M}_{sup}$ and $\texttt{M}_{sub}$ are exactly $\texttt{l}_{i_1},\ldots,\texttt{l}_{i_m}$, and the type of each $\texttt{l}_{i_j}$ in $\texttt{M}_{sub}$ is a subtype of the type of the same label in $\texttt{M}_{sup}$,
- $\mathcal{C}' = \mathcal{C} \cup \{\texttt{SelfType} = \texttt{VisObjectType}\,(\texttt{IV}^{ref}, \texttt{M})\}$,
- $\mathcal{E}' = \mathcal{E} \cup \{\texttt{self}\colon \texttt{SelfType},\ \texttt{close}\colon \texttt{SelfType} \rightarrow \texttt{ObjectType M}\}$, and
- $\texttt{SelfType}$ does not occur in $\texttt{IV}$ or $\texttt{M}$.

Inst Vble
$$\frac{\mathcal{C},\mathcal{E} \vdash \texttt{E}\colon \texttt{VisObjectType}\,(\texttt{IVR},\texttt{M})}{\mathcal{C},\mathcal{E} \vdash \texttt{E.l}_k\colon \texttt{T}_k}$$

where $\texttt{IVR} = \{\!|\ \texttt{l}_1\colon \texttt{T}_1,\ldots,\texttt{l}_n\colon \texttt{T}_n |\!\}$ and $1 \le k \le n$.

VisObj Message
$$\frac{\mathcal{C},\mathcal{E} \vdash \texttt{E}\colon \texttt{VisObjectType}\,(\texttt{IVR},\texttt{M})}{\mathcal{C},\mathcal{E} \vdash \texttt{E} \Leftarrow \texttt{m}_j\colon \texttt{U}_j}$$

where $\texttt{M} = \{\!|\ \texttt{m}_1\colon \texttt{U}_1,\ldots,\texttt{m}_k\colon \texttt{U}_k |\!\}$ and $1 \le j \le k$.

Record
$$\frac{\mathcal{C},\mathcal{E} \vdash \texttt{E}_i\colon \texttt{T}_i,\ \textit{for } 1 \le i \le n}{\mathcal{C},\mathcal{E} \vdash \{\!|\ \texttt{l}_1\colon \texttt{T}_1 = \texttt{E}_1,\ldots,\texttt{l}_n\colon \texttt{T}_n = \texttt{E}_n |\!\}\colon \{\!|\ \texttt{l}_1\colon \texttt{T}_1,\ldots,\texttt{l}_n\colon \texttt{T}_n |\!\}}$$

Block
$$\frac{\begin{array}{c}\mathcal{C},\mathcal{E} \vdash \texttt{TDefs} \diamond \mathcal{C}_1,\mathcal{E},\ \ \mathcal{C}_1,\mathcal{E} \vdash \texttt{CDefs} \diamond \mathcal{C}_1,\mathcal{E}_1,\\ \mathcal{C}_1,\mathcal{E}_1 \vdash \texttt{S}\colon \texttt{Command},\ \ \mathcal{C}_1,\mathcal{E}_1 \vdash \texttt{E}\colon \texttt{T}\end{array}}{\mathcal{C},\mathcal{E} \vdash \texttt{TDefs CDefs}\,\{\,\texttt{S return E};\}\colon \texttt{T}}$$

where `T` does not contain any type identifiers defined in `TDefs`.

Type Abbrev
$$\frac{\mathcal{C},\mathcal{E} \vdash \texttt{E}\colon \mathcal{C}(T)}{\mathcal{C},\mathcal{E} \vdash \texttt{E}\colon \texttt{T}}$$

Figure 10.9 Type-checking rules for expressions of $\mathcal{SOOL}$, part 2.

The notation $\mathtt{R} \oplus \mathtt{R'}$ represents the combination of two record types. The result is a record type with all of the fields of both record types. If there are overlapping labels, then the type associated with each such label in the result is the corresponding type from R′. Thus we can think of $\oplus$ as a right dominant operation that combines record types.

The next two rules provide type-checking rules for message sends and instance variable extraction from elements of type VisObjectType(IVR, M). Recall that the type of self in type checking methods in classes and subclasses is of the form VisObjectType(IVR, M). The result of extracting an instance variable has the same type as the corresponding field of the type of the record of instance variables. The type checking rule for message sends to visible object types is essentially the same as for object types and is based only on the type of the methods.

The type of a record, E, is determined by the types associated with the individual fields. Notice that record expressions include type expressions along with the values for each field.

A block consists of the declarations and body of either a program or function. By rule *Block*, it is type checked by processing the declarations to obtain a new type environment, which is then used to type check the body and return expression. The body should be a statement, and the type of the return expression determines the type of the block. The type of the block should not involve any local type definitions as they will not be visible outside of the scope of the block.

By using the rule *Identifier*, we end up typing expressions by first removing all user-introduced type identifiers. That is, when x: T is included in $\mathcal{E}$, we assigned x the type $\mathcal{C}(\mathtt{T})$ rather than just T. However, as a convenience, we would also like to report the types of expressions with type expressions that use these abbreviations. By rule *Type Abbrev*, we can provide an expression with a type expression involving abbreviations once we determine that the expression has the type obtained by removing the abbreviations.

Figure 10.10 contains type-checking rules for statements. Recall that well-typed statements have type Command.

The special constant, nop, is a command, by rule *No Op*. By rule *Assn*, an assignment statement is well-typed if the type of the left side is a reference to the type of the expression appearing on the right side. If-then-else and while statements are well-typed if the guard expressions have type Boolean and the component statements are themselves well-typed. Individual statements in statement lists should be well-typed. However, unlike the lambda calculus, we do not allow non-statements, *i.e.*, expressions with

No Op $\quad \mathcal{C},\mathcal{E} \vdash \texttt{nop}\colon \texttt{Command}$

Assn $$\frac{\mathcal{C},\mathcal{E} \vdash \texttt{id}\colon \texttt{Ref T}, \qquad \mathcal{C},\mathcal{E} \vdash \texttt{E}\colon \texttt{T}}{\mathcal{C},\mathcal{E} \vdash \texttt{id} := \texttt{E}\colon \texttt{Command}}$$

Cond $$\frac{\begin{array}{c}\mathcal{C},\mathcal{E} \vdash \texttt{E}\colon \texttt{Boolean},\\ \mathcal{C},\mathcal{E} \vdash \texttt{S}_1\colon \texttt{Command}, \qquad \mathcal{C},\mathcal{E} \vdash \texttt{S}_2\colon \texttt{Command}\end{array}}{\mathcal{C},\mathcal{E} \vdash \texttt{if E then } \{\,\texttt{S}_1\,\}\texttt{ else }\{\,\texttt{S}_2\,\}\colon \texttt{Command}}$$

While $$\frac{\mathcal{C},\mathcal{E} \vdash \texttt{E}\colon \texttt{Boolean}, \qquad \mathcal{C},\mathcal{E} \vdash \texttt{S}\colon \texttt{Command}}{\mathcal{C},\mathcal{E} \vdash \texttt{while E do } \{\,\texttt{S}\,\}\colon \texttt{Command}}$$

StmtList $$\frac{\mathcal{C},\mathcal{E} \vdash \texttt{S}_1\colon \texttt{Command}, \qquad \mathcal{C},\mathcal{E} \vdash \texttt{S}_2\colon \texttt{Command}}{\mathcal{C},\mathcal{E} \vdash \texttt{S}_1;\ \texttt{S}_2\colon \texttt{Command}}$$

Program $$\frac{\mathcal{C},\mathcal{E} \vdash \texttt{Block}\colon \texttt{Command}}{\mathcal{C},\mathcal{E} \vdash \texttt{Program id};\ \texttt{Block.}\colon \texttt{Command}}$$

Figure 10.10 Type-checking rules for statements of $\mathcal{SOOL}$.

type different from `Command`, to occur in statement lists. Finally, a program is well-typed if the block it contains is well-typed.

We have now defined type-checking rules for the complete language, but we have not yet mentioned subtyping. Rather than using the very simple invariant subtyping rules used in Java and early versions of C++, we instead use the more general rules discussed in Chapter 5. The $\mathcal{SOOL}$ subtyping rules can be found in Figure 10.11.

Because $\mathcal{C}$ does not contain subtyping assumptions, we need $\mathcal{C}$ only to expand user-defined type definitions. Rule *TypeDef* $_{<:}$ specifies that to determine if two types are subtypes under the assumptions in $\mathcal{C}$, simply expand the type definitions and determine whether the expanded types are subtypes using no assumptions.

The remaining subtyping rules make no direct use of $\mathcal{C}$. Rule *Reflex* $_{<:}$ specifies that a type is always a subtype of itself, while rule *Trans* $_{<:}$ asserts that subtyping is closed under transitivity. Subtyping for functions is covariant in the argument type and contravariant in the argument types by rule *Function* $_{<:}$, just as we would expect based on our earlier discussions of subtyping for function types in Section 5.1.2.

$$\textit{TypeDef}_{<:} \qquad \frac{\emptyset \vdash \mathcal{C}(\mathsf{S}) <: \mathcal{C}(\mathsf{T})}{\mathcal{C} \vdash \mathsf{S} <: \mathsf{T}}$$

$$\textit{Reflex}_{<:} \qquad \mathcal{C} \vdash \mathsf{S} <: \mathsf{S}$$

$$\textit{Trans}_{<:} \qquad \frac{\mathcal{C} \vdash \mathsf{S} <: \mathsf{T}, \qquad \mathcal{C} \vdash \mathsf{T} <: \mathsf{U}}{\mathcal{C} \vdash \mathsf{S} <: \mathsf{U}}$$

$$\textit{Function}_{<:} \qquad \frac{\begin{array}{c}\mathcal{C} \vdash \mathsf{T}_i <: \mathsf{S}_i, \textit{ for } 1 \leq i \leq n, \\ \mathcal{C} \vdash \mathsf{S}_{n+1} <: \mathsf{T}_{n+1}\end{array}}{\mathcal{C} \vdash \mathsf{S}_1 \times \ldots \times \mathsf{S}_n \to \mathsf{S}_{n+1} <: \mathsf{T}_1 \times \ldots \times \mathsf{T}_n \to \mathsf{T}_{n+1}}$$

$$\textit{Record}_{<:} \qquad \frac{m \leq n \ \textit{ and } \mathcal{C} \vdash \mathsf{S}_i <: \mathsf{T}_i \textit{ for all } 1 \leq i \leq m}{\mathcal{C} \vdash \{\!|\ \mathtt{l}_1 \colon \mathsf{S}_1, \ldots, \mathtt{l}_n \colon \mathsf{S}_n |\!\} <: \{\!|\ \mathtt{l}_1 \colon \mathsf{T}_1, \ldots, \mathtt{l}_m \colon \mathsf{T}_m |\!\}}$$

$$\textit{Object}_{<:} \qquad \frac{\mathcal{C} \vdash \mathtt{RType'} <: \mathtt{RType}}{\mathcal{C} \vdash \mathtt{ObjectType\ RType'} <: \mathtt{ObjectType\ RType}}$$

$$\textit{Subsumption} \qquad \frac{\mathcal{C}, \mathcal{E} \vdash \mathsf{E} \colon \mathsf{S}, \qquad \mathcal{C} \vdash \mathsf{S} <: \mathsf{T}}{\mathcal{C}, \mathcal{E} \vdash \mathsf{E} \colon \mathsf{T}}$$

Figure 10.11 Subtyping rules for $\mathcal{SOOL}$.

Rule *Record* $_{<:}$ indicates that record subtyping includes both width and depth subtyping. Finally, rule *Object* $_{<:}$ specifies that two object types are subtypes if the types of their records of methods are subtypes.

The last rule, *Subsumption*, is actually a type-checking rule rather than a subtyping rule. It states that if an expression has a type, then it can also be assigned any supertype of that type. In Chapter 2 we defined types `ColorCellType` and `CellType` where `ColorCellType` $<:$ `CellType`. If `colorCell` has type `ColorCellType`, then by the *Subsumption* rule it also has type `CellType`.

10.2.4 A type checking example

In the previous section we presented a number of formal type-checking rules for $\mathcal{SOOL}$. In this section we present an example showing how these rules can be used to type-check $\mathcal{SOOL}$ programs. At the end of the section we include a brief discussion of how the type-checking rules can be used to create a type-checking algorithm for $\mathcal{SOOL}$.

We show how to use the type-checking rules by verifying that the program `PointExample` in Figure 10.4 is type-correct. We start with the program as a whole, and with a type constraint system and type environment, $\mathcal{C}_0, \mathcal{E}_0$, that are both empty.

It follows from the *Program* rule in Figure 10.10 that the program as a whole is well-typed iff its block has type `Command`. Looking back at Figure 10.9, we see that a block is type-checked by processing the type and constant definitions to get an enriched type constraint system and type environment. Then the body of the block is type-checked to make sure that it is well-typed. Finally the type of the entire block is the type of the return expression. In this case the return expression is `nop`, which by rule *No Op* has type `Command`. Thus the block as a whole will be well-typed if we can first successfully type-check the constant definitions to obtain an enriched type environment, and then can use that enriched type environment to ensure the body of the block is well-typed.

There are two type definitions in the program. By the type-checking rules, both `PointType` with its definition, and `PtClassType` with its definition are added to $\mathcal{C}_0$, resulting in

$$\begin{aligned}\mathcal{C}_1 = \mathcal{C}_0 \cup \{&\texttt{PointType} = \texttt{ObjectType}\ \{\!|\ldots|\!\},\\ &\texttt{PtClassType} = \texttt{ClassType}\,(\ldots)\}\end{aligned}$$

There are only two constant definitions in the program. The first is of the constant `Point`, which is defined to be a class of type `PtClassType`. We must show that the class expression in Figure 10.12 has type `PtClassType`:

For simplicity, we will use the abbreviations:

$$\begin{aligned}\texttt{PIV} &= \{\!|\ \texttt{x}\colon \texttt{Integer}; \texttt{y}\colon \texttt{Integer}\ |\!\}\\ \texttt{PIVR} &= \{\!|\ \texttt{x}\colon \texttt{Ref Integer}; \texttt{y}\colon \texttt{Ref Integer}\ |\!\}\\ \texttt{PM} &= \{\!|\ \texttt{move}\colon \texttt{Integer} \times \texttt{Integer} \rightarrow \texttt{Command};\\ &\qquad \texttt{getx}\colon \texttt{Void} \rightarrow \texttt{Integer};\\ &\qquad \texttt{gety}\colon \texttt{Void} \rightarrow \texttt{Integer}\ |\!\}\end{aligned}$$

We begin by showing that

$$\mathcal{C}_1, \mathcal{E}_0 \vdash \{\!|\ \texttt{x}\colon \texttt{Integer} = \texttt{0};\ \texttt{y}\colon \texttt{Integer} = \texttt{0}\ |\!\}\colon \texttt{PIV}.$$

```
class ( {|
     x: Integer = 0;
     y: Integer = 0;
  |}, {|
     move: Integer × Integer → Command =
          function(dx: Integer, dy: Integer): Command is
     { self.x := (val self.x) + dx;
       self.y := (val self.y) + dy;
       return nop};

     getx: Void → Integer =
        function(): Integer is
     { return val self.x };

     gety: Void → Integer = function(): Integer is
     { return val self.y };
  |});
```

Figure 10.12 Class definition from `PointExample`.

But this is easy as the *Constant* rule implies that $\mathcal{C}_1, \mathcal{E}_0 \vdash$ `0`: `Integer`.[1] The desired typing of the record of instance variables follows from rule *Record*.

We must next show that the record of methods in `Point` has the type `PM` under the type assumption

$$\mathcal{E}_1 = \mathcal{E}_0 \cup \{\texttt{self}\colon \texttt{SelfType},\ \texttt{close}\colon \texttt{SelfType} \to \texttt{ObjectType PM}\}$$

and type constraint system

$$\mathcal{C}_2 = \mathcal{C}_1 \cup \{\texttt{SelfType} = \texttt{VisObjectType}(\texttt{PIVR}, \texttt{PM})\}.$$

By the *Record* rule, it is sufficient to show that each individual method has the appropriate type. We will only type check the `move` method, as the others are similar, but simpler.

We must show that the function expression,

```
  function(dx: Integer, dy: Integer): Command is {
```

1. We did not list all constants of the language in $\mathcal{EC}$, but instead assume familiar constants (like numerals) have the obvious types.

```
        self.x := (val self.x) + dx;
        self.y := (val self.y) + dy;
        return nop }
```

has type `Integer` $\times$ `Integer` $\rightarrow$ `Command` under assumption $\mathcal{C}_2, \mathcal{E}_1$.

We type check the function using the *Function* rule by checking the body under type constraint $\mathcal{C}_2$ and type assumption

$$\mathcal{E}_2 = \mathcal{E}_1 \cup \{\texttt{dx}: \texttt{Integer}, \texttt{dy}: \texttt{Integer}\}.$$

The first line of the function is an assignment. By rule *Inst Vble* and the assumption on the type of `self` in $\mathcal{E}_2$, it follows that

$$\mathcal{C}_2, \mathcal{E}_2 \vdash \texttt{self.x}: \texttt{Ref Integer}.$$

By the *Value* rule, the type of `val self.x` is `Integer`. The type of `dx` is also `Integer` by the *Identifier* rule, because `dx`: `Integer` is in $\mathcal{E}_2$.

At this point we must cheat a bit since we did not include a type-checking rule for operations like "+". The operation "+" should actually be written as a prefix function `Plus` with type `Integer`$\times$`Integer`$\rightarrow$`Integer`. Then `val self.x + dx` abbreviates `Plus(val self.x, dx)`, which, by the *Application* rule, has type `Integer`. Now the *Assignment* rule allows us to infer from $\mathcal{C}_2, \mathcal{E}_2$ that statement

```
    self.x := (val self.x) + dx;
```

has type `Command`, because the left side has type `Ref Integer`, and the right side has type `Integer`. Showing the other assignment statement has type `Command` is similar. By the rule *StmtList*, the sequence of the two assignments also has type `Command`.

The expression `nop` has type `Command` by rule *No Op*, so by rule *Block* the whole function body has type `Command` when typed with respect to $\mathcal{C}_2, \mathcal{E}_2$. By rule *Function*, the function expression associated with method `move` has type `Integer` $\times$ `Integer` $\rightarrow$ `Command` when typed with $\mathcal{C}_2, \mathcal{E}_1$.

Similar arguments show that the other two method bodies have the desired types using the assumptions in $\mathcal{C}_2, \mathcal{E}_1$. Hence the record of methods has type `PM` under $\mathcal{C}_2, \mathcal{E}_1$. Finally the *Class* rule implies that the entire `Point` body has type `PtClassType`. By rule *ConstDef*,

$$\mathcal{E}_3 = \mathcal{E}_0 \ \cup \ \{\texttt{Point}: \texttt{PtClassType}\}$$

results from processing the definition of `PointClass`, starting with $\mathcal{C}_1, \mathcal{E}_0$.

The other definition in the program is

```
pt: Ref PointType = ref nil;
```

By rule *Nil*, `nil` can be assigned type `PointType` (this is the one place where we use information about context in type checking). By rule *Reference*, `ref nil` has type `Ref PointType`. Therefore by rule *ConstDef*,

$$\mathcal{E}_4 = \mathcal{E}_3 \cup \{\texttt{pt}: \texttt{Ref PointType}\}$$

results from processing the definition of `pt`, starting with $\mathcal{C}_1, \mathcal{E}_3$.

By rules *ConstSec* and *ConstDefList* and the results of processing the two constant definitions above, processing the `const` section of the program results in static type environment $\mathcal{E}_4$. We use this environment along with $\mathcal{C}_1$ to type check the body of the program.

The first assignment statement is well-typed because `new Point` has type `PointType` by the *New* rule. The message send of `pt` $\Leftarrow$ `move` has type `Integer`$\times$`Integer` $\rightarrow$ `Command` by rule *Message*, and, by rule *Application*, `pt` $\Leftarrow$ `move(3,2)` has type `Command`. As before, `nop` has type `Command` by rule *No Op*. Combining rules *StmtList* and *Block*, it follows that the block consisting of the program declarations and code has type `Command`. Thus by rule *Program*, the entire program is well-typed.

As the description above shows, type checking a program is fairly mechanical. All constants are declared with their types, so we need only verify that the definitions are of the appropriate types. The expressions representing class and function definitions include sufficient type information in the expressions that the type can be extracted from the expression. For classes (and subclasses) this information can be extracted from the types of the record of instance variables and the record of methods. The types of individual components of records are included in record expressions, so all necessary type information can be extracted from those subexpressions. Then it is only required to check that the subexpressions have the appropriate types.

Most of the other expressions have types that can be simply composed from the types of their subexpressions. For example, the type of a function application will always be the return type of the function, the type of an expression of the form `val E` is obtained by stripping the `Ref` off of the type of `E`, etc. Identifiers' types can be found in the type environment and constants' types are predefined.

That really leaves only `nil` expressions as problematic. Their types may depend on the context, as with the definition of `pt` in the sample program. There it was necessary to look at the type declared with the constant in order to determine which type `nil` should be assigned (it was `PointType`). We could get around this by creating a new type, `Bottom`, which is a subtype of all object types, and give `nil` type `Bottom`. The *Subsumption* rule could then be used to promote the type of `nil` to whatever object type is needed in the context. We will forgo adding this new type and simply presume enough information is available from context to allow us to infer an appropriate type for `nil`.

10.3 Summary

In this chapter we provided a formal definition of the language $\mathcal{SOOL}$ via a context-free grammar and type-checking rules. As noted earlier, we will usually use an abbreviated version of the language that is closer to existing common object-oriented languages, as it will be easy to algorithmically convert from the simplified language to the stricter formal language.

Simply providing a formal language definition and type-checking rules does not guarantee that a statically typed language is type-safe. For example, how do we know that at run time an expression of some type `T` will actually hold a value of type `T`? How do we know that all message sends at run time will be to objects that have methods that can execute in response to the message?

We will address such questions after we have defined the semantics of $\mathcal{SOOL}$. In the next chapter, we begin our investigation of a translational semantics for $\mathcal{SOOL}$.