Chapter 8

Abstraction, Correspondence, and Qualification

The title of this chapter refers to three language design principles proposed by Tennent. We also study a fourth principle, *parameterization.* Many important language constructs are derived from the principles: subroutines, parameters, block-structuring constructs, and encapsulation mechanisms. Denotational semantics is a useful tool for analyzing the design principles and the constructs that they derive. The language in Figure 8.1 is used as a starting point. We apply each of the principles to the language and study the results.

Figure 8.1

Abstract syntax:

P∈ Program
D∈ Declaration
T∈ Type-structure
C∈ Command
E∈ Expression
L∈ Identifier-L-value
S∈ Subscript
I∈ Identifier
N∈ Numeral

P ::= C.

D ::= D_1;D_2 | **var** I:T

T ::= **nat** | **array** [N_1..N_2] **of** T | **record** D **end**

C ::= C_1;C_2 | L:=E | **begin** D;C **end** | · · ·

E ::= E_1+E_2 | I | IS | N | · · ·

L ::= I | IS

S ::= [E] | .I | [E]S | .IS

8.1 ABSTRACTION

The first design principle is the principle of abstraction. Programmers sometimes use the term *abstraction* for a specification that hides some irrelevant computational details. In Chapter 3 we used the term to describe a function expression of form ($\lambda x.M$). The usage is appropriate, for the expression specifies a function, hiding the details regarding the value x that is used in M. We also gave abstractions names, e.g., $square = (\lambda n.\ n\ times\ n)$. The name enhances the abstraction's worth, for we can refer to the abstraction by mentioning its name, e.g., *square*(*two*).

Most programming languages support the creation of named expressions; a Pascal procedure is an abstraction of a command. We execute the command by mentioning its name. Both a *definition* mechanism and an *invocation* mechanism are necessary. Tennent coined the noun *abstract* to describe a named expression that is invoked by mentioning its name. An abstract has both a *name* and a *body*. If its body is an expression from a syntax domain B, the abstract can be invoked by using its name any place in a program where a B-expression is syntactically legal.

The *principle of abstraction* states that any syntax domain of a language may have definition and invocation mechanisms for abstracts.

A Pascal procedure is an example of a *command abstract.* We might also create expression abstracts, declaration abstracts, type abstracts, and so on. Let ⟦**define** I=V⟧ be an abstract. ⟦I⟧ is the abstract's name and ⟦V⟧ is its body. The denotable value of ⟦I⟧ is **V**⟦V⟧. If **V**⟦V⟧ is a function denotation, then are the arguments to the function provided at the point of definition of the abstract or at the point of its invocation? This is an important question and we study its answer through an example.

We augment the language in Figure 8.1 with definition and invocation mechanisms for command abstracts, which we call *procedures.* The definition mechanism is added to the BNF rule for declarations:

D ::= D_1;D_2 | **var** I:T | **proc** I=C

and the invocation mechanism appears in the rule for commands:

C ::= C_1;C_2 | L:=E | **begin** D;C **end** | I | $\cdots$

Recall that the valuation function used in Chapter 7 for commands has functionality **C:** Command $\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$ $Poststore_\perp$. The denotation of the abstract's body can be any of the following:

1. **C**⟦C⟧ : *Environment* $\rightarrow$ *Store* $\rightarrow$ $Poststore_\perp$: the environment and store that are used with the body are the ones that are active at the point of invocation. This corresponds to *dynamic scoping.*
2. (**C**⟦C⟧*e*) : *Store* $\rightarrow$ $Poststore_\perp$: the environment active at the point of definition is bound to the body, and the store that is used is the one active at the point of invocation. This corresponds to *static scoping.*
3. (**C**⟦C⟧*e s*) $\in$ $Poststore_\perp$: the procedure is completely evaluated at the point of definition, and ⟦I⟧ is bound to a constant $Poststore_\perp$ value. This option is unknown in existing languages for command abstracts.

These three options list the possible *scoping mechanisms* for command abstracts. For now, let us choose option 2 and define the semantic domain of procedures to be $Proc = Store \to Poststore_{\perp}$. The denotations of procedure identifiers come from *Proc*:

$$Denotable\text{-}value = Natlocn + Array + Record + Proc$$

The semantic equations for procedure definition and invocation are:

$$\mathbf{D}[\![\textbf{proc I=C}]\!] = \lambda e.\lambda s.\ ((updateenv[\![\mathrm{I}]\!]\ \text{in}Proc(\mathbf{C}[\![\mathrm{C}]\!]e)\ e),\ (return\ s))$$

$$\mathbf{C}[\![\mathrm{I}]\!] = \lambda e.\lambda s.\ \text{cases}\ (accessenv[\![\mathrm{I}]\!]\ e)\ \text{of}$$
$$\text{is}Natlocn(l) \to (signalerr\ s)$$
$$\cdots$$
$$[]\ \text{is}Proc(q) \to (q\ s)\ \ \text{end}$$

(Recall that **D**: Declaration $\to$ *Environment* $\to$ *Store* $\to$ (*Environment* $\times$ *Poststore*).) Since we chose static scoping, (**C**[[C]]*e*) is bound to [[I]] in the environment, and the store is supplied at invocation-time. The definitions of the other two options are left as exercises.

Similar issues arise for expression abstracts (*functions*). For:

D ::= $\cdots$ | **fcn** I=E
E ::= E_1+E_2 | $\cdots$ | I

an ALGOL68-like constant definition results when the environment and store are bound at the point of definition. If just the environment is bound at the point of definition, a FORTRAN-style function results. If neither are bound at the point of definition, a text macro definition results.

An interesting hybrid of procedure and function is the *function procedure*, which is a command abstract that is invoked by an Expression-typed identifier. This construct is found in Pascal. The function procedure must return an expressible value as its result. A possible syntax for a function procedure is:

D ::= $\cdots$ | **fcnproc** I= C **resultis** E

Its inclusion causes a profound change in the semantics of expressions, for an expression can now alter the value of the store. Further, an invoked function procedure might not terminate. The valuation function for expressions must take the form:

$$\mathbf{E}: \text{Expression} \to Environment \to Store \to (Expressible\text{-}value \times Poststore)_{\perp}$$

If we use static scoping, the equations for definition and invocation are:

$$\mathbf{D}[\![\textbf{fcnproc I= C resultis E}]\!] = \lambda e.\lambda s.\ ((updateenv[\![\mathrm{I}]\!]$$
$$\text{in}Fcn\text{-}proc((check(\mathbf{E}[\![\mathrm{E}]\!]e)) \circ (\mathbf{C}[\![\mathrm{C}]\!]e))\ \ e),\ \ (return\ s))$$
$$\text{where}\ check: (Store \to (Expressible\text{-}value \times Poststore)_{\perp})$$
$$\to (Poststore \to (Expressible\text{-}value \times Poststore)_{\perp})$$
traps errors and nontermination

$$\begin{aligned}&\mathbf{E}[\![\mathrm{I}]\!] = \lambda e.\lambda s.\ \text{cases } (\mathit{accessenv}[\![\mathrm{I}]\!]\ e) \text{ of}\\ &\qquad\qquad \text{is}\mathit{Natlocn}(l) \rightarrow ((\mathit{access}\ l\ s),\ (\mathit{return}\ s))\\ &\qquad\qquad \cdots\\ &\qquad\qquad [\,]\ \text{is}\mathit{Fcn\text{-}proc}(f) \rightarrow (f\ s)\ \ \text{end}\end{aligned}$$

All of the other semantic equations for the **E** function must be revised to cope with the complications arising from side effects and nontermination. This is left as an exercise.

Declaration abstractions follow the pattern seen thus far for commands and expressions. The syntax is:

$$\mathrm{D} ::= \mathrm{D}_1;\mathrm{D}_2 \mid \textbf{var}\ \mathrm{I}{:}\mathrm{T} \mid \cdots \mid \textbf{module}\ \mathrm{I}{=}\mathrm{D} \mid \mathrm{I}$$

We call the new construct a *module*. The invocation of a module activates the declarations in the module's body. Since there is no renaming of declarations, multiple invocations of the same module in a block cause a redefinition error.

We also have type abstracts:

$$\begin{aligned}&\mathrm{D} ::= \cdots \mid \textbf{type}\ \mathrm{I}{=}\mathrm{T}\\ &\mathrm{T} ::= \textbf{nat} \mid \cdots \mid \mathrm{I}\end{aligned}$$

The definition and invocation of type abstracts follow the usual pattern:

$$\begin{aligned}&\mathbf{D}[\![\textbf{type}\ \mathrm{I}{=}\mathrm{T}]\!] = \lambda e.\lambda s.\ ((\mathit{updateenv}[\![\mathrm{I}]\!]\ \ \text{in}\mathit{Type}(\mathbf{T}[\![\mathrm{T}]\!]e)\ e),\ (\mathit{return}\ s))\\ &\mathbf{T} : \text{Type-structure} \rightarrow \mathit{Environment} \rightarrow \mathit{Store} \rightarrow (\mathit{Denotable\text{-}value} \times \mathit{Poststore})\\ &\mathbf{T}[\![\mathrm{I}]\!] = \lambda e.\lambda s.\ \text{cases } (\mathit{accessenv}[\![\mathrm{I}]\!]\ e) \text{ of}\\ &\qquad\qquad \text{is}\mathit{Natlocn}(l) \rightarrow (\text{in}\mathit{Errvalue}(),\ (\mathit{signalerr}\ s))\\ &\qquad\qquad \cdots\\ &\qquad\qquad [\,]\ \text{is}\mathit{Type}(v) \rightarrow (v\ s)\ \ \text{end}\end{aligned}$$

An issue raised by type abstraction is: when are two variables equivalent in type? There are two possible answers. The first, *structure equivalence*, states that two variables are type-equivalent if they have identical storage structures. Structure equivalence is used in ALGOL68. The second, *occurrence equivalence*, also known as *name equivalence*, states that two variables are type-equivalent if they are defined with the same occurrence of a type expression. A version of occurrence equivalence is used in Pascal. Consider these declarations:

```
type M = nat;
type N = array [1..3] of M;
var A: nat;
var B: M;
var C: M;
var D: N;
var E: array [2..4] of nat
```

Variables A and B are structure-equivalent but not occurrence-equivalent, because they are defined with different occurrences of type expressions, A with **nat** and B with M. Variables B and C are both structure- and occurrence-equivalent; C and D are neither. Variables D and E are clearly not occurrence-equivalent, but are they structure-equivalent? The two have the same structure in the *store* but have unequal structures, due to different range bounds, in the *environment.* The question has no best answer. A similar problem exists for two variable record structures that differ only in their components' selector names or in the ordering of their components. The semantics of declaration and assignment given in Chapter 7 naturally enforces structure equivalence on types.

If we wish to define the semantics of these variants of type-checking, we must add more information to the denotable values. For arrays, the domain

$$Array = (Index \rightarrow Denotable\text{-}value) \times Index \times Index$$

is inadequate for occurrence equivalence checking and barely adequate for structure equivalence checking. (Why?) A formal description of either kind of equivalence checking is not simple, and the complexity found in the semantic definitions is mirrored in their implementations. The area is still a subject of active research.

8.1.1 Recursive Bindings

The semantics of a recursively defined abstract is straightforward; for the hypothetical abstract:

D ::= ··· | **rec abs** I=M | ···

where **abs** could be **proc**, **fcn**, **class**, or whatever, a statically scoped, recursive version is:

$$\mathbf{D}[\![\textbf{rec abs } I{=}M]\!] = \lambda e.\lambda s.\ (e', (return\ s))$$
$$\text{where } e' = (updateenv[\![I]\!]\ \text{in} M(\mathbf{M}[\![M]\!]e')\ e)$$

As we saw in Chapter 7, the recursively defined environment e' causes a reference to identifier [[I]] in [[M]] to be resolved with e'. This produces a recursive invocation.

What sort of abstracts make good use of recursive bindings? Certainly procedures do. Perhaps the most important aspect of recursive invocations is the means for terminating them. In this regard, the **if-then-else** command serves well, for it makes it possible to choose whether to continue the recursive invocations or not. We can increase the utility of recursive expression and type abstracts by adding conditional constructs to their domains:

E ::= E_1+E_2 | ··· | **if** E_1 **then** E_2 **else** E_3
T ::= **nat** | ··· | **if** E **then** T_1 **else** T_2

Some applications are given in the exercises.

8.2 PARAMETERIZATION

Abstracts usually carry parameters, which are dummy identifiers that are replaced by values when the abstract is invoked. The dummy identifiers are the *formal parameters* and the expressions that replace them are the *actual parameters.* If a formal parameter ⟦I⟧ is used in an abstract's body in positions where a B-construct is syntactically allowed, then the actual parameter bound to ⟦I⟧ must be an expression from the B syntax domain. Abstracts may have expression parameters, command parameters, type parameters, and so on.

The *principle of parameterization* states that a formal parameter to an abstract may be from any syntax domain. The denotation of an abstract's body ⟦V⟧ parameterized on identifiers $⟦I_1⟧, \cdots, ⟦I_n⟧$ is a function of form $(\lambda p_1. \cdots .\lambda p_n.\ \mathbf{V}⟦V⟧ \cdots)$. What are the denotations of the actual parameters? There are a number of options, and an example is the best means for study.

All parameterized abstracts in this section use only one parameter. Consider a procedure, parameterized on a member of the Expression domain, defined by the syntax:

$$\begin{aligned} &D ::= \cdots \mid \textbf{proc}\ I_1(I_2)=C \\ &C ::= C_1;C_2 \mid \cdots \mid I(E) \end{aligned}$$

If the abstract is statically scoped, then the domain of procedures is $Proc = Param \to Store \to Poststore_{\perp}$. The semantics of the abstract is:

$$\begin{aligned} &\mathbf{D}⟦\textbf{proc}\ I_1(I_2)=C⟧ = \lambda e.\lambda s.\ ((updateenv⟦I_1⟧ \\ &\qquad \text{in} Proc(\lambda a.\ \mathbf{C}⟦C⟧(updateenv⟦I_2⟧\ a\ e))\ \ e),\ \ (return\ s)). \end{aligned}$$

$$\begin{aligned} &\mathbf{C}⟦I(E)⟧ = \lambda e.\lambda s.\ \text{cases}\ (accessenv⟦I⟧\ e)\ \text{of} \\ &\qquad\qquad \text{is}Natlocn(l) \to (signalerr\ s) \\ &\qquad\qquad \cdots \\ &\qquad\qquad [\,]\ \text{is}Proc(q) \to (q\ (\cdots \mathbf{E}⟦E⟧ \cdots)\ s) \\ &\qquad\qquad \text{end} \end{aligned}$$

The expression $(\cdots \mathbf{E}⟦E⟧ \cdots)$ represents the denotation of the actual parameter, a member of domain *Param.* Recall that **E**: Expression $\to$ *Environment* $\to$ *Store* $\to$ *Expressible-value* is the functionality of the valuation function for expressions. The options for the denotation of the actual parameter are:

1. $(\mathbf{E}⟦E⟧e\ s) \in$ *Expressible-value*: the actual parameter is evaluated with the environment and store active at the point of invocation. This is implemented as *call-by-value.*
2. $(\mathbf{E}⟦E⟧e)$: *Store* $\to$ *Expressible-value*: the actual parameter is given the invocation environment, but it uses the stores active at the occurrences of its corresponding formal parameter in the procedure body. This is implemented as ALGOL60-style *call-by-name.*
3. **E**⟦E⟧: *Environment* $\to$ *Store* $\to$ *Expressible-value*: the actual parameter uses the environment and the store active at the occurrences of its corresponding formal parameter in the procedure. This is implemented as *call-by-text.*
4. A fourth option used in some languages is to take the actual parameter domain *Param* to be *Location.* This is implemented as *call-by-reference.* Call-by-reference transmission

presents a problem when a nonvariable expression is used as an actual parameter. Since the denotation must be a location value, the usual strategy is to allocate a new location that holds the expressible value of the parameter. This problem is a result of improper language design. A solution is to make an explicit distinction in the language's syntax definition between Identifier L-values and Identifier R-values, as we did in Figure 2.1. A call-by-reference parameter is not an expressible value, but the denotation of an L-value construct, and identifiers that are Expression parameters are R-value constructs.

We conducted the above analysis assuming that expression evaluation always terminated; that is, *Expressible-value* had no $\underline{\perp}$ element. When nontermination of actual parameter evaluation is a possibility, then each of the four options mentioned above may evaluate their actual parameter expressions in two ways:

1. *Immediate evaluation*: the value of the actual parameter is calculated before its binding to the formal parameter. This is described by making *a*'s binding strict in:

$$\mathbf{D}[\![\mathbf{proc}\ \mathrm{I}_1(\mathrm{I}_2)=\mathrm{C}]\!] = \lambda e.\lambda s.\ ((\mathit{updateenv}[\![\mathrm{I}_1]\!]\ \mathrm{in}\mathit{Proc}(\underline{\lambda} a.\ \mathbf{C}[\![\mathrm{C}]\!](\mathit{updateenv}[\![\mathrm{I}_2]\!]\ a\ e))\ e),\ (\mathit{return}\ s))$$

2. *Delayed evaluation*: the value of the actual parameter need be calculated only upon its use in the body of the procedure. In this case, the semantic equation for procedure definition is left in its original form— the value, whether it be proper or improper, is bound into the procedure's environment.

The term *call-by-value* is normally used to mean immediate evaluation to an expressible value, while *call-by-need* and *lazy evaluation* are used to mean delayed evaluation to an expressible value. (The difference between the latter two is that, once an evaluation of an argument does proceed, call-by-need is required to finish it, whereas lazy evaluation need only evaluate the argument to the point that the required subpart of the argument is produced.) Most applications of options 2 through 4 use immediate evaluation. The parameter domain and strictness questions can be raised for all syntactic domains of actual parameters. You should consider these issues for command, declaration, and type parameters to procedures and functions.

One of the more interesting parameterized abstracts is the parameterized type expression. For syntax:

D ::= ··· | **type** $\mathrm{I}_1(\mathrm{I}_2)$=T | ···
T ::= ··· | I(T) | ···

type structures such as:

```
type STACKOF(T) = record
                    var ST: array [1..k] of T;
                    var TOP: nat
                  end
```

can be written. An invocation such as **var** X: STACKOF(**nat**) allocates storage for the two components of the record: X.ST refers to an array of *k* number variables, and X.TOP refers to

a number variable. The semantics of parameterized type abstractions is left as an exercise.

8.2.1 Polymorphism and Typing

An operation is *polymorphic* if its argument can be from more than one semantic domain. The answer it produces is dependent upon the domains of its arguments. As an example, a general purpose addition operation might produce an integer sum from two integer arguments and a rational sum from two rational arguments. This operation might be assigned functionality:

$$(Integer \times Integer) \cup (Rational \times Rational) \rightarrow Integer \cup Rational$$

Unfortunately, the dependence of the codomain on the domain isn't clearly stated in this description. The graph of the operation is the union of the integer addition and rational addition operations. Polymorphic operations do not fit cleanly into the domain theory of Chapter 3, and our semantic notation does not include them.

Polymorphism does appear in general purpose programming languages. Strachey distinguished between two kinds: *ad hoc polymorphism* (also called *overloading*) and *parametric polymorphism*. An ad hoc polymorphic operator ''behaves differently'' for arguments of different types, whereas a parametric polymorphic operation ''behaves the same'' for all types. (We won't attempt more specific definitions because the concepts have proved notoriously difficult to formalize.) In Pascal, a typed language, the + symbol is overloaded, because it performs integer addition, floating point addition, and set union, all unrelated operations. A Pascal compiler determines the context in which the operator appears and associates a specific meaning with +. In contrast, the **hd** operator in Edinburgh ML is parametric. It can extract the head integer from a list of integers, the head character from a list of characters, and, in general, the head α from an α-list. **hd** is a general purpose function, and it is implemented as a general purpose operation. Regardless of the type of argument, the same structural manipulation of a list is performed.

The denotational semantics of an overloaded operator is straightforward to express. Here is a semantic equation for the Pascal addition expression:

$$\begin{aligned}
&\mathbf{E}[\![\mathrm{E}_1 + \mathrm{E}_2]\!] = \lambda e.\lambda s.\ \text{cases } (\mathbf{E}[\![\mathrm{E}_1]\!] e\, s) \text{ of}\\
&\quad \text{is}Nat(n_1) \rightarrow (\text{cases } (\mathbf{E}[\![\mathrm{E}_2]\!] e\, s) \text{ of}\\
&\qquad\qquad \text{is}Nat(n_2) \rightarrow \text{in}Nat(n_1\ plus\ n_2)\\
&\qquad\qquad \cdots \ \text{end})\\
&\quad \cdots\\
&\quad [] \ \text{is}Rat(r_1) \rightarrow (\text{cases } (\mathbf{E}[\![\mathrm{E}_2]\!] e\, s) \text{ of}\\
&\qquad\qquad \cdots\\
&\qquad\qquad [] \ \text{is}Rat(r_2) \rightarrow \text{in}Rat(r_1\ addrat\ r_2)\\
&\qquad\qquad \cdots \ \text{end})\\
&\quad \cdots\\
&\quad [] \ \text{is}Set(t_1) \rightarrow \cdots \ \text{in}Set(t_1\ union\ t_2) \cdots\\
&\quad \cdots \ \text{end}
\end{aligned}$$

A pre-execution analysis like that performed in Figure 7.3 can determine which of *plus*, *addrat*, or *union* is the denotation of the +.

Parametric polymorphic operations are more difficult to handle; we give one method for doing so. Consider the **hd** operator again. In ML, numbers, lists, tuples, sums, function spaces, and the like, are all data types. Its expressible value domain balloons to:

$$Exprval = (Nat + Exprval^* + (Exprval \times Exprval) + (Exprval + Exprval) + (Exprval \to Exprval) + Errvalue)_\perp$$

The **hd** operator manipulates an expressible value list:

$$\begin{aligned}&\mathbf{E}[\![\mathbf{hd}\ \mathrm{E}]\!] = \lambda e.\ \text{let } x = \mathbf{E}[\![\mathrm{E}]\!]e \text{ in cases } x \text{ of}\\ &\qquad \text{is}Nat(n) \to \text{in}Errvalue()\\ &\qquad [] \ \text{is}Exprval^*(l) \to hd\ l\\ &\qquad [] \ \cdots \text{ end}\end{aligned}$$

The disadvantage of this formulation is that *Nat*-lists such as (*two cons one cons nil*) become in$Exprval^*$(inNat(*two*) *cons* inNat(*one*) *cons nil*). We would prefer that **hd** operate directly upon *Nat*-lists, *Nat*×*Nat*-lists, and the like, but both theoretical problems (mathematically, the **hd** operation literally becomes too ''large'' to be well defined) and notational problems (try to define **hd** in the existing semantic notation) arise. The *real* problem lies in our version of domain theory. Our domains live in a rigid hierarchy, and there exists no ''universal domain'' that includes all the others as subdomains. If a universal domain *U* did exist, we could define a single operation $hd_U : U \to U$ that maps those elements of *U* that ''look like'' α-lists to elements in *U* that ''look like'' α-values. Then $\mathbf{E}[\![\mathbf{hd}\ \mathrm{E}]\!] = \lambda e.\ hd_U(\mathbf{E}[\![\mathrm{E}]\!]e)$. A number of researchers, most notably McCracken and Reynolds, have developed domain theories for universal domains and parametric polymorphism.

We next consider polymorphic parameterized abstracts. The parameterized abstracts in the previous section are untyped— no restrictions (beyond syntax domain compatibility) are placed on the formal and actual parameters. This is the version of abstraction used in untyped languages such as BCPL and LISP. If the untyped version of parameterized abstract is used in a typed language such as Pascal, the abstraction acts polymorphically. Consider a command abstract whose denotation lies in the domain $Proc = Expressible\text{-}value \to Store \to Poststore_\perp$. A procedure's actual parameter can be an integer, a truth value, an array, or whatever else is a legal expressible value.

However, a typed programming language like Pascal requires that formal parameters be labeled with type expressions. The expression acts as a precondition or guard: only actual parameters whose type structures match the formal parameter's are allowed as arguments. The advantages of typing— pre-execution type equivalence verification, increased user understanding of abstracts, and efficient execution— are well known.

We use the syntax

$$\begin{aligned}&\mathrm{D} ::= \cdots \mid \mathbf{proc}\ \mathrm{I}_1(\mathrm{I}_2{:}\mathrm{T}){=}\mathrm{C}\\ &\mathrm{C} ::= \cdots \mid \mathrm{I(E)} \mid \cdots\end{aligned}$$

for procedures that receive actual parameters from the Expression syntax domain. The value bound to $[\![\mathrm{I}_2]\!]$ must have type structure $[\![\mathrm{T}]\!]$.

The semantics of typed parameters can be handled in two ways: (1) the type information can guard entry to the abstract at invocation; (2) the abstract's denotation is restricted at definition to a function whose domain is exactly that specified by the type. To define the first version, we use a valuation function:

$$\mathbf{T}_{\text{'}}\text{: Type-structure} \to \textit{Environment} \to \textit{Expressible-value} \to (\textit{Expressible-value} + \textit{Errvalue})$$

such that ($\mathbf{T}_{\text{'}}[\![\text{T}]\!]e\ x$) determines whether or not x has data type $[\![\text{T}]\!]$. If it does, the result is in*Expressible-value*(x); otherwise it is in*Errvalue*(). $\mathbf{T}_{\text{'}}$ is a type-equivalence checker. The semantics of statically scoped procedure declaration is:

$$\begin{aligned}&\mathbf{D}[\![\textbf{proc}\ \text{I}_1(\text{I}_2\text{:T})\text{=C}]\!] = \lambda e.\lambda s.\ ((\textit{updateenv}[\![\text{I}_1]\!]\\ &\quad \text{in}\textit{Proc}(\lambda x.\ \text{cases}\ (\mathbf{T}_{\text{'}}[\![\text{T}]\!]e\ x)\ \text{of}\\ &\qquad\qquad \text{is}\textit{Expressible-value}(x_{\text{'}}) \to \mathbf{C}[\![\text{C}]\!](\textit{updateenv}[\![\text{I}_2]\!]\ x_{\text{'}}\ e)\\ &\qquad\qquad [\,]\ \text{is}\textit{Errvalue}() \to \textit{signalerr}\ \ \text{end})\\ &\quad e),\ (\textit{return}\ s))\end{aligned}$$

The second version fragments the *Proc* domain by making it into a family of procedure domains:

$$\begin{aligned}&\textit{Nat-proc} = \textit{Nat} \to \textit{Store} \to \textit{Poststore}_{\perp}\\ &\textit{Array-proc} = \textit{Array} \to \textit{Store} \to \textit{Poststore}_{\perp}\\ &\textit{Record-proc} = \textit{Record} \to \textit{Store} \to \textit{Poststore}_{\perp}\end{aligned}$$

For simplicity, we give only three kinds of parameter domains. (You are given the problem of formulating a complete hierarchy of domains for Pascal.) Another version of the $\mathbf{T}_{\text{'}}$ function is needed. First, we define:

$$\begin{aligned}&\textit{Type-tag} = \textit{Nat-tag} + \textit{Array-tag} + \textit{Record-tag} + \textit{Err-tag}\\ &\quad \text{where}\ \textit{Nat-tag} = \textit{Array-tag} = \textit{Record-tag} = \textit{Err-tag} = \textit{Unit}\end{aligned}$$

Each non-*Err-tag* corresponds to one of the parameter domains. The valuation function $\mathbf{T}_{\text{'}}$: Type-structure $\to$ *Environment* $\to$ *Type-tag* maps the formal parameter's type information to a type tag value in the obvious fashion. A declaration of a typed procedure has semantics:

$$\begin{aligned}&\mathbf{D}[\![\textbf{proc}\ \text{I}_1(\text{I}_2\text{:T})\text{=C}]\!] = \lambda e.\lambda s.\ \text{cases}\ (\mathbf{T}_{\text{'}}[\![\text{T}]\!]e)\ \text{of}\\ &\qquad \text{is}\textit{Nat-tag}() \to ((\textit{updateenv}[\![\text{I}_1]\!]\ \text{in}\textit{Nat-proc}(\lambda n.\ \mathbf{C}[\![\text{C}]\!](\textit{updateenv}[\![\text{I}_2]\!]\\ &\qquad\qquad \text{in}\textit{Nat}(n)\ e))\ e),\ (\textit{return}\ s))\\ &\qquad \text{is}\textit{Array-tag}() \to\\ &\qquad\quad ((\textit{updateenv}[\![\text{I}_1]\!]\ \text{in}\textit{Array-proc}(\lambda a.\ \mathbf{C}[\![\text{C}]\!](\textit{updateenv}[\![\text{I}_2]\!]\\ &\qquad\qquad \text{in}\textit{Array}(a)\ e))\ e),\ (\textit{return}\ s))\\ &\qquad \text{is}\textit{Record-tag}() \to\ \text{(similar to above)}\\ &\qquad \text{is}\textit{Err-tag}() \to (e,\ (\textit{signalerr}\ s))\\ &\quad \text{end}\end{aligned}$$

This semantics explicity restricts the abstract's argument domain by selecting a particular function at the point of definition. More specific information exists about the data type of the parameter, and a more thorough pre-execution analysis can be performed.

Both of these two description methods have drawbacks. You are left with the problem of finding a better solution to parameter-type enforcement.

8.3 CORRESPONDENCE

The *principle of correspondence* is simply stated: for any parameter binding mechanism, there may exist a corresponding definition mechanism, and vice versa. That is, if elements from a domain D may be denotable values of formal parameter identifiers, then elements from D may be denotable values of declared identifiers, and vice versa. Since an environment is a map from identifiers to denotable values, and a declared identifier is used no differently than a parameter identifier, the correspondence principle makes full use of the semantic domains.

The correspondence between the two forms of binding becomes clear when their semantic equations are compared. Consider once again a statically scoped command abstract with a single parameter; let D be the domain of parameter denotations. Equations for definition and invocation read:

$$\mathbf{D}[\![\textbf{proc } \mathrm{I}_1(\mathrm{I}_2)=\mathrm{C}]\!] = \lambda e.\lambda s.\,((\mathit{updateenv}[\![\mathrm{I}_1]\!] \text{ in}\mathit{Proc}(\lambda d.\,\mathbf{C}[\![\mathrm{C}]\!](\mathit{updateenv}[\![\mathrm{I}_2]\!] \text{ in}D(d)\ \ e))\ \ e),\ \ (\mathit{return}\ s))$$

$$\mathbf{C}[\![\mathrm{I}(\mathrm{M})]\!] = \lambda e.\lambda s.\ \text{cases } (\mathit{accessenv}[\![\mathrm{I}]\!]\ e) \text{ of}$$
$$\cdots$$
$$[]\ \text{is}\mathit{Proc}(q) \rightarrow q(\cdots \mathbf{M}[\![\mathrm{M}]\!] \cdots)\ s$$
$$\cdots \ \text{end}$$

We see that $\text{in}D(\cdots \mathbf{M}[\![\mathrm{M}]\!] \cdots)$ is bound to $[\![\mathrm{I}_2]\!]$ in $[\![\mathrm{C}]\!]$'s environment. We can build a definition construct with similar semantics:

$$\mathbf{D}[\![\textbf{define } \mathrm{I}=\mathrm{M}]\!] = \lambda e.\lambda s.\,((\mathit{updateenv}[\![\mathrm{I}]\!] \text{ in}D(\cdots \mathbf{M}[\![\mathrm{M}]\!] \cdots)\ \ e),\ (\mathit{return}\ s))$$

Thus, $\mathbf{C}[\![\textbf{begin proc } \mathrm{I}(\mathrm{I}_1)=\mathrm{C};\ \mathrm{I}(\mathrm{M})\ \textbf{end}]\!] = \mathbf{C}[\![\textbf{begin define } \mathrm{I}_1=\mathrm{M};\ \mathrm{C}\ \textbf{end}]\!]$ for phrases $[\![\mathrm{M}]\!]$ and $[\![\mathrm{C}]\!]$ that contain no free occurrences of $[\![\mathrm{I}]\!]$. The questions we raised in Section 8.2 regarding the domains of formal parameters now apply to declared identifiers as well.

The correspondence principle may also be practiced in the other direction. When we consider the definition form for variables:

$$\mathbf{D}[\![\textbf{var } \mathrm{I}:\mathrm{T}]\!] = \lambda e.\lambda s.\ \text{let } (d, p) = (\mathbf{T}[\![\mathrm{T}]\!] e\ s) \text{ in } ((\mathit{updateenv}[\![\mathrm{I}]\!]\ d\ e),\ p)$$

we see that the value bound to $[\![\mathrm{I}]\!]$ is an activated type denotation; that is, a reference to newly allocated storage, rather than an expressible value. (Perhaps we should write variable definitions as $[\![\mathrm{I} = \textbf{ref } \mathrm{T}]\!]$.) The corresponding binding mechanism is:

$$\mathbf{D}[\![\textbf{proc } I_1(I_2)=C]\!] = \lambda e.\lambda s.\ ((\mathit{updateenv}[\![I_1]\!]$$
$$\quad \text{in}\mathit{Proc}(\lambda d.\ \mathbf{C}[\![C]\!](\mathit{updateenv}[\![I_2]\!]\ d\ e))\ \ e,\ \ (\mathit{return}\ s))$$

$$\mathbf{C}[\![I(T)]\!] = \lambda e.\lambda s.\ \text{cases}\ (\mathit{accessenv}[\![I]\!]\ e)\ \text{of}$$
$$\cdots$$
$$[]\ \text{is}\mathit{Proc}(q) \rightarrow \text{let}\ (d, p) = (\mathbf{T}[\![T]\!] e\ s)\ \text{in}\ (\mathit{check}(q\ d))(p)$$
$$\cdots\ \ \text{end}$$

Storage for a data object of type [[T]] is allocated when the procedure is invoked, and a reference to the storage is the denotation bound to the parameter. This form of parameter transmission allocates local variables for a procedure. You should study the differences between the version of **C**[[I(T)]] given above and the version induced by the correspondence principle from [[**type** I=T]].

The principle of parameterization can be derived from the principles of abstraction and correspondence by first uniformly generating all possible abstraction forms and then deriving the parameter forms corresponding to the definitions.

8.4 QUALIFICATION

The *principle of qualification* is that every syntax domain may have a block construct for admitting local declarations. The language of Figure 8.1 already has a block construct in the Command domain. Blocks for the other domains take on similar forms:

E ::= $\cdots$ | **begin** D **within** E **end** | $\cdots$
D ::= $\cdots$ | **begin** D_1 **within** D_2 **end** | $\cdots$
T ::= $\cdots$ | **begin** D **within** T **end** | $\cdots$

and so on. For a syntax domain M, the semantics of an M-block [[**begin** D **within** M **end**]] is **M**[[M]] with an environment augmented by the definitions [[D]]. The definitions' scope extends no further than [[M]]. Assuming the usual static scoping, we state the semantics of the M-block as:

$$\mathbf{M}: M \rightarrow \mathit{Environment} \rightarrow \mathit{Store} \rightarrow (\mathit{M} \times \mathit{Poststore})$$
$$\mathbf{M}[\![\textbf{begin } D \textbf{ within } M \textbf{ end}]\!] = \lambda e.\lambda s.\ \text{let}\ (e', p) = (\mathbf{D}[\![D]\!] e\ s)$$
$$\text{in}\ (\mathit{check}(\mathbf{M}[\![M]\!] e'))(p)$$

This format applies to Command blocks, Expression blocks, Type blocks, and so on. A technical problem arises for Declaration blocks. The scope of local definitions [[D_1]] of block [[**begin** D_1 **within** D_2 **end**]] should extend only as far as [[D_2]], but the **D** valuation function defined in Figure 7.2 processes its environment argument as if it were a store— the additions to the environment are retained beyond the scope of the declaration block. A solution is to make the denotation of a declaration be a list of binding pairs:

$$\mathbf{D}: \text{Declaration} \rightarrow \textit{Environment} \rightarrow \textit{Store} \rightarrow ((\text{Identifier} \times \textit{Denotable-value})^* \times \textit{Poststore})$$

Thus, (**D**[[D]]*e s*) denotes the bindings (and the post-store) defined by [[D]] and not the environment that results when those bindings are added to *e*. **D**'s definition and the semantics of declaration blocks are left as exercises.

A number of useful constructions result from the the qualification principle. For example, a Declaration block used within a Declaration abstract creates a Modula-style **module.** Here is an example that models a natural number-containing stack:

```
module STACK-OF-NAT =
    begin
        var ST: array [1..k] of nat;
        var TOP: nat
    within
        proc PUSH(I: nat) = if TOP=k then skip
            else (TOP:=TOP+1; ST[TOP]:=I);
        proc POP = if TOP=0 then skip else TOP:=TOP−1;
        fcn TOP = if TOP=0 then error else ST[TOP];
        proc INITIALIZE = TOP:=0
    end
```

The declaration **var** STACK-OF-NAT creates procedures PUSH, POP, TOP, INITIALIZE, and function TOP. The variables ST and TOP are local definitions that are hidden from outside access.

Type blocks are also useful. First, recall that the syntax of a record structure is:

T ::= ··· | **record** D **end** | ···

Since the body of a record is a declaration, records of variables, procedures, functions, or whatever are allowed. These make semantic sense as well, for *Record* = *Identifier* → *Denotable-value* and *Denotable-value* = (*Natlocn* + *Array* + *Record* + *Proc* + ···)$_\perp$. Since Type blocks allow local variables to a type structure, we can create a SIMULA-style *class*. Here is an example of a type definition for a stack class parameterized on the element type:

```
type STACK-OF(X) =
    begin
        var ST: array[1..k] of X;
        var TOP: nat
    within record
        proc PUSH(I:X) = ··· (as before)
        proc POP = ···
        fcn TOP = ···
        proc INITIALIZE = ···
```

```
    end
end
```

A definition **var** A: STACK-OF(**nat**) creates a record with components A.PUSH, A.POP, A.TOP, and A.INITIALIZE. One technical point must be resolved: the type parameter X has two roles in the definition. It induces storage allocation in the definition **var** ST: **array** [1..k] **of** X and it does type equivalence checking in **proc** PUSH(I:X). Both the **T** and $\mathbf{T}_\iota$ valuation functions are needed.

8.5 ORTHOGONALITY

The introduction to this chapter stated that the abstraction, parameterization, correspondence, and qualification principles were tools for programming language design. Any programming language can be uniformly extended along the lines suggested by the principles to produce a host of user conveniences. The design principles encourage the development of an *orthogonal* language.

What is orthogonality? A precise definition is difficult to produce, but languages that are called *orthogonal* tend to have a small number of core concepts and a set of ways of uniformly combining these concepts. The semantics of the combinations are uniform; no special restrictions exist for specific instances of combinations. Here are two examples. First, the syntax domain Expression is an example of a core concept. An expression should have equal rights and uniform semantics in all contexts where it can be used. In ALGOL68, any legal member of Expression may be used as an index for an array; e.g., ''A[4+(F(X)-1)]'' is acceptable. The semantics of the expression interacts uniformly with the semantics of the array-indexing operation, regardless of what the expression is. This does not hold in FORTRAN IV, where there are restrictions on which forms of Expression can be used as indexes— the expression $4 + (F(X) - 1)$ is too complex to be a FORTRAN array index. The semantics of expressions is not uniformly handled by the FORTRAN-indexing operation. A second example is the specification of a result type for a function. In Pascal, only values from the scalar types can be results from function procedures. In contrast, ML allows a function to return a value from any legal type whatsoever.

Orthogonality reduces the mental overhead for understanding a language. Because it lacks special cases and restrictions, an orthogonal language definition is smaller and its implementation can be organized to take advantage of the uniformity of definition. The principles introduced in this chapter provide a methodology for introducing orthogonal binding concepts. In general, the denotational semantics method encourages the orthogonal design of a language. A valuation function assigns a uniform meaning to a construct regardless of its context. Further, the semantic domains and function notation encourage uniform application of concepts— if *some* members of a semantic domain are processed by an operation, then arrangements must be made to handle *all* of them. The compactness of a language's denotational definition can be taken as a measure of the degree of the language's orthogonality.

SUGGESTED READINGS

Semantics of abstraction and parameterization: Berry 1981; Gordon 1979; Plotkin 1975; Tennent 1977b

Semantics of qualification and correspondence: Ganzinger 1983; Goguen & Parsaye-Ghomi 1981; Tennent 1977b, 1981

Polymorphism & typing: Demers, Donohue, & Skinner 1978; Kahn, MacQueen, & Plotkin 1984; McCracken 1984; MacQueen & Sethi 1982; Reynolds 1974, 1981, 1985

EXERCISES

1. Describe the different scoping mechanisms possible for the Declaration, Type-structure, Identifier-L-value, and Subscript abstracts derived from Figure 8.1. Write the semantic equations for the various scoping mechanisms and give examples of use of each of the abstracts.

2. Consider the interaction of differently scoped abstracts:

   a. Which forms of scoping of expression abstracts are compatible with statically scoped command abstracts? With dynamically scoped command abstracts?
   b. Repeat part a for differently scoped declaration abstracts and command abstracts; for differently scoped type abstracts and command abstracts; for differently scoped declaration abstracts and expression abstracts.

3. Apply the abstraction principle to the language in Figure 5.2 to create command abstracts. Define the semantics of command abstracts in each of the two following ways:

   a. The denotations of command abstracts are kept in a newly created semantic argument, the environment. Since variable identifiers are used as arguments to the store, what problems arise from this semantics? Show how the problems are solved by forcing variable identifiers to map to location values in the environment.
   b. The denotations of command abstracts are kept in the store; that is, $Store = \text{Identifier} \rightarrow (Nat + Proc)_{\perp}$, where $Proc = Store_{\perp} \rightarrow Store_{\perp}$. What advantages and drawbacks do you see in this semantics?

4. Consider the interaction of parameter-passing mechanisms and scoping mechanisms. Using their semantic definitions as a guide, comment on the pragmatics of each of the parameter transmission methods in Section 8.2 with statically scoped command abstracts; with dynamically scoped command abstracts.

5. In addition to those mentioned in Section 8.2, there are other parameter transmission methods for expressions. Define the semantics of:

   a. Pascal-style call-by-value: the actual parameter is evaluated to an expressible value, a new location is allocated, and the expressible value is placed in the new location's

cell. Assignment to the new location is allowed within the abstract's body.

b. PL/1-style call-by-value-result: an Identifier-L-value parameter is evaluated to a location. The value in that location is copied into a newly allocated cell. Upon the termination of the abstract, the value in the new cell is copied into the location that the actual parameter denotes.

c. Imperative call-by-need: like ALGOL60-style call-by-name, except that the first time that the actual parameter is evaluated to an expressible value, that value becomes the value associated with the parameter in all subsequent uses. (That is, for the first use of the parameter in the abstract, the parameter behaves like a call-by-name parameter. Thereafter, it behaves like a call-by-value parameter.)

What are the pragmatics of these parameter-passing mechanisms?

6. Define the syntax and semantics of a command abstract that takes a tuple of parameters.

7. An alternative to defining recursive abstracts via recursively defined environments is defining them through the store. Let:

$$Store = Location \to (Nat + \cdots + Proc + \cdots)_\perp$$
$$\text{where } Proc = Store \to Poststore_\perp$$

be a version of store that holds command abstracts. For $\mathbf{D}$: Declaration $\to$ *Environment* $\to$ *Store* $\to$ (*Environment* $\times$ *Poststore*), let:

$$\mathbf{D}[\![\textbf{proc}\ \mathrm{I} = \mathrm{C}]\!] = \lambda e.\ \lambda s.\ \text{let } (l, p) = \textit{allocate-locn}\ s \text{ in}$$
$$\text{let } e' = \textit{updateenv}\ [\![\mathrm{I}]\!]\ l\ e$$
$$\text{in } (e',\ (\textit{check}(\textit{return} \circ (\textit{update}\ l\ \text{in}\textit{Proc}(\mathbf{C}[\![\mathrm{C}]\!]e'))))(p))$$

be the semantics of procedure declaration.

a. Write the semantic equation for procedure invocation. Explain the mechanics of recursive invocation. Why is it that recursive calls can occur, but *fix* does not appear in the semantics of declaration? Under what circumstances does the use of [[I]] in [[C]] *not* cause a recursive invocation?

b. Suggest how this semantics might be extended to allow self-modifying procedures.

8. A variation on expression parameters that was not mentioned in Section 8.2 is the following: $(\lambda e.\ \mathbf{E}[\![\mathrm{E}]\!]e\ s) : \textit{Environment} \to \textit{Expressible-value}$.

a. Revise the semantic equations to fit this form. Explain how this would be implemented.

b. Show why this form of parameter transmission could easily lead to access errors in the store.

9. Revise the list-processing language given in Figure 7.5 to be typed:

i. Set the domain *Atom* to be *Nat.*

ii. Assign to each well-formed member of the Expression syntax domain a type. For example, [[1]] has type ''**nat**,'' [[1 CONS NIL]] has type ''**nat list**,'' and

⟦(LAMBDA (X: **nat list**) (HEAD X))⟧ has type ''**nat list → nat**.''

a. Alter the language's semantics so that the data typing is enforced; that is, ill-typed expressions have an erroneous denotable value.
b. What is the data type of ⟦NIL⟧? Is the construct overloaded or is it parametrically polymorphic?
c. Are there any expressions that have well-defined denotations in the untyped language but have erroneous denotations in the typed language? (Hint: consider the example of self-application in Section 7.2.2.)
d. Which constructs in the language could be profitably made polymorphic?
e. Are the recursively defined semantic domains absolutely needed to give a denotational semantics to the typed language? (Hint: consider your answer to part c and study the hierarchy *MDArray* in Section 7.3.)

10. Install occurrence equivalence type-checking into the semantics of an imperative language that uses the definition structures of Figure 8.1.

11. a. Define the semantics of these constructs:

E ::= ··· | **if** E_1 **then** E_2 **else** E_3
T ::= ··· | **if** E **then** T_1 **else** T_2

b. Define the semantics of these abstracts:

D ::= ··· | **rec fcn** $I_1(I_2) = E$ | **rec type** $I_1(I_2) = T$

for expression parameters I_2.
c. Give examples of useful recursively defined functions and types using the constructs defined in parts a and b.

12. Add parameterized command abstracts to the language of Figure 5.6 but do *not* include data type information for the formal parameters to the abstracts. In Section 8.2.1, it was suggested that this form of abstract appears to be polymorphic to the user. Why is this form of polymorphism appropriate for this particular language? But why do the polymorphic abstracts have limited utility in this example? How must the language's core operation set be extended to make good use of the polymorphism? Make these extensions and define their semantics.

13. Here is an example of a parameterized abstract in which formal parameters are parameterized on other formal parameters: ⟦**proc** stack(T; op: T→T); C⟧.

a. Show how this example is derived from the correspondence principle.
b. Give a denotational semantics to this example.

14. Formalize the semantics of the principle of correspondence.

15. Give the semantics of the Expression, Declaration, and Type blocks listed in Section 8.4.

16. What form of parameter transmission is induced by the correspondence principle from the ALGOL68 variable declaration ⟦**loc int** I:=E⟧?

17. For each language listed below, apply the design principles described in this chapter. For each principle, document your design decisions regarding syntax, semantics, and pragmatics.

    a. The calculator language in Figure 4.3.
    b. The imperative language in Figures 5.1 and 5.2.
    c. The applicative language in Figure 7.5.

18. Using the language in Figure 7.2:

    a. Use the principle of abstraction to develop expression and command abstract definition constructs. Give their semantics.
    b. Use the principle of correspondence to develop the corresponding parameter forms and parameter transmission mechanisms for the abstracts. Give their semantics.
    c. What other forms of parameters and parameter transmission would we obtain if part b was redone using the principle of parameterization? Give their semantics.

19. Milne has proposed a variety of composition operations for declarations. Three of them are:

    a. ⟦D_1 **and** D_2⟧: the declarations in ⟦D_1⟧ and ⟦D_2⟧ are evaluated simultaneously, and the resulting bindings are the union of the two.
    b. ⟦D_1 **within** D_2⟧: ⟦D_1⟧'s bindings are given to ⟦D_2⟧ for local use. The bindings that result are just ⟦D_2⟧'s.
    c. ⟦D_1; D_2⟧: ⟦D_1⟧'s bindings are passed on to ⟦D_2⟧. The result is ⟦D_2⟧'s bindings unioned with those bindings of ⟦D_1⟧ that are not superceded by ⟦D_2⟧'s.

    Define the semantics of these forms of composition, paying careful attention to erroneous forms of composition (e.g., in part a, ⟦D_1⟧ and ⟦D_2⟧ share a common identifier).

20. Use the version of **T**: Type-structure → *Environment* → (*Denotable-value* × *Environment*) defined in Exercise 24 of Chapter 7 with the version of **D**: Declaration → *Environment* → *Environment* in Figure 7.1 of Chapter 7 to redefine all of the examples of the language design principles in Chapter 8. Describe the pragmatics of the new definitions versus the ones in Chapter 8.

21. The design principles in this chapter set useful bounds for language extension. Nonetheless, economy of design is another valuable feature of a programming language. After you have worked either Exercise 17 or 18, comment on the pragmatics of the constructs you have derived. Which of the new constructs are better discarded? Why aren't language design and formal semantics definition the same thing?