Chapter 9

High Order Programming

First-order programming uses variables of a type whose values are simple entities like Booleans, integers, and finite structures containing such values. Consequently, each type has at most a countable range of values, and a test of the equality of two such values will always terminate. In high order programs, these constraints are relaxed; program variables are themselves allowed to hold programs. In principle, the number of different programs is non-denumerable, and there is no computable test which ever tests whether two variables have an equal value. A stored program is commonly called a *procedure.*

The only operation permitted on a procedure is to invoke its execution. No test is allowed of its properties. The method by which it was constructed and the method by which it computes its result are totally concealed. An implementation may store a textual or binary representation of the code of a procedure. However, all reasoning about the program should assume that the variable actually stores an abstract specification of the procedure as its value. Furthermore, any assignment that improves the value of a procedure-valued variable also improves the value of the whole program. Finally, the inclusion of high order variables does not increase the power of the language. Every program that contains a local high order variable can be transformed to the same kind of normal form as one that does not. This reproduces a result originally due to Kleene [110]. We limit our exploration of high order programming to languages which satisfy this criterion.

The introduction of procedures into a programming language can involve considerable complexity, particularly in their parameter passing mechanisms [70, 86, 119, 155, 176]. We therefore take care to introduce the complexities gradually. To begin with, in Section 9.1 there is no parameter, and all interactions are through global variables. Section 9.2 introduces a single parameter at a time, and describes value and result parameters as special cases of name parameters [149]. Certain care with the alphabet is necessary to prevent interference between parameters and global variables. Section 9.3 considers a language in which there is a single

variable (out') in the output alphabet. Further restriction to the deterministic case then gives a functional programming language.

Declarative programming, as described in Section 9.4, uses high order procedure variables to specify the desired properties of a program. If the expression of the properties is restricted to the notations of a programming language (usually functional or logical), it is possible to interpret the text directly, so as to implement a program that meets the given specification. Declarative programming is a first step towards an ideal of the abolition of programming, and its replacement by direct execution of specifications.

In this chapter we assume that all declarations are properly nested with the end of their scopes, and that all predicates are at least designs, if not programs. As always, the relationship between designs and programming notations is a central issue for study.

9.1 Procedures without parameters

In this section, we introduce a new type of variable, the *procedure* variable, whose values range over predicates, or rather some subset of predicates (or programs) with a given alphabet A. Such variables may be declared, assigned and ended in the same way as variables ranging over familiar data types such as integers or characters. The purpose of procedure assignment is to determine the effect of a later call of that procedure by its variable name. If the predicate assigned to the variable is expressed wholly in the notations of the same programming language, the call can be easily implemented by execution of that program text, or (in practice) by running its machine code translation.

A procedure variable is declared

$$\textbf{var}\ p : proc_A$$

where A determines the alphabet of the predicate that may be assigned to p. It will usually be the same as (or at most a subset of) the alphabet current at the place of the declaration; it will always exclude p itself. We will see that strict accounting of the types and alphabets of procedures is even more necessary than for variables of first-order type.

Since the alphabet of a procedure may contain procedure variables with different alphabets, the type structure has the general form of a tree, defined by the following recursion

$$\begin{aligned} <\mathit{alphabet}> &::= \text{list of } (<\mathit{variable}>:<\mathit{type}>) \\ <\mathit{type}> &::= <\mathit{procedure\ type}> \mid <\mathit{other\ type}> \\ <\mathit{procedure\ type}> &::= \mathit{proc}(<\mathit{alphabet}>) \end{aligned}$$

However, we do not want a type to contain a cycle, that is a component variable with the same alphabet as the whole program. This is one reason why the procedure variable is forbidden to appear in its own alphabet. Another reason is to simplify reasoning about procedure calls. A consequence is that we must continue to use the fixed point notation to define recursive procedures.

As with other data types, we need a way of denoting procedure values by constants, like numerals and strings. Clearly, the most direct way of denoting a procedure constant is just to write the intended predicate or program in the normal way. But this risks confusion with predicates describing the rest of the program. To distinguish what is to be stored from what is to be executed, we enclose procedure constants in brackets, for example

$$\{\![x := x + 1]\!\}$$

The brackets here have no semantic significance; they are simply omitted when the procedure is executed.

In a high order programming language, assignment of a value to a procedure variable will be written in the usual way

$$p := \{\![P]\!\}$$

where P itself is a design or a program text (called the *body* of the procedure). It has to have exactly the same alphabet as that declared for the variable p. This will deliberately exclude all variables declared later than p but earlier than the assignment to p. The restriction is needed to prevent what is known as the *dangling reference* problem [155]. On termination of the program

$$\textbf{var } p : proc_A;\ \textbf{var } x;\ p := \{\![x := x + 1]\!\};\ \textbf{end } x; \ldots$$

the value of p is a program which updates a variable x that no longer exists, because its scope has been ended. The alphabet constraint prevents this by disallowing the occurrence of x on the right hand side of the assignment to p. In many languages, the restriction is implicit in an even stronger restriction: that the only assignment allowed to a procedure variable is one that is written at the point of its declaration. For this case we will use the notation

$$\textbf{proc } p := \{\![P]\!\}$$

A text written in brackets like this is always a constant; it is not subject to substitutions for the variables that it contains. For example,

$$(x := x + 1)\,;\,(p := \{\![y := x]\!\}) \;=\; (p := \{\![y := x]\!\})\,;\,(x := x + 1)$$

It is certainly *not* equal to

$$(p := \{\![y := x + 1]\!\})\,;\,(x := x + 1)$$

Assignment of constants is just a special case of assignment of expressions. A procedure-valued expression may be built from constants and variables combined with the notations of the programming language, for example

$$p := (\{\![x := x + 1]\!\}; q)$$

assigns to p a value similar to the value of q, except that its execution will be preceded by incrementation of x. Note again that a procedure value is an abstract object of mathematics, and not just a text. For example,

$$\{\![x := x + 1]\!\}; \{\![x := x + 3]\!\} \;=\; \{\![x := x + 4]\!\}$$

This is in exact analogy to the equation between numeric constants

$$1 + 3 \;=\; 4$$

The ability of programs to write programs is called *metaprogramming*, and there is hope that its wider application will lead to software that is easier to adapt to varying needs and changing environments.

To invoke the procedure which is the current value of the procedure p, we just write the variable as an executable statement of the program. The effect is clearly shown in the algebraic law

$$(p := \{\![Q]\!\})\,;\, p \;=\; (p := \{\![Q]\!\})\,;\, Q$$

Since p is not allowed in the alphabet of Q, this is also equal to

$$Q\,;\, (p := \{\![Q]\!\})$$

In an implementable programming language, a procedure value must obviously be written as a program, most usually in the same implemented language. But in reasoning about program design, we want to use the design of a program in place of the program text that has not yet been written. The later substitution of the program itself for the design should be justified by the usual appeal to monotonicity, as described for stepwise refinement in Section 2.4. This means that procedure assignment should be monotonic in the assigned value

$$P \sqsubseteq Q \;\Rightarrow\; (p := \{\![P]\!\}) \sqsubseteq (p := \{\![Q]\!\})$$

Unfortunately, this law would be invalid for the standard definition of assignment which equates the final value of the variable with the assigned value

$$p' \;=\; Q$$

The solution is to redefine assignment to allow an implementation to replace the assigned value by any value that is actually stronger than it; the effect is described by weakening the previous equation to an inequation

$$p' \sqsupseteq Q$$

In fact, every assignment can be defined by an inequation in this same way. In the case of first-order data, two values are comparable only if they are equal

$$v' \sqsupseteq v \quad \textbf{iff} \quad v' = v$$

The replacement of equality by an ordering in the definition of assignment also helps in reasoning about the use of computer resources (Example 7.2.1). One of the main objectives of such reasoning is to ensure that an implementation may validly replace a program by one that consistently uses less resources. But the present theory of assignment dictates that programs which assign different values to the resource variable r will be totally incomparable. The solution is to use numerical ordering in the definition of assignment to r

$$r := r + e \ =_{df} \ \textbf{true} \vdash r' \leq r + e \wedge y' = y \wedge \ldots$$

As a result, we can use normal implication ordering to validate replacement of any program by a more efficient one, for example

$$r := r + 4 \ \sqsubseteq \ r := r + 3$$

Because equality is a special case of an ordering, the following definitions are entirely consistent with the previous definitions used up to this point. The definition of alphabet extension needs the same kind of generalisation to allow procedure values to improve instead of staying the same.

Definition 9.1.1 (Procedure assignment)

Let $\alpha p = \alpha Q$

$$p := \{\!|Q|\!\} \ =_{df} \ (\textbf{true} \vdash (p' \sqsupseteq Q) \ \wedge \ (v' \sqsupseteq v))$$

where $\alpha(p := \{\!|Q|\!\}) = \{p, p', v, v'\}$ □

Examples 9.1.2

(1) The declaration **proc** *double* $:= \{\!|x := 2 \times x|\!\}$ expands to

$$\textbf{true} \vdash (v' \sqsupseteq v) \wedge (double' \sqsupseteq x := 2 \times x)$$

(2) Assignment $inc := \{\!|x := x + 1|\!\}$ is described by

$$\textbf{true} \vdash (v' \sqsupseteq v) \wedge (inc' \sqsupseteq x := x + 1)$$

(3) Execution of $(inc := \{\!|x := x + 1|\!\})\,;\,(p := (\{\!|x := x + 2|\!\}; inc))$ has the following effect

$$\textbf{true} \vdash (p' \sqsupseteq x := x + 3) \wedge (inc' \sqsupseteq x := x + 1) \wedge (v' \sqsupseteq v)$$ □

Definition 9.1.3 (Alphabet extension)

Let $p \notin \alpha P$

$$P_{+\{p\}} \quad =_{df} \quad P \| (\mathbf{true} \vdash (p' \sqsupseteq p)) \qquad \square$$

Alphabet extension will frequently be needed to extend the alphabet of a procedure call to that of its context. For example,

$$(\mathbf{proc}\ p := \{\!|Q|\!\});\ \mathbf{var}\ x := x + 1;\ p$$
$$= (\mathbf{proc}\ p := \{\!|Q|\!\});\ \mathbf{var}\ x := x + 1;\ Q_{+\{p,x\}}$$

In many languages a problem arises if the variable name x is already in the alphabet of the procedure p and is declared again local to the call, for example

$$(inc := \{\!|x := x + 1|\!\});\ \mathbf{var}\ x := 1;\ inc; \ldots; \mathbf{end}\ x$$

Here it would be wrong to replace the call by its meaning

$$(inc := \{\!|x := x + 1|\!\});\ \mathbf{var}\ x := 1;\ x := x + 1; \ldots; \mathbf{end}\ x$$

The reason is that the x in the alphabet of *inc* is a different variable from the more recently declared x at the place where *inc* is called. The problem is one of collision of local (bound) and global (free) variables. This problem cannot occur in our programming language because local variables must have different names from global ones (see Section 2.9).

As a result of the change in our basic definitions, the meaning of $I\!I$ also changes

$$I\!I \quad =_{df} \quad (\mathbf{true} \vdash v' \sqsupseteq v)$$

We therefore need new healthiness conditions to ensure that this new identity is still a left unit and a right unit of composition.

Definition 9.1.4 (Healthiness conditions of procedure variables)

A predicate Q is said to be **P1** if it satisfies the following left unit law

P1 $\quad I\!I ; Q \ = \ Q$

It is said to be **P2** if it obeys the following right unit law

P2 $\quad Q ; I\!I \ = \ Q$ $\qquad \square$

Theorem 9.1.5 (Closure of healthy predicates)

Predicates satisfying the healthiness conditions **P1** and **P2** form a complete lattice which is closed under ;, $\sqcap$, $\lhd b \rhd$ and μ.

Proof From the facts that $\Pi \sqsubseteq (v' = v)$ and $\Pi ; \Pi = \Pi$ it follows that

$$L =_{df} \lambda X \bullet (\Pi ; X ; \Pi)$$

is a $\{;, \sqcap, \lhd b \rhd\}_{\sqsupseteq}$-link. The conclusion that **P1** and **P2** are preserved by all programming operators comes from Theorems 4.1.15 and 4.1.19. □

In spite of the changed definitions of this section, all the laws given in Chapter 5 are still valid. As a result every finite program which has no procedure variables in its alphabet can be reduced to finite normal form, using additional laws given below. Local procedure declarations are removed by normalisation. The normalisation procedure is to move all procedure declarations rightward in the text of the program, replacing all its calls by the current value.

L1 $\mathbf{var}\, p : proc_A ; (v := e) = (v := e) ; \mathbf{var}\, p : proc_A$

L2 $\mathbf{var}\, p : proc_A ; p = \mathbf{true}$

L3 $(p := \{\!|Q|\!\}) ; p = Q_{+\{p\}} ; (p := \{\!|Q|\!\})$

9.2 Parameter mechanism

In the previous section, procedure variables took values ranging over programs, or (more generally) predicates which describe program behaviour. In this section, we introduce variables whose values are not themselves predicates but rather functions from some other domain to predicates. The application of such a function to a particular parameter value from that domain will yield a predicate which describes the behaviour of a procedure call. Thus the parameter passing mechanism in the programming language is given the same meaning that it has in the realm of mathematics: just the application of a function. The intention, as always, is to provide the maximum assistance in reasoning about the design of programs from their specifications.

Programming convenience is the main reason for introducing parameters into a programming language. The alphabet constraints imposed in the previous section on procedure variables, assignments and call are both strict and burdensome. To get a procedure to operate on some desired initial values, these have to be copied before the call to certain fixed global locations. To get the results of the procedure into the place where they will be used afterwards, they have to be copied after the call out of fixed global variables . These variables, and all other global resources used by the call, are fixed at the time of declaration of the procedure variable. Such inconvenience is accepted in machine code programming, but it is greatly eased in high order programming language by including a facility for passing the *name* of a variable (even a local variable) as a parameter to a procedure. These name parameters are often efficiently implemented by passing the address of the

machine location allocated to store the variable. Within the machine code of the body of the procedure, the variable is accessed by indirect addressing.

Let p be a procedure variable, and let x be a variable of type T in the alphabet of p which has been selected as a name parameter. An assignment of the procedure will be written

$$p := \{\![\lambda x : var(T) \bullet P]\!\}$$

The procedure p needs to have the appropriate type, which is a function from a variable y of type T to predicates with alphabet

$$\alpha P - \{x, x'\} \cup \{y, y'\}$$

Note that the alphabet of the result of applying the function depends on the choice of parameter. Functions whose type is dependent on the argument of each call are called *polymorphic* [128]. A procedure with two or more parameters can be defined by successive lambda abstraction

$$\lambda x, y : var(T) \bullet P \;=\; \lambda x : var(T) \bullet (\lambda y : var(T) \bullet P)$$

In principle, such a procedure is a function of the single parameter x, whose result is itself a function of the single parameter y. When (and only when) both parameters have been supplied, the result is a predicate which can be called in the same way as in the previous section.

A call of a parameterised procedure is written in the usual way as $p(y)$, where y must be a variable of type T, and one which is *not* in the alphabet of p. This restriction applies to all the arguments of a procedure: not only must they be disjoint from the alphabet of the procedure body, they must also be distinct from each other. The reason for this restriction will soon be abundantly clear.

A variable x in a program is represented in the corresponding predicate as two observational variables x and x'. The notation $\lambda x : var(T)$ in a programming language is therefore represented in the language of predicates by lambda abstraction $\lambda x, x'$ over both the corresponding observational variables. With this interpretation, the meaning of λ is exactly the same as that familiar in mathematics. The value of the function applied to particular parameters y and y' is the predicate obtained on replacing x by y and x' by y' in the body of the procedure. But there is an extra constraint: the second parameter must be the dashed variant of the first.

Example 9.2.1

$$\begin{aligned} & \{\![\lambda x : var(T) \bullet \mathbf{true} \vdash (x' = x + 3 \wedge z' = z + 4)]\!\}(y) \\ = \; & \{\![\lambda x, x' \bullet \mathbf{true} \vdash (x' = x + 3 \wedge z' = z + 4)]\!\}(y, y') \\ = \; & \mathbf{true} \vdash (y' = y + 3 \wedge z' = z + 4) \end{aligned}$$

□

The rule that prohibits sharing between the argument and the body of the procedure is essential to avoid contradiction, as shown below.

Counterexample 9.2.2

$$\begin{aligned} & \{\![\lambda x : var(T) \bullet \mathbf{true} \vdash (x' = x + 3 \wedge z' = z + 4)]\!\}(z) \\ = \ & \mathbf{true} \vdash (z' = z + 3 \wedge z' = z + 4) \\ = \ & \mathbf{true} \vdash \mathbf{false} \end{aligned}$$

□

Many programming languages have a variety of parameter passing mechanisms to meet various needs. For example, a *result* parameter is one that is used solely to determine where to store a result of the procedure call: the initial value of the corresponding argument is irrelevant. Furthermore, it is safe to relax the disjointness constraint on result arguments. A suitable notation might be

$$\{\![\lambda x : res(T) \bullet P]\!\}$$

An advantage of a result parameter is a potential increase in efficiency of implementation, by avoiding the use of indirect addressing. Another advantage is the calculation of the effect of a procedure call by the following law

$$\{\![\lambda x : res(T) \bullet P]\!\}(y) \ = \ \mathbf{var}\ x; P; y := x; \mathbf{end}\ x$$

A definition that gives us this law also gives a strong hint on an efficient method of implementation.

Definition 9.2.3 (Result parameter)

$$\lambda x : res(T) \bullet P \ =_{df} \ \lambda y : var(T) \bullet (\mathbf{var}\ x; P; y := x; \mathbf{end}\ x)$$

where y is a fresh variable. □

A similar treatment can be given to the *value* parameter; the corresponding argument specifies where to find an initial value for the call, and the procedure body is not allowed to change the argument. A suitable notation might be

$$\{\![\lambda x : val(T) \bullet P(x)]\!\}$$

In this case it is usual to allow the argument to be an arbitrary expression of type T, without any extra restriction. The meaning is explained by the law

$$\{\![\lambda x : val(T) \bullet P]\!\}(e) \ = \ \mathbf{var}\ x := e; P; \mathbf{end}\ x$$

A definition that gives us this law is as follows.

Definition 9.2.4 (Value parameter)

$$\lambda x : val(T) \bullet P \ =_{df} \ \lambda y : T \bullet (\mathbf{var}\ x := y; P; \mathbf{end}\ x)$$

where y is a fresh variable. Note that in this case it ranges over *values* of type T rather than over *variables*. That is what allows the actual argument to be an expression. □

For purposes of direct implementation, it is essential that the body of a called procedure be expressed as a program, using only the notations of the programming language, and these certainly do not include dashed variables. For procedure bodies that are already programs, we need to ensure that the result of parameter substitution is still a program that can be directly executed. In fact, the simplest way of obtaining that program is to substitute the undashed argument variable for all occurrences of the parameter in the program itself, even those on the left of an assignment. If the resulting program is expressed as a predicate, this predicate is the *same* as if the dashed and undashed versions of the variable had been substituted into the corresponding predicate.

Example 9.2.5

$$
\begin{aligned}
& \{\!|\lambda x : var(T) \bullet (x := x + 3;\ z := z + 4)|\!\}(y) && \{\text{substitution in the program}\} \\
= \ & (y := y + 3;\ z := z + 4) && \{\text{def of assignment and composition}\} \\
= \ & \mathbf{true} \vdash (y' = y + 3 \land z' = z + 4) && \{\text{substitution in the predicate}\} \\
= \ & (\mathbf{true} \vdash (x' = x + 3 \land z' = z + 4))[y, y'/x, x'] && \{\text{function application}\} \\
= \ & \{\!|\lambda x, x' \bullet \mathbf{true} \vdash (x' = x + 3) \land (z' = z + 4)|\!\}(y, y') && \square
\end{aligned}
$$

Coincidence of meaning of two methods of substitution (in the program and in the predicate) depends on the restriction that the actual argument should be outside the alphabet of the procedure.

Counterexample 9.2.6

$$
\begin{aligned}
& \{\!|\lambda x : var(T) \bullet (x := x + 3;\ z := z + 4)|\!\}(z) \\
= \ & (z := z + 3;\ z := z + 4) \\
\neq \ & \mathbf{true} \vdash \mathbf{false} \\
= \ & \{\!|\lambda x, x' \bullet \mathbf{true} \vdash (x' = x + 3) \land (z' = z + 4)|\!\}(z, z') && \square
\end{aligned}
$$

This counterexample shows the need for a proof that the result of substitution of names in correctly written programs is exactly the same as in correctly written predicates. That is the topic of the following theorem. It shows (e.g. in Cases 1, 4 and 5) that name substitution in a program requires substitution to be applied even in procedure alphabets and procedure bodies.

Theorem 9.2.7 (Substitution)

Let $\mathtt{P(x)}$ be a program text using $\mathtt{x}$ as a variable name. Let $\lceil \mathtt{P(x)} \rceil$ be its meaning. Let $\mathtt{ax}$ be a variable name. Suppose that

(A1) $\mathtt{ax}$ does not occur in $\mathtt{P(x)}$, and

(A2) none of the global variable names of $\mathtt{P(x)}$ is used as a local variable.

Then $\quad \lceil \mathtt{P(ax)} \rceil \;=\; \lceil \mathtt{P(x)} \rceil [ax, ax'/x, x']$

Proof Define

$$R(X) \;=_{df}\; (\mathbf{true} \vdash (ax = x')); X; (\mathbf{true} \vdash (x = ax'))$$

From Theorem 3.1.4 it follows that for all predicates $P(x, x')$ satisfying the healthiness conditions **H1** to **H3**

$$R(P(x, x')) \;=\; P(ax, ax') \;=\; P(x, x')[ax, ax'/x, x']$$

The following proof is based on structural induction on the program text $\mathtt{P(x)}$.

Case 1 $\mathtt{P(x)} = \mathtt{(x, y, \ldots := e(x), f(x), \ldots)}$.

$$\begin{array}{lll}
 & \lceil \mathtt{P(ax)} \rceil & \{\text{def of P}\} \\
= & \lceil \mathtt{(ax, y, \ldots := e(ax), f(ax), \ldots)} \rceil & \{\text{def of assignment}\} \\
= & \mathbf{true} \vdash (ax' = e(ax)) \wedge (y' = f(ax)) \wedge \ldots & \{\text{assumption (A2)}\} \\
= & R(\mathbf{true} \vdash (x' = e(x)) \wedge (y' = f(x)) \wedge \ldots) & \{\text{def of assignment}\} \\
= & R(\lceil \mathtt{P(x)} \rceil) &
\end{array}$$

Case 2 $\mathtt{P(x)} = \mathtt{P1(x)} \lhd \mathtt{b(x)} \rhd \mathtt{P2(x)}$.

$$\begin{array}{lll}
 & \lceil \mathtt{P(ax)} \rceil & \{\text{def of P}\} \\
= & \lceil \mathtt{P1(ax)} \lhd \mathtt{b(ax)} \rhd \mathtt{P2(ax)} \rceil & \{\text{def of conditional}\} \\
= & \lceil \mathtt{P1(ax)} \rceil \lhd b(ax) \rhd \lceil \mathtt{P2(ax)} \rceil & \{\text{inductive hypothesis}\} \\
= & R(\lceil \mathtt{P1(x)} \rceil) \lhd b(ax) \rhd R(\lceil \mathtt{P2(x)} \rceil) & \{\text{Theorem 4.4.8}\} \\
= & R(\lceil \mathtt{P1(x)} \rceil \lhd b(x) \rhd \lceil \mathtt{P2(x)} \rceil) & \{\text{def of conditional}\} \\
= & R(\lceil \mathtt{P(x)} \rceil) &
\end{array}$$

Case 3 $\mathtt{P(x)} = \mathbf{var}\ \mathtt{lv}; \mathtt{Q(lv, x)}; \mathbf{end}\ \mathtt{lv}$.

$$\begin{array}{lll}
 & \lceil \mathtt{P(ax)} \rceil & \{\text{def of P and assumption (A2)}\} \\
= & \lceil \mathbf{var}\ \mathtt{lv}; \mathtt{Q(lv, ax)}; \mathbf{end}\ \mathtt{lv} \rceil & \{\text{def of declaration}\} \\
= & \exists lv, lv' \bullet \lceil \mathtt{Q(lv, ax)} \rceil & \{\text{inductive hypothesis}\}
\end{array}$$

$$
\begin{aligned}
&= \exists lv, lv' \bullet R(\lceil \mathtt{Q(1v, x)} \rceil) && \{\text{(A1) and (A2)}\}\\
&= R(\exists lv, lv' \bullet \lceil \mathtt{Q(1v, x)} \rceil) && \{\text{def of declaration}\}\\
&= R(\lceil \mathtt{P(x)} \rceil)
\end{aligned}
$$

Case 4 $\mathtt{P(x)} = \mathbf{proc}\ (\mathtt{p} := \mathtt{Q(x)});\ \mathtt{p};\ \mathbf{end}\,\mathtt{p}.$

$$
\begin{aligned}
&\lceil \mathtt{P(ax)} \rceil && \{\text{def of P and assumption (A2)}\}\\
&= \lceil \mathbf{proc}\ \mathtt{p} := \mathtt{Q(ax)};\ \mathtt{p};\ \mathbf{end}\,\mathtt{p} \rceil && \{\text{def of procedure call}\}\\
&= \lceil \mathtt{Q(ax)} \rceil && \{\text{inductive hypothesis}\}\\
&= R(\lceil \mathtt{Q(x)} \rceil) && \{\text{def of procedure call}\}\\
&= R(\lceil \mathbf{proc}\ \mathtt{p} := \mathtt{Q(x)};\ \mathtt{p};\ \mathbf{end}\,\mathtt{p} \rceil) && \{\text{def of P}\}\\
&= R(\lceil \mathtt{P(x)} \rceil)
\end{aligned}
$$

Case 5 $\mathtt{P(x)} = \mathbf{proc}\ \mathtt{p} := (\lambda \mathtt{f} : \mathtt{var(T)} \bullet \mathtt{Q(f, y)});\ \mathtt{p(x)};\ \mathbf{end}\ \mathtt{p}.$

From procedure alphabet constraints it follows that $\mathtt{x} \notin \alpha\mathtt{Q}$.

$$
\begin{aligned}
&\lceil \mathtt{P(ax)} \rceil && \{\mathtt{x} \notin \alpha\mathtt{Q}\}\\
&= \lceil \mathbf{proc}\,\mathtt{p} := \lambda \mathtt{f} : \mathtt{var(T)} \bullet \mathtt{Q(f, y)};\\
&\quad \mathtt{p(ax)}; \mathbf{end}\,\mathtt{p} \rceil && \{\text{def of procedure call}\}\\
&= \lceil \mathtt{Q(f, y)} \rceil [ax, ax'/f, f'] && \{\text{(A2) and inductive hypothesis}\}\\
&= \lceil \mathtt{Q(ax, y)} \rceil && \{\text{inductive hypothesis}\}\\
&= R(\lceil \mathtt{Q(x, y)} \rceil) && \{\text{inductive hypothesis}\}\\
&= R(\lceil \mathtt{Q(f, y)} \rceil [x, x'/f, f']) && \{\text{def of procedure call}\}\\
&= R(\lceil \mathtt{P(x)} \rceil)
\end{aligned}
$$

Case 6 $\mathtt{P(x)} = \mathbf{proc}\ (\mathtt{p} := \lambda \mathtt{f} : \mathtt{var(T)} \bullet \mathtt{Q(x, f)});\ \mathtt{p(y)};\ \mathbf{end}\,\mathtt{p}.$

Similar to **Case 5**.

Case 7 $\mathtt{P(x)} = \mathtt{P1(x)}\ \mathrm{op}\ \mathtt{P2(x)}$ where $\mathrm{op} \in \{\sqcap, ;\}$.

Similar to **Case 2**.

Case 8 $\mathtt{P(x)} = \mathbf{proc}\ (\mathtt{p} := \mathtt{Q(x)});\ \mathtt{S(x)};\ \mathtt{p};\ \mathtt{T(x)};\ \mathbf{end}\,\mathtt{p},$

where $\mathtt{S(x)}$ does not mention p.

The conclusion follows from **Case 7** and the fact that

$$\lceil \mathtt{P(ax)} \rceil = \lceil \mathtt{S(ax)}; (\mathbf{proc}\,(\mathtt{p} := \mathtt{Q(ax)}); \mathtt{p}; \mathbf{end}\,\mathtt{p}); (\mathbf{proc}\,(\mathtt{p} := \mathtt{Q(ax)}); \mathtt{T(ax)}; \mathbf{end}\,\mathtt{p}) \rceil$$

Case 9 $\mathtt{P(x)} = \mu \mathtt{X} \bullet \mathtt{F(b(x), Q(x), X)},$

where F is wholly composed by operators ;, $\sqcap$ and $\lhd\,\rhd$.

$$
\begin{array}{lll}
 & \lceil \mathtt{P(ax)} \rceil & \{\text{(A1) and (A2)}\} \\
= & \lceil \mu \mathtt{X} \bullet \mathtt{F(b(ax), Q(ax), X)} \rceil & \{\text{def of } \mu\} \\
= & \mu X \bullet F(b(ax), \lceil \mathtt{Q(ax)} \rceil, X) & \{\text{inductive hypothesis}\} \\
= & \mu X \bullet F(b(ax), R(\lceil \mathtt{Q(x)} \rceil), X) & \{\text{Theorem 4.2.12)}\} \\
= & R(\mu X \bullet F(b(x), \lceil \mathtt{Q(x)} \rceil, X)) & \{\text{def of } \mu\} \\
= & R(\mathtt{P(x)}) & \Box
\end{array}
$$

This is a long and arduous proof, but the result is very important in establishing the mathematical coherence of the programming notation: that it is appropriately independent of the choice of the names of local variables. The details of the proof are essential to check the need for all the alphabet restrictions imposed on parameter passing. Similar proofs will be needed by the implementors of any tool which processes program texts or their proofs.

9.3 Functions

A function in a programming language is often introduced as a special case of a procedure, one that delivers just a single value as its only result. A call of such a procedure is written as an operand in an expression; during evaluation of the expression, the procedure is called, and the result that it delivers is used in further calculation of the value of the whole expression. In this way, a function defined by the programmer is no different from the functions and operators (e.g. plus and minus) which are built into the programming language.

The simplest way of defining the result of a function call is by a body consisting just of an expression e that delivers the required value. The expression will usually depend on the value of some parameter, occurring as a variable in e (say x of type T). The function is written in the usual lambda notation

$$\lambda x : T \bullet e$$

The actual argument of the function must be an expression (say f) of type T, and the result of the call can be calculated by systematic substitution of this expression for the parameter in the body of the function. The substituted body is evaluated in the normal way.

$$(\lambda x : T \bullet e(x))(f) \;=\; e(f)$$

This familiar rule of the lambda calculus is called beta-reduction. In languages which do not exclude meaningless expressions (say $1/0$), its consequences may be rather surprising. Consider the constant function of real numbers

$$K2 =_{df} \lambda x : real \bullet 2$$

Since the defining expression 2 does not contain x, substitution of $1/0$ for x makes no difference, so we get

$$K2(1/0) = 2$$

The result of the call is well defined as 2, even though the argument is not. This minor paradox is similar to that encountered in Section 2.5, and it can be resolved in the same two ways.

The more elegant resolution is to adopt a method of implementation that conforms to the principle of beta-substitution. This is known as *lazy* evaluation, because it attempts to postpone evaluation of the argument of a function until it can be no longer avoided; that is when the value is known to be needed for computation of the overall result of the program. The computed value of the argument is then stored, so that it can be used again without recalculation. In the case of the constant function described above, the argument value is never needed, because the answer can be given straight away. In other cases, the need to evaluate the argument will depend on the values of the other arguments and global variables in the body of the function.

Lazy evaluation is quite standard in a purely functional language, but unusual in an imperative language, because of its inefficiency. It is generally more efficient to evaluate all arguments fully before calling the function. This *strict* implementation is the same as that for the value argument, introduced for procedures in the previous section. We might as well use the same notation

$$\lambda x : val(T) \bullet e$$

The result of applying this function to an actual argument f is defined by substitution in exactly the same way as in the lazy case. The difference between the two parameter mechanisms lies solely in the definition of the $\mathcal{D}$ operator (see Section 3.1), which gives the weakest precondition under which the expression is sure to deliver a value. In the strict case, the argument is required to be defined as well as the body of the function.

Definition 9.3.1 ($\mathcal{D}$ for lazy and strict functions)

$$\mathcal{D}((\lambda x : T \bullet e)(f)) \;=_{df}\; \mathcal{D}(e[f/x])$$
$$\mathcal{D}((\lambda x : val(T) \bullet e)(f)) \;=_{df}\; \mathcal{D}(f) \wedge \mathcal{D}(e[f/x])$$ □

Note that if functions are definable in the language by recursion, it is no longer possible to ensure that $\mathcal{D}(e)$ will itself always be computable. The only way a computer can test its truth is to attempt to evaluate e. If this terminates, then $\mathcal{D}(e)$ is true. If it does not terminate, the computer will never find out that $\mathcal{D}(e)$ is false. Like most mathematical notations, the fact that a computer cannot compute it is of no concern. Quite the reverse: introduction of the notation is even more

necessary, for assistance in reasoning about avoidance of exactly these failures in a program.

Many programming languages permit the result of a function to be computed by a block of imperative program, which assigns the intended function value to some local variable of the body, say *res*. Such a program block is turned into an expression by the operator τ, which declares the name of the variable from which the result will be taken.

$$(\tau\ res : T \bullet P) \qquad \text{(the } res \text{ such that } P\text{)}$$

This construction may appear anywhere as a component in a larger expression, for example

$$x := (\tau\ t : T \bullet Q) + (\tau\ s : S \bullet R) \times 3$$

Such result blocks are often useful as bodies of functions, and they assist in explaining the intention of value parameters, as shown in the law

$$(\lambda x : val(S) \bullet (\tau\ res : T \bullet P))(e) \ = \ \tau\ res : T \bullet (\mathbf{var}\ x := e; P; \mathbf{end}\ x)$$

The inclusion of programs inside expressions is a facility which faces the language designer with the question of *side-effect*. What happens if the body of the function makes an assignment to some global variable of the program? In many programming languages, the effect is explained by the manner of its implementation, which is quite simple if parameter passing is treated strictly. All function calls in an expression can be converted into procedure calls with a result parameter. These calls are executed before evaluating the expression, and the results are used whenever the corresponding function calls appear within the expression. To reason successfully about the program, the programmer has to make the same transformation in exactly the same way as the implementor. But if this has to be done by both programmer and implementor, there was no good reason to use functional notation in the first place. There are many good reasons not to. Hidden side-effects in function calls are a potent source of program misunderstanding and error. They can readily be banned by a syntactically checkable restriction, which enforces the rule that the variable that holds the result must be the only one in the output alphabet of the function body. In the formalisation and use of a theory of programming, this is the restriction that we recommend.

In a language which permits non-determinism, a more serious question arises. What happens if there is more than one value of *res* that satisfies P? An early answer was given by Russell's theory of descriptions. But we adopt the version given by Church, who introduced the choice operator as the basic method of quantification in higher order logic [39, 69]; Church's answer was taken over in the logic of the specification language Z [171]. Consider the set of possible answers

$$\{res' : T \mid P(res')\}$$

The operator τ acts like a choice function, selecting a member of this set. If the set is non-empty, its result is definitely a member of the set. We don't know which member it is, but every time a selection is made from the same set, the same member is selected. τ is a genuine function, so these properties of τ are encapsulated in the axiom

$$(\exists res' \bullet P(res')) \Rightarrow P(\tau\, res : T \bullet P)$$

Unfortunately, it is practically impossible for a non-deterministic programming language implementation to match the logical properties of Church's choice function. The only reasonable way of implementing non-determinism is by selecting an arbitrary one of the possible results. But suppose the same possible set of values crops up again later in the execution of the program. Church's axioms require the implementor to detect this fact and always deliver the same value on the later occasion as the earlier. The inefficiency is unthinkable. But if a different choice is made non-deterministically on the two occasions, there is virtually no hope of reasoning successfully about the results: one cannot even rely on the most elementary property of all mathematics, namely that

$$e = e$$

There is only one case in which the problem is avoided – the case in which the implemented choice function is guaranteed to agree with Tarski's choice: that is, when the set has only one member anyway. We must therefore prohibit the application of τ to programs that are non-deterministic in their results. The obligation to avoid non-determinism can be placed upon the programmer by a suitable definition of the $\mathcal{D}$ operator applied to expressions that contain programs.

Definition 9.3.2 ($\mathcal{D}$ for design blocks)

$$\mathcal{D}(\tau\, res : T \bullet (p \vdash P)) =_{df} p \wedge \text{there is at most one } res' \text{ such that } P \qquad \square$$

Another very strong motive for this harsh definition is given by the lemma

Lemma 9.3.3 τ is monotonic. $\square$

At the other end of the scale, what happens if τ is applied to an empty set? Church's logic then selects a particular value of the intended type of the result. In Z, this may be a non-standard value. We do not know which value it is, *but it is always the same value.* This logic is frequently implemented for built-in functions by computer hardware: for example, if the result of a calculation cannot be represented as a floating point number, it is given as a bitpattern called a *nan* (not a number). In fact this solution is not needed for programmer-defined functions,

because the feasibility conditions for program blocks guarantee that the set of possible results is non-empty.

The introduction of $\mathcal{D}$ into reasoning about expressions that may be undefined is closely analogous to the introduction of ok, ok' (or $\vdash$) into reasoning about programs that may fail to terminate, and $\mathcal{D}$ obeys a very similar collection of healthiness conditions. In exploring these properties, we abandon the rule that $\mathcal{D}x$ is always true for a variable x. In fact, if x is a lazy parameter, it will be false whenever the corresponding argument is undefined. Note that $\mathcal{D}x$ is *not* a function of the value of x. It is a separate Boolean variable associated with the variable x; like all logical variables, it admits quantification and substitution.

The first healthiness condition guarantees that when the operands of an expression become more defined, the result cannot become less defined.

D1 $\mathcal{D}(e)$ is monotonic in $\mathcal{D}x$.

The second healthiness condition deals with the case when the value of an expression e is defined, even though the value of one of its operands x is undefined. This can only happen if the value of e is genuinely independent of the value of x. These conditions are formalised in the law

D2 $\mathcal{D}e \wedge \neg\mathcal{D}x \;\Rightarrow\; \forall x, \forall \mathcal{D}x \bullet \mathcal{D}e$

It is reasonable to insist that $\mathcal{D}e$ is always defined, that is

D3 $\mathcal{D}(\mathcal{D}e) \;=\; \mathit{true}$

Exercise 9.3.4

Here is a structural definition of $\mathcal{D}$ over a simple language of expressions

$$\begin{aligned}
\mathcal{D}(x+y) &= \mathcal{D}x \wedge \mathcal{D}y \\
\mathcal{D}(x \times y) &= \mathcal{D}x \wedge (x \neq 0 \Rightarrow \mathcal{D}y) \\
\mathcal{D}(x \vee y) &= (x \wedge \mathcal{D}x) \vee (y \wedge \mathcal{D}y) \vee (\mathcal{D}x \wedge \mathcal{D}y)
\end{aligned}$$

(1) Extend the definition of $\mathcal{D}$ to conditional expressions.

(2) Prove that all the definitions are healthy. □

The definition of $\mathcal{D}$ for disjunction in this exercise would cause a serious problem for implementation. It decrees that the only method of evaluating $(x \vee y)$ is to embark on the evaluation of both x and y; if either evaluation terminates with the answer *true*, that is the right answer, and evaluation of the other operand can be discontinued. If one evaluation terminates with $false$, evaluation of the other operand continues and eventually gives the right answer. If neither evaluation terminates, that is the right answer too. This method of implementation

is called *non-sequential*, because the two evaluations must in principle be undertaken concurrently, or by some simulation of concurrency such as interleaving. The implementation overhead for non-sequential calculations is severe, but what is worse is the wasted effort on evaluation of alternatives that are never used. Non-sequentiality is usually avoided in programming language design. For example, the definition of $\mathcal{D}(x \vee y)$ is often given as

$$\mathcal{D}x \wedge (x \vee \mathcal{D}y)$$

which specifies that evaluation of y must be *short circuited* if the value of x is found to be true. This definition is sequential, because every reference to an operand x is protected by a positive occurrence of $\mathcal{D}x$. But it does introduce an unwelcome asymmetry into the normally symmetric meaning of disjunction.

Exercise 9.3.5

Define a syntactic normal form for expressions in x, $\mathcal{D}x$, y, $\mathcal{D}y$, ... which guarantees sequentiality, for example every occurrence of x is protected by $\mathcal{D}x$. Define an ordering which permits the limit of infinite chains of such normal forms. Prove that if $\mathcal{D}e$ is in normal form, then

$$\mathcal{D}(e[f/x]) = \mathcal{D}e[f, \mathcal{D}f/x, \mathcal{D}x] \qquad \square$$

9.4* Declarative programming

A purely functional programming language like Haskell [95] is one in which all programs are defined as functions, and there are no assignments, jumps or side-effects. Such programs are commonly defined by a set of equations which describe the results of the function in all the relevant special cases.

Example 9.4.1 (Factorial)

A factorial function is defined by a pair of equations

$$\begin{aligned} fac(0) &= 1 \\ fac(n) &= n \times fac(n-1), \qquad \text{if } n > 0 \end{aligned} \qquad \square$$

If the equations specifying the function are expressed within the rules of the programming language, an implementation can undertake the evaluation of any call of the function; it does so by repeatedly replacing each call of a function by the corresponding right hand side of the relevant equation, with appropriate substitutions of arguments for parameters, and calculation of any operations with operands which are now constants. This is called the *procedural* interpretation of the equations. If

there exists a unique function that satisfies all the equations of the program, *and* if the above implementation terminates, then the result given by the implementation will be the correct value of the intended function.

In the case of the factorial, the successive steps in the evaluation of $fac(2)$ might be

$$\begin{aligned} fac(2) &= 2 \times fac(2-1) && \{\text{since } 2 > 0\} \\ &= 2 \times fac(1) && \{\text{since } 1 > 0\} \\ &= 2 \times 1 \times fac(1-1) && \{\text{calculation}\} \\ &= 2 \times fac(0) && \{\text{substitution}\} \\ &= 2 \times 1 && \{\text{calculation}\} \\ &= 2 \end{aligned}$$

But the promise of functional programming is that the programmer need not be concerned with the execution details, and can check each equation separately as a true statement of a property of the function in question. In this view, the set of simultaneous equations shown in Example 9.4.1 is regarded not as a definition of the factorial, but rather as true and provable facts about a function already known to mathematics – in this case it is the one defined as the product of all numbers up to n. This view of programming is known as *declarative*. Its use is not confined to functional programs: logic programming and even imperative programming can share the same benefit. This section investigates the circumstances in which programs can be successfully specified in the declarative style as the solutions of simultaneous equations. It highlights the problem of mismatch of declarative and procedural interpretations, and works towards their reconciliation.

In standard mathematics, a set of simultaneous equations contains n equations for n unknowns, and each unknown occurs (by itself) on the left hand side of exactly one equation. Of course, any unknown may also occur any number of times on the right hand side of any number of the equations. Fortunately, all the important ideas can be illustrated on an example with only two unknowns and only two equations

$$\begin{aligned} X &= P(X, Y) \\ Y &= Q(X, Y) \end{aligned}$$

In an imperative programming language, such a set of equations defines X and Y by *mutual* recursion, because each of the programs X and Y is called from within the body of the other. We will use Tarski's analysis [175] to show the existence of a weakest solution to such mutual recursions.

As always, we start with the assumption that each variable (X, Y) ranges over a complete lattice, and that all the functions involved are monotonic in all

their operands. The main theorem that we need is

Theorem 9.4.2 (Product of complete lattices)

If A and B are complete lattices, so is their Cartesian product $A \times B$. The ordering is defined

$$(X1, Y1) \sqsubseteq_{A \times B} (X2, Y2) \quad \textbf{iff} \quad X1 \sqsubseteq_A X2 \textbf{ and } Y1 \sqsubseteq_B Y2 \qquad \square$$

In the Cartesian product space, the pair of equations can be expressed as a single equation

$$(X, Y) \;=\; (P(X, Y), Q(X, Y))$$

Furthermore the right hand side is monotonic in X and Y, because both P and Q are. It therefore has a weakest fixed point, which is also a pair. The pair consists of values of X and Y respectively which solve the original simultaneous equations. The same construction works for any number of equations; it also gives the same answer if the equations are replaced by inequations, for example

$$X \;\sqsupseteq\; P(X, Y)$$
$$Y \;\sqsupseteq\; Q(X, Y)$$

Programming in the declarative style does not necessarily restrict itself to n equations for n unknowns. The example of the factorial already uses two equations for a single unknown. In general, a greater number of equations leads to an *over-determined* or *inconsistent* system, for which there is *no* solution, not even in a complete lattice. Solubility can be restored by the trick suggested above: replace all the equations by inequations

$$X \;\sqsupseteq\; P(X, Y)$$
$$X \;\sqsupseteq\; Q(X, Y)$$

By the definition of the least upper bound in a lattice, two equations like this can always be amalgamated into the single equation

$$X \sqsupseteq P(X, Y) \sqcup Q(X, Y)$$

Thus the number of inequations can be reduced to the number of unknowns, which guarantees their consistency and solubility. This treatment of over-determinacy is the one that is most consistent with the declarative interpretation of programs.

But the price of consistency is that the top of the lattice has to be acceptable as a valid solution. For example, the inequation

$$X \sqsupseteq (x := 3) \sqcup (x := 4)$$

has the miracle $\top$ as its weakest solution. Although the theory has shown the

existence of a solution within the complete lattice, this solution does not satisfy the healthiness conditions that are necessary for implementability. Any attempt to base an implementation on the given inequations must (sometimes at least) give an answer that does not satisfy them. This is an explanation of the notorious gap between the declarative semantics and the procedural semantics of many implemented declarative programming languages.

Exercise 9.4.3

Explore the consequence of turning the inequations the other way, and of taking the *strongest* fixed point of the inequations

$$X \sqsubseteq P(X, Y)$$
$$Y \sqsubseteq Q(X, Y)$$

Under what circumstances is the result an unimplementable program? □

In many cases of individual programs, the problem of implementability does not arise. For example, consider the pair of inequations

$$X \sqsupseteq (b \Rightarrow P) \qquad\qquad X \sqsupseteq (\neg b \Rightarrow Q)$$

The declarative and the procedural semantics of these inequations are the same, because when they are combined into a single equation, the result is the perfectly implementable

$$X \sqsupseteq (P \lhd b \rhd Q)$$

Many higher level declarative languages introduce compile-time checkable syntactic constraints which reconcile procedural and declarative semantics in a similar way. But we will take the complementary approach of giving a realistic abstract semantics to the procedural interpretation of the program. This will enable the programmer to prove of a particular program that the procedural and the declarative interpretations coincide, or at least that the executed program anyway meets some independently formulated specification.

The main problem in matching a procedural reading with a declarative reading of a set of equations is surprisingly simple. According to the declarative reading, the inequation

$$X \sqsupseteq X$$

is always true, so its addition to or removal from any set of inequations makes absolutely no difference. But for an implementation, the effect of adding this vacuous equation in general would be disastrous. The computer would loop forever, replacing one side of the inequation (X) by the other (X). If we want to reason about the actual effect of executing a program that is specified by general inequations,

we are forced to move at least a short way from the purely declarative ideal.

The central assumption of the declarative reading is that all the inequations of the program, like all the requirements in a specification, are connected simply by conjunction. But $X \sqsupseteq X$ is a unit of conjunction, and all our problems arise from that. They are the same problems as those of Section 7.1, where a solution was given in the form of a strict parallel combinator to replace the non-strict conjunction. In this section we adopt the same solution. To begin with, we need to write dashes on all the variables on the left hand sides of the inequations, for example

$$X' \sqsupseteq P(X, Y)$$
$$Y' \sqsupseteq Q(X, Y)$$
$$X' \sqsupseteq R(X, Y) \qquad \text{etc.}$$

Each inequation thereby takes the form of an assignment, as defined in Section 9.1. The conjunction of all the inequations is therefore a simultaneous assignment to all the variables. The programs specified by the set of inequations are just the weakest ones that are unchanged by execution of the assignment.

If each identifier occurs exactly once on the left hand side of an assignment, nothing has been gained (or lost) by the more elaborate construction. But if the the same dashed variable appears in two or more inequations, we have a potential conflict. As in the case of concurrency, this must be resolved by applying an associative merging operator $\oplus$ on the values contributed by two assignments. To deal with the original problem of $X' \sqsupseteq X$, $\oplus$ must be strict. It must also be idempotent: repetition of an equation has no effect. In a language which assigns no significance to the ordering between clauses, $\oplus$ will be commutative, but we will also show how to deal with languages that give some sort of precedence to the earlier clauses.

The final important property of $\oplus$ is that it should have a unit – at least a left unit. This is introduced to deal with the case that a dashed variable never appears on the left hand side of any of the inequations. On executing the inequations, an implementation can clearly detect this case, and report it to the user or programmer, maybe with a request for further instructions. This phenomenon is known as *finite* failure, to distinguish it from the infinite failure which results from an ill-designed recursion, and which can never be detected by the computer itself. We have already encountered finite failure which results from parallelism; in a reactive system it takes the form of deadlock. We will therefore represent it by the familiar $STOP$

$$STOP \quad =_{df} \quad wait := true$$

This involves introducing the variable *wait* into the alphabet, together with the healthiness conditions

W1 $P = I\!I \lhd wait \rhd P$

W2 $P = P \vee (P \wedge wait' \wedge v' \neq v); \mathbf{true}$

where the healthiness condition **W2** ensures that the program variables remain unchanged when P stops. However, there is no need for ref or tr. And we will forbid $STOP$ to appear explicitly in any program: its only use is in the context of Definition 9.4.4 below.

A first candidate for selection as the merge operator is the non-deterministic choice $\sqcap$. This has all the required algebraic properties. But it has to be rejected because its unit is the unimplementable $\top$. The closest available alternative is the $\|$ operator defined in Section 8.2; its unit is the perfectly implementable $STOP$. This gives us an excellent procedural interpretation of the conditional equations which feature so strongly in declarative programming.

Definition 9.4.4 (Conditional equation)

$$(X = P \text{ if } b) =_{df} X' \sqsupseteq (P \lhd b \rhd STOP) \qquad \square$$

The assembly of many such clauses is denoted by $\|$.

Definition 9.4.5 (Assembly)

$$P \| Q =_{df} P \|_M Q$$

where $$M =_{df} X' \sqsupseteq (0.X \| 1.X) \wedge Y' \sqsupseteq (0.Y \| 1.Y) \wedge \ldots. \qquad \square$$

This encoding appears to solve all remaining semantic problems, as shown in the following examples, where P and Q are programs free of finite failures, that is they do not contain $STOP$.

Example 9.4.6 (Non-determinism)

$$(X' \sqsupseteq P \parallel X' \sqsupseteq Q) = (X' \sqsupseteq (P \| Q)) = (X' \sqsupseteq (P \sqcap Q))$$

Unconditional equations lead to a non-deterministic choice. $\square$

Example 9.4.7 (Determinism)

$$(X = P \text{ if } b) \| (X = Q \text{ if } \neg b) = (X' \sqsupseteq (P \lhd b \rhd Q))$$

Conditional equations give the power to avoid non-determinism. $\square$

Example 9.4.8 (A mixture)

$$(X = P \text{ if } b) \| (X = Q \text{ if } c)$$
$$= (X' \sqsupseteq (P \sqcap Q) \lhd b \wedge c \rhd (P \lhd b \rhd (Q \lhd c \rhd STOP)))$$

If b (or c) is false, it inhibits selection of P (or Q). If they are both true, the choice between them is non-deterministic. In logic programming, this is *committed*

non-determinism. Once the choice is made, it cannot be reversed by backtracking. □

Example 9.4.9 (Divergence)

$$(X' \sqsupseteq X \parallel X' \sqsupseteq x := 3) \;=\; (X' \sqsupseteq (X \parallel x := 3))$$

The weakest fixed point of any set of disjoint inequations containing this one will assign the bottom value **true** to X. □

In a functional programming language, the intrusion of non-determinism leads to the severe logical problems discussed in Section 9.3. The solution suggested there places responsibility for avoiding non-determinism upon the programmer. But the simplest way of avoiding a risk is to avoid all notations which entail that risk, and obviously the merging operator $\parallel$ must be avoided too. For this reason, many functional programming languages give significance to the ordering between the clauses of a program. If several of the conditions of different clauses are all true, the first of the clauses is always selected. The relevant merging operator is similar to the overriding operator of Z, which extends the domain of a function, and we denote it by $\bullet\!\parallel$. We require it to obey the algebraic law

$$(P \lhd b \rhd STOP) \bullet\!\parallel (Q \lhd c \rhd STOP) \;=\; P \lhd b \rhd (Q \lhd c \rhd STOP)$$

provided that P is free of finite failure.

The idea of the definition of $P \bullet\!\parallel Q$ is that the $STOP$ of P is treated as a jump to Q, which is otherwise not executed.

Definition 9.4.10

$$P \bullet\!\parallel Q \;=_{df}\; \Pi \lhd wait \rhd (P;\, ((wait := false;\, Q) \lhd wait \rhd \Pi))$$ □

This definition uses the healthiness condition **W2** to ensure that the initial values of program variables will pass to program Q after P gets deadlocked.

Exercise 9.4.11

Prove that $\bullet\!\parallel$ enjoys the following algebraic laws.

L1 $P \bullet\!\parallel P \;=\; P$

L2 $(P \bullet\!\parallel Q) \bullet\!\parallel R \;=\; P \bullet\!\parallel (Q \bullet\!\parallel R)$

L3 $STOP \bullet\!\parallel P \;=\; P \;=\; P \bullet\!\parallel STOP$

L4 $CHAOS \bullet\!\parallel P \;=\; CHAOS$

L5 $P \bullet\!\parallel Q \;=\; P$, whenever P is free of finite failure

L6 $P \parallel Q \;=\; (P \bullet\!\parallel Q) \sqcap (Q \bullet\!\parallel P)$ □

Exercise 9.4.12

The asymmetry of $\bullet|$ is displeasing to declarative programmers. Define an alternative operator that punishes non-determinism by abortion, and prove its algebraic properties. Hint: Define a function that leaves a wholly deterministic predicate unchanged, but maps every other predicate to $\perp$. □

Most declarative programming languages offer considerable flexibility in writing the left hand sides of the equations. For example, the single high order variable can be followed by a list of formal parameters, as in

$$X(n) \;=\; n \times X(g(n))$$

As a declaration, this equation is intended to be true of all n. Consequently, it is just another way of writing

$$X \;=\; \lambda n : val(\mathcal{N}) \bullet n \times X(g(n))$$

Certain kinds of function are often allowed in the formal parameter list, for example

$$X(f(n)) \;=\; (n+1) \times X(n)$$

Here again, restrictions are imposed to ensure implementability. The function f must have an easily testable image and an easily computable inverse. In that case, the equation given above is equivalent to

$$X \;=\; \lambda n : val(\mathcal{N}) \bullet (f^{-1}(n)+1) \times X(f^{-1}(n)), \qquad \text{if } n \in image(f)$$

These restrictions in many languages require f to be a constructor for some data type, perhaps one defined by the programmer.

The original idea of declarative programming is that programs can serve as their own specifications, so that doubts about program correctness are logically inconceivable. Programming languages designed in pursuit of this ideal tend to be clear and simple and easy to use. Particular attention is given to rigorous compile-time checking of data types, often defined by recursion and implemented by automatic storage allocation and deallocation. Even if the original idea of declarative programming is not achieved, it certainly eases the task of programming. It does this by splitting the design process into clearly defined stages. The first is to construct a set of inequations which, in their declarative interpretation, correctly describe the intended function. The second stage is to prove that the procedural interpretation of the same program text gives the same meaning as the declarative. At either stage, the program may be made more efficient, though perhaps less clear, by correctness-preserving transformations. The current section has given an unusually abstract and general treatment of the procedural semantics, which facilitates proof of the relevant algebraic laws. An alternative treatment can

be based on an operational semantics, which treats each equation of the program as defining a step in its execution, whereby an appropriate chosen instance of the left hand side of the equation is replaced by an appropriate modified copy of the right hand side. Clearly, this operational semantics must correspond correctly to the other models of semantic presentation; that is the topic of the next chapter.