Chapter 6

Implementation

The main goal of a theory of programming is to help in the design of programs that meet their intended specification. But the whole exercise would be pointless unless there is some effective mechanism for implementing the program once it has been written. And it would be even worse than pointless, perhaps even dangerous, if the implementation itself is not correct. To help in the design of correct implementations is the main goal of this chapter.

Fortunately, we can reuse exactly the same approach to correctness of implementations as we have to correctness of programs. But now it is the program that plays the role of the specification; its implementation is also described by a predicate satisfying even more stringent implementation-oriented constraints, and it is correct if it logically implies the program. The implementation predicate will usually have a different alphabet from that of the program, so that it can describe additional details of the internal working of the implementation. In the case of computers, these will include registers and other storage devices not relevant to an abstract high level programming language. A simulation is often needed to relate the two levels of abstraction, as described in Example 4.4.7.

The construction of an efficient program to meet a general specification is a task that is usually guided by human insight, ingenuity and inventiveness. But the transition between a program and its implementation is much more routine; in fact it can be completely automated. This is done by subjecting the program to a series of transformations, whose validity is put beyond doubt by mathematical proof. The correctness of the implementation is therefore guaranteed by a kind of normal form theorem, which shows that every program can be transformed automatically into an implementation that implies it as a specification.

There are two familiar ways of implementing a program, by compilation or by interpretation. Interpretation executes the text of the program directly, and is described in Section 6.3. Compilation first translates the program into the machine

code of some computer, which then executes it more quickly by hardware. All the high level control structures of the language have to be replaced by machine code jumps. The principles of correct compiler design are described in Section 6.2. Both implementation methods split the execution of a program into a large number of steps, and introduce a special control variable to select the action to be performed on each step. This basic idea is described in Section 6.1, where the relevant theory is developed.

The final section of this chapter develops a theory for a language which combines high level structures with explicit jumps that lead directly from one part of the program to a labelled statement in some other part. Provided that there are no jumps into the middle of a structure, the complexity of adding this new feature is not serious, and none of the laws for the original language is invalidated.

6.1 Execution

The execution of a computer program is normally delegated to an automatic device, which splits the task into a collection of atomic actions, and performs them one after another in sequence. The potential for parallel execution will be described in later chapters: in this chapter we will concentrate on programs whose execution can be expressed as a sequential repetition of an individual step. This step is described in the usual way by a predicate relating the values of the variables before each action to the values after that action. The predicate describes the behaviour of an assembled block of machine code instructions. Each instruction too is described by a predicate, and their combined effect is given by an assembly operation on these predicates. Assembly into disjoint regions of machine store is the only form of composition for machine code instructions. In this section we explore its simple mathematical properties. In the next section we will see how it can be used in various ways to implement high level control structures of a programming language.

To define the total behaviour of the executing mechanism, it is necessary also to specify the circumstances under which it terminates. A perfectly general and convenient way of doing this is to introduce into the alphabet of the step a specially named variable (say l), which serves as a *control* variable. Before each repetition of the action, the value of l is tested. As long as it remains within some designated set of values, the execution continues; otherwise it terminates. We will take the set αl of continuation points and the choice of the name l of the control variable as an inseparable and unchangeable part of the description of the implementation; for this reason we include both l and αl as part of the alphabet of the predicate describing the step. This convention greatly simplifies the statement and use of the relevant algebraic laws.

Definition 6.1.1 (Continuations and execution)

Let P be a predicate describing a step, and let a special variable named l in the alphabet of P be reserved to denote the control variable for its execution. Then $\alpha l P$ denotes the set of *continuations* of P. The *execution* of P is defined as a loop, which iterates the step as long as l remains in the continuation set.

$$P^* =_{df} (l \in \alpha l P) * P \qquad \square$$

The control variable l may be used within the step P in the same way as any other variable in its alphabet. For example, l may be used initially to select which of the many possible actions described by P will be executed on this iteration, and an assignment of a new value to l provides the way in which one action can nominate the action that will be performed on the next iteration of the step.

Example 6.1.2 (Machine code)

Let $P = (A, l := 27, l + 1) \lhd l = 20 \rhd (ram[56],\ l := A,\ l + 1 \lhd l = 21 \rhd I\!I)$

and $\alpha l P = \{20, 21\}$

This describes the behaviour of a small fragment of a machine code program, consisting of just two instructions stored in locations 20 and 21 of the program store of the machine. The control variable l stands for the program pointer (sequence control register). If its initial value is 20, the first instruction loads a constant value 27 into register A, and adds one to the program pointer, thereby ensuring that the next instruction will be taken from location 21. But if the fragment of program is entered by a jump, the initial value of l may be 21, so only the second instruction is executed. In either case, the second instruction stores the value of A in location 56 of the random access memory ram; it also sets l to 22, which is outside $\alpha l P$, and as a result, execution of the entire fragment of program terminates. The overall effect might be described in a high level language as a simple assignment

$$x := 27 \qquad \square$$

Example 6.1.3 (Algebraic normal form)

In Chapter 5, it was shown how any program can be reduced to the form

$$\sqcup_l (b(l) \vee P(l))$$

where the indexing variable has been selected as the control. As explained in Section 5.6, this normal form can be executed by iteration of the step

$$S =_{df} ((l := l + 1) \lhd b(l) \rhd (P(l); l := -1)) \lhd l \geq 0 \rhd I\!I$$

Execution terminates with $l = -1$; all non-negative integers are continuations

$$\alpha l S \;=\; \mathcal{N}$$

This example shows in principle that every program can be expressed as S^* for some step S. □

Example 6.1.4 (Symbolic interpreter)

In a symbolic interpreter, execution is controlled by the text of the program being interpreted. So it is convenient to arrange that the control variable l ranges over all sequences of program texts expressible in the language. The continuations consist of all sequences of program texts except the empty sequence, which triggers termination of the interpreter. This example is elaborated in Section 6.3. □

What happens if the initial value of l lies outside the continuation set $\alpha l P$? In this case, the effect of the whole execution is to do nothing, as expressed by the idempotence principle

$$P^* \;=\; P^* \lhd l \in \alpha l P \rhd I\!I$$

The same question may be asked about the individual step P. Since P is never even started with l outside its continuation set, it really does not matter what answer is given. However, to simplify the test of equality between predicates describing the step, it is convenient to standardise the behaviour of P even in this impossible case, and for reasons which will soon become apparent, we will choose the behaviour of $I\!I$ as standard. We therefore stipulate that each step must satisfy the same idempotence principle as the whole execution.

Definition 6.1.5 (Step)

A predicate P is a *step* if $l \in \alpha P$ and

$$P \;=\; (P \lhd l \in \alpha l P \rhd I\!I)$$

An immediate consequence is

$$(l \notin \alpha l P)_\perp ; P \;=\; (l \notin \alpha l P)_\perp$$ □

In this chapter we will confine attention to programs which satisfy this definition. We shall want to define particular steps using all the operators of our programming language except μ, and for this purpose, we need appropriate definitions of the continuation sets for the result of each operation, and a closure theorem that guarantees that their results still satisfy the defining condition for a step.

Definition 6.1.6 (Continuations of operators)

$$\alpha l(P \, op \, Q) \;=_{df}\; (\alpha lP \cup \alpha lQ), \qquad \text{where } op \in \{;, \sqcap, \lhd b \rhd\} \qquad \square$$

Theorem 6.1.7 (Step closure)

If P and Q are steps, then

(1) $P;Q$ is a step.

(2) $P \sqcap Q$ and $P \lhd b \rhd Q$ are also steps whenever $\alpha lP = \alpha lQ$.

(3) The set of steps $\{P \mid \alpha lP = L\}$ is a complete lattice.

Proof of (1) Notice that $l \notin \alpha l(P;Q) \;\Rightarrow\; (l \notin \alpha lP) \wedge (l \notin \alpha lQ)$.

$$\begin{array}{lll} & P;Q & \{2.1\mathbf{L1}\} \\ = & (P;Q) \lhd l \in \alpha l(P;Q) \rhd (P;Q) & \{(l \notin \alpha lP)_\perp ; P = (l \notin \alpha lP)_\perp\} \\ = & (P;Q) \lhd l \in \alpha l(P;Q) \rhd Q & \{(l \notin \alpha lQ)_\perp ; Q = (l \notin \alpha lQ)_\perp\} \\ = & (P;Q) \lhd l \in \alpha l(P;Q) \rhd \Pi & \square \end{array}$$

This theorem justifies the use of all familiar programming notations for steps as for complete programs. However, we still have the freedom (and duty) to assign continuation alphabets to the primitive components of a program. This can be done either explicitly or implicitly by a definition which takes the form of the idempotence principle of Definition 6.1.5.

Example 6.1.8

We adopt the definition $\alpha l\Pi =_{df} \{\}$. This convention maintains validity of the unit laws

$$\Pi;P \;=\; P \;=\; P;\Pi$$

Furthermore it ensures that the execution of Π also has no effect

$$\Pi^* \;=\; (l \in \{\}) * \Pi \;=\; \Pi \qquad \square$$

Example 6.1.9

$$\begin{array}{rll} s : \mathtt{jump}\ f & =_{df} & (l := f \lhd l = s \rhd \Pi) \\ \alpha l(s : \mathtt{jump}\ f) & =_{df} & \{s\} \end{array}$$

This notation describes a machine code jump instruction with destination f, which is stored in location s. The classical "tight loop" of machine code is represented

$$(s : \mathtt{jump}\ s)^* \;=\; (l \neq s)_\perp$$

This simple mathematical calculation gives the appropriate warning against allowing control to reach such an instruction. $\square$

The following law is used to peel off the first action of the execution of a step.

Lemma 6.1.10

$P^* = P; P^*$

Proof

$$\begin{aligned} & P^* && \{\text{fixed point}\} \\ = \ & (P; P^*) \lhd l \in \alpha lP \rhd I\!I && \{2.1\mathbf{L6}\} \\ = \ & (P; P^*) \lhd l \in \alpha lP \rhd (l \notin \alpha lP)_\perp && \{5.5\mathbf{L2} \text{ and Def. } 6.1.1\} \\ = \ & (P; P^*) \lhd l \in \alpha lP \rhd P^* && \{2.2\mathbf{L2} \text{ and Def. } 6.1.5\} \\ = \ & P; P^* && \square \end{aligned}$$

If a step is guaranteed to assign to l a value outside the continuation set, then the step will be executed exactly once, and this condition is also necessary.

Lemma 6.1.11

$P^* = P$ **iff** $P = P; (l \notin \alpha lP)_\perp$

Proof

$$\begin{aligned} & P = P; (l \notin \alpha lP)_\perp \\ \Rightarrow \ & P = P; (l \notin \alpha lP)_\perp \ \wedge \\ & P; P^* = P; (l \notin \alpha lP)_\perp; P^* && \{5.5\mathbf{L2}\} \\ \Rightarrow \ & P = P; (l \notin \alpha lP)_\perp \ \wedge \\ & P; P^* = P; (l \notin \alpha lP)_\perp && \{\text{Lemma } 6.1.10\} \\ \Rightarrow \ & P = P^* && \{\text{Corollary of } 5.5\mathbf{L4}\} \\ \Rightarrow \ & P = P; (l \notin \alpha lP)_\perp && \square \end{aligned}$$

From Lemma 6.1.11 and Corollary of 5.5**L4** we establish an idempotence law for execution.

Theorem 6.1.12

$(P^*)^* = P^*$ □

Two steps are said to be *disjoint* if their execution can never be interleaved, in the sense that neither of them can be started until the other has finished. This can be guaranteed by a simple condition on their alphabets: it ensures that in any circumstance at most one of the two steps can be executed, and the value of l determines which.

Definition 6.1.13 (Disjointness)

Two steps P and Q are *disjoint* if their continuations are disjoint, that is

$$\alpha lP \cap \alpha lQ = \{\}$$ □

Two blocks of machine code program which occupy disjoint areas of store will be disjoint in this sense. Two such blocks can be joined together by an assembly operation defined as follows.

Definition 6.1.14 (Assembly)

Let P and Q be disjoint steps. Define

$$\begin{array}{rcl} P \| Q & =_{df} & (P \lhd l \in \alpha l P \rhd Q) \lhd (l \in \alpha l P \cup \alpha l Q) \rhd I\!I \\ \alpha l(P \| Q) & =_{df} & \alpha l P \cup \alpha l Q \end{array}$$ □

Because of disjointness, this operator (when defined) enjoys extra algebraic properties, though for the same reason, it cannot be idempotent.

Theorem 6.1.15 (Algebraic properties of assembly)

$\|$ is associative and commutative and has unit $I\!I$. □

Execution of an assembly consists of an interleaving of the executions of its operands. Because of disjointness, each operand is repeated as many times as possible before the other one can start again.

Theorem 6.1.16 (Execution of an assembly)

$$(P \| Q)^* \;=\; (P^* \| Q)^* \;=\; (P^* \| Q^*)^*$$

Proof Let $b =_{df} (l \in \alpha l P \cup \alpha l Q)$ and $c =_{df} (l \in \alpha l P)$. Define

$$\begin{array}{l} F(X) =_{df} ((P \| Q); X) \lhd b \rhd I\!I \\ G(X) =_{df} ((P^* \| Q^*); X) \lhd b \rhd I\!I \end{array}$$

So $\mu F = (P \| Q)^*$ and $\mu G = (P^* \| Q^*)^*$.

From 5.5**L5** we conclude that

$$\mu F \;=\; ((l \in \alpha l P) * (P \| Q)); \mu F \;=\; P^*; \mu F \qquad (\dagger)$$

$$\mu G \;=\; ((l \in \alpha l P) * (P^* \| Q^*)); \mu G \;=\; (P^*)^*; \mu G \;=\; P^*; \mu G \qquad (\ddagger)$$

$$\begin{array}{lll} & \mu F & \{\text{fixed point}\} \\ = & F(\mu F) & \{\text{def of } F \text{ and } 2.2\mathbf{L2}\} \\ = & (P; \mu F \lhd c \rhd Q; \mu F) \lhd b \rhd I\!I & \{(\dagger)\} \\ = & (P; P^*; \mu F \lhd c \rhd Q; Q^*; \mu F) \lhd b \rhd I\!I & \{\text{Lemma 6.1.10 and } 2.2\mathbf{L2}\} \\ = & G(\mu F) & \end{array}$$

Using ($\ddagger$) we can show

$$\mu G \;=\; F(\mu G)$$

in a similar way. The conclusion $(P\|Q)^* = (P^*\|Q^*)^*$ follows from the weakest fixed point theorem. Furthermore from Theorem 6.1.12 it follows that

$$(P^*\|Q)^* = ((P^*)^*\|Q^*)^* = (P^*\|Q^*)^* \qquad \square$$

An even stronger disjointness condition on two steps is that one of them cannot be started after the other is terminated.

Definition 6.1.17

If P and Q are disjoint, then Q is said to *inhibit* P if

$$Q = (Q;(l \notin \alpha l P)_\perp) \lhd l \in \alpha l Q \rhd I\!I \qquad \square$$

Example 6.1.18 (Machine code)

A block of machine code inhibits another if it contains no jumps to any location within the other. $\square$

Lemma 6.1.19

If Q inhibits P, then

$$(l \notin \alpha l P)_\perp; Q^* = (l \notin \alpha l P)_\perp; Q^*; (l \notin \alpha l P \cup \alpha l Q)_\perp$$

Proof From Definition 6.1.17 and Exercise 5.5.1(2). $\square$

In the translation of a high level symbolic program to machine code, the only operation available is assembly of larger blocks of code out of smaller. The effect of the various high level program combinators must be achieved by placing constraints on jumps between the blocks, as shown by the following theorem

Theorem 6.1.20 (Sequential assembly)

If Q inhibits P then

$$(P\|Q)^* = P^*; Q^*$$

Proof

$$\begin{aligned}
& (P\|Q)^* && \{5.5\mathbf{L5}\}\\
= \; & (l \in \alpha l P) * (P\|Q)\,; \\
& (l \in \alpha l Q) * (P\|Q)\,; (P\|Q)^* && \{P \text{ and } Q \text{ are disjoint}\}\\
= \; & P^*; Q^*; (P\|Q)^* && \{\text{Corollary of } 5.5\mathbf{L4}, \text{ Lemma } 6.1.19\}\\
= \; & P^*; Q^*; (l \notin \alpha l (P\|Q))_\perp; (P\|Q)^* && \{5.5\mathbf{L2}\}\\
= \; & P^*; Q^* && \square
\end{aligned}$$

Theorem 6.1.21 (Complete disjointness)

If P inhibits Q and Q inhibits P then

$$P^* \| Q^* \;=\; (P \| Q)^*$$

Proof Let $b \;=\; (l \in \alpha lP \cup \alpha lQ)$.

$$\begin{aligned}
& P^* \| Q^* && \{\text{Def. 6.1.14}\} \\
=\; & (P^* \lhd l \in \alpha lP \rhd Q^*) \lhd b \rhd I\!I && \{P \text{ and } Q \text{ are disjoint}\} \\
=\; & (((l \notin \alpha lQ)_\perp ; P^*) \lhd l \in \alpha lP \rhd ((l \notin \alpha lP)_\perp ; Q^*)) \\
& \lhd b \rhd I\!I && \{\text{Lemma 6.1.19}\} \\
=\; & ((P^* ; (\neg b)_\perp) \lhd l \in \alpha lP \rhd (Q^* ; (\neg b)_\perp)) \lhd b \rhd I\!I && \{\text{Def. 6.1.14 and 2.2}\mathbf{L2}\} \\
=\; & (P^* \| Q^*) ; (\neg b)_\perp
\end{aligned}$$

From Lemma 6.1.11 and Theorem 6.1.16 it follows that

$$P^* \| Q^* \;=\; (P^* \| Q^*)^* \;=\; (P \| Q)^* \qquad \square$$

The representation of program execution as the repetition of members of a specified set of steps is the basic idea behind the UNITY model of parallel computation [38] and also of action systems [13]. In UNITY, there is an additional fairness constraint that insists that no selectable step is infinitely often rejected in any execution sequence. In the case of disjointness, fairness is automatic, because there is only one selectable step. We will also use disjointness and inhibition to ensure that the single operation of assembly can correctly implement a range of different program structures in Section 6.2.

Exercises 6.1.22

(1) Prove $(l \in \alpha lP) * (P \| Q) \;=\; P^*$.

(2) Prove that if both P and Q inhibit R, so does $P \| Q$. $\square$

6.2 Compilation

The efficient execution of a computer program is normally preceded by a preliminary transformation (compilation) of the program into a target program, expressed in the idiosyncratic machine code of the computer that is to execute it. The machine code usually does not include mathematical and program structuring features such as arithmetic expressions, conditionals or iterations. Instead, it provides simple arithmetic operations, and a selection of conditional or unconditional jumps that may be used in combination to achieve the same effect. In this section we explore a range of valid transformation rules whose repeated application will

achieve the task of program compilation, without changing the meaning of the program. We shall concentrate on compilation of control structures. Translation of the primitive assignments has been treated in Section 2.9, and translation of symbolic addresses to machine variables has been introduced in Section 4.4.

An obvious obligation on the compiler designer is to know the meaning of all the machine code instructions of the target computer. Fortunately, the effect of each instruction is easily defined using the standard familiar programming notations, particularly assignments and conditionals. The designers of the computer hardware are willing to accept such descriptions as a specification of what is to be implemented on silicon, and the correctness of the hardware implementation is not the responsibility of the designer of a compiler.

Examples 6.2.1

Consider a simple machine with a single register A. An instruction to load the constant n into this register is specified

$$\texttt{LDL}\ n \ =_{df}\ (A, l\ :=\ n, l+1), \qquad \text{where } 0 \leq n < 64$$

This instruction itself occupies one location: so incrementation of l ensures it will point to the next instruction when this one is complete. Similarly a store instruction can be defined

$$\texttt{STO}\ b \ =_{df}\ (ram[b], l := A, l+1)$$

Arithmetic instructions take the form

$$\texttt{ADD}\ b \ =_{df}\ (A, l\ :=\ A + ram[b], l+1)$$

A conditional jump instruction skips over the n following locations if A is negative

$$\texttt{cjump}\ n \ =_{df}\ (l\ :=\ (l+n+1 \lhd A < 0 \rhd l+1)) \qquad \square$$

A single machine code instruction stored in location m of the code store is represented by a step which has the singleton set $\{m\}$ as its continuation set, for example

$$\texttt{LDL}\ 27 \lhd l = m \rhd I\!I$$

We will introduce a special notation for this case.

Definition 6.2.2 (Single instruction)

If INST is a machine code instruction, then

$$m : \texttt{INST} \ =_{df}\ \texttt{INST} \lhd l = m \rhd I\!I$$

is called a single instruction. $\square$

A block of machine code is defined in terms of its component instructions. For example,

$$(20 : \texttt{LDL 27} \parallel 21 : \texttt{STO 56})$$

is a block of two instructions. After expansion of the definitions, it is shown to be the same as Example 6.1.2.

Definition 6.2.3 (Machine code block)

A *machine code block* is a program expressible as an assembly of single instructions

$$S_0 \parallel S_1 \parallel \ldots \parallel S_n$$ □

A machine code block can in general be entered at any of its constituent continuation points. In practice, it is beneficial to designate one of these as its normal *start point*, and call it by the standard name s. This singles out the instruction that will be activated when control passes sequentially into the code; any other point would have to be entered by a jump. Similarly, we single out the standard *finishing point* (named f) of a block as the value of l when control leaves the block normally, that is not by a jump.

The start point will usually be the first of the locations in which the code is stored, and the finishing point will be the location just following the code. The code will usually be packed into contiguous locations; all its other continuations lie numerically between s and f so that their difference $(f - s)$ gives a count of the length of the code. Our theory will not depend on a contiguous range of addressing for the program store. We will use symbolic values for continuation points, relying only on the reasonable hypothesis that there is an unbounded supply of them.

In any machine code program there will be many blocks which are never entered by a jump instruction, nor do they use a jump instruction to exit. An obligation to terminate normally through the end f is expressed by the assertion $(l = f)_\perp$, and the assumption of normal entry through the beginning s is expressed $(l = s)^\top$. Blocks of code that have these as postcondition and precondition will be called *structured*, by analogy with the recommended programming practice of avoiding explicit jumps.

Definition 6.2.4 (Structured block)

A *structured block* is a program of the form

$$(l = s)^\top; P^*; (l = f)_\perp$$

where P is a machine code block. The value s is called its start point and f is its finishing point. □

Example 6.2.5 (Identity)

There is only one useful structured block for which the start and finish are the same. Since it occupies no storage the only thing that it can do is nothing.

$$(l = s)^{\top}; \Pi^*; (l = s)_{\perp} \;=\; (l = s)^{\top}$$

This can be used to place a label s anywhere within the code. □

Example 6.2.6 (Abort)

$$(l = s)^{\top}; (s : \mathtt{jump}\ s)^*; (l = f)_{\perp} \;=\; (\perp \lhd l = s \rhd \top)$$ □

Example 6.2.7

A step with only one continuation point can never be entered in the middle by a jump. So a block consisting of a single instruction is easy to convert to structured form, for example

$$(l = 20)^{\top}; (20 : \mathtt{LDL}\ 27)^*; (l = 21)_{\perp}$$ □

Example 6.2.8

A block which has been translated from a single assignment will usually be structured. For example, the assignment

$$ram[b] \;:=\; ram[a] + 27$$

could be translated to

$$(l = 20)^{\top}; (20 : \mathtt{LDL}\ 27 \| 21 : \mathtt{ADD}\ a \| 22 : \mathtt{STO}\ b)^*; (l = 23)_{\perp}$$ □

The form of a structured block has been defined to model closely the behaviour of an actual computer executing a block of machine code instructions. The task of a compiler faced with an arbitrary source program P is to compile it to a target program $\hat{P}$ expressed in the machine language of the computer, ensuring that $\hat{P}$ has the same effect as P (or better). The target program has l in its alphabet, but the source does not, so a declaration is needed to match the two sides of the inequation

$$P \;\sqsubseteq\; (\mathbf{var}\ l; \hat{P}; \mathbf{end}\ l)$$

We define our target code to match the right hand side of this inequation.

Definition 6.2.9 (Target code)

A program is in *target code* if it is expressed in the form

$$\begin{aligned} \langle s, Q, f\rangle &=_{df} \textbf{var } l; (l = s)^{\top}; Q^*; (l = f)_{\perp}; \textbf{end } l \\ &= \textbf{var } l := s; Q^*; (l = f)_{\perp}; \textbf{end } l \end{aligned}$$

where Q is a machine code block □

The fundamental theorem of compilation states that every program P can be expressed as target code. Its proof is by structural induction on P, based on a series of lemmas which display the transformations used by a compiler. For simplicity, declarations are omitted, and iteration is the only form of recursion.

Theorem 6.2.10 (Compilation)

For each program P expressed in the language of Chapter 5, and for any start point s and for any set L of continuation points (where $s \notin L$), there is a finish point f and a machine code block $\hat{P}$ such that $\alpha l \hat{P}$ is disjoint from L and

$$P \sqsubseteq \langle s, \hat{P}, f\rangle$$

Proof By structural induction, based on the following lemmas. □

The unit $I\!I$ can be translated either as an empty segment of code, or as a machine code jump. The jump may be anywhere and lead to anywhere else.

Lemma 6.2.11 (Skip)

$I\!I = \langle s,\ I\!I,\ s\rangle$

Proof

$$\begin{aligned} & \langle s,\ I\!I,\ s\rangle && \{\text{Lemma 6.1.11}\} \\ = \ & \textbf{var } l := s; I\!I; \textbf{end } l && \{2.9\textbf{L4}, \textbf{L6} \text{ and } \textbf{L8}\} \\ = \ & I\!I && \square \end{aligned}$$

Lemma 6.2.12 (Skip)

If $s \neq f$, then $I\!I = \langle s,\ (s : \texttt{jump}f),\ f\rangle$

Proof

$$\begin{aligned} & \langle s,\ (s : \texttt{jump } f),\ f\rangle && \{\text{Lemma 6.1.11}\} \\ = \ & \textbf{var } l := s; (s : \texttt{jump } f); (l = f)_{\perp}; \textbf{end } l && \{3.1\textbf{L3} \text{ and } 2.1\textbf{L5}\} \\ = \ & \textbf{var } l := f; (l := f)_{\perp}; \textbf{end } l && \{3.1\textbf{L3}, 2.9\textbf{L6} \text{ and } \textbf{L8}\} \\ = \ & I\!I && \square \end{aligned}$$

For all reasonable machines, a single assignment can be translated into a sequence of machine code instructions, but the compilation strategy and its proof must depend on details of the machine, so they are omitted.

Lemma 6.2.13 (Assignment)

For every s there is an f and a machine code block $\hat{P}$ such that

$$(v := e) \sqsubseteq \langle s,\ \hat{P},\ f\rangle \qquad \square$$

The compilation of structured statements depends on the fact that the compiler can arrange that the operands can be compiled into disjoint regions of program store with no jumps between them, and that a fresh unused location can be found whenever needed to store an extra compiled instruction. The general strategy is shown by compilation of a non-deterministic choice. Of course in practice no computer has in its repertoire the non-deterministic jump, so a more practical compiler would resolve the non-determinism by selecting and compiling just one of the alternatives.

Lemma 6.2.14 (Non-deterministic choice)

Let $C = s : (l := s_1 \sqcap l := s_2)$ and $s \notin \{s_1, s_2\}$
and P inhibits Q and C and $s_1 \notin \alpha l Q$
and Q inhibits P and C and $s_2 \notin \alpha l P$.
Then $\langle s_1, P, f\rangle \sqcap \langle s_2, Q, f\rangle = \langle s,\ P \| C \| Q,\ f\rangle$

Proof From Exercise 6.1.22(2) it follows that $P \| Q$ inhibits C.

$$\begin{array}{lll}
 & \langle s,\ P\|C\|Q,\ f\rangle & \{\text{Theorem 6.1.20}\} \\
= & \mathbf{var}\, l := s; C^*; (P\|Q)^*; (l = f)_\perp; \mathbf{end}\, l & \{\text{Lemma 6.1.11}\} \\
= & \mathbf{var}\, l := s; C; (P\|Q)^*; (l = f)_\perp; \mathbf{end}\, l & \{\text{Theorem 6.1.20}\} \\
= & \mathbf{var}\, l := s_1; (Q^*; P^*); (l = f)_\perp; \mathbf{end}\, l \sqcap & \{s_1 \notin \alpha l Q,\ s_2 \notin \alpha l P \\
 & \mathbf{var}\, l := s_2; (P^*; Q^*); (l = f)_\perp; \mathbf{end}\, l & \text{and } 5.5\mathbf{L2}\} \\
= & \langle s_1, P, f\rangle \ \sqcap\ \langle s_2, Q, f\rangle & \square
\end{array}$$

The treatment of the conditional is similar to that of non-determinism. The two operands must share the same normal exit; this can be readily achieved by planting a jump at the end of one of them.

Lemma 6.2.15 (Conditional)

Let $B = s : (l := s_1 \lhd b \rhd s_2)$ and $s \notin \{s_1, s_2\}$
and P inhibits B and Q and $s_1 \notin \alpha l Q$
and Q inhibits B and P and $s_2 \notin \alpha l P$.
Then $\langle s_1, P, f\rangle \lhd b \rhd \langle s_2, Q, f\rangle = \langle s,\ P\|B\|Q,\ f\rangle$

Proof Similar to Lemma 6.2.14. $\square$

The compilation of sequential composition requires that the finish point of the first operand must be the same as the normal entry of the second operand. Of course, the operands must be disjoint and the second must not jump back to the first. In this case, the two machine code blocks can be simply assembled, without any extra code to glue them together.

Lemma 6.2.16 (Sequential composition)

If Q inhibits P, then

$$\langle s, P, h\rangle; \langle h, Q, f\rangle \sqsubseteq \langle s, P \| Q, f\rangle$$

Proof

$$\begin{array}{lll} & \langle s, P \| Q, f\rangle & \{\text{Theorem 6.1.20}\} \\ = & \mathbf{var}\, l := s; P^*; Q^*; (l = f)_\perp; \mathbf{end}\, l & \{2.9\mathbf{L7}\} \\ \sqsupseteq & \langle s, P, h\rangle; \langle h, Q, f\rangle & \square \end{array}$$

Iteration is a special case of recursion that is relatively easy to implement in machine code.

Lemma 6.2.17 (Iteration)

Let s, f, s_1 and f_1 be distinct labels and $\alpha l P \cap \{s, f_1, f\} = \{\}$

and $B = s : (l := s_1 \lhd b \rhd f)$

and $J = f_1 : (l := s)$.

Then $b * \langle s_1, P, f_1\rangle \sqsubseteq \langle s, B \| P \| J, f\rangle$

Proof

$$\begin{array}{lll} & \langle s, B \| P \| J, f\rangle & \{5.5\mathbf{L5} \text{ and Lemma } 6.1.11\} \\ = & \mathbf{var}\, l := s; B; (B \| P \| J)^*; (l = f)_\perp;\ \mathbf{end}\, l & \{3.1\mathbf{L3}, 2.9\mathbf{L5} \text{ and } 5.5\mathbf{L2}\} \\ = & (\mathbf{var}\, l := s_1; (B \| P \| J)^*; (l = f)_\perp; \mathbf{end}\, l) & \\ & \lhd b \rhd (\mathbf{var}\, l := f; (l = f)_\perp; \mathbf{end}\, l) & \{5.5\mathbf{L5} \text{ and } J \text{ inhibits } P\} \\ = & (\mathbf{var}\, l := s_1; P^*; J^*; (B \| P \| J)^*; (l = f)_\perp; & \\ & \mathbf{end}\, l) \lhd b \rhd I\!I & \{\text{Lemma } 6.1.11 \text{ and } 2.9\mathbf{L7}\} \\ \sqsupseteq & (\langle s_1, P, f_1\rangle; \mathbf{var}\, l := f_1; J; (B \| P \| J)^*; & \\ & (l = f)_\perp; \mathbf{end}\, l) \lhd b \rhd I\!I & \{\text{def of } J\} \\ = & (\langle s_1, P, f_1\rangle\, ;\, \langle s, B \| P \| J, f\rangle) \lhd b \rhd I\!I & \end{array}$$

The conclusion follows from the weakest fixed point theorem. $\square$

Exercise 6.2.18

Prove that Theorem 6.2.17 can be strengthened to an equation. $\square$

6.3 Interpretation

An interpreter is a program that accepts as input the *text* of an arbitrary program expressed in its source language, and then behaves exactly as described by the predicate which the text denotes. Interpretation is in principle slower than direct execution of compiled code; this is in practice tolerable when the interpreter is written in a language which is executed with inherently greater efficiency than the interpreted language. But there is considerable theoretical interest in an interpreter that is written in the *same* language that it interprets. For example, a universal Turing machine is an interpreter capable of simulating all other Turing machines, including itself. Any language which is powerful enough to program all computable functions must be capable of writing its own interpreter. And any method of reasoning about programs should be powerful enough to prove that this interpreter is correct.

The construction and proof of an interpreter requires a clear way of describing and manipulating the texts of a program, quite independent of their meaning. We will use typewriter font variables $\mathtt{P}, \mathtt{Q}, \ldots, \mathtt{b}, \mathtt{e}, \mathtt{v} \ldots$ to stand for program texts, including conditions, expressions and lists of variables. We will use the normal programming operators to stand for the textual operation of combining texts into a larger program text. For example, $\mathtt{P};\mathtt{Q}$ denotes the text obtained by writing the text *denoted* by $\mathtt{P}$ (*not* the letter "P" itself), followed by a semicolon (the symbol ";" itself), followed by the text denoted by $\mathtt{Q}$; the whole result may be enclosed in brackets (if necessary). All these textual operations are independent of the meaning of the text, although they are closely related. If $\mathtt{P}$ denotes a text which has a meaning described by the predicate P, and $\mathtt{Q}$ similarly has meaning Q, then the text $\mathtt{P};\mathtt{Q}$ will clearly have the meaning $(P;Q)$, as defined in Chapter 2. The same applies to all the other symbols of the programming language. In general, we will use an italic font letter to stand for the variable, expression or predicate whose text is denoted by the corresponding typewriter font letter. These rather informal conventions can be formalised by a rigorous separation of syntax from the semantics of the language, and the definition of an explicit function mapping between them. This is the standard practice in the exposition of denotational semantics, but we prefer the lighter notation of just changing the font.

In our interpreter, the control variable l will take as its value a list of texts representing the rest of the program which currently remains to be executed; its initial value is the whole program taken as a unit list, and its final value is empty. Each step of the interpreter analyses the value of l to find the first action to be performed; it also updates the value of l to maintain its record of remaining actions. When no action remains, the interpreter terminates. Apart from l, we assume that the interpreter has exactly the same alphabet of assignable variables as the program that it interprets. Thus the assignment $\mathtt{v} := \mathtt{e}$ has an effect which is described simply by $v := e$.

As in previous sections, the interpreter is designed as an assembly of steps, each of which deals with a particular programming feature. For example, the step which deals assignments is called A, defined by

$$A =_{df} (v, l := e, tail(l)) \lhd l_0 = (\mathtt{v} := \mathtt{e}) \rhd \mathit{II}$$

where l_0 denotes the first member of the list l,

and $tail(l)$ represents the list l after the removal of its first member.

The condition tests whether the first action specified by the current value of l is an assignment. If so, the assignment is performed, and its text is removed from the list. The continuation alphabet of this step is the set of all lists of program texts that begin with an assignment. The condition $l_0 = (\mathtt{v} := \mathtt{e})$ in the clause A is tested by pattern matching; the list of variables v is what is found to match the left hand side v of the assignment $\mathtt{v} := \mathtt{e}$, and e is similarly determined by e.

The other steps of the interpreter deal with the control structures of the language, and they update only the control variable l. In the following we use $l^\frown m$ to denote the catenation of lists l and m, and $< \mathtt{P} >$ to denote the unit list with just the single element P.

$$B =_{df} ((l :=< \mathtt{P} > {}^\frown tail(l)) \lhd b \rhd (l :=< \mathtt{Q} > {}^\frown tail(l))) \lhd l_0 = (\mathtt{P} \lhd \mathtt{b} \rhd \mathtt{Q}) \rhd \mathit{II}$$

$$C =_{df} ((l :=< \mathtt{P} > {}^\frown tail(l)) \sqcap (l :=< \mathtt{Q} > {}^\frown tail(l))) \lhd l_0 = (\mathtt{P} \sqcap \mathtt{Q}) \rhd \mathit{II}$$

$$J =_{df} (l := tail(l)) \lhd l_0 = \mathit{II} \rhd \mathit{II}$$

$$R =_{df} (l :=< \mathtt{F}(\mu\mathtt{X}.\mathtt{F}(\mathtt{X})) > {}^\frown tail(l)) \lhd l_0 = (\mu\mathtt{X}.\mathtt{F}(\mathtt{X})) \rhd \mathit{II}$$

$$S =_{df} (l :=< \mathtt{P} > {}^\frown < \mathtt{Q} > {}^\frown tail(l)) \lhd l_0 = (\mathtt{P}; \mathtt{Q}) \rhd \mathit{II}$$

$$Z =_{df} \mathit{II} \lhd l_0 = \bot \rhd \mathit{II}$$

The clauses B and C choose which of P and Q to execute; J ignores any occurrence of II. S deals with sequential composition by deleting leftmost brackets and adding the two components in front of l. Eventually, this will reveal the first primitive action to be performed. The clause R deals with the case of recursion. It executes $\mu\mathtt{X}.\mathtt{F}(\mathtt{X})$ by executing its body $\mathtt{F}(\mathtt{X})$, but first, all instances of X in the body have been replaced by the whole of the original recursive construction. These copies will be ready in place when they are needed for further recursive activations. The clause Z is used to deal with abortion by ensuring that the interpreter never terminates.

Each of the steps defined above has its own continuation alphabet; it is the set of all lists which begin with a text of the form displayed in the condition. All of these forms are distinct, and because they are texts, their values are also all different. We can therefore define the interpreter as an assembly of steps with

disjoint alphabets

$$INT =_{df} A|B|C|J|R|S|Z$$

Its alphabet is the union of the alphabets of its constituent steps. It includes all lists of program texts except the empty list. So the interpreter will terminate as soon as it encounters <>, which is not dealt with by INT. We therefore define the continuation alphabet of the interpreter to be all lists but <>. To make this work, the interpreter assigns the program text as the only member of the list to the variable l before starting to execute it. And, of course, the variable l is declared as a local variable of the interpreter.

Definition 6.3.1 (Interpreter)

$$I(\mathtt{Q}) =_{df} \langle < \mathtt{Q} >, INT, <> \rangle \qquad \text{(see Definition 6.2.9)} \qquad \square$$

The definitions given above show that it is rather easy to write an interpreter in its own language, especially if pattern matching is used to analyse the program text. It is also rather easy to see that the interpreter does what is expected of it. Nevertheless, it is worthwhile to formulate a theorem expressing what it means for such an interpreter to be correct, and then to see whether the algebraic laws which have been listed and proved in Chapter 5 are adequate to prove its correctness.

Theorem 6.3.2 (Correctness of the interpreter)

Let Q be the text of a program described by the predicate Q. Then

$$Q \;=\; I(\mathtt{Q})$$

Proof The proof is by structural induction on the text of the program Q, and it is based on the following lemmas. $\square$

As in Section 6.2, each case is proved by algebraic laws justifying symbolic execution. The most difficult case is sequential composition, where it is necessary to show that the interpretation of $l\hat{}m$ passes through an intermediate stage when l has been fully executed, and only m remains. This case needs two lemmas.

Lemma 6.3.3

$$(l \neq <>)_{\perp}; (l := l\hat{}m); INT \;=\; (l \neq <>)_{\perp}; INT; (l := l\hat{}m)$$

Proof Based on case analysis on the value of l and the definition of the step function INT. $\square$

Lemma 6.3.4

$$(l := l\hat{}m)\,;\, ((l \neq <> \wedge l \neq m) * INT) \;=\; INT^*\,;\, (l := ,,)$$

Proof Let $f(X) =_{df} (l := l^\frown m)\ ;\ X$

$g(X) =_{df} (INT; X) \lhd (l \neq <>) \wedge (l \neq m) \rhd I\!I$

$h(X) =_{df} (INT; X) \lhd l \neq <> \rhd (l := m)$

	Lemma 6.3.3	{def of f, g and h}
$\Rightarrow$	$f \circ g = h \circ f$	{Theorem 4.2.12}
$\Rightarrow$	$f(\mu g) = \mu h$	{5.5**L4**}
$\Rightarrow$	$LHS = RHS$	□

Lemma 6.3.5 (Sequential composition)

$I(\mathtt{P};\mathtt{Q}) = I(\mathtt{P}); I(\mathtt{Q})$

Proof Let $b =_{df} (l \neq <>) \wedge (l \neq <\mathtt{Q}>)$.

	$I(\mathtt{P};\mathtt{Q})$	{def of INT and 5.5**L1**}
=	$\mathbf{var}\, l :=< \mathtt{P} >^\frown < \mathtt{Q} >; INT^*; (l =<>)_\perp; \mathbf{end}\, l$	{5.1**L3** and 5.5**L5**}
=	$\mathbf{var}\, l :=< \mathtt{P} >;\ l := l^\frown < \mathtt{Q} >;\ (b * INT);$ $INT^*; (l =<>)_\perp; \mathbf{end}\, l$	{let $m =< \mathtt{Q} >$ in Lemma 6.3.4}
=	$\mathbf{var}\, l :=< \mathtt{P} >; INT^*; (l :=< \mathtt{Q} >); INT^*; (l =<>)_\perp; \mathbf{end}\, l$	{5.5**L3**}
=	$\mathbf{var}\, l :=< \mathtt{P} >; INT^*; (l =<>)_\perp;$ $(l :=< \mathtt{Q} >); INT^*; (l =<>)_\perp; \mathbf{end}\, l$	{2.9**L7**}
=	$I(\mathtt{P}); I(\mathtt{Q})$	□

The only other significant case is recursion.

Lemma 6.3.6 (Recursion)

$I(\mu \mathtt{X}.\mathtt{F}(\mathtt{X})) = \mu X.\, F(X)$

Proof	$I(\mu \mathtt{X}.\mathtt{F}(\mathtt{X}))$	{5.5**L1**}
=	$\mathbf{var}\ l :=< \mathtt{F}(\mu \mathtt{X}.\mathtt{F}(\mathtt{X})) >; INT^*; (l =<>)_\perp; \mathbf{end}\ l$	{induction}
=	$F(I(\mu \mathtt{X}.\mathtt{F}(\mathtt{X})))$	

which implies that $I(\mu \mathtt{X}.\mathtt{F}(\mathtt{X})) \sqsupseteq \mu X.\, F(X)$.

To show correctness of the interpreter I, this inequation is sufficient. To establish the inequation in the other direction, we construct a more abstract interpreter T as an infinite assembly of clauses, each of which interprets a whole sequence of programs in a single step

$$T =_{df} (\exists \mathtt{Q1}, \ldots, \mathtt{Qn} \bullet ((l = <\mathtt{Q1}, \ldots, \mathtt{Qn}>)^\top; Q1; \ldots; Qn) \lhd l \neq <> \rhd I\!I); \mathbf{end}\, l$$

T is not a program, but it is reasonable to regard it as a predicate, and for purposes of proof that is good enough. In the case of a unit sequence, we have

$$(\mathbf{var}\, l :=< \mathtt{Q} >; T) \;=\; Q \qquad (\dagger)$$

Furthermore

$$\begin{array}{lll} & T & \{2.1\mathbf{L1}\} \\ = & T \lhd l \in \alpha l INT \rhd T & \{\text{def of } INT\} \\ = & (INT; T) \lhd l \in \alpha l INT \rhd \mathbf{end}\, l & \end{array}$$

from which and 5.5**L4** it follows that

$$T \;\sqsupseteq\; \mu X \bullet ((INT; X) \lhd l \in \alpha l INT \rhd \mathbf{end}\; l) \;=\; (INT^*; \mathbf{end}\, l) \qquad (\ddagger)$$

and

$$\begin{array}{lll} & \mu X.\, F(X) & \{(\dagger)\} \\ = & \mathbf{var}\, l :=< \mu \mathtt{X.F(X)} >; T & \{(\ddagger)\} \\ \sqsupseteq & \mathbf{var}\, l :=< \mu \mathtt{X.F(X)} >; (INT^*); \mathbf{end}\, l & \{\text{def of } I\} \\ \sqsupseteq & I((\mu \mathtt{X.F(X)})) & \Box \end{array}$$

These lemmas show the main clauses of the proof of the correctness of the interpreter I.

6.4* Jumps and labels

The process of compilation described in Section 6.2 translates a language with high level control structures like conditionals, compositions and iterations into a language which replaces all such structures by conditional and unconditional jumps to labels placed elsewhere in the program. Many of the older programming languages provide jumps and labels in addition to more abstract program structures. Such a mixture of levels of abstraction is not always recommended in programming practice, but it provides a good exercise for the extensibility of our theory. Our goal is to preserve the validity of all the laws relating to structures, and just to add the laws needed for reasoning about the additional unstructured features. Our method is to concentrate mainly on *forward* jumps, and *backward* jumps are treated separately, because they can give rise to iteration.

To reason about the structure of a program P in a language with jumps, it is necessary to keep an account of all the *exit* points of P, that is all the labels that may be destinations outside P of jumps which originate inside P. This is nothing but the set of expected final values of the control variable l, and it will be denoted $\alpha l' P$. The set plays the same role as the single finishing point f of Section 6.2.

Similarly, we single out the complementary set $\alpha l_0 P$, containing the permissible set of initial values of l before execution of P. It denotes the subset of $\alpha l P$ that contains the intended entry points of P; it will contain the labels placed internally within P, but exclude labels which are intended only as destinations for jumps which are also internal to P. It plays the same role as the single start point s of Section 6.2.

We will also single out a special value **n**, which is the value that l takes when P is entered normally through the beginning, and not by a jump. This is the value that l will also take on normal exit from P, which occurs not by a jump but by falling through the end. It is convenient to exclude **n** from both $\alpha l_0 P$ and $\alpha l' P$; it will *never* be used for a backward jump.

The structure and control flow of a program with jumps and labels can often be made instantly obvious to the eye by means of a flow chart. This is a pictorial representation in which the program text is written in boxes, and the label values are written on arrows drawn between pairs of boxes, and on arrows connecting a box to the outside.

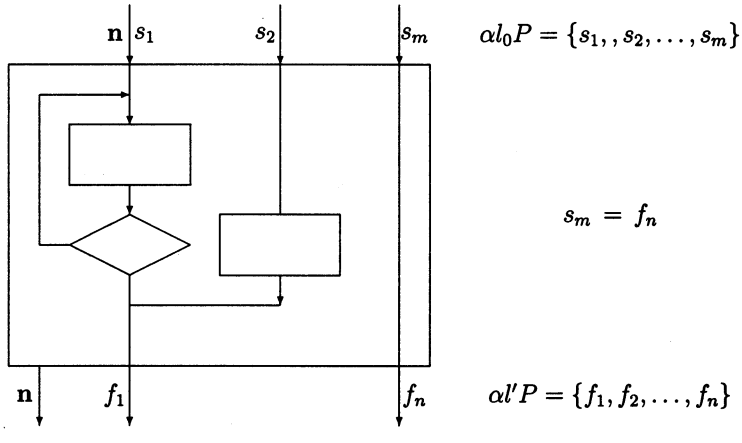

Figure 6.4.1 Flow chart

$\alpha l_0 P$ is the set of labels (other than **n**) on arrows leading into the box surrounding P, $\alpha l' P$ is the set of labels on arrows leading out. We allow structuring, that is flow charts may be drawn within the boxes of a flow chart, as shown in Figure 6.4.1. □

Suppose k is an internal label inside P. If k occurs as a label *after* all the jumps to k within P, then k will not be an exit label of P. It may or may not be in

$\alpha l_0 P$, depending on whether an external jump to that label is also to be allowed. If, however, a jump to k occurs *after* the label k, then k is definitely in $\alpha l' P$. If the jump is intended to lead back to the label k, then k must also be among the entries of P. In this case, the execution P^* will model explicitly the iterations that result from executing backward jumps; they will be detected by the fact that their labels are in the intersection $\alpha l_0 P \cap \alpha l' P$. If there are only forward jumps, the entry labels will be disjoint from the exit labels, and we will single out this special case as easier to reason about.

The expectation that the initial value of l will lie in $\alpha l_0 P \cup \{\mathbf{n}\}$ is encoded in the same way as for steps (Definition 6.1.5) by making the action of P vacuous otherwise. The obligation for l to terminate in the set $\alpha l' P \cup \{\mathbf{n}\}$ is encoded as in Definition 6.2.4 (structured block) by an appropriately placed assertion.

Definition 6.4.2 (Blocks and proper blocks)

Let S and F be sets of labels, and $\mathbf{n} \notin S$ and $\mathbf{n} \notin F$.

$$(P : S \Rightarrow F) \;=_{df}\; (P \;=\; (P; (l \in (F \cup \{\mathbf{n}\})_\perp) \lhd l \in (S \cup \{\mathbf{n}\}) \rhd I\!I)$$

A program P is a *block* if it satisfies

$$(P : \alpha l_0 P \Rightarrow \alpha l' P)$$

A block P is called a *proper* block if

$$\alpha l_0 P \cap \alpha l' P \;=\; \{\}$$

In this section all the programs mentioned will be blocks. □

Examples 6.4.3

(1) If L is the set of all labels, then $(P : L \Rightarrow L)$ for all P.

(2) At the opposite extreme, if P contains no labels or jumps, that is

$$\alpha l_0 P \;=\; \alpha l' P \;=\; \{\}$$

then it is a proper block. It is entered only at the beginning (with $l = \mathbf{n}$) and exits only at the end (with l again equal to $\mathbf{n}$). □

A label s is placed within a program by the construction **label** s, at the point intended as the destination of any jump to s. It may be entered either normally or by a jump, but it always exits normally. A jump plays a complementary role. It is entered normally but it always exits by means of a jump. Neither of them has any effect on anything except l.

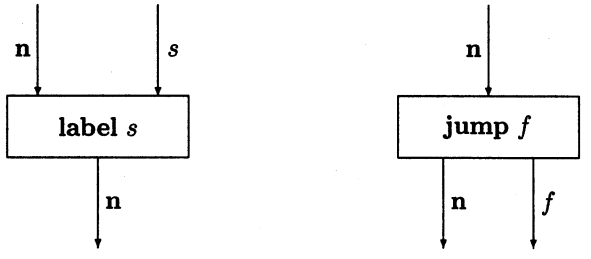

Figure 6.4.4 **label** s and **jump** f

Definition 6.4.5 (Labels and jumps)

$$\begin{array}{lll}
\textbf{label}\ s & =_{df} & (l := \mathbf{n}) \lhd l \in \{s, \mathbf{n}\} \rhd I\!I \\
\alpha l_0\,\textbf{label}\ s & =_{df} & \{s\} \\
\alpha l'\,\textbf{label}\ s & =_{df} & \{\} \\
\textbf{jump}\ f & =_{df} & (l := f) \lhd l = \mathbf{n} \rhd I\!I \\
\alpha l_0\,\textbf{jump}\ f & =_{df} & \{\} \\
\alpha l'\,\textbf{jump}\ f & =_{df} & \{f\}
\end{array}$$

□

The following laws show how a forward jump can be executed by omitting the code that follows it.

Theorem 6.4.6

L1 If $P : \{\} \Rightarrow \{\}$, then $(\textbf{jump}\ f\,;\, P) = \textbf{jump}\ f$

L2 $(\textbf{jump}\ s\,;\,\textbf{jump}\ f) = \textbf{jump}\ s$

L3 $(\textbf{label}\ s\,;\,\textbf{label}\ f) = (\textbf{label}\ f\,;\,\textbf{label}\ s)$

L4 $(\textbf{label}\ s\,;\,\textbf{label}\ s) = \textbf{label}\ s$

Proof of L2

$$\begin{array}{lll}
& \textbf{jump}\ s;\textbf{jump}\ f & \{\text{Def. 6.4.5 and 2.2}\mathbf{L2}\} \\
= & (l := s;\textbf{jump}\ f) \lhd l = \mathbf{n} \rhd I\!I & \{\text{Def. 6.4.5 and 2.3}\mathbf{L4}\} \\
= & ((l := s; l := f) \lhd \mathit{false} \rhd l := s) & \\
& \lhd l = \mathbf{n} \rhd I\!I & \{\text{Def. 6.4.5 and 2.1}\mathbf{L5}\} \\
= & \textbf{jump}\ s &
\end{array}$$

□

Our goal is to apply to blocks all the structuring notations of our programming language. For blocks with the same alphabet of entry and exit points, the permitted operators are defined by the following theorem.

Theorem 6.4.7 (Block closure)

The set of blocks $\{P \mid P : S \Rightarrow F\}$ is a complete lattice, and closed with respect to non-deterministic choice and conditional. The same applies to proper blocks. □

The main omission from this theorem is sequential composition. This is because the alphabets of the two operands of composition are usually different, and the alphabet of the result is different yet again. However, the result alphabet can be defined as a function of the operand alphabets in such a way that the relevant closure theorem can be proved. In general, a sequential composition may be entered at any entry point of either of its operands, and it may exit by a jump out of either of its operands. However, any label by which the first operand jumps into the second now becomes an internal forward jump, and will not lead to an exit from the sequential composition.

Definition 6.4.8 (Alphabets for sequential composition)

$$\begin{aligned} \alpha l_0(P;Q) &=_{df} \alpha l_0 P \cup \alpha l_0 Q \\ \alpha l'(P;Q) &=_{df} (\alpha l' P \setminus \alpha l_0 Q) \cup \alpha l' Q \end{aligned}$$ □

This definition is essentially restated by the following theorem.

Theorem 6.4.9 (Composition closure)

If $P : S \Rightarrow F$ and $Q : T \Rightarrow G$, then

$$(P;Q) : (S \cup T) \Rightarrow ((F \setminus T) \cup G)$$ □

To reason effectively about labelled programs, we need a method of abstracting from internal label values which are never intended to be seen or used from outside. In the interests of modularity, the hiding should be selective; as soon as an assembly P has been built up to include all jumps to an internal label value, that value can be hidden, while all the other labels are still visible. Labels that are used only for entry or only for exit are easy to hide, by just removing them from the alphabet. But a label in both alphabets can only be hidden by making explicit the internal iteration. This case must be dealt with first.

A backward jump (**jump** f) executed in a program P is one that leads to a label f in $\alpha l' P \cap \alpha l_0 P$. It also gives rise to a repetition of P whenever l takes f as its final value. On completion of the repetitions, the label can no longer be a final value of l; it is therefore removed from $\alpha l' P$. Let H be a set of labels which are used as destinations of backward jumps in P. We define $P^{\setminus H}$ to represent the desired effect of the iteration; in addition the labels of H are hidden by exclusion from the *exit* alphabet. This operation can be used to turn any block P into a proper block, by hiding all labels in $\alpha l_0 P \cap \alpha l' P$. Once backward jumps have been dealt with, all remaining jumps are forward, and all remaining blocks are proper.

H is usually a subset of both the exit and the entry alphabets: any extraneous elements are irrelevant; they are ignored in the following definitions.

Definition 6.4.10 (Hiding exits)

$$\begin{aligned} P^{\backslash H} &=_{df} P;(l \in H \cap \alpha l_0 P) * P \\ \alpha l_0\, P^{\backslash H} &=_{df} \alpha l_0 P \\ \alpha l'\, P^{\backslash H} &=_{df} \alpha l' P \setminus (\alpha l_0 P \cap H) \end{aligned}$$ □

The theorem effectively shows that the definition has its intended effect.

Theorem 6.4.11 (Hiding exits)

If $P : S \Rightarrow F$, then $P^{\backslash H} : S \Rightarrow (F \setminus (S \cap H))$. □

The hiding of exits has all the expected properties of a hiding operator. Hiding what is not there has no effect; the order of hiding different labels is irrelevant, and if hiding is irrelevant for one operand of a composition, it distributes to the other.

Theorem 6.4.12

L5 $P^{\backslash H} = P$ whenever $H \cap \alpha l' P = \{\}$

L6 $(P^{\backslash H})^{\backslash G} = P^{(H \cup G)}$

L7 $(P;Q)^{\backslash H} = P;(Q^{\backslash H})$ whenever $H \cap \alpha l_0 P = \{\}$

L8 $(P;Q)^{\backslash H} = (P^{\backslash H});Q$ whenever $H \cap (\alpha l_0 Q \cup \alpha l' Q) = \{\}$

Proof of **L7** Let $A = \alpha l_0 P \cup \alpha l_0 Q$.

$$\begin{aligned} & (l \in H \cap A) * (P;Q) && \{5.5\mathbf{L1}\} \\ = \; & (l \in H \cap A) * ((l \in H \cap A)_\perp; P; Q) && \{\text{Def. } 6.4.2\} \\ = \; & (l \in H \cap A) * ((l \in H \cap A)_\perp; && \{b_\perp; c_\perp = (b \wedge c)_\perp \\ & (P; Q \lhd l \in \alpha l_0 P \cup \{\mathbf{n}\} \rhd Q)) && \text{and } H \cap \alpha l_0 P = \{\}\} \\ = \; & (l \in H \cap A) * ((l \in H \cap A)_\perp; Q) && \{5.5\mathbf{L1}\} \\ = \; & (l \in H \cap A) * Q && \{H \cap \alpha l_0 P = \{\}\} \\ = \; & (l \in H \cap \alpha l_0 Q) * Q \end{aligned}$$

which implies

$$LHS = P;Q;((l \in H \cap A) * (P;Q)) = P;Q;((l \in H \cap \alpha l_0 Q) * Q) = RHS$$ □

Hiding of an entry label is simpler than hiding an exit label. It is applied to a label in P which is intended only as the destination of a forward jump that is also within P, and from nowhere else. Indeed, the correctness of the program

may depend on observance of this restriction. Such a restriction can be enforced syntactically by removing the label from the alphabet of permitted entry points of the block.

Definition 6.4.13 (Hiding entries)

Let H be a set of labels to be removed from the entry points of P.

$$\begin{aligned} {}^{H/}P &=_{df} \mathit{\Pi} \lhd l \in H \rhd P \\ \alpha l_0\, {}^{H/}P &=_{df} \alpha l_0 P \setminus H \\ \alpha l'\, {}^{H/}P &=_{df} \alpha l' P \end{aligned}$$

□

This definition also has the properties expected of any hiding operator.

Theorem 6.4.14 (Hiding entries)

If $P : S \Rightarrow F$, then ${}^{H/}P : (S \setminus H) \Rightarrow F$. □

Theorem 6.4.15

L9 ${}^{H/}P = P$ whenever $H \cap \alpha l_0 P = \{\}$

L10 ${}^{H/}({}^{G/}P) = {}^{(H \cup G)/}P$

L11 ${}^{H/}(P;Q) = ({}^{H/}P);Q$ whenever $H \cap \alpha l_0 Q = \{\}$

L12 ${}^{H/}(P;Q) = P;({}^{H/}Q)$ whenever $H \cap (\alpha l_0 P \cup \alpha l' P) = \{\}$

Proof of **L12** Let $B = \alpha l_0 P \cup \{\mathbf{n}\}$.

$$\begin{aligned} & P;({}^{H/}Q) && \{\text{Def. 6.4.2 and 2.2}\mathbf{L2}\} \\ = \; & (P;(l \in \alpha l' P)_\perp;({}^{H/}Q)) \lhd l \in B \rhd ({}^{H/}Q) && \{2.1\mathbf{L5} \text{ and } H \cap \alpha l' P = \{\}\} \\ = \; & (P;Q) \lhd l \in B \rhd ({}^{H/}Q) && \{\text{Def. 6.4.13 and 2.1}\mathbf{L2}\} \\ = \; & (\mathit{\Pi} \lhd l \in H \rhd Q) \lhd l \notin B \rhd (P;Q) && \{2.1\mathbf{L3} \text{ and } H \cap B = \{\}\} \\ = \; & \mathit{\Pi} \lhd l \in H \rhd (Q \lhd l \notin B \rhd (P;Q)) && \{\text{Def. 6.4.2 and 2.2}\mathbf{L2}\} \\ = \; & \mathit{\Pi} \lhd l \in H \rhd (P;Q) && \{\text{Def. 6.4.13}\} \\ = \; & {}^{H/}(P;Q) \end{aligned}$$

□

Our theory of labels and jumps is powerful enough to treat *exception handling* as a special case. An exception is just a specially restricted kind of label e that can never appear in an entry alphabet, and can never be the destination of a backward jump. To raise the exception within P, an ordinary forward jump can be used. If Q is the handler designed to catch exception e within P, their combined effect can be defined

$$P;((l := \mathbf{n};Q) \lhd l = e \rhd \mathit{\Pi})$$

This is a block with $\alpha l_0 P \cup \{e\}$ as its entry label set, and $(\alpha l' P \setminus \{e\}) \cup \alpha l' Q$ as its exit label set. If Q raises the same or a different exception, it will automatically be directed forward to a more global handler.

It is now possible to hide *all* the labels and jumps within a block P, and thereby obtain a module of program that can be treated as though it were expressed in a language without any jumps and labels at all. This is in fact recommended practice: jumps should be confined in range to a self-contained unit of program, for example the body of a subroutine. We therefore define an *encapsulation* operator which hides all labels, and even conceals the control variable l.

Definition 6.4.16 (Encapsulation)

If $P : S \Rightarrow F$ then

$$\langle P \rangle \ =_{df} \ \mathbf{var}\ l; (l := \mathbf{n}); {}^{S/}(P^{\setminus F}); \mathbf{end}\ l \qquad \square$$

Theorems 6.4.7 and 6.4.9 give the constraints under which structured notations can be mixed with jumps and labels. In the case of recursion, the entry and exit alphabets of the recursive call must exactly match those of the whole body, and if there are any labels local to the body, the alphabet of the recursive call must be extended to accept and ignore any of these labels (Exercise 6.4.17). Extension may also be used to equalise the alphabets of the two components of a conditional. But the effect may be rather different from what is expected. Our theory requires that if a conditional is entered by a jump, the condition still has to be tested to determine whether to go to the label in the first operand or the label in the second operand. Implementors of structured programming languages have not been prepared to take this trouble. They have therefore forbidden jumps into a conditional, effectively requiring the input alphabet to be empty.

Exercise 6.4.17

The converse operation to hiding is alphabet extension. Under reasonable conditions, define this operation and prove that it has the required properties. $\square$