Refinement: An Overview

Ana Cavalcanti[1], Augusto Sampaio[2], and Jim Woodcock[1]

[1] Department of Computer Science
University of York
York, UK
[2] Centro de Informática
Universitates Federal de Pernambuco
Recife - PE, Brazil

The purpose of this initial chapter is to introduce concepts and techniques assumed as general background in the remaining chapters of this book. The relevant notions are introduced using a simple and well-known programming notation: Dijkstra's language of guarded commands [81], presented in Section 1.

Three classical approaches to assigning semantic meaning to programs are then explored. In Section 2 we discuss the annotation of programs with assertions and the associated reasoning framework (Hoare Logic). Section 3 is dedicated to a calculational style where the behaviour of a program is defined in terms of a predicate transformer: its weakest precondition. Partial and total correctness of programs are contrasted in these two sections. The important notion of program refinement is introduced in Section 4. We start with some intuition and then we give a weakest precondition based definition, followed by an alternative (but equivalent) definition in terms of nondeterminism.

In Section 5, we explore another approach to program semantics, known as refinement algebra, which is based on equations and inequations (laws) relating programming constructs; algebraic laws allow a term rewriting style of program transformation. We then show, in Section 6, how the programming constructs can be embedded into a more abstract space of specifications; we introduce Morgan's specification statement and illustrate Morgan's refinement calculus concerning both algorithmic and data refinement. In Section 7 we discuss how a programming (or specification) language with a refinement ordering can be regarded as a lattice. This allows using well-established results of lattice theory in programming methodologies. We conclude this chapter with a brief discussion of refinement in other programming paradigms and the importance of tools to support program refinement in practice.

1 A Simple Programming Notation

The version of Dijkstra's Guarded Command Language (GCL) adopted here is summarised below; c stands for a command, x for a list of variables, e for a list of expressions, and ψ for a predicate.

A. Cavalcanti, A. Sampaio, and J. Woodcock (Eds.): PSSE 2004, LNCS 3167, pp. 1–17, 2006.

$c ::= \mathbf{skip} \mid \mathbf{abort}$	do nothing, abortion
$\mid x := e \mid c;\ c$	assignment, sequence
$\mid \mathbf{if}\ [] i \bullet \psi_i \rightarrow c_i\ \mathbf{fi}$	conditional
$\mid \mathbf{do}\ [] i \bullet \psi_i \rightarrow c_i\ \mathbf{od}$	iteration
$\mid c \sqcap c$	nondeterminism
$\mid \mathbf{var}\ x : T \bullet c$	local variable block

The primitive constructs are standard. The command **skip** has no effect and, when executed, terminates immediately. In contrast, the command **abort** has a completely arbitrary behaviour; rather than being deliberately written by a programmer, it may arise as a result of some undesirable computation like division by zero, or an infinite loop without any externally visible effect.

The remaining primitive command of GCL is assignment. The program $x := e$ is a multiple (or simultaneous) assignment, where x is a list of distinct variables and e an equal-length list of expressions. The components of e are evaluated and assigned to the corresponding components of x in the same position. All the contributing chapters of this book assume that expressions have no side-effect. In this chapter we further assume that expressions are always well-defined (their evaluation is always successful). In Chapter 2, on refinement of object-oriented programs, we consider assignments whose expressions might fail when evaluated, and the impact of side effects in expressions is discussed. As a simple example of a multiple assignment, the command

$$x, y := y, x$$

swaps the values of x and y.

In GCL, the body of the conditional is a guarded command set. A guarded command takes the form $\psi \rightarrow c$, where ψ, the guard, is a predicate. The choice of which command executes is between those whose guards evaluate to true. If more than one guard is satisfied, the choice is nondeterministic; if no guard evaluates to true, the conditional behaves like **abort**. As an example, we consider the following command that assigns to z the greatest value held by x or y.

$$\begin{array}{l}\mathbf{if}\ (x \leq y) \rightarrow z := y \\ []\ (y \leq x) \rightarrow z := x \\ \mathbf{fi}\end{array}$$

When $x = y$, the choice of which assignment executes is nondeterministic.

The body of an iteration ($\mathbf{do}\ [] i \bullet \psi_i \rightarrow c_i\ \mathbf{od}$) is also a guarded command set. Similarly to the conditional, the choice of which guarded command executes depends on the evaluation of the guards. If more than one guard is satisfied, the choice is nondeterministic; if no guard evaluates to true, the iteration terminates successfully, behaving like **skip**. In the program fragment below, the final value of r is the factorial of the natural number assigned to n.

$$r := 1;\ \mathbf{do}\ (n > 1) \rightarrow r := r * n;\ n := n - 1\ \mathbf{od}$$

The program $c_1 \sqcap c_2$ denotes an arbitrary (also known as *demonic*) choice between the commands c_1 and c_2, in the sense that either one can be selected for execution. For instance, consider the program fragment below.

$x := 1 \sqcap x := 2$;
if $(x = 1) \rightarrow$ **skip**
$[]$ $(x = 2) \rightarrow$ **abort**
fi

In this context, either $x := 1$ or $x := 2$ will be selected, so that is possible that the execution of the conditional leads to abortion. It is important to observe, however, that, according to equality and refinement notions that reflect total correctness, any program that might abort (like the one above) is actually identified with abort.

Some specification languages (like the one introduced in the next chapter) include a complementary notion of nondeterminism known as *angelic* choice. In this case the most suitable command for a given context is selected for execution. If the choice in the above example were angelic, the assignment $x := 1$ would have been selected. Operationally, the view is that of *backtracking* in the search for the best possible execution of the program. For further considerations on demonic and angelic choices see, for instance, [17].

The program (**var** $x : T \bullet c$) declares the variable x of type T for use in the command c. Local blocks of this form may appear anywhere a command is expected. The occurrence of a variable x in the scope of a local declaration is *bound*, and *free* otherwise. For example, y is bound in **var** $y : T \bullet x := y$, but free in $x := y$. The program fragment that computes the factorial of n, previously presented, can be redesigned to leave n untouched, using a local variable.

$$r := 1;\ \textbf{var}\ t : \mathbb{N} \bullet t := n;\ \textbf{do}\ (n > 1) \rightarrow r := r * t;\ t := t - 1\ \textbf{od}$$

In the following two sections we discuss two well-established approaches to define a formal semantics to programming languages; the guarded command language introduced in this section is used as illustration.

2 Assertions and Hoare Logic

A classical approach to assigning formal meaning to (and reasoning about) programs, well-known as Hoare logic, is presented in [114]. Like in other branches of mathematics, the basis of this approach is to define the behaviour of programming constructs in terms of axioms and inference rules. Axioms define the semantics of the primitive commands like **skip**, **abort** and assignment. Each axiom takes the form of a Hoare triple, $P\,\{c\}\,Q$, where c is a command and P and Q are logical assertions, playing the role of pre- and postcondition, respectively. A Hoare triple is interpreted as follows: if P is true and c terminates successfully, then Q must be established. The semantics of language operators like sequential composition and conditional is defined by inference rules which assume that a Hoare triples hold for the arguments.

The original formulation of Hoare logic is for *partial correctness*, which means that the axioms and inference rules assume successful termination of a program, as can be inferred from the above interpretation. Nevertheless, several subsequent formulations of Hoare logic as, for instance, [74, 106, 203], address *total correctness*. In this case, the interpretation of a triple $P\,\{c\}\,Q$ is: if P is true, then c must terminate successfully and establish Q. While the semantics described here regards partial correctness, the weakest precondition semantics defined in the next section embodies termination.

The Hoare triples for the language presented in the previous section are summarised in Table 1.

Table 1. Hoare triples for GCL

$P\,\{\mathbf{skip}\}\,P$
$\mathbf{true}\,\{\mathbf{abort}\}\,\mathbf{false}$
$P[e/x]\,\{x := e\}\,P$
If $P\,\{c_1\}\,Q$ and $Q\,\{c_2\}\,R$ then $P\,\{c_1;\ c_2\}\,R$
If, for all i, $(\psi_i \wedge P)\,\{c_i\}\,Q$ then $P\,\{(\mathbf{if}\ [\!]\, i \bullet \psi_i \rightarrow c_i\ \mathbf{fi})\}\,Q$
If, for all i, $(\psi_i \wedge P)\,\{c_i\}\,P$ then $P\,\{\mathbf{do}\ [\!]\, i \bullet \psi_i \rightarrow c_i\ \mathbf{od}\}\,(P \wedge (\bigwedge i \bullet \neg\psi_i))$
If $P\,\{c_1\}\,Q$ and $P\,\{c_2\}\,Q$ then $P\,\{c_1 \sqcap c_2\}\,Q$
If $P\,\{c\}\,Q$ then $(\forall\, x : T \bullet P)\,\{\mathbf{var}\ x : T \bullet c\}\,(\exists\, x : T \bullet Q)$

As **skip** has no effect, any logical assertion that is true before its execution, remains true after it terminates. Being totally unpredictable, nothing can be guaranteed concerning what **abort** could establish as postcondition, if it terminates. The axiom for assignment formalises that if P is to be established after the assignment of e to x, then the assertion obtained from P, by replacing all free occurrences of x with e, must be true before the assignment.

The semantics of the remaining constructs is defined by inference rules. For $c_1;\ c_2$, the precondition is that of c_1, and the postcondition is that established by c_2; furthermore, the postcondition established by c_1 coincides with the precondition of c_2. Concerning the conditional, each of its guarded commands must establish the expected postcondition, provided the corresponding guard is true. In the case of a nondeterministic execution of c_1 and c_2, the expected result can only be ensured if both produce such a result.

The semantics of iteration is based on an invariant P that is assumed to hold for each guarded command in the body of the loop, whenever the corresponding guard is true. This invariant is also assumed to hold before the entire iteration starts executing. Then, P must also hold after (possible) termination of the iteration, when none of the guards holds any longer.

Assuming that an assertion holds for the body of a local declaration block (where occurrences of a variable, say x, might be free), the precondition for the entire block is assumed to hold for all possible values of x. Then there must be at least one possible value of x such that the postcondition holds. When P and Q do not mention x the quantifiers have no effect and can be eliminated.

Apart from the axioms and inference rules that define the semantics of the programming constructs, Hoare logic includes additional rules for reasoning, like the *rules of consequence* [114] displayed below.

If $Q\,\{c\}\,R$ and $P \Rightarrow Q$ then $P\,\{c\}\,R$
If $Q\,\{c\}\,R$ and $R \Rightarrow S$ then $P\,\{c\}\,S$

As an example of the application of these rules, we can observe that the behaviour of **abort** is really unpredictable, since a Hoare triple such as

$$x = 1\ \{\mathbf{abort}\}\ x = 2$$

holds. The proof follows from the fact that $x = 1 \Rightarrow \mathbf{true}$ and $\mathbf{false} \Rightarrow x = 2$.

3 Weakest Preconditions

The seminal work [81] presented an alternative technique to reason about programs: weakest preconditions calculus. In this approach, the semantics of a program is characterised by a predicate transformer: a function from predicates to predicates usually called *wp*. When applied to a program p and to predicate ψ, *wp* gives a predicate that defines all states in which execution of p terminates and leads to a state in which ψ holds. The predicate ψ is called a postcondition, and $wp.p.\psi$ is the weakest precondition that guarantees that the program establishes ψ; a period is used here to denote function application. In contrast with the previous chapter, here we consider total correctness.

The weakest precondition semantics of the simple language presented in Section 1 is shown in Table 2. The semantics of a loop is given in terms of the semantics of recursion, which is discussed in Section 7.

Table 2. Weakest precondition semantics of GCL

$wp.\mathbf{skip}.\psi$	ψ
$wp.\mathbf{abort}.\psi$	$false$
$wp.x := e.\psi$	$\psi[e/x]$
$wp.(c_1;\ c_2).\psi$	$wp.c_1.(wp.c_2.\psi)$
$wp.(\mathbf{if}\ [] i \bullet \psi_i \rightarrow c_i\ \mathbf{fi}).\psi$	$(\bigvee i \bullet \psi\_i) \wedge (\bigwedge i \bullet \psi_i \Rightarrow wp.c_i.\psi)$
$wp.(c_1 \sqcap c_2).\psi$	$(wp.c_1.\psi) \wedge (wp.c_2.\psi)$
$wp.(\mathbf{var}\ x : T \bullet c).\psi$	$\forall x : T \bullet wp.c.\psi$

Since **skip** terminates, but does not affect any variables, the only way in which it can establish a postcondition ψ is if it already holds. Because **abort** may not even terminate, it can never provide a guarantee to establish any postcondition. The assignment $x := e$ establishes a postcondition ψ if it holds when the variables x take the values e (and all other variables are not changed).

The semantics of sequence is function composition. The weakest precondition for $c_1 ; c_2$ to establish ψ is the weakest precondition for c_1 to establish the weakest precondition for c_2 to establish ψ.

For a conditional to be guaranteed at least to terminate, one of its guards has to be true. Moreover, if it is to be guaranteed that it establishes ψ, then the weakest precondition for each of the commands c_i associated with guards ψ_i that are true have to be satisfied. This is because any of these commands may be chosen for execution.

This also explains the semantics of the choice $c_1 \sqcap c_2$. If it is to be guaranteed that it establishes ψ, then both c_1 and c_2 have to provide the guarantee. Finally, arbitrary choice is also embedded in the semantics of a variable block ($\mathbf{var}\ x : T \bullet c$); the initial value of x is nondeterministically chosen. It is only guaranteed to establish ψ, if c does, for every value that x may take.

4 Refinement Notions

During development, sometimes the resulting program does not behave exactly as the original program, but is possibly better, from the point of view of the user. In this case, we say that we have a refinement of the original program. Formally, an ordering relation on programs is used: $c_1 \sqsubseteq c_2$ holds when c_2 is at least as good as c_1 in the sense that it will meet every purpose and satisfy every specification satisfied by c_1.

A refinement relation is a pre-order: it is reflexive and transitive.

Law 1 (Refinement reflexive). $c \sqsubseteq c$

Law 2 (Refinement transitive). $(c_1 \sqsubseteq c_2) \land (c_2 \sqsubseteq c_3) \Rightarrow (c_1 \sqsubseteq c_3)$

Often, and in this book, $\sqsubseteq$ is a partial ordering, further satisfying the antisymmetry law.

Law 3 (Refinement antisymmetric). $(c_1 \sqsubseteq c_2) \land (c_2 \sqsubseteq c_1) \Rightarrow (c_1 = c_2)$

While the transitivity property of the refinement relation supports stepwise refinement, antisymmetry reduces proofs of equivalence to proofs of mutual refinement, just like equivalence of predicates in the predicate calculus can be established using mutual implication.

Apart from these properties, to allow compositional transformations (independent refinement of subcomponents of compound programs) the language operators should preferably be monotonic with respect to $\sqsubseteq$. For example, $c_1 \sqsubseteq c_2$ must imply $c_1 ; c_3 \sqsubseteq c_2 ; c_3$. In general, we have the result below, where F is a context: a function on programs built from the language operators.

Law 4 (Refinement compositional). $(c_1 \sqsubseteq c_2) \Rightarrow (F(c_1) \sqsubseteq F(c_2))$

Ideally, this must hold for all valid contexts F. Some languages, nevertheless, allow constructs which are not monotonic with respect to $\sqsubseteq$, and therefore re-

strictions must be imposed on F so that the above law holds. This is the case, for instance, of private attributes in object-oriented languages. They cannot be used to build contexts since an improved class does not necessarily has the same private attributes of the original class.

A formal definition of the refinement relation can be given in the weakest precondition model, in a simple and intuitive way. A refined program must work in at least the same set of states as the original program, but possibly in a larger set. The stronger a predicate is, the smaller is the set of elements it defines. Therefore, for all postconditions ψ, the weakest precondition of a refined program must be no stronger than that of the original program.

$$(c_1 \sqsubseteq c_2) \mathrel{\widehat{=}} wp.c_1.\psi \Rightarrow wp.c_2.\psi$$

Refinement can also be understood as a reduction of nondeterminism. Therefore, if the nondeterministic choice between c_1 and c_2 always yields c_1, this means that c_2 refines c_1. This gives an alternative characterisation of $\sqsubseteq$.

$$(c_1 \sqsubseteq c_2) \mathrel{\widehat{=}} (c_1 \sqcap c_2 = c_1)$$

Nondeterminism is used in specifications to provide abstraction: choices that are better made during design or implementation are left open.

Exercise 1. Derive the above definition of refinement from the previous one and the weakest precondition semantics of $\sqcap$ given in Section 2.

5 Refinement Algebra

Program transformation with the preservation of semantics can be formally justified in terms of a semantic model like weakest precondition or Hoare logic, as discussed in previous sections. For instance, a program c_1 can be safely transformed into a program c_2 provided c_1 and c_2 have the same weakest precondition; the transformation is also valid if c_2 is a refinement of c_1 (the weakest precondition of c_1 implies that of c_2).

In an algebraic style of reasoning, the properties of the programming constructs are captured by equations and inequations (laws) that directly relate these constructs. An attractiveness of algebraic reasoning, therefore, is that it is entirely conducted at the programming level; at least in principle, this seems more appealing for programmers. In this approach, given that the algebraic laws are sound, transformations based on their application are also correct by construction. Soundness of the laws is considered a separate issue; this is done by proving the laws in a mathematical model, like weakest precondition. The focus here is on the presentation of the laws, rather than on their proofs.

First we consider simple properties of **skip**, assignment and sequential composition. For example, the following law states that the assignment of the value of a variable to itself has no effect.

Law 5 (Void assignment). $(x := x) = \mathbf{skip}$

Such a vacuous assignment can also occur as part of a multiple assignment.

Law 6 (Identity assignment). $(x, y := e, y) = (x := e)$

The list of variables and expressions may be subjected to the same permutation, without changing the effect of the assignment.

Law 7 (Assignment symmetry). $(x, y := e, f) = (y, x := f, e)$

Two assignments to the same variables can be readily combined into a single assignment.

Law 8 (Combine assignments). $(x := e;\ x := f) = (x := f[e/x])$

The notation $f[e/x]$ denotes the substitution of e the free occurrences of x in f.

As **skip** has no effect, it is both the left and the right identity of sequence.

Law 9 (Composition identity). $(\mathbf{skip};\ c) = c = (c;\ \mathbf{skip})$

A comprehensive set of laws for imperative programming can be found in [116]. The purpose here is to illustrate the algebraic reasoning style. As a simple example, we prove that assignments can be swapped when there is no interference.

Example 1 (Swap assignments). Consider x, y, w and z are distinct identifiers. Then $(x := y;\ w := z) = (w := z;\ x := y)$

Proof.

$$
\begin{array}{ll}
x := y;\ w := z & \text{[Law 6]} \\
x, w := y, w;\ w, x := z, x & \text{[Law 7]} \\
w, x := w, y;\ w, x := z, x & \text{[Law 8]} \\
w, x := z, y & \text{[Law 8]} \\
w, x := z, x;\ w, x := w, y & \text{[Law 7]} \\
w, x := z, x;\ x, w := y, w & \text{[Law 6]} \\
w := z;\ x := y &
\end{array}
$$

The laws allow us to prove that the two sequences of assignments (although syntactically different) behave the same and, therefore, are semantically equivalent.

A nice feature of the algebraic style is modularity. One can explore program properties incrementally, considering one construct at a time. For example, let us now deal with variable declaration. A simple property is that, if the declared variable is not used in its scope, then the declaration has no effect.

Law 10 (Void declaration). $(\mathbf{var}\ x : T \bullet c\ ) = c$ **provided** *x is not free in c*

Recall that an occurrence of a variable x in c is *bound* (or local) if it is in the scope of a declaration of x in c, and *free* (or global) otherwise.

Another property of local variable declaration is that assigning to a variable at the end of its scope has no effect.

Law 11 (Assignment elimination).

$$(\mathbf{var}\ x : T \bullet c;\ x, y := e, f\ ) = (\mathbf{var}\ x : T \bullet c;\ y := f\ )$$

As is usual in an algebraic presentation, the introduction of the new laws for declaration has no impact on the previous laws; actually they contribute to the set of properties that hold of our simple programming language. Therefore, the proof of Example 1 does not need to be revised. Transformations involving declarations can now be performed using the previous and the new laws. The following exercise serves as an illustration.

Exercise 2. Assuming that z is not free in $x := y$, and that these variables have type T, prove the following equivalence:

$$(\mathbf{var}\ z : T \bullet z := y;\ x := z\ ) = (x := y)$$

As previously discussed, during program transformation, sometimes the resulting program does not behave exactly as the original program, but is possibly better than (a refinement of) it. The following is an example of a refinement law.

Law 12 (Declaration initialisation).

$$(\mathbf{var}\ x : T \bullet c\ ) \sqsubseteq (\mathbf{var}\ x : T \bullet x := e;\ c\ )$$

Since the initial value of a declared variable is totally arbitrary, initialisation of a variable may reduce nondeterminism, leading to a more predictable program.

Nondeterminism can be understood as allowing choices to be made. Program development usually starts with abstract specifications which leave several design decisions for the programmer to take. One important issue in refining a specification into a program is reducing nondeterminism. This is addressed in further detail in the next section.

6 Specification and Program Development

While the notation introduced so far exemplifies well-known (executable) programming constructs, it is not suitable for writing abstract specifications. In the view followed by consolidated approaches to program development, a mathematical trick is applied: the programming language is embedded within a more general specification notation. In this way, a single notation is used both for programming and for specification; programs appear as a special kind of specification. Therefore, program development reduces to transformations of specifications within a uniform framework. Examples of approaches which adopt this view are the refinement calculi by Back [14], Morgan [192] and Morris [199].

A distinguishing feature of Morgan's calculus is the *specification statement*:

$$w : [pre, post]$$

which describes a program that, when executed in a state satisfying the precondition *pre*, terminates in a state satisfying the postcondition *post*, possibly modifying the values of variables in the list (frame) w.

As an example, consider the specification statement

$$s, r : [e \notin s, s = s_0 \cup \{e\} \land r = \texttt{"}Okay\texttt{"}]$$

whose effect is to add a new element e to a set s, and assign to r the constant "*Okay*", indicating successful execution of the operation. By convention, occurrence of framed variables in the precondition refer to their initial values, whereas in the postcondition such occurrences refer to the final values of the framed variables. To reference initial values of framed variables in the postcondition, a subscript is adopted. Therefore, s_0 stands for the initial value of s in the postcondition above.

This specification can be refined into executable code, as discussed in the sequel. In this way, the language allows us to start with an abstract specification of a program and progressively refine it by mixing code and specifications, and then finally obtain a program with executable constructs only.

Some extreme specifications are of particular interest for reasoning. For example, we can write **abort** as a specification.

$$\mathbf{abort} = x : [\mathbf{false}, \mathbf{true}]$$

It is the worst possible specification. It is never guaranteed to terminate (precondition **false**), and even when it does, its outcome is completely arbitrary (postcondition **true**). It allows any refinement; for instance, programs setting x to arbitrary values. At the other extreme, we have the best possible specification

$$\mathbf{miracle} = x : [\mathbf{true}, \mathbf{false}]$$

which can execute in any state (precondition **true**) and establishes as outcome the impossible postcondition **false**. This is an infeasible specification; it cannot be realised as an executable program. In fact, it is not refined by any other specification or code. So, arriving at this specification during development indicates that the developer should return to a previous development step and make alternative design choices in order to be able to implement the initial specification.

It is also useful in program derivation or transformation to assume that a condition b holds at a given point in the program text. This can be written as $\{b\}$, and defined as follows.

$$\{b\} \mathrel{\widehat{=}} : [b, \mathbf{true}]$$

If b is **false**, $\{b\}$ reduces to **abort**. Otherwise, it behaves like **skip**: always terminates and does nothing. In [192], and in this book, this is called an *assumption*; it coincides with the concept of *assertion* in the setting of Hoare logic.

We can also give a simple specification to **skip**.

$$\mathbf{skip} =: [\mathbf{true}, \mathbf{true}]$$

The empty frame guarantees that no variables are changed.

6.1 Algorithmic Refinement

Morgan's calculus is perhaps the most appealing to practising programmers, since it includes several laws that allow transforming specification statements into executable programs. Some laws relate specification statements. Two of these capture the notion of refinement in program development. The first states that a program can be made more applicable (defined for a larger domain or set of states) when refined; in other words, its precondition can be weakened.

Law 13 (Precondition weakening). $w : [pre, post] \sqsubseteq w : [pre', post]$
provided $pre \Rightarrow pre'$

Concerning the effect (postcondition), refinement might lead to a more deterministic or predictable program, as already discussed in the previous section.

Law 14 (Postcondition strengthening). $w : [pre, post] \sqsubseteq w : [pre, post']$
provided $pre[w_0/w] \wedge post' \Rightarrow post$

This states that strengthening the postcondition, in states which satisfy the precondition, leads to refinement. The substitution of w_0 for w in the proviso is necessary due to the convention that initial values of framed variables in the postcondition are subscripted.

A refinement of our example specification is to increase its applicability, recording an error message in r when the element is already in the set.

$$\begin{aligned} s, r : [\mathbf{true}, &(e \notin s_0 \wedge s = s_0 \cup \{e\} \wedge r = \text{"}Okay\text{"}]) \vee \\ &(e \in s_0 \wedge s = s_0 \wedge r = \text{"}AlreadyMember\text{"})] \end{aligned}$$

Whenever the precondition of the original specification is satisfied (the element is not in the set), the refined version exhibits exactly the same behaviour. When the element is already in the set, the values of z and r are not defined in the original specification, and therefore totally arbitrary. In the refined version, when this happens, s is not modified and r takes the defined value "*AlreadyMember*".

Exercise 3. Prove the refinement:

$$\begin{aligned} & s, r : [e \notin s, s = s_0 \cup \{e\} \wedge r = \text{"}Okay\text{"}] \\ \sqsubseteq \\ & s, r : [\mathbf{true}, (e \notin s_0 \wedge s = s_0 \cup \{e\} \wedge r = \text{"}Okay\text{"}]) \vee \\ & \qquad\qquad (e \in s_0 \wedge s = s_0 \wedge r = \text{"}AlreadyMember\text{"})] \end{aligned}$$

Most of the laws of Morgan's refinement calculus relate particular forms of specification statements to programming constructs, serving as tools to refine abstract

specifications into code which can be effectively executed. The process is known as algorithmic or control refinement. For example, the law below allows the introduction of an assignment.

Law 15 (Assignment introduction). $w, v : [pre, post] \sqsubseteq v := e$
provided $(v = v_0) \land pre \Rightarrow post[e/v]$

It states that if the value of the assigned expression is suitable to establish the postcondition, in contexts where the precondition holds, then the assignment is a valid implementation of such a specification. In the proviso, the condition $(v = v_0)$ is necessary due to the convention that initial values of framed variables in the postcondition are subscripted. For example, in a specification statement of the form $v : [\mathbf{true}, v = v_0 + 1]$, v_0 denotes the initial value of v in the postcondition, whereas in an assignment of the form $v := v + 1$ no subscript variables are used. Identifying $v = v_0$ in the formulation allows to justify this kind of refinement. For this example, the proof obligation is $v = v_0 \land \mathbf{true} \Rightarrow v + 1 = v_0 + 1$, which clearly holds.

As another example, consider the following law.

Law 16 (Following assignment).

$$w, v : [pre, post] \sqsubseteq w, v : [pre, post[e/v]];\ v := e$$

It allows the extraction of an assignment from a specification statement , but still keeps the remaining behaviour as a (modified) specification statement.

Exercise 4. Prove the following refinement:

$$s, r : [e \notin s, s = s_0 \cup \{e\} \land r = \text{"}Okay\text{"}] \sqsubseteq (s := s \cup \{e\};\ r := \text{"}Okay\text{"})$$

6.2 Data Refinement

Complementarily to algorithmic refinement, program development normally involves change of data representation. A typical development starts with a specification whose data structures are abstract and, possibly, not even available in the target programming language. As the development progresses, the abstract data types give rise to more concrete representations.

As a simple example, consider a specification statement similar to that of the previous section, which adds a new element to a set.

$$s : [e \notin s, s = s_0 \cup \{e\}]$$

Then consider a possible implementation using a sequence, as a concrete representation of a set.

$$t : [e \notin \mathsf{set}\ t, t = t_0 \frown \langle e \rangle]$$

We use $\mathsf{set}\ t$ to denote the set with the elements of the sequence t; $\langle e \rangle$ stands for the singleton sequence with element e, and $\frown$ represents sequence concatenation.

Intuitively, the latter specification refines the former, since sequences (lists or arrays) are well-known structures used for implementing sets. Furthermore, both operations have the same observable effect of adding a new element to the relevant data structure. Nevertheless, attempting to prove this refinement using the laws for weakening precondition and strengthening postcondition soon reveals that a direct comparison between the two statements is not possible at all. The reason is that they operate on different data spaces: a set s and a sequence t.

The missing connection is a relation between the abstract and the concrete states. For the particular example,

$$s = \mathsf{set}\ t$$

Based on this relation, it is possible to prove this data refinement. In general, relations between abstract and concrete states can be arbitrary. In many cases of practical interest, nevertheless, these relations are functional, and are called *abstraction functions*. In our example, the relation is functional and the abstraction function is set.

In Morgan's calculus, data refinement is formulated at the level of programming modules. A module includes state variables, a state initialisation, and procedures which act on the module state. Broadly, the technique involves adding the concrete variables to the module being data refined, making the abstract variables auxiliary, and then removing the auxiliary (abstract) variables. When the relation between abstract and concrete states is functional, these steps are combined into a single step.

In this view, our abstract module would include the declaration of the variable s (say, a set of natural numbers), a state initialisation (say, the empty set), and several operations, including the one to insert new elements into the set, as presented above. To proceed with the data refinement, several transformations are proposed by Morgan to deal with initialisation, assignments, specification statements, and so on.

To illustrate the technique, we present a transformation for specification statements. A specification statement

$$w, x : [pre, post]$$

becomes

$$w, z : [pre[\mathsf{af}\ z/x], post[\mathsf{af}\ z_0, \mathsf{af}\ z/x_0, x]]$$

where af stands for the relevant abstraction function, x for the abstract variables, and z for the concrete variables. Observe that this transformation replaces x with z in the frame, and occurrences of x in the pre and in the postcondition with $\mathsf{af}\ z$. In the postcondition, both the initial and final values of x need to be replaced.

Exercise 5. Formalise the data transformation of the statement previously presented: $s : [e \notin s, s = s_0 \cup \{e\}]$ into $t : [e \notin \mathsf{set}\ t, t = t_0 \frown \langle e \rangle]$ considering the abstraction function: $s = \mathsf{set}\ t$.

7 Refinement and Lattices

We have not given a semantics for recursion yet, and presented only a partial semantics for iteration in Section 2. In this section, we come to this point as part of a discussion of refinement as a partial order in a lattice of monotonic predicate transformers [199, 97, 16]. A partial order is a reflexive, anti-symmetric and transitive relation between elements of a set. If, given any two elements of the set, they are always related by the order in some way, then we have a total order. Refinement, however, is a partial order between programs.

To give the semantics and reason about simple (non-recursive) procedures, we can use a copy rule: basically, it allows calls to the procedures to be replaced with their bodies. As an example, we consider the program below, which defines a procedure *Inc*, and calls it twice.

$$\mathbf{proc}\ Inc \mathrel{\widehat=} x := x + 1 \bullet Inc;\ Inc$$

It is equivalent to $x := x + 1;\ x := x + 1$. In the case of a recursive procedure, however, this approach does not work. As a second example, we take a procedure *Sum* that adds the value of x to that of another variable y (and sets x to 0).

$$\begin{array}{l}\mathbf{proc}\ Sum \mathrel{\widehat=} \mathbf{if}\ (x = 0) \rightarrow \mathbf{skip} \\ \qquad\qquad\quad [] (x > 0) \rightarrow y := y + 1;\ x := x - 1;\ Sum \\ \qquad\qquad\quad \mathbf{fi} \\ \bullet\ y := 5;\ x := 10;\ Sum\end{array}$$

In the main program, we call *Sum*; if we replace this call by the body of *Sum*, another call of *Sum* is introduced, so the copy rule does not really sort out the reasoning problem.

The declaration of *Inc* can be seen as a definition of this procedure through the equation

$$Inc = x := x + 1;\ x := x + 1$$

In the case of *Sum*, this equation is

$$Sum = \mathbf{if}\ (x = 0) \rightarrow \mathbf{skip}\ [] (x > 0) \rightarrow y := y + 1;\ x := x - 1;\ Sum\ \mathbf{fi}$$

The body of *Sum* can be regarded as a context: a function from programs to programs, which can be defined as follows using the λ notation.

$$\lambda X \bullet \mathbf{if}\ (x = 0) \rightarrow \mathbf{skip}\ [] (x > 0) \rightarrow y := y + 1;\ x := x - 1;\ X\ \mathbf{fi}$$

Therefore, the equation that defines *Sum* requires that it is a program that, when given as argument to the above function, the result is itself. For any function F, the arguments X for which $F(X) = X$ are called fixed points of F. So, the equation for *Sum* requires it to be a fixed point of the function on programs characterised by its body.

The problem is that fixed points may not exist, or there may be lots of them. So, for the equation characterised by the declaration of a recursive procedure to

be a valid definition, we need to guarantee that there is at least one fixed point, and that, when there are several, we have a way of choosing one. With this end, it is usual taking lattices with various properties as semantic models.

A lattice is a set S with a partial order $\leq$ that satisfies a few extra properties. They are based on the existence of lower and upper bounds for certain subsets of S. For any subset T of S, an upper bound u of T is an element of S such that $t \leq u$ for every t in T. Similarly, a lower bound l is such that $l \leq t$, for every $t \in T$. The least upper-bound is an upper bound that is smaller than all others; likewise, the greatest lower-bound is a lower bound that is bigger than all others.

Definition 1 (Lattice and complete lattice). *A* lattice *is a partially ordered set, in which all* non-empty finite *subsets have both a least upper-bound (join) and a greatest lower bound (meet). A* complete lattice *is a lattice in which* all *subsets have both a join and a meet.* □

Every complete lattice has a bottom (smallest element) and a top (biggest element).

Example 2 (Complete lattices). Complete lattices are often found in mathematics and computer science. We give two examples drawn from mathematics.

1. The power set of a given set S ordered by inclusion forms a complete lattice. The least element is the empty set and the greatest element is S itself. Join is union and meet is intersection of subsets.
2. The set of natural numbers ordered by divisibility forms a complete lattice. Divisibility gives $m \sqsubseteq n$ exactly when $(\exists\, k \bullet k \times m = n)$, and so forms a partial order. The bottom of this lattice is the number *1*, since it exactly divides every other number. The top element of the lattice is *0*, since it can be divided exactly by every other number. The join of finite sets is given by the least common multiple; for infinite sets, the join will always be *0*. The meet is the greatest common divisor, and for infinite sets this may well be greater than *1*. For example, the set of all even numbers has *2* as the greatest common divisor. □

An example of a complete lattice drawn from the area of refinement is the set of monotonic predicate transformers ordered by refinement. A discussed in Section 3, we can use a function *wp* to characterise (the semantics of) programs. For a program p, $wp.p$ is a predicate transformer; the set of all such predicate transformers pt is a convenient model for programs. The partial order $\sqsubseteq$ for predicate transformers defined as

$$pt_1 \sqsubseteq pt_2 \mathrel{\widehat{=}} pt_1.\psi \Rightarrow pt_2.\psi, \text{ for all predicates } \psi$$

corresponds to the refinement relation defined in Section 4.

A function f from a set S with a partial order $\leq_S$ to a set T with a partial order $\leq_T$ is monotonic if, for every x and y in S, if $x \leq_S y$, then $f(x) \leq_T f(y)$.

For every program p, the function $wp.p$ is monotonic. The partial order considered for the set of predicates is implication. If a postcondition ψ_1 implies another postcondition ψ_2, we say that ψ_1 is stronger than ψ_2, because less states satisfy ψ_1, and every state that satisfies ψ_1 also satisfies ψ_2. In this case, for every program p, $wp.p.\psi_1$ implies $wp.p.\psi_2$; this is because if from a particular initial state p is guaranteed to establish ψ_1, then it is also guaranteed to establish ψ_2.

The bottom of the complete lattice of monotonic predicate transformers is **abort**; the top is **miracle**. The join operator the demonic choice, and the meet operator is angelic choice.

Well-known results establish the existence of fixed points for functions on complete lattices. For example, we know that every monotonic function on a complete lattice has a fixed point. Since the body of a recursive procedure is a monotonic function on programs (see Law 4), this is in the direction of what we need to give meaning to recursive procedures. What we still need to sort out is the fact that such functions may have several fixed points. An extreme example is an infinite recursion: $\mathbf{proc}\, Inf \mathrel{\widehat{=}} Inf \bullet \ldots$. The function defined by its body is the identity $\lambda X \bullet X$. Every program is a fixed point of this function.

In such situations, we take the least fixed point to be the definition of the recursive procedure; it is denoted by $\mu X \bullet F(X)$, where F is the body of the procedure written as a function of X. This is the least refined program that satisfies the equation characterised by the definition of the recursive procedure. From the point of view of program development, this is the natural solution, since, as already explained, we want to impose as few restrictions as possible on a specification, and leave design decisions open. The meaning of Inf, for example, is taken to be the least refined program: **abort**.

The Knaster-Tarski fixed point theorem is very much used since it gives an explicit characterisation for the least fixed point of a monotonic function on a complete lattice; it is stated below.

Theorem 1 (Knaster-Tarski fixed point). *For every monotonic function $F(X)$ on a complete lattice with order $\leq$*

$$\mu X \bullet F(X) = \sqcap \{ X : L \mid F(X) \leq X \}$$

We use $\sqcap S$ to denote the greatest lower-bound of the set S.

The greatest fixed point of F also exists; it is denoted by $\nu X \bullet X$, and it is possible to prove that $\nu X \bullet F(X) = \bigsqcup \{ X : L \mid X \leq F(X) \}$. For a set S, $\bigsqcup S$ is its least upper-bound.

As explained above, the top of the lattice of monotonic predicate transformers is **miracle**, an infeasible program that is not implementable. Some semantic models do not include this program, or any program that may behave like **miracle** in some situations. In this case, it is usual that, instead of a complete lattice, the semantic model is a CPO: complete partially ordered set.

Definition 2 (Complete partially ordered set). *A* CPO *(complete partially ordered set) is a partially ordered set, which has a bottom, and in which every* directed *subset has a least upper-bound.*

A set is directed if all its finite subsets have an upper bound as one of its own elements. Monotonic functions on a CPO also have fixed points. If, in addition, the function is continuous, then it has a least fixed point as characterised in the theorem below. A function F is continuous if, and only if, it distributes over least upper-bounds of directed sets: $F(\bigsqcup D) = \bigsqcup\{\, d : D \bullet F(d) \,\}$, for every directed set D.

Theorem 2. *For every continuous function $F(X)$ on a CPO with order $\leq$ and bottom element $\bot$*

$$\mu\, X \bullet F(X) = \bigsqcup\{\, n : \mathbb{Z} \bullet F^n(\bot) \,\}$$

For a function F, we define F^0 to be the identity function ($F^0(X) = X$), and $F^n(X) = F(F^{n-1}(X))$, for $n > 0$. Continuity can be a strong property, and is not satisfied by many programs involving unbounded nondeterminism [31]. A more comprehensive account of lattice theory, including proofs for the theorems presented above, can be found in [77].

8 Final Considerations

There are several program and programming models. We have briefly discussed here Hoare logic, refinement algebra, and weakest preconditions. Other models are presented in later chapters of this book. Chapters 3 and 6 give different relational models for the process algebra CSP, and Chapter 4 gives a weakest precondition model for probabilistic programs. Chapter 5 adopts a more operational model (transition systems) and the TLA (Temporal logic of Actions) notation to explore specification, verification and scheduling of real-time and fault-tolerant systems. A common feature to all of them is a formal characterisation of refinement; this is central to any model that supports a development technique.

Potential applications of algebraic reasoning in the context of object-oriented programming is the major objective of the next chapter. Laws of the process algebra CSP are explored in Chapter 3. Laws of programming involving probability are explored in Chapter 4.

Typically, the use of refinement techniques is potentially a very error-prone activity that involves copious formula manipulations. If the benefits of the use of a rigorous programming approach are not to be lost, and are to be available for large-scale industrial systems, the use of tools is essential. Many products are available. Model checking techniques have been particularly attractive to industry due to their high level of automation; they are discussed in detail in Chapter 8. A set of tools that have been very successful in industry for the verification of control system in the area of avionics is presented in Chapter 7.