Chapter 23

Semantics

23.1 Introduction

'Semantics' means meaning. In this final chapter we look back and examine the mathematical meaning of our program development method.

There are several reasons for taking an interest in foundations. The first, for the new reader in this subject, is to provide references to the research from which this style of program development has arisen.

The second reason is to address the nervous reader: Why are these dozens of laws necessarily consistent? Where have they come from? Why do they, or indeed does refinement itself, have any connection with real programming?

Questions like those should not be asked during program development, for there it is already too late; those who only *apply* the method need not read this chapter. But those who *select* a program development method regard such questions as crucial: they have to be sure that it works.

The answers are usually given by deciding first of all on a reasonable way of viewing programs — in fact, a mathematical model for them. That model should correspond closely with the way that computers operate, so that there is no doubt about its appropriateness. (Otherwise the questions will only have been moved, not answered.) Then the laws are checked, one by one, against the model. If the check is passed, and one has accepted the model, then by that very fact one has accepted the law also. As an example, we check *assignment* 5.2 in Section 23.3.3 below.

Our mathematical model is the predicate transformer, popularised by E.W. Dijkstra [Dij76]. The treatment of it below is brief, even condensed; for the novice, pursuit of the introductory references is probably essential. Any of [Dij76, Gri81, Heh84, Bac86, DF88] is a good starting point.

23.2 Predicate transformers

In Section 1.4.3 the operation of a specification was described as follows:

> *If* the initial state satisfies the precondition *then* change only the variables listed in the frame so that the resulting final state satisfies the postcondition.

In fact, that description could apply to the behaviour of any command: if for *any* postcondition we know which preconditions will guarantee termination in a final state satisfying the postcondition, then we say that we know the meaning of the command.

For command *prog* and postcondition $\mathcal{A}$, let

$$wp(prog, \mathcal{A})$$

be the *weakest* precondition sufficient to ensure termination in a state described by $\mathcal{A}$. In that way we can see *prog* as a predicate transformer, because it transforms the postcondition $\mathcal{A}$ into the weakest precondition $wp(prog, \mathcal{A})$. And with it we know the meaning of *prog*: a precondition $\mathcal{B}$ will guarantee that *prog* terminates in a state described by $\mathcal{A}$ precisely when

$$\mathcal{B} \quad \Rrightarrow \quad wp(prog, \mathcal{A}) \ .$$

For example, the meaning of $x := x + 1$ is a predicate transformer that takes the postcondition $x > 0$ to the precondition $x \geq 0$, because that precondition is the weakest one whose truth initially guarantees termination of $x := x + 1$ and truth of $x > 0$ finally. Thus $x \geq 0$ is the weakest precondition of $x := x + 1$ with respect to $x > 0$, and we write

$$wp(x := x + 1, x > 0) \quad \equiv \quad x \geq 0 \ . \tag{23.1}$$

The next section gives the meaning of our program development method in terms of predicate transformers.

23.3 Semantic definitions

23.3.1 Guarded commands

Semantics for the ordinary guarded commands are introduced in [Dij76] and repeated in [Heh84, Gri81]. In [Bac86], similar definitions are given for Pascal. Here for example we give the semantics of assignment:

Definition 23.1 assignment For any postcondition $\mathcal{A}$,

$$wp(w := E, \mathcal{A}) \ \widehat{=} \ \mathcal{A}[w \backslash E] \ .$$

□

$$
\begin{array}{rcl}
wp(\mathbf{skip}, \mathcal{A}) & \widehat{=} & \mathcal{A} \\
wp(\mathbf{abort}, \mathcal{A}) & \widehat{=} & \mathsf{false} \\
wp(x := E, \mathcal{A}) & \widehat{=} & \mathcal{A}[x \backslash E] \\
wp(P;\ Q, \mathcal{A}) & \widehat{=} & wp(P, wp(Q, \mathcal{A}))
\end{array}
$$

$$
\begin{array}{l}
wp(\mathbf{if}\ ([\!]\, i \cdot G_i \rightarrow P_i)\ \mathbf{fi}, \mathcal{A}) \\
\qquad \widehat{=} \qquad (\vee i \cdot G_i) \\
\qquad\qquad \wedge \ \ (\wedge i \cdot G_i \Rightarrow wp(P_i, \mathcal{A}))
\end{array}
$$

Iteration **do** $\cdots$ **od** is a special case of recursion, dealt with in Section 23.3.9.

Figure 23.1 Predicate transformers for guarded commands

With Definition 23.1 we verify the claim (23.1) above:

$$
\begin{array}{ll}
 & wp(x := x + 1, x > 0) \\
\equiv & \text{“}\textit{assignment}\ 23.1\text{”} \\
 & (x > 0)[x \backslash x + 1] \\
\equiv & x + 1 > 0 \\
\equiv & x \geq 0\ .
\end{array}
$$

Figure 23.1 gives predicate transformers for all the basic guarded commands except iteration.

23.3.2 Specifications

Our most significant extension to the ordinary guarded commands is the specification, and this is its meaning:

Definition 23.2 specification

$$wp(w\colon [pre\ ,\ post], \mathcal{A}) \ \widehat{=}\ pre \wedge (\forall\, w \cdot post \Rightarrow \mathcal{A})\, [v_0 \backslash v]\ ,$$

where the substitution $[v_0 \backslash v]$ replaces *all* initial variables by corresponding final variables.

□

Note that initial variables v_0 never occur in postconditions $\mathcal{A}$.

As an example of the above, suppose we use $x := x \pm 1$ to abbreviate

$$x\colon [x = x_0 + 1 \vee x = x_0 - 1]\ .$$

Then we have

$$
\begin{array}{ll}
 & wp(x := x \pm 1, \mathcal{A}) \\
\equiv & wp(x\colon [x = x_0 + 1 \vee x = x_0 - 1], \mathcal{A}) \\
\equiv & \mathsf{true} \wedge (\forall\, x \bullet (x = x_0 + 1 \vee x = x_0 - 1) \Rightarrow \mathcal{A})\,[x_0 \backslash x] \\
\equiv & \text{“Predicate laws A.34, A.65”} \\
 & (\forall\, x \bullet x = x_0 + 1 \Rightarrow \mathcal{A})\,[x_0 \backslash x] \wedge (\forall\, x \bullet x = x_0 - 1 \Rightarrow \mathcal{A})\,[x_0 \backslash x] \\
\equiv & \text{“Predicate law A.56”} \\
 & \mathcal{A}[x \backslash x_0 + 1][x_0 \backslash x] \wedge \mathcal{A}[x \backslash x_0 - 1][x_0 \backslash x] \\
\equiv & \text{“}\mathcal{A} \text{ contains no } x_0\text{”} \\
 & \mathcal{A}[x \backslash x + 1] \wedge \mathcal{A}[x \backslash x - 1]\ .
\end{array}
$$

‘Specifications’ were first added to the guarded command language in [Bac78, Bac80], though not in our form; in particular, miracles were not allowed (see Section 23.3.6 below). Later they appeared in [Mee79]; most recently they appear in [Mor87, MR87, Mor88d, Bac88]. References [MR87, Mor88d] are closest to this book.

23.3.3 Refinement

Refinement $\sqsubseteq$ is a relation between commands (just as $\leq$ is a relation between numbers): for any commands *prog*1 and *prog*2, either *prog*1 $\sqsubseteq$ *prog*2 holds or it does not. This is the definition:

Definition 23.3 <u>refinement</u> For any commands *prog*1 and *prog*2, we say that *prog*1 is refined by *prog*2, writing *prog*1 $\sqsubseteq$ *prog*2, exactly when for all postconditions $\mathcal{A}$ we have

$$wp(prog1, \mathcal{A}) \quad \Rrightarrow \quad wp(prog2, \mathcal{A})\ .$$

□

Definition 23.3 is used in all approaches to the refinement calculus (for example [Bac80, MR87, Mor87, Mor88d]), and in other places as well [Heh84, Abr87]. It seems to be the only reasonable definition for sequential program development based on weakest preconditions.

With our definitions so far, we can see refinement in action. For example, it is easy to verify both of these:

$$
\begin{array}{lcl}
x := x \pm 1 & \sqsubseteq & x := x + 1 \\
x := x \pm 1 & \sqsubseteq & x := x - 1\ .
\end{array}
$$

More interesting, however, is showing the proof of one of our laws. Suppose we have $w = w_0 \wedge pre \Rrightarrow post[w \backslash E]$, the proviso of *assignment* 5.2; then we reason

$$\begin{array}{ll}
 & wp(w\colon [pre\ ,\ post]\,, \mathcal{A}) \\
\equiv & \text{“}\textit{specification}\ 23.2\text{”} \\
 & pre \wedge (\forall\, w \cdot post \Rightarrow \mathcal{A})\,[v_0 \backslash v] \\
\equiv & \text{“Predicate law A.56; } \textit{pre} \text{ contains no } v_0\text{”} \\
 & (\forall\, v_0 \cdot v_0 = v \Rightarrow pre \wedge (\forall\, w \cdot post \Rightarrow \mathcal{A})) \\
\Rrightarrow & \text{“assumed proviso; } v_0 \text{ includes } w_0\text{”} \\
 & (\forall\, v_0 \cdot v_0 = v \Rightarrow post[w \backslash E] \wedge (\forall\, w \cdot post \Rightarrow \mathcal{A})) \\
\Rrightarrow & \text{“Predicate law A.86”} \\
 & (\forall\, v_0 \cdot v = v_0 \Rightarrow post[w \backslash E] \wedge (post[w \backslash E] \Rightarrow \mathcal{A}[w \backslash E])) \\
\Rrightarrow & (\forall\, v_0 \cdot v = v_0 \Rightarrow \mathcal{A}[w \backslash E]) \\
\equiv & \text{“}\mathcal{A},\ E \text{ contain no } v_0\text{”} \\
 & \mathcal{A}[w \backslash E] \\
\equiv & \text{“}\textit{assignment}\ 23.1\text{”} \\
 & wp(w := E, \mathcal{A})\ .
\end{array}$$

With that, and *refinement* 23.3, we have proved *assignment* 5.2.

But we have done more: we have shown how to prove a law which formerly we had to take on faith. Such proofs have been used to establish the consistency of all the laws in this book, ensuring that no contradictions can occur. And new laws can be added, supported by similar proofs.

23.3.4 Local variables

Local variable declarations are code, and their meaning is this:

Definition 23.4 local variable

$$wp(|[\ \mathbf{var}\ x \cdot prog\ ]|, \mathcal{A})\ \ \widehat{=}\ \ (\forall\, x \cdot wp(prog, \mathcal{A}))\,,$$

provided $\mathcal{A}$ contains no free x.
□

Definition 23.4 is well known but not often quoted. The proviso, similar to others below, is easily circumvented by renaming x in *prog* to some other fresh variable.

Typed local variables are discussed in Section 23.3.11.

23.3.5 Logical constants

Logical constant declarations are *not* code, but they have meaning nevertheless:

Definition 23.5 logical constant

$$wp(|[\ \mathbf{con}\ x \cdot prog\ ]|, \mathcal{A})\ \ \widehat{=}\ \ (\exists\, x \cdot wp(prog, \mathcal{A}))\,,$$

provided $\mathcal{A}$ contains no free x.
□

Definition 23.5 is not so well known; it appears in [MG90]. In [Mor] it is introduced specifically for procedures. Our use of it is independent of procedures: we make precise the long-established use of logical constants for referring in the postcondition to initial values. Other uses have been discovered, in data refinement [MG90] for example.

As noted in [MG90], logical constants do not satisfy E.W. Dijkstra's Property 3 [Dij76, p.18]: that is, they do not distribute conjunction.

23.3.6 Feasibility

A command is feasible if it obeys the *Law of the Excluded Miracle* [Dij76, p.18]. That gives the following definition:

Definition 23.6 feasibility Command *prog* is *feasible* exactly when

$$wp(prog, \mathsf{false}) \quad \equiv \quad \mathsf{false} \ .$$

Otherwise it is *infeasible*.
□

Infeasible commands, because they break the law, are called *miracles*. They were introduced in [Mor87, Mor88d], and in [Nel89] (but not for refinement). Miracles are used also in [Abr87], and in [Mor88b] for data refinement.

It is easy to show that miracles refine only to miracles (just apply the definitions above), and hence never to code: E.W. Dijkstra's law paraphrased reads 'all code is feasible'.

23.3.7 Annotations

These definitions follow from the above, given *assumption* 1.6 and *coercion* 17.1. Assumptions are defined independently in [Mor87, Bac88, MV89]; coercions are defined in [MV89].

Definition 23.7 assumption

$$wp(\{pre\}, \mathcal{A}) \ \widehat{=} \ pre \wedge \mathcal{A} \ .$$

□

Definition 23.8 coercion Provided *post* contains no initial variables,

$$wp([post], \mathcal{A}) \ \widehat{=} \ post \Rightarrow \mathcal{A} \ .$$

□

All assertions are feasible; no coercion is feasible except [true].

23.3.8 Substitutions

The following definitions come from [Mor88c]:

Definition 23.9 substitution by value

$$wp(prog[\textbf{value}\ f \backslash E], \mathcal{A}) \ \widehat{=}\ wp(prog, \mathcal{A})[f \backslash E]\ ,$$

provided f does not occur free in $\mathcal{A}$.
□

Definition 23.10 substitution by result

$$wp(prog[\textbf{result}\ f \backslash a], \mathcal{A}) \ \widehat{=}\ (\forall f \cdot wp(prog, \mathcal{A}[a \backslash f]))\ ,$$

provided f does not occur free in $\mathcal{A}$.
□

Definition 23.11 substitution by value-result

$$wp(prog[\textbf{value result}\ f \backslash a], \mathcal{A}) \ \widehat{=}\ wp(prog, \mathcal{A}[a \backslash f])[f \backslash a],$$

provided f does not occur free in $\mathcal{A}$.
□

Those three definitions account for all the simple substitution laws of Chapter 11. Procedures and parameters are treated in [Bac87, Mor] also.

23.3.9 Recursion

This is the standard definition:

Definition 23.12 recursion Let $\mathcal{C}(p)$ be a program fragment in which the name p appears. Then

$$\textbf{re}\ p \cdot \mathcal{C}(p)\ \textbf{er}$$

is the *least-refined* program *fix* such that $\mathcal{C}(fix) = fix$.
□

Take **re** $p \cdot p$ **er**, for example. Since $prog = prog$ holds for all programs *prog*, the *fix* in Definition 23.12 is in this case the least-refined of all programs: it is **abort**.

Definition 23.12 gives indirectly the meaning of iteration, since iteration can be viewed as a certain kind of recursion:

$$\begin{array}{lcl} \textbf{do}\ G \rightarrow prog\ \textbf{od} & = & \textbf{re}\ P \cdot \\ & & \quad \textbf{if}\ G\ \textbf{then}\ prog;\ \ P\ \textbf{fi} \\ & & \textbf{er}\ . \end{array}$$

The meaning above generalises the standard meaning of iteration [Dij76, Gri81] in a way first explored in [Boo82]. Recursion is treated in [Nel89], and recursive procedures in [Bac87, Mor]. Recursion is much used in [Heh84].

23.3.10 Data refinement

Data refinement is abundantly defined; our definition appears in [GM91, Mor88a, Mor89, MG90, CU89]:

Definition 23.13 data refinement Let a be a list of variables called *abstract*, let c be a list of variables called *concrete*, and let I be a formula called the *coupling invariant*. Then command *progA* is data-refined to *progC* by a, c, I exactly when for all postconditions $\mathcal{A}$ not containing c we have

$$(\exists\, a \cdot I \wedge wp(progA, \mathcal{A})) \quad \Rrightarrow \quad wp(progC, (\exists\, a \cdot I \wedge \mathcal{A}))\ .$$

□

The approach of Chapter 17 is based mainly on [Mor88a, MG90].

23.3.11 Types

A precise treatment of types, and invariants, appears in [MV89], where the semantic function *wp* is extended to take them into account. The effect is roughly as follows.

Typed local variable and logical constant declarations are a combination of an untyped declaration and the imposition of a local invariant in which the type information appears. For example, the typed declaration $n : \mathbb{N}$ appears as $n \in \mathbb{N}$ in the local invariant. At any point in a program, the surrounding local invariants are conjoined and called the *context*, and the semantic function *wp* takes that context as an extra argument.

In a context C, commands behave as if C were assumed initially (aborting otherwise), and they are guaranteed to establish C if they terminate. For specifications, that effect is gained by imagining the context conjoined to both pre- and postcondition. Thus in the context $x \geq 0$ we have

$$\begin{array}{ll}
 & x := x \pm 1 \\
= & x\colon [x = x_0 + 1 \vee x = x_0 - 1] \\
= & \text{“impose invariant in pre- and postcondition”} \\
 & x\colon [x \geq 0\ ,\ x \geq 0 \wedge (x = x_0 + 1 \vee x = x_0 - 1)] \\
= & \mathbf{if}\ x > 0 \rightarrow x := x - 1 \\
 & [\!]\ \ x \geq 0 \rightarrow x := x + 1 \\
 & \mathbf{fi}\ .
\end{array}$$

(The last equality can be shown using weakest preconditions directly; our laws would show only refinement.)

Note that the alternation above can abort when $x < 0$ initially, and when $x = 0$ initially the possibility $x := x - 1$ is automatically avoided.

Commands that cannot avoid breaking the invariant go ahead and break it, but establish it too: thus they are miracles. For example, in the same context we have

$$
\begin{array}{ll}
 & x := -1 \\
= & x\colon [x = -1] \\
= & \text{“impose invariant in pre- and postcondition”} \\
 & x\colon [x \geq 0\ ,\ x \geq 0 \wedge x = -1] \\
= & x\colon [x \geq 0\ ,\ \mathsf{false}]\ .
\end{array}
$$

Thus $x := -1$, normally called ill-typed, is just a miracle when the declaration $x : \mathbb{N}$ is in effect: type checking is just feasibility checking.

Local invariants resulting from type declarations are called *implicit*; those introduced by **and** are called *explicit*. Explicit local invariants are not code; they are removed by laws that distribute them through compound programs towards atomic components. Removing a local invariant immediately surrounding, say, an assignment amounts only to checking that the assignment preserves it.