Chapter 17

State transformation and data refinement

In the last chapter, where we met modules for the first time, the notion of module refinement was introduced: it was argued that refinement of a module's individual procedures refined the module overall.

We now consider a much more radical possibility, in which we can carry out quite startling changes within the module while still being sure of its overall refinement. The technique is known as state transformation, or data refinement.

17.1 What we cannot yet show

Consider the module Tag' of Figure 17.1. It is a refinement of module Tag of Figure 16.1, although we cannot yet show it to be — for suppose the contrary, that $Tag \not\sqsubseteq Tag'$: then a client expecting Tag would have to be disappointed by Tag'. That means in turn that there is some program whose behaviour would be detectably different if Tag were replaced by Tag' — otherwise, the client would have no grounds for complaint! But there is no such program: using Tag', any series of *Acquire* and *Return* will produce successively higher values of t, starting from some randomly chosen value. And using Tag, exactly the same could have happened. (One might argue that it is unlikely: but still it is *possible*.) Thus $Tag \not\sqsubseteq Tag'$ is *not* true.

On the other hand, we can see easily that $Tag' \not\sqsubseteq Tag$ is indeed the case. For Tag can *Acquire* 1 then 0, and that is something Tag' could never do. Thus $Tag \sqsubseteq Tag'$ but $Tag' \not\sqsubseteq Tag$, and so $Tag \sqsubset Tag'$: a strict refinement. The former could be a definition module, and the latter one of its implementation modules.

Although we have argued that $Tag \sqsubseteq Tag'$, indeed we cannot show that rigorously with our techniques so far: the individual procedures Tag' do not refine their counterparts in Tag; moreover, the states of the two modules are completely different. Such differences, however, are of great importance in program development. Module Tag contains a *set* of natural numbers, and few programming languages

module *Tag'*
  var $n : \mathbb{N}$;

  procedure *Acquire* (**result** $t : \mathbb{N}$)
  $\widehat{=}$ $t, n := n, n + 1$;

  procedure *Return* (**value** $t : \mathbb{N}$)
  $\widehat{=}$ **skip**
end

Figure 17.1 A refinement of Figure 16.1

accept that as code. But *Tag'* contains only a single natural number, and that is far more realistic.

We see shortly that it is precisely because the state of a module cannot be accessed directly — its local variables — that we are free to change that state; we can replace more abstract variables (like sets) by more concrete variables (like simple numbers), provided the difference cannot be detected by use of the exported procedures.

Such a change of state is sometimes known as *change of representation*; if the change tends from abstract to concrete (towards code, in other words), it is known as *data refinement.*

17.2 State transformation

State transformation, carried out on the interior of a module, results in refinement of its external behaviour. We consider two specific transformations: one adds variables to a module; the other removes variables from a module.

To add variables, a *coupling invariant* is chosen, relating the existing variables to the new ones; it can be any formula over the local and exported variables of the module. (It may not refer to imported variables.) Declarations of the new variables are added; the initialisation is strengthened by conjoining the coupling invariant; every guard may assume the coupling invariant; and every command is extended by modifications to the new variables that *maintain* the coupling invariant. The resulting module then refines the original.

To remove variables, they must be first made auxiliary by refining the procedures of the module individually. A set of variables is *auxiliary* if its elements occur only in assignments or specifications whose changing variables are in the same set, so that other variables cannot depend on them. Then the declarations and all occurrences of those variables are removed. Again, the resulting module refines the original.

Often the two steps are carried out in succession — augmentation to add variables, then diminution to remove them — though in special cases we can bundle them together in one step.

Before we look at augmentation however, the first of the two, we must take a brief detour.

17.3 Coercions

We have already met assumptions, formulae $\{pre\}$ between braces that act as **abort** unless the formula holds as that point in the program. Complementary to assumptions are *coercions*, which *make* a formula true at that point in the program. Here is the abbreviation:

Abbreviation 17.1 coercion Provided *post* contains no initial variables,

$$[post] \;\widehat{=}\; : [\mathsf{true}\ ,\ post]\ .$$

□

A coercion to *post* behaves like **skip** if *post* is true, and **magic** otherwise. As do assumptions, coercions have an empty frame, and true as one of their constituent formulae. For both, the explicit appearance of the sequential composition operator is optional.

Here are two simple laws for coercions:

Law 17.2 absorb coercion A coercion following a specification can be absorbed into its postcondition.

$$w\colon [pre\ ,\ post]\,;\ [post'] \quad = \quad w\colon [pre\ ,\ post \wedge post']\ .$$

□

(Compare *absorb assumption* 1.8.)

Law 17.3 introduce coercion **skip** is refined by any coercion.

$$\mathbf{skip} \quad \sqsubseteq \quad [post]\ .$$

□

(Compare *remove assumption* 1.10.)

We should also mention the following law for absorbing an assumption *following* (rather than preceding) a specification. It is

Law 17.4 establish assumption An assumption after a specification can be removed after suitable strengthening of the precondition.

$$\begin{array}{ll} & w\colon [pre\ ,\ post]\,;\ \ \{pre'\} \\ = & w\colon [pre \wedge (\forall\, w \cdot post \Rightarrow pre')\,[w_0\backslash w]\ ,\ post]\ \ . \end{array}$$

□

Law 17.4 exploits the fact that we do not distinguish a program that can abort later (because *post* can be true but pre' false) from one that aborts sooner (because in its precondition $(\forall\, w \cdot post \Rightarrow pre')\,[w_0\backslash w]$ is false).

Assumptions and coercions are together known as *annotations.* Coercions in particular have many surprising — and useful — properties that are explored in the exercises for this chapter. One specific use is in augmentation, to which we now turn.

17.4 Adding variables: augmentation

Each of the following laws deals with an aspect of adding new variables. We assume throughout that the new variables are c, and that the coupling invariant is CI. Note that they are not *refinement* laws, and so do not contain the symbol $\sqsubseteq$. Rather they are transformations, for which we use the word '*becomes*'.

In our examples below, we suppose the module already contains a variable p, and that the new variables are q and r. The coupling invariant is $p = q + r$, and all three variables have type $\mathbb{N}$.

17.4.1 Declarations

Declarations of the new variables are added to the module. For the example, we would add **var** $q, r : \mathbb{N}$.

17.4.2 Initialisation

The coupling invariant is conjoined to the initialisation.

Law 17.5 augment initialisation The initialisation I *becomes* $I \wedge CI$.
□

If the initialisation were $p = 1$, it would become $p = 1 \wedge p = q + r$.

17.4.3 Specifications

The coupling invariant is conjoined to both the pre- and postcondition of specifications, and the frame is extended to allow the new variables to change.

Law 17.6 augment specification The specification w: $[pre\ ,\ post]$ *becomes*

$$w, c\colon [pre\ ,\ CI\ ,\ post]\ .$$

□

For example, the command p: $[p > 0\ ,\ p < p_0]$ *becomes*

$$p, q, r\colon [p > 0\ ,\ p = q + r\ ,\ p < p_0]\ .$$

17.4.4 Assignments

Assignments are extended so that they can change the new concrete variable, but they too must preserve the coupling invariant:

Law 17.7 augment assignment The assignment $w := E$ can be replaced by the fragment

$$\{CI\}\ w, c := E, ?\ [CI]\ .$$

□

Note the similarity between Laws 17.6 and 17.7: in each, the frame is widened to include c, and the coupling invariant appears before (in the precondition, or assumption) and after (in the postcondition, or coercion). As a special case of *augment assignment* 17.7, however, we have

Law 17.8 augment assignment The assignment $w := E$ can be replaced by the assignment $w, c := E, F$ provided that

$$CI \quad \Rrightarrow \quad CI[w, c \backslash E, F]\ .$$

□

Law 17.8 is easily obtained by refining the right-hand side of Law 17.7. (See Exercise 17.32.)

As an example of augmenting assignments, the command $p := p + 1$ can be replaced by

$$\{p = q + r\}\ p, q, r := p + 1, ?, ?\ [p = q + r]\ ,$$

which in turn can be refined to

$$p, q := p + 1, q + 1\ ,$$

for example. The effect of the coercion is to force the two open assignments ?,? to resolve to values that make the formula hold (the formula is $p = q + r$, in this case). There may be many such values, and so there are of course other possibilities for the augmentation. If the notion of coercion seems too surprising, remember that the alternative, *augment assignment* 17.8, reaches $p, q := p + 1, q + 1$ in a single step.

17.4.5 Guards

Each guard can be replaced by another whose equivalence to the first follows from the coupling invariant.

Law 17.9 augment guard The guard G may be replaced by G' provided that

$$CI \quad \Rrightarrow \quad (G \Leftrightarrow G') .$$

□

Note that $CI \wedge G$ is always a suitable G' above, as is $CI \Rightarrow G$. The guards $p > 0$ and $p < 0$ could become $p > 0 \wedge p = q + r$ and $p < 0 \wedge p = q + r$.

17.5 Removing auxiliary variables: diminution

Each of the following laws deals with an aspect of removing variables. In each one we assume that the auxiliary variables are a and that their type is A, and we continue with the example of the previous section. There is no coupling invariant for diminutions.

17.5.1 Declarations

The declarations of auxiliary variables are simply deleted. In our example, the declaration **var** $p : \mathbb{N}$ is removed.

17.5.2 Initialisation

Existential quantification removes auxiliary variables from the initialisation.

Law 17.10 diminish initialisation The initialisation I *becomes*

$$(\exists\, a : A \bullet I) .$$

□

The example initialisation, augmented in Section 17.4.2, *becomes* $q + r = 1$ when p is removed.

17.5.3 Specifications

The following laws remove variables from a specification. In many practical cases, however, these laws are not needed: often the variables can be removed by ordinary refinement. (See our more substantial example, in Section 17.6.2.)

The first law is used when the variable a, to be removed, is in the frame:

Law 17.11 diminish specification The specification w, a: $[pre\ ,\ post]$ *becomes*

$$w\colon [(\exists\, a : A \bullet pre)\ ,\ (\forall\, a_0 : A \bullet pre_0 \Rightarrow (\exists\, a : A \bullet post))]\ ,$$

where pre_0 is $pre[w, a \backslash w_0, a_0]$. The frame beforehand *must* include a.
□

Law 17.11 may appear surprisingly complex (a good reason perhaps to use ordinary refinement where possible to eliminate the abstract variables, and to reserve *diminish specification* 17.11 as a last resort). In fact it performs three operations: first, since a is auxiliary, we can have no interest in its final value — we care only that there *is* one. That explains the quantification $\exists\, a$ in the postcondition.

Second, we strengthen the postcondition so that it no longer depends on the initial value of a; it must apply for *all* such values. That explains the quantification $\forall\, a_0$ in the postcondition. The antecedent pre_0 is optional (and *strengthen postcondition* 5.1 allows it in any case); it makes the postcondition weaker, and less likely to be infeasible.

Finally, we cannot refer in the precondition to the actual value of a, although we can be sure it has *some* value. That explains the quantification $\exists\, a$ in the precondition, which weakens it.

The example from Section 17.4.3 yields, after several applications of Predicate law A.56,

$$q, r\colon [q + r > 0\ ,\ q_0 + r_0 > 0 \Rightarrow q + r < q_0 + r_0]\ .$$

And by strengthening the postcondition that refines to

$$q, r\colon [q + r > 0\ ,\ q + r < q_0 + r_0]\ .$$

If a specification does not contain a in the frame, we can still use Law 17.11 provided we use *expand frame* 8.3 first. Or we can use this law, derived from those:

Law 17.12 diminish specification The specification w: $[pre\ ,\ post]$ *becomes*

$$w\colon [(\exists\, a : A \bullet pre)\ ,\ (\forall\, a : A \bullet pre_0 \Rightarrow post)]\ ,$$

where pre_0 is $pre[w \backslash w_0]$. The frame beforehand need not include a, and *post* must not contain a_0.

Proof:

$$\begin{array}{ll}
 & w\colon [pre\ ,\ post] \\
\sqsubseteq & \text{“\textit{expand frame} 8.3”} \\
 & w, a\colon [pre\ ,\ post \wedge a = a_0] \\
\textit{becomes} & \text{“\textit{diminish specification} 17.11”} \\
 & w\colon [(\exists\, a : A \bullet pre)\ , \\
 & \qquad \left(\forall\, a_0 : A \bullet \begin{array}{ll} & pre[w, a \backslash w_0, a_0] \\ \Rightarrow & (\exists\, a : A \bullet post \wedge a = a_0) \end{array}\right)]
\end{array}$$

$$
\begin{array}{ll}
\sqsubseteq & \text{“Predicate law A.56”} \\
& w{:}\ \left[(\exists\, a : A \bullet pre)\ ,\ \left(\forall\, a_0 : A \bullet \begin{array}{ll} & pre[w, a\backslash w_0, a_0] \\ \Rightarrow & a_0 \in A \wedge post[a\backslash a_0] \end{array}\right)\right] \\
\sqsubseteq & \text{“remove } a_0 \in A \text{, rename bound variable”} \\
& w{:}\ [(\exists\, a : A \bullet pre)\ ,\ (\forall\, a : A \bullet pre[w\backslash w_0] \Rightarrow post)]\ .
\end{array}
$$

□

For example, $n{:}\ [p > 0\ ,\ 0 \le n < p]$ is taken by *augment specification* 17.6 to

$$n, q, r{:}\ [p > 0\ ,\ p = q + r\ ,\ 0 \le n < p]\ ,$$

and then by *diminish specification* 17.12, *strengthen postcondition* 5.1, and *contract frame* 5.4 to

$$n{:}\ [q + r > 0\ ,\ 0 \le n < q + r]\ .$$

17.5.4 Assignments

The auxiliary part of the assignment is removed.

Law 17.13 diminish assignment If E contains no variables a, then the assignment $w, a := E, F$ can be replaced by the assignment $w := E$.
□

The example yields $q := q + 1$.

17.5.5 Guards

Guards must be rewritten so that they contain no auxiliary variables. Our earlier law *alternation guards* 4.3 can be used for that, since it is applicable to the refinement of alternations generally. In the example, we get $q+r > 0$ and $q+r < 0$.

17.6 An example of data refinement

As our first serious[1] example, consider the module of Figure 17.2 for calculating the mean of a sample of real numbers. We write $\sum b$ and $\#b$ for the sum and size respectively of bag b.

The module is operated by: first *clearing*; then *entering* the sample values, one at a time; then finally taking the *mean* of all those values.

[1]It is serious: the refinement we calculate here is exactly the one used in pocket calculators.

module *Calculator*
  var b : **bag** $\mathbb{R}$;

  procedure *Clear* $\hat{=}$ $b := \llbracket\,\rrbracket$;

  procedure *Enter* (**value** $r : \mathbb{R}$)
  $\hat{=}$ $b := b + \llbracket r \rrbracket$;

  procedure *Mean* (**result** $m : \mathbb{R}$)
  $\hat{=}$ $\{b \neq \llbracket\,\rrbracket\}\ m := \sum b / \# b$
end

Figure 17.2 The mean module

We transform the module, replacing the abstract bag b by a more concrete representation s, n, a pair of numbers. Throughout, we refer to b as the abstract variable, and to s, n as the concrete variables. First s and n are added, then b is removed.

17.6.1 Adding concrete variables

We shall represent the bag by its sum s and size n:

abstract variable:	b : **bag** $\mathbb{R}$
concrete variables:	$s : \mathbb{R}$; $n : \mathbb{N}$
coupling invariant:	$s = \sum b \wedge n = \# b$.

The first step is to add the declarations of new variables s, n and apply the augmentation techniques of Section 17.4 to the initialisation and the three procedures.

- For the initialisation, we have from *augment initialisation* 17.5

$$s = \sum b \wedge n = \# b \ .$$

- For *Clear*, we have from *augment assignment* 17.8

$$b, s, n := \llbracket\,\rrbracket, 0, 0 \ .$$

- For *Enter*, we have from *augment assignment* 17.8

$$b, s, n := b + \llbracket r \rrbracket, s + r, n + 1 \ .$$

- For *Mean* we have from *augment specification* 17.6 (after rewriting)

$$m, s, n\colon [b \neq \llbracket\rrbracket \ , \ s = \sum b \wedge n = \# b \ , \ m = \sum b/\# b] \ ,$$

and we can carry on, making these refinements immediately:

$$\begin{array}{ll} \sqsubseteq & m, s, n\colon [n \neq 0 \ , \ s = \sum b \wedge n = \# b \ , \ m = s/n] \\ \sqsubseteq & m\colon [n \neq 0 \ , \ s = \sum b \wedge n = \# b \ , \ m = s/n] \\ \sqsubseteq & \text{“\textit{remove invariant} 7.1”} \\ & m\colon [n \neq 0 \ , \ m = s/n] \ . \end{array}$$

The result is shown in Figure 17.3.

Remember that augmentation (or diminution) is not in itself a refinement: the assignment $\{b \neq \llbracket\rrbracket\}\ m := \sum b/\# b$ is *not* refined by $\{n \neq 0\}\ m := s/n$. The relation between them is augmentation (or diminution), relative to the abstract and concrete variables and the coupling invariant. That is why we write *becomes* rather than $\sqsubseteq$.[2]

17.6.2 Removing abstract variables

The abstract variable is b, and its removal from Figure 17.3 is straightforward for the assignment commands, because it is auxiliary (*diminish assignment* 17.13). Its removal from *Mean* is unnecessary — it has been removed already! That leaves only the initialisation. We use *diminish initialisation* 17.10, giving

$$n = 0 \Rightarrow s = 0 \ .$$

Now the abstract b has been removed altogether; the result is given in Figure 17.4. But the appearance of an explicit initialisation may be surprising, and it is in circumstances like this that being careful pays off. Suppose at some later stage an alternative abstract definition of *Clear* were given, such as this one:

$$\begin{array}{ll} \multicolumn{2}{l}{\textbf{procedure } \mathit{Clear}} \\ \widehat{=} & \textbf{if } b \neq \llbracket\rrbracket \rightarrow b := \llbracket\rrbracket \\ & [\!]\ \ b = \llbracket\rrbracket \rightarrow \textbf{skip} \\ & \textbf{fi} \ . \end{array}$$

It is equal to the original, in Figure 17.2, but it could be more efficient if the operation $b := \llbracket\rrbracket$ were very expensive, to be avoided if at all possible. With our augmentation and diminution it *becomes*

[2] Compare change of variable in an integral: faced with $\int_a^b dx/\sqrt{1-x^2}$ we might consider the substitution $x = \sin\theta$ (which is the analogue of the coupling invariant). But it would be wrong to claim that $dx/\sqrt{1-x^2}$ and $(\cos\theta\, d\theta)/\cos\theta$ were *equal*, even though the two definite integrals as a whole are equal.

module *Calculator*
  var $b : \mathbf{bag}\,\mathbb{R}$;
    $s : \mathbb{R};\;\; n : \mathbb{N}$;

  procedure *Clear* $\;\widehat{=}\;\; b, s, n := \llbracket\,\rrbracket, 0, 0$;

  procedure *Enter* (**value** $r : \mathbb{R}$)
  $\widehat{=}\;\; b, s, n := b + \llbracket r \rrbracket, s + r, n + 1$;

  procedure *Mean* (**result** $m : \mathbb{R}$)
  $\widehat{=}\;\; m\colon [n \neq 0\;,\; m = s/n]\;$;

  initially $s = \sum b \wedge n = \#b$
end

Figure 17.3 After addition of concrete variables

procedure *Clear*
$\widehat{=}$ **if** $n \neq 0 \rightarrow s, n := 0, 0$
  $[\!]\;\; n = 0 \rightarrow$ **skip**
  fi ,

and a subtle bug has crept in — suppose n and s were initially 0 and 1, for example (as they might be with no *explicit* initialisation)!

So our explicit concrete initialisation is necessary, after all, even though there was no abstract initialisation (other than true); and it is our good fortune that a rigorous approach brings that naturally to our attention. Note however that by *refine initialisation* 16.1 we could replace it by the simpler $s = 0$.

17.7 Abstraction functions

The laws of Section 17.4 dealt with a very general case of data refinement, in which the coupling invariant linking the abstract and concrete states could be anything whatever. In particular, several abstract variables could be collapsed onto a single concrete representation, as shown in the example of Section 17.6:

$$\begin{array}{ll} \text{both} & b = \llbracket 1, 2, 3 \rrbracket \\ \text{and} & b = \llbracket 2, 2, 2 \rrbracket \end{array} \;\text{ are represented by } s = 6 \wedge n = 3.$$

That is actually a fairly rare occurrence in everyday program development however: it is much more common for the coupling invariant to be functional from

> **module** *Calculator*
> **var** $s : \mathbb{R}$; $n : \mathbb{N}$;
>
> **procedure** *Clear* $\widehat{=}$ $s, n := 0, 0$;
>
> **procedure** *Enter* (**value** $r : \mathbb{R}$)
> $\widehat{=}$ $s, n := s + r, n + 1$;
>
> **procedure** *Mean* (**result** $m : \mathbb{R}$)
> $\widehat{=}$ m: $[n \neq 0\ ,\ m = s/n]$;
>
> **initially** $n = 0 \Rightarrow s = 0$
> **end**

Figure 17.4 The mean module, after transformation

concrete to abstract. (The above is not, but our earlier example $p = q + r$ is.) An example is the representation of sets by sequences, in which many distinct sequences may represent a given set: the elements may appear in different orders, may be duplicated, or even both. But to each sequence there corresponds at most one set; that is the functional nature of the abstraction, and what distinguishes it from the calculator example at the beginning of Section 17.6.

The general form for such coupling invariants, called *functional abstractions* is

$$a = \mathsf{af}\ c \wedge \mathsf{dti}\ c\ , \tag{17.1}$$

where af is a function, called the *abstraction function* and dti is a predicate, in which a does not appear, called the *data-type invariant.* In the case of sets and sequences, for example, the abstraction function is set, the function that makes a set from a sequence. Various data-type invariants may be included as well, for example that the sequences are kept in order (in which case we would write $a = \mathsf{set}\ c \wedge \mathsf{up}\ c$), or that the sequences contain no duplicated elements.

The reason for our interest in the special cases of data refinement is that when the augmentation and diminution laws are specialised to coupling invariants of the form (17.1) they become very much simpler, and the augmentation and diminution may be done together in one step.

17.7.1 Data-refining initialisations

Suppose that here (and in the following subsections) the coupling invariant is in the form (17.1), whence we may speak of abstraction function af and a data-type invariant dti. Given abstract initialisation I we would with *augment initialisation* 17.5 calculate $I \wedge a = \mathsf{af}\ c \wedge \mathsf{dti}\ c$; then *diminish initialisation* 17.10 would produce

$$(\exists\, a \cdot I \wedge a = \mathsf{af}\ c \wedge \mathsf{dti}\ c)\ .$$

But we can simplify that as follows:

$\equiv$ "Predicate law A.80"
$(\exists\, a \cdot I \wedge a = \mathsf{af}\ c) \wedge \mathsf{dti}\ c$
$\equiv$ "Predicate law A.56"
$I[a \backslash\ \mathsf{af}\ c] \wedge \mathsf{dti}\ c\ .$

Thus one merely replaces all occurrences of abstract variables a by their concrete counterparts $\mathsf{af}\ c$, conjoining the data-type invariant $\mathsf{dti}\ c$ to the result. That gives

Law 17.14 data-refine initialisation Under abstraction function af and data-type invariant dti, the initialisation I *becomes*

$$I[a \backslash\ \mathsf{af}\ c] \wedge \mathsf{dti}\ c\ .$$

□

In our original example (Section 17.4.2) that would take us in just one step from abstract initialisation $p = 1$ to concrete initialisation $q + r = 1$. (The data-type invariant is just true.)

As an example, let us represent a set $as : \mathbf{set}\, A$ by a sequence $aq : \mathbf{seq}\, A$ kept in strictly ascending order (thus excluding duplicates, and making the assumption that A is an ordered type). By analogy with up, we define

$$\mathsf{sup}\ aq \ \widehat{=}\ (\forall\, i, j : 0 {\rightarrow} \#aq \cdot i < j \Rightarrow aq[i] < aq[j])\ ,$$

and so take as our coupling invariant

$$as = \mathsf{set}\ aq \wedge \mathsf{sup}\ aq\ . \qquad (17.2)$$

If we now suppose that our abstract initialisation was $as = \{\}$, we calculate the formula $\mathsf{set}\ aq = \{\} \wedge \mathsf{sup}\ aq$ for the concrete initialisation, and continue

$\equiv$ $aq = \langle\rangle \wedge \mathsf{sup}\ aq$
$\equiv$ $aq = \langle\rangle\ .$

Thus to implement an abstract initialisation to the empty set, we provide a concrete initialisation to the empty sequence, whose strictly ascending order is trivial.

17.7.2 Data-refining specifications

Here as above we are going to carry out augmentation and diminution in succession; since *diminish specification* 17.11 will require an abstract a in the frame, we shall start with one there in readiness. Thus we begin with $w, a\colon [pre\ ,\ post]$ and apply *augment specification* 17.6 to get

$$w, a, c\colon [pre\ ,\ a = \mathsf{af}\ c \wedge \mathsf{dti}\ c\ ,\ post]\ .$$

Law *diminish specification* 17.12 then produces the (rather complicated-looking)

$$\begin{array}{l} w, c\colon [(\exists\, a \bullet pre \wedge a = \mathsf{af}\ c \wedge \mathsf{dti}\ c)\ , \\ \qquad\qquad \left(\forall\, a_0 \bullet \begin{array}{ll} & pre_0 \wedge a_0 = \mathsf{af}\ c_0 \wedge \mathsf{dti}\ c_0 \\ \Rightarrow & (\exists\, a \bullet post \wedge a = \mathsf{af}\ c \wedge \mathsf{dti}\ c) \end{array}\right)]\ . \end{array}$$

As before, however, the one-point laws A.56 apply, and we can simplify as follows:

$$\begin{array}{ll} \sqsubseteq & \text{“Predicate laws A.78, A.80, A.56”} \\ & w, c\colon \left[pre[a \backslash \mathsf{af}\ c] \wedge \mathsf{dti}\ c\ ,\ \begin{array}{ll} & pre_0[a_0 \backslash \mathsf{af}\ c_0] \wedge \mathsf{dti}\ c_0 \\ \Rightarrow & post[a_0, a \backslash \mathsf{af}\ c_0, \mathsf{af}\ c] \wedge \mathsf{dti}\ c \end{array}\right]\ . \\ \sqsubseteq & \text{“\textit{strengthen postcondition} 5.1”} \\ & w, c\colon [pre[a \backslash \mathsf{af}\ c]\ ,\ \mathsf{dti}\ c\ ,\ post[a_0, a \backslash \mathsf{af}\ c_0, \mathsf{af}\ c]]\ . \end{array}$$

The pattern is again substitution (abstraction function) and conjunction (data-type invariant). The law is thus

Law 17.15 data-refine specification Under abstraction function af and data-type invariant dti, the specification $w, a\colon [pre\ ,\ post]$ *becomes*

$$w, c\colon [pre[a \backslash \mathsf{af}\ c]\ ,\ \mathsf{dti}\ c\ ,\ post[a_0, a \backslash \mathsf{af}\ c_0, \mathsf{af}\ c]]\ .$$

□

Earlier that would have taken us from

$$p\colon [p > 0\ ,\ p < p_0]$$

directly to $q, r\colon [q + r > 0\ ,\ q + r < q_0 + r_0]$ in just a single step.

Continuing with our more recent example above, we consider now the specification

$$as\colon [a \in as\ ,\ a \notin as \wedge \{a\} \cup as = as_0]$$

that removes a given element a from our abstract set as (and which may abort if the element is not there). With the coupling invariant (17.2) we proceed

$$\begin{array}{l} \textit{becomes}\ \text{“\textit{data-refine specification} 17.15”} \\ \qquad aq\colon [a \in \mathsf{set}\ aq\ ,\ \mathsf{sup}\ aq\ ,\ a \notin \mathsf{set}\ aq \wedge \{a\} \cup \mathsf{set}\ aq = \mathsf{set}\ aq_0]\ , \end{array}$$

and are confronted immediately with one of the 'facts of life' in such derivations: after a calculated data refinement there still may be considerable work to do at the concrete level. In this case, we carry on as follows:

$$
\begin{array}{lll}
& \sqsubseteq & \mathbf{var}\ n : \mathbb{N} \bullet \\
& & n\colon [a \in aq\ ,\ aq[n] = a]\,; \qquad\qquad \text{(i)} \\
& & aq, n\colon [aq[n] = a \wedge \mathsf{sup}\ aq\ , \\
& & \quad \mathsf{sup}\ aq \wedge a \notin \mathsf{set}\ aq \wedge \{a\} \cup \mathsf{set}\ aq = \mathsf{set}\ aq_0] \qquad\qquad \text{(ii)} \\
\text{(i)} & \sqsubseteq & \text{“invariant } a \in aq{\downarrow}n\text{”} \\
& & n := 0; \\
& & \mathbf{do}\ aq[n] \neq a \rightarrow n := n + 1\ \mathbf{od} \\
\text{(ii)} & \sqsubseteq & n, aq := ?, aq{\uparrow}n +\!\!+\ aq{\downarrow}(n+1) \\
& \sqsubseteq & \mathbf{con}\ AQ : \mathbf{seq}\, A \bullet \\
& & n, aq\colon [AQ = aq{\uparrow}n +\!\!+\ aq{\downarrow}(n+1)\ ,\ aq = AQ] \\
& \sqsubseteq & I \ \widehat{=}\ AQ = aq{\uparrow}n +\!\!+\ aq{\downarrow}(n+1) \bullet \\
& & n, aq\colon [I\ ,\ I \wedge n \geq \#aq - 1]\,; \qquad\qquad \triangleleft \\
& & aq := \mathsf{fr}\ aq \\
& \sqsubseteq & \text{“invariant } I\text{”} \\
& & \mathbf{do}\ \ n < \#aq - 1 \rightarrow \\
& & \quad aq[n], n := aq[n+1], n+1 \\
& & \mathbf{od}\ .
\end{array}
$$

17.7.3 Data-refinement of assignments

In this case by *augment assignment* 17.8 we can replace $w, a := E, F$ by the assignment $w, a, c := E, F, G$ provided

$$a = \mathsf{af}\ c \wedge \mathsf{dti}\ c \Rrightarrow F = \mathsf{af}\ G \wedge \mathsf{dti}\ G\ . \tag{17.3}$$

(Note that w, a here are together playing the role of w in *augment assignment* 17.8.)

If we assume additionally that E, G contain no a, then *diminish assignment* 17.13 takes us immediately from $w, a, c := E, F, G$ to $w, c := E, G$. The law is thus, after simplification of the proviso,

Law 17.16 data-refine assignment Under abstraction function af and data-type invariant dti, the assignment $w, a := E, F$ can be replaced by the assignment $w, c := E[a\backslash\ \mathsf{af}\ c], G$ provided that G contains no a, and that

$$
\begin{array}{ll}
& \mathsf{dti}\ c \Rrightarrow F[a\backslash\ \mathsf{af}\ c] = \mathsf{af}\ G \\
\text{and} & \mathsf{dti}\ c \Rrightarrow \mathsf{dti}\ G\ .
\end{array}
$$

□

That our earlier $p := p + 1$ *becomes* $q := q + 1$ follows immediately.

For our later example we take the abstract $a := \{2, 0\}$ (with A as $\mathbb{N}$), and we propose $aq := \langle 0, 2 \rangle$ for the concrete assignment. The proviso is (the two taken together)

$$\mathsf{sup}\, aq \quad \Rrightarrow \quad \{2, 0\} = \mathsf{set}\langle 0, 2\rangle \wedge \mathsf{sup}\langle 0, 2\rangle \ ,$$

and is easily verified (even without its antecedent). But note how the proviso would not hold had we chosen $aq := \langle 2, 0 \rangle$, where the sequence is not in order.

17.7.4 Data-refinement of guards

Law *augment guard* 17.9 allows us to replace G by $G[a \backslash \mathsf{af}\ c] \wedge \mathsf{dti}\ c$, where as in *augment guard* 17.9 there is some flexibility: the conjunct $\mathsf{dti}\ c$ is optional. Subsequent adjustments may be made by *alternation guards* 4.3 as before. We have

Law 17.17 data-refine guard Under abstraction function af and data-type invariant dti, the guard G may be replaced by $G[a \backslash \mathsf{af}\ c] \wedge \mathsf{dti}\ c$, or if desired simply by $G[a \backslash \mathsf{af}\ c]$ on its own.

□

Consider for example the alternation

$$\begin{array}{l} \textbf{if}\ a \in as \rightarrow prog1 \\ [\!]\ \ a \notin as \rightarrow prog2 \\ \textbf{fi}\ . \end{array}$$

By *data-refine guard* 17.17 (and other laws) that *becomes*

$$\begin{array}{l} \textbf{if}\ a \in \mathsf{set}\ aq \rightarrow prog1' \\ [\!]\ \ a \notin \mathsf{set}\ aq \rightarrow prog2' \\ \textbf{fi}\ , \end{array}$$

where $prog1'$ and $prog2'$ data-refine $prog1$ and $prog2$ respectively. We could continue

$$\begin{array}{ll} \sqsubseteq & \textbf{var}\ n : \mathbb{N} \bullet \\ & n\colon [\mathsf{sup}\, aq\ ,\ a \in \mathsf{set}\, aq \Leftrightarrow a = aq[n]]\ ; \qquad\qquad \lhd \\ & \textbf{if}\ a = aq[n] \rightarrow prog1' \\ & [\!]\ \ a \neq aq[n] \rightarrow prog2' \\ & \textbf{fi} \\ \sqsubseteq & n\colon [\mathsf{sup}\, aq\ ,\ aq{\uparrow}n < a \leq aq{\downarrow}n] \\ \sqsubseteq & ???\ . \end{array}$$

17.8 Exercises

Ex. 17.1 Give other possible new assignment commands for the example $p := p + 1$ in Section 17.4.4.

Ex. 17.2 Use *diminish specification* 17.12 to remove the variable a from the following:

$$x\colon [x = a \ , \ x = a + 1] \ .$$

Ex. 17.3 (See Exercise 17.2.) Remove a from that specification *without* using *diminish specification* 17.12.

Ex. 17.4 ♡ Use *diminish specification* 17.11 to remove a from

$$a, x\colon [x = a_0] \ .$$

Now remove it from $a, x\colon [x = a]$. Comment on the difference: is a auxiliary in both?

Ex. 17.5 ♡ *Log-time multiplication* The following program terminates in time proportional to $\mathsf{log}\, N$:

$$\begin{array}{l} l, m, n := 0, 1, N; \\ \textbf{do}\ n \neq 0 \rightarrow \\ \quad \textbf{if}\ \mathsf{even}\, n \rightarrow m, n := 2 \times m, n \div 2 \\ \quad [\!]\ \ \mathsf{odd}\, n \ \ \rightarrow l, n := l + m, n - 1 \\ \quad \textbf{fi} \\ \textbf{od}\ . \end{array}$$

Propose an iteration invariant that could be used to show that the program refines

$$l, m, n\colon [l = N] \ ,$$

given the declarations $l, m, n, N : \mathbb{N}$.

Augment the program by variables l' and m', coupled as follows:

$$\begin{array}{rcl} l' & = & M \times l \\ m' & = & M \times m \ . \end{array}$$

What is the resulting program, and what value is then found in l' on its termination?

Now go on to diminish the program by removing all variables not needed for the calculation of l'; then rename variables to remove primes. What is the resulting program?

Ex. 17.6 *Log-time exponentiation* Augment the program of Exercise 17.5 by variables l' and m', coupled as follows:

$$\begin{aligned} l' &= M^l \\ m' &= M^m \,. \end{aligned}$$

What is the resulting program, and what value is then found in l' on its termination?

Diminish the program by removing all variables not needed for the calculation of l'; then rename variables to remove primes. What is the resulting program?

Ex. 17.7 *Log-time transitive closure* Let A be given, and define

$$\mathsf{tc}\, n \;\widehat{=}\; (+i : \mathbb{N} \mid i < n \cdot A^i) \,.$$

(The function tc could be said to be forming the transitive closure of A, if A were an incidence matrix for a graph; but that point of view is not necessary for this exercise.)

Augment the program of Exercise 17.5 by variables l' and m', coupled as follows:

$$\begin{aligned} l' &= \mathsf{tc}\, l \\ m' &= \mathsf{tc}\, m \,. \end{aligned}$$

What is the resulting program, and what value is then found in l' on its termination?

You will need an identity that gives $\mathsf{tc}(a+b)$ in terms of $\mathsf{tc}\, a$, $\mathsf{tc}\, b$ and A^b; what is it? How does that identity help you to decide what the 'definition' of $\mathsf{tc}\, 0$ should be?

Your augmented program should not contain any occurrences of tc, but may contain expressions A^m.

Further augment the program — add another variable, suitably coupled — so that the exponentiation can be removed. Note that the coupling invariant may be assumed when simplifying expressions.

Now diminish the program so that, after suitable renaming, a program remains that calculates $\mathsf{tc}\, N$ in logarithmic time.

Ex. 17.8 ♡ Exercise 17.6 showed that A^N can be calculated in time logarithmic in N, and so the equality

$$\mathsf{tc}\, N \quad = \quad (A^N - 1)/(A - 1)$$

appears to extend that logarithmic efficiency to the calculation of $\mathsf{tc}\, N$ itself, where tc is as defined in Exercise 17.7.

Under what circumstances might the program of Exercise 17.7 still be a better way to proceed? (The case $A = 1$ is on its own not a sufficient answer!)

Ex. 17.9 ♡ Suppose *pre* and *post* and w contain no a or a_0. What effect does *diminish specification* 17.12 have on the following?

$$w\colon [pre\ ,\ post]$$

Ex. 17.10 ♡ Use *assumption* 1.6 to formulate laws for adding and removing variables from assumptions.

Ex. 17.11 ♡ Use *coercion* 17.1 to formulate laws for adding and removing variables from coercions.

Ex. 17.12 Apply *diminish specification* 17.12 directly to

$$m, s, n\colon \left[b \neq \llbracket\,\rrbracket\ ,\ s = \sum b \wedge n = \#b\ ,\ m = \sum b/\#b\right]\ ,$$

without first doing the refinements on p.172. Then simplify the result. Which is easier: this exercise, or p.172?

Ex. 17.13 ♡ Suppose the mean procedure were instead

$$\begin{array}{l}\textbf{procedure}\ \mathit{Mean}\ (\textbf{result}\ m : \mathbb{R})\\ \quad \widehat{=}\ \textbf{if}\ b \neq \llbracket\,\rrbracket \rightarrow m := \sum b/\#b\\ \qquad [\!]\ \ b = \llbracket\,\rrbracket \rightarrow \textbf{error}\\ \qquad \textbf{fi}\ ,\end{array}$$

where **error** is some definite error indication unaffected by data refinement. Use *augment guard* 17.9 and *alternation guards* 4.3 to calculate the concrete procedure.

Ex. 17.14 ♡ In Exercise 16.2, Module *Tag* of Figure 16.1 was refined so that *Acquire* acquired only even numbers. The result was infeasible, because the precondition $u \neq \mathbb{N}$ was not strong enough to ensure that $\mathbb{N} - u$ contained any even numbers still to be acquired. Use augmentation with *no* concrete variables but still a coupling invariant of $u \in \textbf{finset}\,\mathbb{N}$ to show that *Acquire* can be transformed to

$$t, u\colon [u \in \textbf{finset}\,\mathbb{N}\ ,\ t \notin u_0 \wedge u = u_0 \cup \{t\}]\ .$$

How does that help with Exercise 16.2?

Ex. 17.15 ♡ Show that in the *Tag* module of Figure 16.1, the body of *Return* can be replaced by **skip**. *Hint*: Remember that you cannot transform just part of a module. Use new variable v and coupling invariant $u \subseteq v \wedge v \in \textbf{finset}\,\mathbb{N}$ to transform all of it. The appearance of changing only *Return* is then gained by renaming v back to u.

Ex. 17.16 (See Exercise 17.15). Why is $v \in \textbf{finset}\,\mathbb{N}$ necessary in the coupling invariant?

Ex. 17.17 Show that $\mathit{Tag} \sqsubseteq \mathit{Tag}'$ (Figures 16.1 and 17.1).

Ex. 17.18 ♡ Explain the effect of a data refinement with *no* concrete or abstract variables, but still a coupling invariant. *Hint*: Recall Exercise 17.16.

Ex. 17.19 Why isn't false a good idea for a coupling invariant? *Hint*: See Exercise 16.1.

Ex. 17.20 ♡ The example of Section 17.7.2 on data-refining specifications concerned removing an element a from a set as. What does the concrete version (in terms of aq) do if a is *not* in aq? Is that reasonable?

Ex. 17.21 Use the functional abstraction laws to do the data refinement of Section 17.6 in reverse: that is, show that the module of Figure 17.2 is a refinement of that in Figure 17.4. Does that mean that the modules are equal? How does equality differ from refinement?

Hint: Convert the assignments to specifications first.

Ex. 17.22 ♡ In the example of Section 17.7.2 the function fr is left in the code, but considerable trouble was taken to remove $\uparrow$, $\downarrow$ and $+\!\!\!+$. Why is fr acceptable but the others not?

Ex. 17.23 Why is it acceptable to use linear search in the example of Section 17.7.2, instead of the more efficient binary search?

Ex. 17.24 ♡ Let the abstract type be (again) a set $as : \mathbf{set}\, A$ and take the concrete type to be a pair $aq : \mathbf{seq}_N\, A;\ n : \mathbb{N}$ with the functional abstraction being $as = \mathsf{set}(aq{\uparrow}n)$. (Thus the data-type invariant is true.) Calculate data refinements for the following:

1. $a\colon [as \neq \{\}\ ,\ a \in as]$;
2. $a, as\colon [as \neq \{\}\ ,\ a \notin as \wedge \{a\} \cup as = as_0]$;
3. $as := as \cup \{a\};\ \{\#as \leq N\}$.

Hint: For the third, consider rewriting it as a specification.

Ex. 17.25 ♡ Justify the last step of the derivation of Section 17.7.4. What might replace the '???' ?

Ex. 17.26 ♡ Use *expand frame* 8.3 to derive a law analogous to *data-refine specification* 17.15 in which the abstract variable does not appear in the frame. Do not assume that *post* contains no a_0.

Ex. 17.27 ♡ Suggest an example where the abstract command does not include a in the frame, but the concrete refinement of it does nevertheless. (See Exercise 17.26.)

Ex. 17.28 ♡ A more abstract database type than those we investigated in Section 15.3 would be $K \rightarrow\!\!\!\!\!+ D$, a partial function from keys K to data D. Give a specification, at that level, of a lookup operation which can be data-refined to the specification you gave as your answer to Exercise 15.6. Write down the coupling invariant and work through the data refinement.

Ex. 17.29 ♡ Section 15.8.2 presented an iterative tree-summing program whose code contained a sequence of trees. How might that be implemented in a more conventional language that had only fixed-size arrays (for sequences) and records-plus-pointers for trees? Would one have to change the specification?

Ex. 17.30 Is it a refinement to strengthen or to weaken coercions? (Recall Exercise 1.13.)

Ex. 17.31 Prove this equality:

Law 17.18 <u>merge coercions</u>

$$[post]\ [post'] \quad = \quad [post \wedge post'] \ .$$

□

Ex. 17.32 Show that *augment assignment* 17.8 indeed follows from *augment assignment* 17.7.

Ex. 17.33 Prove this law:

Law 17.19 <u>introduce assumption</u>

$$[post] \quad \sqsubseteq \quad [post]\ \{post\}.$$

□

Ex. 17.34 Prove this law:

Law 17.20 <u>remove coercion</u>

$$\{pre\}\ [pre] \quad \sqsubseteq \quad \{pre\}.$$

□

Ex. 17.35 ♡ Prove this refinement:

$$\begin{array}{ll}
 & x := 1 \\
\sqsubseteq & \mathbf{if}\ \mathsf{true} \rightarrow x := 1 \\
 & [\!]\ \ \mathsf{true} \rightarrow x := -1 \\
 & \mathbf{fi}; \\
 & [x \geq 0]\ .
\end{array} \qquad \text{(i)}$$

If coercions were code, how would (i) above be executed? *Hint*: Consider backtracking.