Chapter 16

Modules and encapsulation

We have seen already how procedures act as a structuring tool for large programs, and that they simplify the development process. Going further, groups of procedures can themselves be organised, usually into units corresponding to data abstractions. We call those *modules*, and they are the subject of this chapter and the next.

16.1 Module declarations

Suppose in a large program there were a need for uniquely allocated natural number tags. The program would declare a set u of used tag values:

declaration: $\quad$ **var** u : **set** $\mathbb{N}$.

It would contain commands for acquiring and returning tags (assume that t is in $\mathbb{N}$):

$$\begin{array}{rl} \text{acquire new tag:} & t, u\colon [u \neq \mathbb{N} \ , \ t \notin u_0 \wedge u = u_0 \cup \{t\}] \\ \text{return tag:} & u := u - \{t\} \ . \end{array}$$

And it would contain an initialisation of u, placed before any other use of it:

initialisation: $\quad u := \{\}$.

Finally — an important point — the variable u would not be used in any other way.

If the commands occur often, they could be made procedures. But there would still be a lack of organisation in the program: the three aspects of u (declaration, use, initialisation) would be widely separated, though they are all to do with a single abstraction. With a module they can be brought together.

Modules contain all three features: local variable declarations, procedure declarations, and initialisations. They *encapsulate* their data and all aspects of their

module *Tag*
  var u : **set** $\mathbb{N}$;

  procedure *Acquire* (**result** $t : \mathbb{N}$)
  $\hat{=}\ \ t, u\colon [u \neq \mathbb{N}\ ,\ t \notin u_0 \wedge u = u_0 \cup \{t\}]$;

  procedure *Return* (**value** $t : \mathbb{N}$)
  $\hat{=}\ \ u := u - \{t\}$;

  initially $u = \{\}$
end

Figure 16.1 Module declaration

use. Figure 16.1 gives an example; note that a module is a declaration, not a command.

A local block containing a module declaration is equivalent in meaning to one in which the components are distributed back to their normal positions. Its variable declarations are placed with the other variable declarations of the block; its procedure declarations are placed with the other procedure declarations; and its initialisation (made a command) is placed at the beginning of the block body. Figures 16.2 and 16.3 illustrate that distribution. Normally in program development, however, we would move in precisely the reverse direction, from Figure 16.3 to 16.2.

16.2 Exported and local procedures

In Figure 16.2 it is not possible for *prog* to refer to the variable b, because *prog* is outside the module but b is inside the module. The procedures $P1$ and $P2$ can be used in *prog*, however; that is precisely what they are for. They are *exported* from the module. In general, we indicate explicitly the procedures to be exported: an *export list* gives their names. It is written

  export $P1, P2$,

and is placed inside the module. If it is missing, then by default all procedures are exported.

Procedures not exported are *local*, and are available for use only within the module. As for local variables, they can be given fresh names if the module is removed. Figure 16.4 gives an example of the use of a local procedure; it is equivalent to Figure 16.1.

|[**var** a;

module M
var b;
procedure $P1$ $\cdots$;
procedure $P2$ $\cdots$;
initially *init*
end;

var c;

procedure P $\cdots$.

prog
]|

Figure 16.2 Local block with module

|[**var** a;
var b;
var c;

procedure P $\cdots$;
procedure $P1$ $\cdots$;
procedure $P2$ $\cdots$.

b: $[init]$;
prog
]|

Figure 16.3 Equivalent local block without module

Variables may be exported also, in which case they may be accessed but not changed ('read' but not 'written') by commands outside of the module. They may be changed by commands within the module.

module *Tag*
  export *Acquire*, *Return*;

  var u : **set** $\mathbb{N}$;

  procedure *Acquire* (**result** $t : \mathbb{N}$)
  $\widehat{=}$ *Choose* (t); $u := u \cup \{t\}$;

  procedure *Return* (**value** $t : \mathbb{N}$)
  $\widehat{=}$ $u := u - \{t\}$;

  procedure *Choose* (**result** $t : \mathbb{N}$)
  $\widehat{=}$ t: $[u \neq \mathbb{N}\ ,\ t \notin u]$;

  initially $u = \{\}$
end

Figure 16.4 Module with local procedure

16.3 Refinement of modules

We saw in Chapter 11 that refining a procedure body refines the program containing it, and we regard that as refining the procedure itself.

Similarly, we can refine a module, as a whole, by refining its exported procedures. The local procedures can be changed in any way we please (as long as that results in refinement of the exported procedures). The initialisation is refined by strengthening it:

Law 16.1 <u>refine initialisation</u> If $init' \Rrightarrow init$, then

$$\textbf{initially}\ init \quad \sqsubseteq \quad \textbf{initially}\ init'\ .$$

□

Figure 16.5 contains a refinement of the module in Figure 16.4. By refining *Choose*, we refine the exported *Acquire*, which uses it. The refined module acquires the *least* unused tag.

16.4 Imported procedures and variables

Modules developed for one program can often be reused in another, because their encapsulation makes them largely independent of the surrounding program.

module *Tag*
$\vdots$
procedure *Choose* (**result** $t : \mathbb{N}$)
$\widehat{=}\;\; t :=(\sqcap e : \mathbb{N} \mid e \in \mathbb{N} - u)$
$\vdots$
end

Figure 16.5 Refinement of Figure 16.4

For reuse, however, we must be explicit about the dependencies there are. References made by the module to its environment are *imported* using an *import list.* Both variables and procedures may be imported.

An imported variable is redeclared within the module, and that declaration must be implied[1] by the original declaration of the variable.

An imported procedure is redeclared within the module, and its original declaration must refine its redeclaration. The effect of an imported procedure on a module is given by the text associated with its (re)declaration *in the module*, not for example by the text of the actual (external) procedure — a necessary precaution if we are to allow modules to be refined in isolation, without direct reference to the context in which they will be placed.[2]

Both sorts of redeclarations — of variables and procedures — are to make reasoning about the module independent of the surrounding program, and the import list distinguishes redeclarations from declarations.

Imported procedures cannot refer directly to local variables of the module (because of variable capture); for that, they must use parameters. Figure 16.6 continues the example by importing *Choose*. The redeclaration of *Choose*, within the module, records the assumptions made about it. Refining the module means assuming *less*:

Law 16.2 <u>refine module</u> Let E be the list of exported procedures from module M, I its imported procedures, and *init* its initialisation. A module M' refines M if the following three conditions are satisfied:

1. Its exported variables are unchanged.

[1]Consider types as local invariants for that purpose, so that for example the declaration $a : \mathbb{N}$ 'implies' the declaration $a : \mathbb{Z}$.

[2]Naturally an implementor is likely to resolve calls to the imported procedure by using the actual external procedure. Given our rule about imported procedures being refined by the procedures they redeclare, the effect of that is a refinement of the whole program — but one on which the developer *cannot depend.* No more can he depend on the fact that a particular compiler might implement all assumptions as **skip**, or all nondeterministic choices as a deterministic choice of the first enabled alternative.

module *Tag*
  export *Acquire*, *Return*;
  import *Choose*;

  var u : **set** $\mathbb{N}$;

  procedure *Acquire* (**result** $t : \mathbb{N}$)
  $\widehat{=}$ *Choose* $(\mathbb{N} - u, t)$;
    $u := u \cup \{t\}$

  procedure *Return* (**value** $t : \mathbb{N}$)
  $\widehat{=}$ $u := u - \{t\}$;

  procedure *Choose* (**value** s : **set** $\mathbb{N}$; **result** $e : \mathbb{N}$)
  $\widehat{=}$ e: $[s \neq \{\}\ ,\ e \in s]$;

  initially $u = \{\}$
end

Figure 16.6 Module with imported procedure

2. Its exported procedures E' refine E.
3. Its initialisation $init'$ refines $init$.

In addition, the following changes may be made provided the three conditions above are not invalidated as a result:

1. Its imported variables' declarations are weakened.
2. Its imported procedures I' are refined by I.
3. An imported procedure I is replaced by a *local* (neither imported nor exported) procedure I' that refines I.

□

(In fact the third change cannot invalidate any of the earlier conditions, but is mentioned for completeness: in that way a module can be decoupled from parts of its context.)

Note that Law 16.2 says 'if', and not 'only if': we see in the next chapter that there are other much more general ways of achieving $M \sqsubseteq M'$.

module $M1$
  export $P1$;
  import $P2$;

  procedure $P1 \mathrel{\widehat{=}}$ **magic**;
  procedure $P2 \mathrel{\widehat{=}}$ **magic**
end

module $M2$
  export $P2$;
  import $P1$;

  procedure $P2 \mathrel{\widehat{=}}$ **magic**;
  procedure $P1 \mathrel{\widehat{=}}$ **magic**
end

Figure 16.7 Circular export/import

16.5 Definition and implementation modules

In fact we do not need a technical meaning for 'definition' and 'implementation' when applied to modules. They refer only to a discipline of module reuse.

When a module is first formulated, it is likely to be during the development of some program, and will be abstract. Using an import list, it can be extracted from the program, and left for later refinement. That is a definition module.

Later, we can refine the definition module. The refined version can be inserted back into the program, replacing the original. That is an implementation module.

If in the development of some other program the same (definition) module is reached, it can immediately be replaced by its refinement, the implementation module. If the implementation module is code, as often will be the case, then that replacement can be done by linking in machine code just before execution. Thus the discipline we refer to is simply this:

> If for modules D and I we have $D \sqsubseteq I$, then we can call D a *definition* module and I (one of) its *implementation* module(s).

Definition/implementation pairs can be saved for future program developments. The definition module is published, but the implementation module is supplied. Since a module can have many refinements, a definition module can have many corresponding implementation modules.

16.6 Circular export/import

Consider Figure 16.7, and its refinement by *refine module* 16.2 to Figure 16.8. Figure 16.7 is infeasible (it contains **magic**), but Figure 16.8 is code — remember that the imported procedures (for example, $P2$ into $M1$) are there only for reasoning (about $M1$). Imported procedures need not be code.

Recall however from Chapter 1 that infeasible programs can never be refined to code.

module $M1$	**module** $M2$
export $P1$;	**export** $P2$;
import $P2$;	**import** $P1$;
procedure $P1 \mathrel{\widehat{=}} P2$;	**procedure** $P2 \mathrel{\widehat{=}} P1$;
procedure $P2 \mathrel{\widehat{=}}$ **magic**	**procedure** $P1 \mathrel{\widehat{=}}$ **magic**
end	**end**

Figure 16.8 Apparent refinement of Figure 16.7

The contradiction is due to the circular export/import between $M1$ and $M2$; and a simple way of avoiding that is to forgo such circularities between modules. If more care is taken, that can be relaxed to banning only circularities between procedures in separate modules. The most general solution involves variants, as for recursion; but it is seldom needed in practice.

16.7 Initialisation in code

We allow only certain initialisations in code. One is

$$\textbf{initially}\ \mathsf{true},$$

which allows any initial values of the variables consistent with their types. An initialisation true can be omitted.

The other initialisation allowed in code is

$$\textbf{initially}\ w = E,$$

where the variables w are all local, and the code expressions E are consistent with their types. The initialisation true is a special case of that, where w is empty.

16.8 Exercises

Ex. 16.1 The law *refine initialisation* 16.1 allows any initialisation to be refined to false. Why isn't that a good idea?

Ex. 16.2 ♡ Is it possible to refine *Tag* so that it acquires only *even* numbers? Is the result feasible?

Ex. 16.3 Modify *Tag* so that it can abort if 'too many' numbers are acquired but not returned. Is that a refinement? Does the original version refine your modified version?

Ex. 16.4 ♡ Write a module for acquiring pseudo-random numbers. Does it refine *Tag*?

Ex. 16.5 ♡ Recall Figure 16.7. In Module $M1$, can the imported procedure $P2$ be refined to **abort**? Can that be done in Figure 16.8?

Ex. 16.6 ♡ Suppose a module contained the declarations

> **var** $n : \mathbb{N}$;
> **import** In;
> **export** Out, n;
>
> **procedure** In (**result** $m : \mathbb{N}$) $\widehat{=}$ m: $[m = 0 \vee m = 1]$;
> **procedure** Out $\widehat{=}$ $In\ (n)$,

and that the actual procedure supplied for In, by the context, was

> **procedure** In (**result** $m : \mathbb{N}$) $\widehat{=}$ $m := 0$.

What value would a call of Out assign to n under those circumstances?

Suppose now a programmer refined Out to the assignment $n := 1$. Does that refine the module, according to Law 16.2? Is the new behaviour — of the module and its actual procedure — a refinement of the original behaviour?

Can you explain?