Program Specialisation in the Refinement Calculus

Lindsay Groves
School of Mathematical and Computing Sciences
Victoria University of Wellington, New Zealand
lindsay@mcs.vuw.ac.nz

Abstract

This paper presents an approach to program specialisation for nondeterministic imperative programs written in an extension of Dijkstra's guarded commands language. Assertion statements are used to express information about inputs and program variables, and refinement laws are used to show how these assertions can be introduced, propagated through the program structure, and used to specialise various program constructs.

This approach allows more general constraints than other work on program specialisation, which usually considers only fixed values for some of the inputs, and is easier to understand because the transformations used are all expressed entirely in terms of the programming language.

1 Introduction

Program specialisation involves modifying a program, or program fragment, to take account of additional information about the context in which it is to be used. The aim is usually to obtain a simpler and/or more efficient implementation. This is of practical importance when generic program components are reused: a high degree of generality is required to make the components reusable, but most of the functionality of the generic component is not required in any given application. Demonstrating that one program is a specialisation of another may also provide insights that are interesting in their own right; e.g. showing that various efficient string searching algorithms can be viewed as specialisations of a less efficient general algorithm, or that algorithms to recognise various classes of languages are specialisations of a generic recognition algorithm.

The most widely studied form of program specialisation is partial evaluation (e.g. [6, 14]), where one or more input values are fixed and a simplified program derived. Thus, a call on a substring search routine might be replaced by code that searches for a specific string passed as a constant argument in the original call, or a formatting routine might be specialised to doing the formatting required for a specific case. More general forms of specialisation, utilising weaker constraints on the inputs, may also by useful but have not been studies as extensively. For example, a formatting routine may be invoked several times using only four of the twenty possible formatting options — creating a specialised version for each call may be too expensive in terms of the both size of the program created, but a simplified version supporting just the options that are used would lead to a reduction in size and speed of the resulting program.

Program specialisation (and partial evaluation in particular) is usually performed by a form of interpreter which examines constructs in the program to see if they can be simplified given the additional information available. This is easiest to do in the context of functional programs, where an interpreter is (relatively) easy to write and understand, and specialisation of functional programs appears to be the most widely studied case. There has also been considerable work on specialisation of logic programs and imperative programs (e.g. [3, 5]), but the results are generally not as elegant. In any case, the workings of a program specialiser tend to be hard to understand because the transformations that are performed are described in terms of an interpreter (or, equivalently, an operational semantics), rather than at the level of the source program.

The aim of this paper is to demonstrate that the refinement calculus [18, 1] provides a convenient setting in which to study the transformations involved in program specialisation. We describe some simple transformations, using assertion statements to carry additional information supplied by the user, or extracted from the program, to be used in simplifying the given program. This allows a wide range of input restrictions to be expressed (not just specific values), and allows the transformations to described entirely at the source program level.

We are primarily concerned with providing a logical basis for understanding the kinds of transformations required for program specialisation, rather than how to automate the process of program specialisation. Our main interest is in laws for simplifying program constructs when given additional information. In order to get the information to the places where we can use it to make simplifications, we need laws for propagating information about variables through various program constructs. And since some of the information used in simplification is drawn from the program itself, we also need laws for collecting information about program variables that may be useful for this purpose.

Section 2 defines the programming language we use, and explains the relationship between specialisation and program refinement. Sections 3, 4 and 5 present laws for introducing assertions based on program structure and removing assertions, for propagating assertions through programs, and for using assertions to simplify programs. Section 6 demonstrates the approach with some simple examples, and Section 7 concludes by discussing our results, and related and future work. We omit proofs of the laws presented; these can mostly be found in [12].

2 Programs, Assertions and Specialisation

Our programs are written in an extended form of Dijkstra's guarded commands language, with the following syntax:

$$\begin{aligned} \mathit{Prog} ::= {} & \mathbf{skip} \mid \mathbf{abort} \mid \{A\} \mid \mathit{Vars} := \mathit{Exps} \mid \\ & \mathit{Prog}_1 \,;\mathit{Prog}_2 \mid \mathbf{if} \ \big[\!\big]_{i \in \mathcal{I}} \ G_i \to \mathit{Prog}_i \ \mathbf{fi} \mid \\ & \mathbf{do} \ \big[\!\big]_{i \in \mathcal{I}} \ G_i \to \mathit{Prog}_i \ \mathbf{do} \mid \big[\!\big[\mathbf{var}\ \mathit{Var} \bullet \mathit{Prog}_1 \big]\!\big] \mid \\ & \big[\!\big[\mathbf{proc}\ p \ \widehat{=} \ \mathit{Prog}_1 \bullet \mathit{Prog}_2 \big]\!\big] \end{aligned}$$

where A is an assertion (a formula of predicate logic), *Var* is a variable, *Vars* and *Exps* are lists of variables (or array references) and expressions, respectively, of the same length, G_i are Boolean expressions (guards), and $\mathcal{I}$ is a finite index set (this is usually omitted). The scope of the index variable introduced by $\big[\!\big]_i$ (and also later by $\bigvee_i$ and $\bigwedge_i$) extends as far as possible to the right. For simplicity, we omit type specifications and assume that all variables and expressions are of the appropriate type, and we only consider parameterless, non-recursive procedures.

The semantics of this language is defined in terms of weakest preconditions (*wp*), and (apart from assertions) is well enough know not to need repeating here (e.g. see [7, 10]). We sometimes also refer to the strongest postcondition (*sp*), which we also assume to be well understood [7, 8].

As in [7], we assume that all programs are monotonic with respect to postconditions, conjunctive and nonmiraculous (strict).

"$E[x \setminus e]$", where E is an expression, predicate or program, is E with all free occurrences of x replaced by e. When x and e are lists of the same length, $E[x \setminus e]$ denotes simultaneous replacement of free occurrences of x's by the corresponding e's.

An assertion $\{A\}$ is a statement which does nothing (i.e. behaves like **skip**) if A holds, and behaves unpredictably (i.e. behaves like **abort**) otherwise. Formally, $wp(\{A\}, \phi) \ \widehat{=} \ A \wedge \phi$.

If the assertion A expresses some information about the context in which a program S is to be used, we can think of $\{A\} \,; S$ as (a specification for) a specialised version of S which can replace S in that context. We can then attempt to transform this program into a simpler one using the information provided by A.

In performing this transformation, there is no reason to preserve the behaviour of S in cases where A does not hold — if we are assuming that A holds, we don't care what the program would do if A did not hold. And since our language allows nondeterminism, we might also allow the simplified form to be more deterministic. This means that our transformations need to preserve correctness, not operational equivalence. In the language of the refinement calculus, we want the specialised program to be a *refinement* of $\{A\} \,; S$. The refinement relation $\sqsubseteq$ is defined as follows.

Definition 1 (*program refinement*) For any programs S and T, $S \sqsubseteq T$ iff $wp(S, \phi) \Rrightarrow wp(T, \phi)$, for every postcondition ϕ.

Two programs are refinement-equivalent, written $S = T$, if they refine each other, and thus correspond to the same predicate transformer.

Definition 2 (*program equivalence*) For any programs S and T, $S = T$ iff $wp(S, \phi) \equiv wp(T, \phi)$, for every postcondition ϕ.

Thus, in this setting, the task of program specialisation is to find a program T such that $\{A\} \,; S \sqsubseteq T$, or if we wish to preserve refinement-equivalence, $\{A\} \,; S = T$. The refinement calculus provides a collection of laws that can be used in performing such transformations, and we can augment these with additional laws for introducing and propagating assertions, and using them to perform the simplifications required for program specialisation.

3 Introducing and removing assertions

In addition to information supplied externally about the program inputs, we can collect information about program variables by adding assertions that must hold by virtue of the program structure.

Following any program, we can assert the strongest postcondition assuming termination.

Law 1 $S = S; \{sp(S\,,\,true)\}$

The most useful instance of this is when S is an assignment statement. The most general form requires an existential quantifier to refer to the original values of the variable(s) begin assigned; we get a simpler version if these variables do not occur in the right-hand side.

(i) $x := e = x := e; \{\exists x' \bullet x = e[x \setminus x']\}$

(ii) $x := e = x := e; \{x = e\}$, if x not free in e

We can also add assertions on the basis of guards in **if** and **do** statements. Any guard will be true at the beginning of the corresponding branch; following a **do**, all the guards must be false.

Law 2

(i) $\textbf{if}\ \|_i\ G_i \to S_i\ \textbf{fi} = \textbf{if}\ \|_i\ G_i \to \{G_i\}; S_i\ \textbf{fi}$

(ii) $\textbf{do}\ \|_i\ G_i \to S_i\ \textbf{od} = \textbf{do}\ \|_i\ G_i \to \{G_i\}; S_i\ \textbf{od}; \{\bigwedge_i \neg G_i\}$

We can also assert any known loop invariant at the beginning of a loop body and following the loop (see law 12).

We will eliminate assertions when we have finished with them, since we do not regard them as being code. We can always remove assertions using the following law.

Law 3

(i) $\{A\} \sqsubseteq \textbf{skip}$

(ii) $\{A\}; S \sqsubseteq S$

If we wish to preserve refinement-equivalence when eliminating assertions, we can use laws 1 and 2 in the opposite direction, or the following law.

Law 4 $\{A\}; S = S$, if $A \Rrightarrow wp(S\,,\,true)$

4 Propagating assertions

We now consider laws for propagating assertions through various constructs. Some of them also collect information, as discussed in Section 3. We get simple laws to push assertions past atomic statements, and (usually) pairs of laws for pushing assertions into and out of the various compound statements.

We don't need special rules for sequential composition, since if $\{A\}; S \sqsubseteq S; \{B\}$ and $\{B\}; T \sqsubseteq T; \{C\}$, then $\{A\}; S; T \sqsubseteq S; T; \{C\}$ follows by monotonicity of $\sqsubseteq$.

4.1 General laws for propagating assertions

The following is a general law for finding an assertion that will hold after executing a program S, given that a preceding assertion A ensures that S terminates.

Law 5 $\{A\}; S \sqsubseteq S; \{sp(S\,,\,A)\}$

The new assertion given by this law embodies information derived from S as well as from A, and is often both stronger and more complex that we need. It may thus be preferable to use a weaker post-assertion containing only information we expect to be useful later, and only carry forward information derived from the program text if it is specifically required.

The following law can also be used to propagate assertions. Although it is less direct than law 5, it is often easier to work with, as it uses weakest preconditions and can give equivalence results.

Law 6

(i) $\{A\}; S \sqsubseteq S; \{B\}$, if $A \Rrightarrow wp(S\,,\,B)$

(ii) $\{A\}; S = S; \{B\}$, if $A \equiv wp(S\,,\,B)$

Taking $A \equiv B$, part (ii) says that $\{A\}; S = S; \{A\}$, if $A \equiv wp(S\,,\,A)$, which will hold provided S cannot alter any variables in A and always terminates. In particular, an assertion can be propagated unchanged through a **skip**.

(iii) $\{A\}; \textbf{skip} = \textbf{skip}; \{A\}$

4.2 Assignment

We can propagate an assertion through an assignment, $x := e$, by replacing occurrences of x in A by an existentially quantified variable denoting the original value of x. If x occurs in A, we can also add a conjunct relating the new value of x to e, as given by the strongest postcondition for assignment (as in law 1); this gives the same result as law 5.

Law 7

(i) $\{A\}; x := e \sqsubseteq x := e; \{\exists x' \bullet A[x \setminus x']\}$

(ii) $\{A\}; x := e \sqsubseteq x := e; \{\exists x' \bullet A[x \setminus x'] \wedge x = e[x \setminus x']\}$

If x is not free in A or in e, we can remove the existential quantifier and simplify the assertion in (ii) to $\{A \wedge x = e\}$.

Law 6 gives $\{A\}; x := e = x := e; \{A\}$ when x is not free in A, since $wp(x := e\,,\,A) \equiv A$ in this case.

4.3 Conditionals

A preceding assertion can be moved into a conditional, and attached to each of the branches. This gives an equivalence, because the weakest precondition for a conditional implies the disjunction of its guards.

Law 8 $\{A\}$; **if** $[]_i\ G_i \to S_i$ **fi** $=$ **if** $[]_i\ G_i \to \{A\}$; S_i **fi**

Sometimes we have an assertion of the form $\bigwedge_i G_i \Rightarrow A_i$, allowing us to deduce different things about each branch on the basis of the guards. In this case, we can propagate each A_i to the corresponding branch.

Law 9 $\{\bigwedge_i G_i \Rightarrow A_i\}$; **if** $[]_i\ G_i \to S_i$ **fi** $=$ **if** $[]_i\ G_i \to \{A_i\}$; S_i **fi**

An assertion holding at the end of each branch can be moved out of an **if** statement.

Law 10 **if** $[]_i\ G_i \to S_i$; $\{A\}$ **fi** $=$ **if** $[]_i\ G_i \to S_i$ **fi** ; $\{A\}$

If we have different assertions at the end of each branch, we have to resort to a weaker law.

Law 11 **if** $[]_i\ G_i \to S_i$; $\{A_i\}$ **fi** $\sqsubseteq$ **if** $[]_i\ G_i \to S_i$ **fi** ; $\{\bigvee_i A_i\}$

4.4 Loops

An assertion holding prior to a loop will still hold at the beginning of the loop body, and when the loop exits, providing it is preserved by the loop body.

Law 12 $\{A\}$; **do** $G \to S$ **od** $\sqsubseteq$ **do** $G \to \{A\}$; S **od** ; $\{A\}$, if $\{A \wedge G\}$; $S \sqsubseteq S$; $\{A\}$

The requirement that A is preserved by the loop body means that A is an invariant of the loop. The law generalises to multi-branch loops in the obvious way.

4.5 Variable declarations

A preceding assertion can be moved into a variable declaration block, provided the variable declared does not occur in the assertion. If the new variable does occur in the assertion, we need to rename it.

Law 13

(i) $\{A\}$; $[\![$ **var** $x \bullet S]\!] = [\![$ **var** $x \bullet \{A\}$; $S]\!]$, if x not free in A

(ii) $\{A\}$; $[\![$ **var** $x \bullet S]\!] = [\![$ **var** $x' \bullet \{A\}$; $S[x \setminus x']]\!]$, where x' is a fresh variable

A trailing assertion can be moved out of a variable declaration block, provided the variable declared does not occur in the assertion. If the declared variable occurs in the assertion, it is quantified in the new assertion.

Law 14

(i) $[\![$ **var** $x \bullet S$; $\{A\}]\!] \sqsubseteq [\![$ **var** $x \bullet S]\!]$; $\{\exists x \bullet A\}$

(ii) $[\![$ **var** $x \bullet S$; $\{A\}]\!] = [\![$ **var** $x \bullet S]\!]$; $\{A\}$, if x not free in A

4.6 Procedure declarations

We can move assertions into and out of the scope of a procedure declaration block, just as we can for variable declaration blocks.

Law 15 $\{A\}$; $[\![$ **proc** $p \mathrel{\widehat{=}} S \bullet T]\!] = [\![$ **proc** $p \mathrel{\widehat{=}} S \bullet \{A\}$; $T]\!]$, if p not free in A

Law 16 $[\![$ **proc** $p \mathrel{\widehat{=}} S \bullet T$; $\{A\}]\!] = [\![$ **proc** $p \mathrel{\widehat{=}} S \bullet T]\!]$; $\{A\}$, if p not free in A

4.7 Procedure calls

We can propagate an assertion preceding a procedure call into the body of the procedure. Since this alters the meaning of the procedure body we must create a new procedure, unless every call is preceded by the same assertion.

Law 17

(i) $[\![$ **proc** $p \mathrel{\widehat{=}} S \bullet \mathcal{C}[X \setminus \{A\}$; $p]]\!] = [\![$ **proc** $p \mathrel{\widehat{=}} S \bullet [\![$ **proc** $p' \mathrel{\widehat{=}} \{A\}$; $S \bullet \mathcal{C}[X \setminus p']]\!]]\!]$, where $\mathcal{C}$ is a program scheme containing one occurrence of program variable X and p' is a new procedure name.

(ii) $[\![$ **proc** $p \mathrel{\widehat{=}} S \bullet \mathcal{C}[X \setminus \{A\}$; $p]]\!] = [\![$ **proc** $p \mathrel{\widehat{=}} \{A\}$; $S \bullet \mathcal{C}[X \setminus p]]\!]$, where $\mathcal{C}$ is a program scheme containing one or more occurrences of program variable X and no occurrences of p.

We can similarly move an assertion at the end of a procedure, placing it after every call on the procedure.

Law 18 $[\![$ **proc** $p \mathrel{\widehat{=}} S$; $\{A\} \bullet \mathcal{C}[X \setminus p]]\!] = [\![$ **proc** $p \mathrel{\widehat{=}} S \bullet \mathcal{C}[X \setminus p$; $\{A\}]]\!]$, where $\mathcal{C}$ is a program scheme containing one or more occurrences of program variable X and no occurrences of p.

Since laws 17 and 18 alter the meaning of the declared procedure, it is often simpler, and safer, to propagate an assertion through a procedure call by propagating it through the procedure body as a whole.

Propagating an assertion past a procedure call has the same effect as propagating it through the body of the procedure.

Law 19 If p is declared as **proc** $p \mathrel{\widehat{=}} S$, then:

(i) $\{A\};p \;=\; p;\{A'\}$, if $\{A\};S \;=\; S;\{A'\}$

(ii) $\{A\};p \;\sqsubseteq\; p;\{A'\}$, if $\{A\};S \;\sqsubseteq\; S;\{A'\}$

5 Simplifying programs

We now consider laws that can be used to simplify programs on the basis of information expressed as an assertion. As might be expected, the most significant structural simplifications can be made to conditionals and loops; simplifications to expressions and guards can also reduce the amount of computation required.

5.1 Abortion

A program can be replaced by **abort** if a preceding assertion reveals that it may not terminate.

Law 20 $\{A\};S \;=\;$ **abort**, if $A \Rrightarrow \neg\, wp(S\,,\, true)$

5.2 Assignment

An assertion may allow us to simplify an assignment, by substituting a simpler expression, or deleting it if we can show that the variable already has the required value.

Law 21

(i) $\{A\};x := e \;\sqsubseteq\; x := f$, if $A \Rrightarrow e = f$

(ii) $\{A\};x := e \;\sqsubseteq\;$ **skip**, if $A \Rrightarrow x = e$

5.3 Conditionals

A preceding assertion may allow an **if** statement to be simplified in various ways. For example, we may be able to determine that certain guards are always true or always false. We can also use an assertion to modify the guards.

We can reduce a conditional to a branch whose guard is implied by a preceding assertion. We retain the assertion as it may allow further simplification of the result.

Law 22 $\{A\}$ **; if** $[]_{i\in\mathcal{I}}\; G_i \to S_i$ **fi** $\;\sqsubseteq\; \{A\}$ **;** S_k, for any $k \in \mathcal{I}$ where $A \Rrightarrow G_k$.

We can similarly remove a branch whose guard is contradicted by a preceding assertion.

Law 23 $\{A\}$ **; if** $[]_{i\in\mathcal{I}}\; G_i \to S_i$ **fi** $=$ **if** $[]_{i\in\mathcal{I}-\{k\}}\; G_i \to S_i$ **fi**, for any $k \in \mathcal{I}$ where $A \Rrightarrow \neg\, G_k$.

We can weaken the guards while propagating the assertion into the conditional (cf. law 8).

Law 24 $\{A\}$ **; if** $[]_i\; G_i \to S_i$ **fi** $=$ **if** $[]_i\; (A \Rightarrow G_i) \to \{A\}$ **;** S_i **fi**

We can also use the assertion to strengthen the guards. This gives an equivalence, even omitting the assertion in the result, because the weakest precondition for a conditional implies the disjunction of its guards. We can carry the assertion into each branch if required, as in law 8.

Law 25 $\{A\}$ **; if** $[]_i\; G_i \to S_i$ **fi** $=$ **if** $[]_i\; A \wedge G_i \to S_i$ **fi**

In moving an assertion into an **if** statement, we can modify each branch in any of the ways embodied in laws 8, 24 and 25. For example, we can weaken one guard while strengthening the other:

Law 26 $\{A\}$ **; if** $G \to S \;[]\; H \to T$ **fi** $=$ **if** $(A \Rightarrow G) \to \{A\}$ **;** $S \;[]\; A \wedge H \to T$ **fi**

5.4 Loops

An assertion can be used to simplify the body of a loop, provided that the assertion is preserved by the modified loop body. In some cases we may need to refine the loop body in order to ensure that the assertion is preserved by the loop, and can thus be propagated through it.

Law 27

(i) $\{A\}$ **; do** $G \to S$ **od** $\sqsubseteq$ **do** $G \to S'$ **od ;** $\{A\}$, if $\{A \wedge G\};S \;\sqsubseteq\; S';\{A\}$.

(ii) $\{A\}$ **; do** $G \to S$ **od** $=$ **do** $G \to S'$ **od ;** $\{A\}$, if $\{A \wedge G\};S \;=\; S';\{A\}$.

This law can be strengthened in a number of ways. In particular, we know that $A \wedge G$ holds at the beginning of the loop body, A holds at the end of the loop body, and $\neg G$ holds following the loop (by law 2(ii)).

We can use an assertion to weaken loop guards in much the same way as for conditionals (cf. law 24), provided the assertion is preserved by the loop body.

Law 28

(i) $\{A\}$ **; do** $[]_i\; G_i \to S_i$ **od** $\sqsubseteq$ **do** $[]_i\; (A \Rightarrow G_i) \to \{A\};S_i$ **od**, if $\{G_i \wedge A\};S_i \;\sqsubseteq\; S_i;\{A\}$, for each i.

(ii) $\{A\}$ **; do** $[]_i\; G_i \to S_i$ **od** $=$ **do** $[]_i\; (A \Rightarrow G_i) \to \{A\}$ **;** S_i **od**, if $\{G_i \wedge A\}$ **;** $S_i \;=\; \{G_i \wedge A\}$ **;** S_i **;** $\{A\}$, for each i.

We can also use an assertion preserved by the loop to strengthen the guards in a loop (cf. law 25).

Law 29

(i) $\{A\}$ **;** **do** $[]_i\ G_i \to S_i$ **od** $\sqsubseteq$ **do** $[]_i\ A \wedge G_i \to S_i$ **od**, if $\{G_i \wedge A\}$ **;** $S_i \sqsubseteq S_i$ **;** $\{A\}$, for each i.

(ii) $\{A\}$ **;** **do** $[]_i\ G_i \to S_i$ **od** $=$ **do** $[]_i\ A \wedge G_i \to S_i$ **od**, if $\{G_i \wedge A\}$ **;** $S_i = S_i$ **;** $\{A\}$, for each i.

Simplifying the guards may then allow us to make further simplifications to the loop.

An assertion may allow us to simplify a following loop in a variety of ways, for example, by simplifying the body or replacing the whole loop with a simpler statement. The next few laws show some of the possibilities.

If the assertion implies that no guard is true, the loop is equivalent to **skip** and can be deleted.

Law 30 $\{A\}$ **;** **do** $[]_i\ G_i \to S_i$ **od** $= \{A\}$, if $A \Rrightarrow \neg \bigvee G_i$.

If the assertion implies the negation of one of the guards, and is preserved by the other branches, that branch can be deleted and the assertion conjoined to the remaining guards. For simplicity, we state this for the two branch case.

Law 31 $\{A\}$ **;** **do** $G \to S\ []\ H \to T$ **od** $\sqsubseteq$ **do** $A \wedge G \to S$ **od**, if $\{A \wedge G\}$ **;** $S \sqsubseteq S$ **;** $\{A\}$ and $A \Rrightarrow \neg H$.

If the assertion guarantees that the loop guard is true, and is preserved by the loop body, the loop will never terminate and is thus equivalent to **abort**.

Law 32 $\{A\}$ **;** **do** $G \to S$ **od** $=$ **abort**, if $A \Rrightarrow G$ and $\{A \wedge G\}$ **;** $S \sqsubseteq S$ **;** $\{A\}$.

We can often use an assertion to specialise a particular iteration of a loop. The most obvious way to do this is to unroll the loop until the required iteration has been exposed. This is especially useful in a one branch loop, where the assertion ensures that the guard holds initially, and we want to specialise the first iteration.

Law 33

(i) $\{A\}$ **;** **do** $G \to S$ **od** $\sqsubseteq S'$ **;** **do** $G \to S$ **od**, if $A \Rrightarrow G$ and $\{A\}$ **;** $S \sqsubseteq S'$.

(ii) $\{A\}$ **;** **do** $G \to S$ **od** $= S'$ **;** **do** $G \to S$ **od**, if $A \Rrightarrow G$ and $\{A\}$ **;** $S = S'$.

We can also specialise a particular iteration or group of iterations, by introducing an **if** statement in the loop body, splitting the loop into a sequence of two loops, or, if the number of iterations can be determined, unrolling the whole loop into a sequential composition.

5.5 Procedure calls

The additional information provided by an assertion A might suggest that a procedure call be replaced by some other code, or by a call on a different procedure (perhaps a new one introduced specifically for the purpose). If all calls on a procedure occur in the same context (i.e. preceded by the same assertion), we might choose to modify the procedure, so it has $\{A\}$ **;** S is its specification, and then modify the implementation accordingly.

6 Examples

We now present some simple examples to illustrate the kinds of simplifications that can be made using the laws presented above. In each case we use laws from Section 4 to move the assertion to the place where it can be used, then use laws from Section 5 to effect the simplifications, and finally remove the assertions.

Example 1 An assertion may allow repeated calculation to be avoided.

$\{m = (lo + hi)/2\}$ **;**
if $x \leq a[(lo + hi)/2] \to hi := (lo + hi)/2$
$[]\ x \geq a[(lo + hi)/2] \to lo := (lo + hi)/2$
fi

becomes

if $x \leq a[m] \to hi := m$
$[]\ x \geq a[m] \to lo := m$
fi

Example 2 An assertion may allow data structure accesses to be removed. In this case we need to introduce $1 \leq k \leq n$ as a loop invariant so that we can instantiate the universal quantifier.

$\{\forall i : 1 \mathinner{..} n \bullet b[i] = i\}$ **;**
$k := 1$ **;**
do $k \leq n \to$
  $a[k] := a[k] * b[k]$ **;** $k := k + 1$
od

becomes

$k := 1$ **;**
do $k \leq n \to$
  $a[k] := a[k] * k$ **;** $k := k + 1$
od

Example 3 An assertion may allow a case analysis to be removed.

$$\{x \leq y\}\,;$$
$$\mathbf{if}\, x \leq y \rightarrow min := x$$
$$[]\; x \geq y \rightarrow min := y$$
$$\mathbf{fi}$$

becomes

$$min := x$$

Example 4 An assertion may allow number of cases in a case analysis to be reduced. Again, we need to introduce $1 \leq k \leq n$ as a loop invariant so that we can instantiate the universal quantifier.

$$\{\forall i : 1 \,..\, n \bullet a[i] \in \{1, 2\}\}\,;$$
$$k := 1\,;$$
$$\mathbf{do}\, k \leq n \rightarrow$$
$$\quad \mathbf{if}\, a[k] = 1 \rightarrow x := x + 1$$
$$\quad []\; a[k] = 2 \rightarrow y := x + 1$$
$$\quad []\; a[k] = 3 \rightarrow x := x - 1$$
$$\quad []\; a[k] = 4 \rightarrow y := y - 1$$
$$\quad \mathbf{fi}\,;$$
$$\quad k := k + 1$$
$$\mathbf{od}$$

becomes

$$k := 1\,;$$
$$\mathbf{do}\, k \leq n \rightarrow$$
$$\quad \mathbf{if}\, a[k] = 1 \rightarrow x := x + 1$$
$$\quad []\; a[k] = 2 \rightarrow y := x + 1$$
$$\quad \mathbf{fi}\,;$$
$$\quad k := k + 1$$
$$\mathbf{od}$$

Example 5 An assertion might indicate a preference for one alternative in a non-deterministic choice, because some of the values required are already available.[1]

$$\{a = (top - bot) * (x_2 - x_1)\}\,;$$
$$\mathbf{if}\, true \rightarrow$$
$$\quad n, s, w, e := top, bot, x_1, x_2\,;$$
$$\quad a := (top - bot) * (x_2 - x_1)$$
$$[]\; true \rightarrow$$
$$\quad n, s, w, e := y_1, y_2, left, right\,;$$
$$\quad a := (y_2 - y_1) * (right - left)$$
$$\mathbf{fi}$$

becomes

$$n, s, w, e := top, bot, x_1, x_2$$

7 Discussion

We have shown that program specialisation can be understood as a special case of program refinement. Using assertion statements allows us to express a wide range of input constraints and describe the transformations required for program specialisation entirely within the programming language. By using the refinement calculus, we can handle nondeterministic programs and can employ more powerful transformations, since we can use additional information to resolve nondeterminism and do not need to preserve nonterminating behaviour. All the laws presented here, except those concerning procedures, are proved in [12]; laws for procedures with parameters can be obtained by expanding procedure calls using variable declarations and substitutions as described in [18].

In some of our examples, to obtain the desired specialisations, we needed additional information about the program, viz. simple loop invariants concerning variable ranges. If loop invariants are recorded as part of the program, and similarly with procedure specifications, these can be used to assist program specialisation without having to be derived by analysing the program. More powerful simplifications still can be made if we have a formal derivation of the program, allowing us to recognise places where the additional information suggests a completely difference solution — for example, a generic component might look up a key in a table before updating the table, but in the context where it is used we might already know the position of the key in the table and so be able to eliminate the look-up. This is discussed in [12], which also considers a more general language including recursion and non-conjunctive and infeasible programs (in particular, logical constants, angelic choice and magic), also examines the use of coercion statements to impose additional constraints on the output produced, which allows nondeterministic programs to be specialised in a way that reduces nondeterminism.

[1] The program sets n, s, w and e to be the North, South, West and East coordinates of a minimal rectangle containing the points (x_1, y_1) and (x_2, y_2), and extending either from *top* to *bot* or from *left* to *right*, and computes its area.

The techniques described here can be extended to apply to other kinds of control structures that might be found in an imperative programming language, and could also be used in conjunction with operational or denotational semantics if stronger forms of equivalence are required. The major limitation is that we can only utilise information that can be expressed in the form of assertions, so additional notations would be needed, for example, to describe properties of complex data structures.

Program specialisation has been considered in the context of the refinement calculus before in [9, 11], where data refinement is used to specialise an abstract program in various different ways rather than using assertions as is done here. A similar approach is also presented in [19].

Assertions have been used extensively to express correctness conditions, and to prove context dependent refinements (e.g. [2], [18], [1] and [15]). Our use of assertions is rather different, since we are interested in propagating information through a program to places where it can be used to make simplifying transformations. This means that the laws we use are often of a different form.

We have not discussed issues to do with automation, which is a dominant concern in most work on program specialisation (especially partial evaluation). We are currently incorporating the laws discussed here in a program refinement tool, which will allows us to implement a program specialiser as a tactic (see [13, 4, 17, 16]).

Acknowledgements

Part of this work was completed while I was on study leave at Oxford University. I would like to thank the Oxford University Computing Laboratory, Kellogg College, and my host, Jim Woodcock, for their hospitality and support during this period. Thanks also to Ray Nickson for many fruitful discussions on this work, and helpful comments on this paper.

References

[1] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[2] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[3] A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, April 1990.

[4] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A program refinement tool. *Formal Aspects of Computing*, 10(2):97–124, 1998.

[5] A. Coen-Porisini, F. D. Paoli, C. Ghezzi, and D. Mandrioli. Software specialization via symbolic execution. *IEEE Transactions on Software Engineering*, 17(9):884–899, 1991.

[6] C. Consel and O. Davey. Partial evaluation: Principles and perspectives. In *Proc. 20th Annual ACM SIGPLAN-SIGART Symp. om principuples of programming Languages*, 1993.

[7] E. W. Dijkstra. *A Discipline of Programming*. Academic Press, 1976.

[8] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[9] A. Gravell. Specialising abstract programs. In J. M. Morris and R. C. Shaw, editors, *Fourth Refinement Workshop*, Workshops in Computing, pages 1–23. BCS FACS, Springer-Verlag, 1991.

[10] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[11] L. Groves. Deriving language recognition algorithms: A case study in combining program specialisation and data refinement. In D. Till, editor, *Sixth Refinement Workshop*, Workshops in Computing, pages 103–120. BCS FACS, Springer-Verlag, 1994.

[12] L. Groves. *Evolutionary Software Development in the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 2000.

[13] L. Groves, R. Nickson, and M. Utting. A tactic driven refinement tool. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *Fifth Refinement Workshop*, Workshops in Computing, pages 272–297. BCS FACS, Springer-Verlag, 1992.

[14] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, 1996.

[15] L. Laibinis and J. von Wright. Context handling in the refinement calculus framework. TUCS Technical Report 118, Turku Centre for Computer Science, August 1997.

[16] A. Martin, R. Nickson, and M. Utting. A tactic language for Ergo. In L. Groves and S. Reeves, editors, *Formal Methods Pacific '97*, Springer Series in Discrete Mathematics and Theoretical Computer Science, pages 186–207. Springer-Verlag, 1997.

[17] A. P. Martin, P. H. B. Gardiner, and J. C. P. Woodcock. A tactic calculus. *Formal Aspects of Computing*, 1996. To appear.

[18] C. Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.

[19] G. S. Novak. Software reuse by specialization of generic procedures through views. *IEEE Transactions on Software Engineering Software Engineering*, 23(7):401–417, 1997.