VICTORIA UNIVERSITY OF WELLINGTON

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Metavariables and Conditional Refinements in the Refinement Calculus

Raymond G. Nickson* and Lindsay J. Groves†

Technical Report CS-TR-94/9
January 1994

Abstract

We describe two techniques for the refinement calculus that facilitate goal-directed development. The techniques achieve this by allowing the deferring of decisions about the precise form of refinement steps, so high-level choices can be expressed as soon as those choices are appropriate. *Metavariables* are place-holders for components of partly developed programs that will be instantiated when they are suitably constrained by later refinements. The *conditional refinements* technique allows the development of alternative refinements of a specification, and the collection of those alternative refinements into a guarded command set.

We think that programmers developing programs using the refinement calculus make use of both of these techniques informally, but the written derivation does not usually reflect their use. We describe and illustrate a rigorous way to apply these techniques and record their use.

Publishing Information

To appear in: David Till and Roger C. F. Shaw (Editors), *Proceedings of the 6th Refinement Workshop*, London, 5-7 January 1994, Springer-Verlag, 1994.

*Software Verification Research Centre, University of Queensland, Brisbane, Australia. Email: `nickson@cs.uq.oz.au`.

†Email: `lindsay@comp.vuw.ac.nz`.

1 Introduction

The refinement calculus [1, 2, 3] provides a set of rules for deriving programs from formal specifications. Each rule defines a small step from specification to program; by building an appropriate sequence of rule applications, we can develop programs hand in hand with their proofs, with the confidence that the whole proof is correct arising from our belief in the validity of each rule. Specifications and executable code are written in a unified *wide-spectrum* language. Programs in this language are transformed by *refinement rules*, each of which is a schema describing how one program fragment (the *subject*) can be transformed to another (the *result*) providing some *applicability condition* holds. Rules generally contain *schema variables* that must be instantiated to provide a concrete refinement theorem that can be used in a development.

We have implemented a tool supporting the refinement calculus in which higher-level strategy is represented by *tactics* [4], which allow sequences of refinements to be packaged into logical units. One of our aims in doing this is to be able to emulate in the refinement calculus some of the *goal-directed* strategies discussed in Gries's textbook [5] on the formal development of programs from their specifications. The essence of the goal-directed approach is that derivation steps are motivated by the structure of the specification (in particular, the structure of the postcondition). A key aspect of this approach is that the general form of a command may often be suggested by a pattern in the specification, but the details must be calculated, perhaps much later in the development. The result is an overall development strategy that builds sequential compositions backwards and **if** and **do** commands inside out: in general, we postulate a command to address part of the postcondition, then calculate a precondition (often but not always the weakest precondition) for that command to establish our whole postcondition, and then produce preceding commands or surrounding guards to set up the required condition.

This strategy conflicts slightly with the top-down development that is a feature of the refinement calculus. The most obvious way to apply a rule is to instantiate all its schema variables, check its applicability condition and then calculate the result. We would like to apply a rule, leaving some of its variables unbound, then determine appropriate instantiations for those variables later in the development. For example, we might want to introduce a sequential composition in which the intermediate assertion is not known, and then develop the second component and calculate the appropriate intermediate assertion. Similarly, we might want to build an **if** or **do** command by developing the commands in the body and then generating guards to guarantee their preconditions; we may not even know how many guarded commands will be needed. This we normally do informally when performing refinements by hand, and only write down the details of the rule applications afterwards. We want to be able to support the application of rule schemas while deferring the fine details of the application, and to record the choice of schema when it is made. Most importantly, we must do this without compromising the soundness of the refinement method that arises from the rigorous application of fully instantiated rules and checking of applicability conditions.

Our *metavariables* technique supports this approach. A rule can be applied without instantiating all its schema variables; the result of this application itself contains metavariables. The applicability condition too may be only partly instantiated; in that case, it must be posted as a constraint on the later binding of the metavariables in it. A second technique, *conditional refinements*, is used in conjunction with metavariables to allow the development of **if** and **do** commands one guarded command at a time. The value of the techniques is that they allow a development where design decisions can be written down as soon as they are made, instead of waiting for the full details to be worked out. Using these techniques, we can

`Introduce Assignment` $(\tilde{x}, \tilde{e})$

$$\frac{\tilde{x} \subseteq \tilde{w} \qquad pre \Rightarrow post[\tilde{x} := \tilde{e}]}{\tilde{w}\!:\!\left[pre \;\big/\; post\right] \quad \sqsubseteq \quad \tilde{x} := \tilde{e}}$$

Figure 1: The `Introduce Assignment` refinement rule.

develop a program in a more convenient way, avoiding some of the need to work out parts of the development on one side before recording them in the derivation.

Section 2 presents an example that shows the style of development we expect in the 'traditional' refinement calculus, and discusses the shortcomings exposed by that example. The same example is used in Sections 3 and 4 to illustrate the novel techniques. Section 3 describes metavariables and illustrates their use in several small examples. Section 4 describes how we use conditional refinements to develop naked guarded commands, and combine alternative conditional refinements to form guarded command sets. In Section 5 we present a larger example illustrating both techniques — the derivation of an algorithm for calculating the least common multiple.

2 Refinement

In this section, we describe the refinement calculus in a little more detail, and present the derivation of a simple program using refinement rules from [1] and [3], instantiating them fully before application. This serves to introduce the traditional style of development in the calculus, and shows how the goal-directed strategies we wish to use sometimes conflict with the disciplines of this style.

Specifications are written in the wide-spectrum language as *specification statements*. A specification statement $\tilde{w}\!:\!\left[pre \;\big/\; post\right]$ is a command that will establish the postcondition *post* by changing only variables in the frame $\tilde{w}$, providing the precondition *pre* holds initially. The wide-spectrum language allows arbitrary mixing of specification constructs with executable (*code*) constructs like **if** commands and assignments. To produce a finished, executable program, specification constructs must be implemented in code. The process of transforming specification to code proceeds by a sequence of applications of *refinement rules*, which are schemas of the general form

$$\frac{condition}{subject\ pattern \quad \sqsubseteq \quad result\ pattern}$$

Each such rule states that an occurrence of an instance of the subject pattern can be replaced by the corresponding instance of the result pattern providing the instantiated condition holds.

An example of a refinement rule appears in Figure 1. This rule is `Introduce Assignment` from [3]. The subject pattern $\tilde{w}\!:\!\left[pre \;\big/\; post\right]$ will match any specification statement, and the result pattern $\tilde{x} := \tilde{e}$ gives the form of the assignment command that the rule produces. The applicability condition constrains the rule to apply only when the variables $\tilde{x}$ changed by the assignment are all in the frame $\tilde{w}$ and the precondition implies that the substituted postcondition is true.

$$\tilde{w}\!:\!\left[pre \wedge (\bigvee_i b_i) \;\middle/\; post \right] \quad \sqsubseteq \quad \textbf{if } []_i\, b_i \rightarrow \tilde{w}\!:\!\left[pre \wedge b_i \;\middle/\; post \right] \textbf{ fi}$$

Figure 2: Morgan and Robinson's original **if** rule.

The schema variables in `Introduce Assignment` are $\tilde{w}$, *pre*, *post*, $\tilde{x}$ and $\tilde{e}$. Schema variables are subdivided into *subject variables* (occurring in the subject pattern) and *parameters* (listed in the heading of the rule)[1]. Subject variables allow a rule to be applied to any of a collection of possible subjects. Parameters allow us to apply the same rule in the same circumstances and arrive at different results. In `Introduce Assignment`, the subject variables are $\tilde{w}$, *pre* and *post*, and the parameters are $\tilde{x}$ and $\tilde{e}$. Matching the subject against the subject pattern of the selected rule instantiates the subject variables. For the simple matching of abstract syntax we use with refinement rules, this instantiation is unique[2]. The parameters must normally be instantiated explicitly by the rule user when the rule is invoked: our implemented rules are annotated with procedural code to obtain instantiations for parameters.

We now present an example to illustrate this style of development. Consider the specification 'set z to the greater of the two integers x and y', formalized as follows:

$$\begin{array}{ll} [\![& z = max(x, y) \mathrel{\widehat{=}} z \geq x \wedge z \geq y \wedge (z = x \vee z = y) \bullet \\ & z\!:\!\left[true \;\middle/\; z = max(x, y) \right] \\ ]\!] & \end{array}$$

Here the notation $z = max(x, y) \mathrel{\widehat{=}} z \geq x \wedge \cdots$ introduces a local definition of a two-place function *max* that can be used inside the block. The body of the definition block is a specification statement, which must be implemented by code that alters z to satisfy $z = max(x, y)$, under the trivial initial assumption *true*.

Our goal directed strategy says that when there is a disjunction in the postcondition, we should consider introducing an **if** command, with one branch for each disjunct. In this case there is an implicit disjunction (in the definition of max) with two disjuncts, so we will introduce an **if** with two branches, though it is not yet clear what the guards should be. We shall compare two versions of the law for introducing **if** (from [1] and [3]), and see how the choice affects the reasoning process required in this problem.

First, consider Morgan and Robinson's original rule for introducing an **if** command (Figure 2). This rule is unconditional and has no parameters; when we apply it to a subject there are no further decisions or proofs required. The obvious interpretation of this rule, however, is it will apply only when the subject is a specification statement whose precondition has the syntactic form $pre \wedge (b_1 \vee \cdots \vee b_n)$. In this case, to make the rule applicable, we must introduce the tautology $x \geq y \vee y \geq x$ into the precondition – in a refinement tool, we must then ensure that the tool does not immediately remove the tautology as a simplification! Thus, before applying the rule, we have to determine how many branches to use and what the guards will be, then manipulate the specification statement so that the disjunction of the guards is explicit in the precondition, which involves showing that one of the guards must be true when *pre* is true.

[1] Our implementation also has *body variables*, which are used to calculate results.

[2] More complex matching, such as associative-commutative matching, is done by our tool in other circumstances.

`Introduce IF` $(b_1, \ldots, b_n)$

$$\frac{pre \Rightarrow (\bigvee_i b_i)}{\tilde{w}{:}\left[pre \;\middle/\; post\right] \quad \sqsubseteq \quad \textbf{if } \;[\!]_i\; b_i \rightarrow \tilde{w}{:}\left[pre \wedge b_i \;\middle/\; post\right] \textbf{ fi}}$$

Figure 3: The `Introduce IF` refinement rule.

`Introduce IF (two branch instance)` (b_1, b_2)

$$\frac{pre \Rightarrow b_1 \vee b_2}{\tilde{w}{:}\left[pre \;\middle/\; post\right] \quad \sqsubseteq \quad \begin{array}{ll} \textbf{if} & b_1 \rightarrow \tilde{w}{:}\left[pre \wedge b_1 \;\middle/\; post\right] \\ [\!] & b_2 \rightarrow \tilde{w}{:}\left[pre \wedge b_2 \;\middle/\; post\right] \\ \textbf{fi} & \end{array}}$$

Figure 4: The two-branch instance of `Introduce IF`.

Now consider the **if** rule from [3], shown in Figure 3. This rule can match any specification statement, and has the guards as parameters. The applicability condition requires that the precondition of the subject implies that at least one of the guards is true. To use this rule, we must determine how many branches are required and select the appropriate guards to provide as parameters to the rule. In our *max* example, the guards required are $x \geq y$ and $y \geq x$. The proof that one of the guards will be true is then associated directly with the rule application, rather than with an application of another rule that simply rewrites the precondition. This fits our desired goal-directed strategy much better than does Morgan and Robinson's original rule. Still, as we shall see, we might want to leave the choice of guards until later.

To introduce the **if** in our *max* example, we invoke the two-branch instance of `Introduce IF` shown in Figure 4. The subject pattern of that rule is $\tilde{w}{:}\left[pre \;\middle/\; post\right]$; matching our subject specification statement against this binds $\tilde{w}$, *pre* and *post* appropriately:

$$\begin{array}{lll} \tilde{w} & \mapsto & z \\ pre & \mapsto & true \\ post & \mapsto & z = max(x, y) \end{array}$$

The parameters supplied for b_1 and b_2 are the guards $x \geq y$ and $y \geq x$. The instantiated applicability condition is $true \Rightarrow x \geq y \vee y \geq x$, whose validity is easily proved. The result of this rule application is the program:

$$\begin{array}{ll} \textbf{if} & x \geq y \rightarrow z{:}\left[x \geq y \;\middle/\; z = max(x, y)\right] \\ [\!] & y \geq x \rightarrow z{:}\left[y \geq x \;\middle/\; z = max(x, y)\right] \\ \textbf{fi} & \end{array}$$

Before applying the rule, we had to decide how many guarded commands were needed, and what the guards should be. In this case, we know we want a two-branch **if**, and have an idea what commands we want inside it, but would rather calculate the guards. In more complex examples, we might not know in advance how many guarded commands are needed.

The specifications in the branches can be satisfied by assignment commands, introduced with the rule `Introduce Assignment` (Figure 1). In the first branch, matching the subject pattern with the specification statement yields the bindings $\tilde{w} \mapsto \{z\}$, $pre \mapsto x \geq y$ and $post \mapsto z = max(x, y)$. The parameters of the rule are $\tilde{x}$ and $\tilde{e}$; we supply the values $\{z\}$ and $\{x\}$ for these, giving the assignment command $z := x$. The second branch behaves similarly; the bindings are $\tilde{x} \mapsto \{z\}$ and $\tilde{e} \mapsto \{y\}$ giving $z := y$. The subject variable and parameter bindings must also be applied to the applicability conditions of the rule, yielding the instances

$$\{z\} \subseteq \{z\}, \quad x \geq y \Rightarrow (z = max(x, y))[z := x] \quad \text{and}$$
$$\{z\} \subseteq \{z\}, \quad y \geq x \Rightarrow (z = max(x, y))[z := y]$$

These are easily discharged by performing the substitutions, expanding max and simplifying. The resulting program is

$$\begin{array}{ll} \textbf{if} & x \geq y \rightarrow z := x \\ [\!] & y \geq x \rightarrow z := y \\ \textbf{fi} & \end{array}$$

3 Metavariables

We suggest that deferring the instantiation of parameters would allow us to develop programs in the goal-directed style, while still retaining the top-down refinement calculus approach. If we invoke a rule without first instantiating all of its parameters, the resulting program contains uninstantiated schema variables. It will be convenient to give these variables names distinct from the names of the parameters in the rule (in part so we can distinguish different applications of the same rule); we shall reserve the term *metavariables* for them, and use initial capital letters for their names. If the applicability condition of the partially instantiated rule contains metavariables, we shall not be able to discharge it immediately. We cannot discard the obligation, as the soundness of the refinement depends on it; instead, we post it as a *constraint* on the metavariables within it, which future instantiations of those metavariables must satisfy. Typically, later refinements of the program containing metavariables will add further constraints, and heuristics can be used to *solve* the set of constraints, suggesting suitable instantiations for the metavariables.

As long as every metavariable in a development is eventually instantiated in a way that satisfies all constraints, the soundness of the development is not compromised. To demonstrate this, the development can be presented as a strict sequence of applications of fully instantiated refinement rules: the motivation for the choice of parameters will be lost, but the proof obligation associated with each refinement will be explicit and can be checked. The use of metavariables and the posting and solving of constraints is recorded in the development, so the choice of a rule schema is associated with the specification that motivated it, and the details of metavariable (parameter) instantiations are associated with the stage in the derivation where the constraints were solved.

3.1 Metavariables and Sequential Composition

The rule that introduces sequential composition (;) is shown in Figure 5, in which the intermediate assertion is the parameter. An alternative is the rule `Weakest Prespecification` (Figure 6), described in [1]. This rule introduces an arbitrary command s (the parameter) as the second component of the composition, and allows the calculation of an appropriate intermediate assertion: this well supports the goal-directed development strategy, since the

`Split Specification` (mid)

$$\tilde{w}{:}\left[pre \;\middle/\; post \right] \quad \sqsubseteq \quad \tilde{w}{:}\left[pre \;\middle/\; mid \right] \,;\, \tilde{w}{:}\left[mid \;\middle/\; post \right]$$

Figure 5: The `Split Specification` refinement rule.

`Weakest Prespecification` (s)

$$\frac{s \text{ changes only } \tilde{w}}{\tilde{w}{:}\left[pre \;\middle/\; post \right] \quad \sqsubseteq \quad \tilde{w}{:}\left[pre \;\middle/\; wp(s, post) \right] \,;\, s}$$

Figure 6: The `Weakest Prespecification` refinement rule.

appropriate command s is often suggested by examining the input specification. `Weakest Prespecification` works well where s is an assignment (in which case it behaves just like *following assignment* [3]) or a sequential composition of assignments, but is usually inconvenient when s is an **if** or **do** command, because the formula for the weakest precondition can be quite complicated. In these cases, it is usually more convenient to construct the required command, then find an adequate precondition satisfying one of Dijkstra's basic theorems for the alternative and repetitive constructs [6, pp37–40]. But we can avoid directly appealing to such theorems, because we have designed our `Introduce IF` and `Introduce DO` rules to take them into account.

We propose to achieve the effect of `Weakest Prespecification` by using `Split Specification` (Figure 5), but deferring the instantiation of the *mid* parameter. The subject specification statement is matched to the subject pattern of `Split Specification`, instantiating $\tilde{w}$, *pre* and *post*. There is no applicability condition. The partially instantiated result pattern (with a metavariable *Int* as the intermediate assertion) replaces the subject in the program. Now we have a sequence of two specifications; the first has *Int* as its postcondition, and the second has *Int* as its precondition. By refining the second component $\tilde{w}{:}\left[Int \;\middle/\; post \right]$ to some command s, we generate constraints on the metavariable whose cumulative effect is $Int \Rightarrow wp(s, post)$; solving these constraints instantiates *Int* so we can proceed with the refinement of the first component of the sequence. We have thus achieved a practical approximation of the effect of `Weakest Prespecification` without actually calculating a weakest precondition (the individual refinements will constrain *Int* to be no weaker than the weakest precondition), without explicitly checking that s changes only the variables $\tilde{w}$ (the refinements themselves will do that), and without necessarily knowing in advance exactly what command s we wanted.

If we use `Split Specification` with a metavariable as before, then refine the first component $\tilde{w}{:}\left[pre \;\middle/\; Int \right]$ to some command s (perhaps less likely in a goal-directed system, since we do not know its postcondition), we get the effect of calculating an approximation to the strongest postcondition $sp(pre, s)$ (actually, it may not be the *strongest* postcondition, but it will be no stronger). By combining the single rule `Split Specification` with the metavariables technique, we have subsumed the two rules `Weakest Prespecification` and `Strongest Postspecification`. If the command s in either case is a simple assignment (introduced by `Intro Assignment`, Figure 1), we have subsumed Morgan's *following assignment* and a powerful version of *leading assignment*.

A simple example

Given the specification $x, y{:}\left[\, true \;\middle/\; x = 1 \wedge y = 2 \,\right]$, the goal directed strategy suggests attacking each conjunct in the postcondition in turn. Since the conjuncts have no variables in common, it seems likely that we shall need one command for each, so we first split the specification, leaving a metavariable *Int* as the intermediate assertion:

$$\begin{array}{ll} & x, y{:}\left[\, true \;\middle/\; x = 1 \wedge y = 2 \,\right] \\ \sqsubseteq & \texttt{Split Specification} \;\; (Int) \\ & x, y{:}\left[\, true \;\middle/\; Int \,\right] \; ; \; x, y{:}\left[\, Int \;\middle/\; x = 1 \wedge y = 2 \,\right] \end{array}$$

No refinement of the first component of the composition suggests itself, since we do not know the form of the postcondition. In the second component, the conjunct $y = 2$ in the postcondition suggests an assignment $y := 2$, so we try `Introduce Assignment`:

$$\begin{array}{ll} & x, y{:}\left[\, Int \;\middle/\; x = 1 \wedge y = 2 \,\right] \\ \sqsubseteq & \texttt{Introduce Assignment} \;\; (y, 2) \\ & y := 2 \end{array}$$

The proof obligation is $Int \Rightarrow (x = 1 \wedge y = 2)[y := 2]$. Simplifying the obligation yields $Int \Rightarrow x = 1$, but no further simplification can be performed without instantiating the metavariable *Int*. We record the simplified obligation as a constraint on the metavariable. In this case, we can immediately solve the constraint by binding *Int* to the value $x = 1$, resulting in the expected sequence $x, y{:}\left[\, true \;\middle/\; x = 1 \,\right] \; ; \; y := 2$, which is then refined in the traditional way to $x := 1 \; ; \; y := 2$. In other circumstances we might leave the binding of the metavariable until later.

3.2 Metavariables and the Alternative Command

Gries gives the following strategy for developing an alternation [5, p174]:

> To invent a guarded command, find a command C whose execution will establish postcondition R in at least some cases; find a Boolean B satisfying $B \Rightarrow wp(C, R)$; and put them together to form $B \rightarrow C$. Continue to invent guarded commands until the precondition of the construct implies that at least one guard is true.

This strategy is based on a theorem [5, p135] which states that Q is an adequate (but not necessarily weakest) precondition for **if** $[\!]_i (b_i \rightarrow s_i)$ **fi** to establish the postcondition R if and only if $Q \Rightarrow \bigvee_i b_i$ and for each i from 1 to n, $Q \wedge b_i \Rightarrow wp(s_i, R)$. It is goal directed in the sense that it makes use of commands built to satisfy parts of the postcondition, and then sets up their preconditions (the guards) to ensure the satisfaction of the whole postcondition.

Consider again the *max* specification:

$$\begin{array}{ll} [\![& z = max(x, y) \mathrel{\hat=} z \geq x \wedge z \geq y \wedge (z = x \vee z = y) \bullet \\ & z{:}\left[\, true \;\middle/\; z = max(x, y) \,\right] \\ ]\!] & \end{array}$$

Again, we want a two-branch **if**, but this time we want to calculate the guards based on refinements of the body. We invoke `Introduce IF`, leaving metavariables C and D for the guard parameters b_1 and b_2:

$$
\begin{array}{ll}
 & z{:}\left[\mathit{true} \;\middle/\; z = \mathit{max}(x, y) \right] \\
\sqsubseteq & \texttt{Introduce IF} \;\; (C, D) \\
 & \mathbf{if}\ C \rightarrow z{:}\left[C \;\middle/\; z = \mathit{max}(x, y) \right] \\
 & [\!]\ D \rightarrow z{:}\left[D \;\middle/\; z = \mathit{max}(x, y) \right] \\
 & \mathbf{fi}
\end{array}
$$

The **if** command was obtained by:

- matching the specification statement (subject) with the subject pattern $\tilde{w}{:}\left[\mathit{pre} \;\middle/\; \mathit{post} \right]$, giving the bindings $\tilde{w} \mapsto \{z\}$, $\mathit{pre} \mapsto \mathit{true}$ and $\mathit{Post} \mapsto z = \mathit{max}(x, y)$;
- supplying metavariables C and D for the parameters b_1 and b_2; and
- substituting into the result pattern and simplifying.

The instantiated applicability condition is $\mathit{true} \Rightarrow (C \vee D)$, which is simplified to $C \vee D$. This contains metavariables, so it cannot be proved immediately. Instead, its proof is deferred until the metavariables become bound. That is, $C \vee D$ constrains the possible values of those metavariables; no later instantiations of (or constraints on) C and D will be accepted that conflict with this constraint.

Then we decide to introduce an assignment command inside each guarded command, suggested by the presence of the *variable* = *value* disjuncts which are still implicit in the postcondition. In the first branch, we introduce $z := x$, and in the second $z := y$. Applying the corresponding subject variable and parameter bindings to the applicability conditions gives the respective instances

$$
\begin{array}{ll}
\{z\} \subseteq \{z\}, & C \Rightarrow (z = \mathit{max}(x, y))[z := x] \quad \text{and} \\
\{z\} \subseteq \{z\}, & D \Rightarrow (z = \mathit{max}(x, y))[z := y]
\end{array}
$$

Performing the substitutions, expanding *max* and simplifying yields $C \Rightarrow x \geq y$ and $D \Rightarrow y \geq x$. These can be simplified no further, and since they contain metavariables, they are posted as constraints.

Now the constraint solver (whether human or machine) detects possible bindings for C and D that satisfy all constraints, and posts the tentative bindings $C \mapsto x \geq y$ and $D \mapsto y \geq x$. This solution is not unique (in particular, there are two other solutions where one of C and D is strengthened to a strict inequality), so we must be prepared to retract those bindings in the event that further inconsistent constraints appear later.

3.3 Sequential Composition and Alternation

We can modify the *max* example slightly to illustrate an extension of Gries's alternative command strategy. The extension allows us to calculate a sequence whose second component is an **if** command, and whose first component prepares the state so that the **if** does not abort. Such constructs arise frequently when using a goal-oriented approach, wherein we introduce commands to achieve part of the postcondition, and then calculate a precondition for those commands to achieve the whole postcondition. The calculated precondition serves as the postcondition of the first component of a sequence.

Consider the specification $v, z{:}\left[\mathit{true} \;\middle/\; z = \mathit{max}(x, y) \wedge v = x + 1 \right]$. It is fairly obvious in this case that we should consider satisfying $v = x + 1$ with an assignment that is independent of the *max* calculation; we can do it before or after the **if**, or even within each branch. For the sake of the example, let us imagine that it is not obvious. First, we anticipate the sequential composition as before; we can make this a general goal-directed strategy for introducing **if**, since in the case where the composition was not necessary, its first component will refine to **skip** and we can use the identity **skip**; $S = S$. A sequential composition is introduced with `Split Specification`, using a metavariable P as the *mid* parameter. There is no applicability condition, so no constraint is posted at this stage.

We cannot do more with the first component now, since the method is goal-directed (based on the form of the postcondition), and we do not yet know its postcondition. We can address the hidden disjunction of equalities in the postcondition of the second component; as before, we make the **if** guards metavariables C and D, and introduce assignments. The derivation looks like:

$$
\begin{array}{rll}
 & v, z{:}\left[\mathit{true} \;\middle/\; z = \mathit{max}(x, y) \wedge v = x + 1 \right] & \\
 & \sqsubseteq \;\; \texttt{Split Specification} \;\; (P) & \\
 & v, z{:}\left[\mathit{true} \;\middle/\; P \right] ; & (i) \\
 & v, z{:}\left[P \;\middle/\; z = \mathit{max}(x, y) \wedge v = x + 1 \right] & (ii) \\
(ii) & \sqsubseteq \;\; \texttt{Introduce IF} \;\; (C, D) & \\
 & \mathbf{if}\ C \rightarrow v, z{:}\left[P \wedge C \;\middle/\; z = \mathit{max}(x, y) \wedge v = x + 1 \right] & (iii) \\
 & [\!]\ D \rightarrow v, z{:}\left[P \wedge D \;\middle/\; z = \mathit{max}(x, y) \wedge v = x + 1 \right] & (iv) \\
 & \mathbf{fi} & \\
(iii) & \sqsubseteq \;\; \texttt{Introduce Assignment} \;\; (z, x) & \\
 & z := x & \\
(iv) & \sqsubseteq \;\; \texttt{Introduce Assignment} \;\; (z, y) & \\
 & z := y &
\end{array}
$$

The constraints posted are:

$$
\begin{array}{ll}
P \Rightarrow C \vee D & \text{from the } \mathbf{if} \text{ rule} \\
P \wedge C \Rightarrow x \geq y \wedge v = x + 1 & \text{from the first assignment} \\
P \wedge D \Rightarrow y \geq x \wedge v = x + 1 & \text{from the second assignment}
\end{array}
$$

Once again, there is no single solution. A heuristic that we have found appropriate in such circumstances [7] is to treat the right-hand sides of the implications in the last two constraints as sets of conjuncts. Conjuncts in the intersection of the sets ($v = x + 1$ in this example) are associated with the common conjunct P on the left-hand side; the rest are associated with the other conjuncts C and D on the left-hand side. This suggests the bindings $P \mapsto v = x + 1$, $C \mapsto x \geq y$, $D \mapsto y \geq x$. These bindings satisfy all three constraints adequately, though once again the solution is not unique and we must be prepared to seek alternatives if later constraints demand it.

Instantiating P instantiates it everywhere, so the first component (i) of the sequential composition becomes $v, x{:}\left[\mathit{true} \;\middle/\; v = x + 1 \right]$, which can be satisfied by a single application of the assignment rule.

4 Conditional Refinements

Metavariables as described above allow us to support the strategy of building the commands inside an **if** or **do** before the guards. They do not give us the full strategy of Gries, in which the guarded commands are added one at a time until all cases are covered. For that, we propose to use *conditional refinements*.

The idea is to build guarded commands one at a time and then combine them into an **if** or **do** command once all the necessary cases have been covered. To do this, we need a semantics for guarded commands independent of the **if** or **do** that normally surrounds them; this is provided by an extension to the guarded commands language ([8, 9]) that treats the normal **if** command as being a composition using distinct operators: guarding ($\rightarrow$), demonically nondeterministic alternative composition ($[\!]$), and **if** itself. Each of these operators has its own weakest preconditions semantics (**do** is defined in terms of **if** and recursion, as usual):

$$\begin{array}{lcl} wp(b \rightarrow s, \psi) & \hat{=} & b \Rightarrow wp(s, \psi) \\ wp(s \,[\!]\, t, \psi) & \hat{=} & wp(s, \psi) \wedge wp(t, \psi) \\ wp(\mathbf{if}\ s\ \mathbf{fi}, \psi) & \hat{=} & wp(s, \psi) \wedge \neg wp(s, \mathit{false}). \end{array}$$

Using these definitions, the meaning of a 'normal' **if** command

$$\mathbf{if}\ b_1 \rightarrow s_1 \,[\!]\, \cdots [\!]\, b_n \rightarrow s_n\ \mathbf{fi}$$

is the same as it is according to Dijkstra, providing the commands s_i are not miraculous.

We can then define refinement rules to introduce a naked guarded command, to combine commands into a nondeterministic composition, and to enclose a command in an **if** or **do**. Possible versions of these are shown in Figure 7. The last three of these rules differ from rules we have seen so far in this important way: some of their applicability conditions are themselves refinements. To apply such a rule, we treat the refinement *lemmas* in the same way as we treat lemmas in deductive reasoning: they are proved independently of the development in which they are used (either before the step or after it). We attempt to prove lemmas before applying the rule, and post as constraints any lemmas that cannot be proven until metavariables within them are bound.

Sadly, the **if** and **do** constructors are not monotonic with respect to the refinement relation. That is, for programs S and T where $S \sqsubseteq T$, it does not necessarily follow that $\mathbf{if}\ S\ \mathbf{fi} \sqsubseteq \mathbf{if}\ T\ \mathbf{fi}$ and $\mathbf{do}\ S\ \mathbf{od} \sqsubseteq \mathbf{do}\ T\ \mathbf{od}$. As a counter-example for either, take $S \mapsto (x \neq 0 \rightarrow x := 0)$ and $T \mapsto (\mathit{false} \rightarrow x := 0)$. The practical effect of this is to prevent us from doing further refinement (other than by equivalence) on the guarded commands once `Introduce IF` or `Introduce DO` has been used. This is acceptable if we are using the rules in Figure 7, since we choose t to be the desired code, prove its derivation (from s, in the case of `Introduce IF`, or from $\tilde{w}\!:\!\left[\mathit{inv} \wedge \mathit{var} = \mathit{var}_0 \middle/ \mathit{inv} \wedge \mathit{var} < \mathit{var}_0\right]$ in the case of `Introduce DO`), and then apply `Introduce IF` or `Introduce DO` with the derivation of t satisfying the refinement lemma. We can make this more practically appealing using metavariables: we introduce the **if** or **do** with a metavariable for t, so the applicability conditions of the rule become constraints on t. We then address the refinement lemma (containing the metavariable), and instantiate the metavariable on the way.

[3] Theodore Norvell has pointed out that this could be weakened to $wp(s, \mathit{true}) \Rightarrow \neg wp(t, \mathit{false})$: in cases where s would fail to terminate, **if** t **fi** need not terminate either, so t need not be feasible. Similarly, $wp(s, \mathit{true})$ could be added to the antecedent of the last applicability condition in `Introduce DO`.

`Introduce Naked Guarded Command` (b)

$$\tilde{w}\!:\!\left[\, pre \;\middle/\; post \,\right] \quad \sqsubseteq \quad b \rightarrow \tilde{w}\!:\!\left[\, pre \wedge b \;\middle/\; post \,\right]$$

`Introduce Demonic Nondeterminism` $(t_1, \ldots, t_n)$

$$\frac{s \sqsubseteq t_1, \quad \ldots, \quad s \sqsubseteq t_n}{s \quad \sqsubseteq \quad t_1 \mathbin{[\!]} \cdots \mathbin{[\!]} t_n}$$

`Introduce IF` (t)

$$\frac{\begin{array}{c} s \sqsubseteq t \\ \neg wp(t, \mathit{false})^3 \end{array}}{s \quad \sqsubseteq \quad \textbf{if } t \textbf{ fi}}$$

`Introduce DO` (var, inv, t)

$$\frac{\begin{array}{c} pre \Rightarrow inv \\ \tilde{w}\!:\!\left[\, inv \wedge var = var_0 \;\middle/\; inv \wedge var < var_0 \,\right] \quad \sqsubseteq \quad t \\ inv \wedge wp(t, \mathit{false}) \Rightarrow post \end{array}}{\tilde{w}\!:\!\left[\, pre \;\middle/\; post \,\right] \quad \sqsubseteq \quad \textbf{do } t \textbf{ od}}$$

Figure 7: The rules for introducing naked guarded commands, demonic nondeterminism, **if** and **do** constructs.

4.1 The *max* Example

Let us address the *max* example using these rules. First, assume we have recognized the need for an **if** command, but have not analysed the number of branches needed or what their guards should be. We invoke `Introduce IF` with a metavariable T as parameter.

$$\begin{array}{ll} & z{:}\left[\mathit{true} \;\middle/\; z = \mathit{max}(x, y) \right] \\ \sqsubseteq & \texttt{Introduce IF} \;\; (T) \\ & \textbf{if } T \textbf{ fi} \end{array}$$

The partly instantiated applicability conditions are $z{:}\left[\mathit{true} \;\middle/\; z = \mathit{max}(x, y) \right] \sqsubseteq T$ and $\neg wp(T, \mathit{false})$. The first of these will be addressed immediately; the second is posted as a constraint on T.

We now wish to find a T satisfying both the refinement lemma and the constraint. The postcondition of the specification statement in the refinement lemma contains implicit disjuncts $z = x$ and $z = y$. Because of the first of these, we want to introduce the assignment $z := x$, as a guarded command. First, we first introduce a naked guarded command, with a metavariable as the guard:

$$\begin{array}{ll} & z{:}\left[\mathit{true} \;\middle/\; z = \mathit{max}(x, y) \right] \\ \sqsubseteq & \texttt{Introduce Naked Guarded Command} \;\; (C) \\ & C \rightarrow z{:}\left[C \;\middle/\; z = \mathit{max}(x, y) \right] \end{array}$$

The rule has no applicability condition, so no constraint on C is posted. Upon refining the body to $z := x$, we obtain the constraint $C \Rightarrow x \geq y$ as before. Now, $C \rightarrow z := x$ is a candidate for T, but this cannot satisfy the constraint on T from `Introduce IF`, since $\neg wp(C \rightarrow z := x, \mathit{false}) = C$, and C is at least as strong as $x \geq y$. Instead, we put that refinement aside, and return to the original specification. We introduce a naked guarded command with a different metavariable D, and this time refine the body to $z := y$, generating the constraint $D \Rightarrow y \geq x$.

We have now produced two alternative refinements of the same specification, and each has generated its own conditions on the values of the metavariables. We can combine the alternative refinements with the rule `Introduce Demonic Nondeterminism`. We already have constraints that guarantee the necessary refinement lemmas, so the partially instantiated refinement rule

$$\frac{\begin{array}{c} z{:}\left[\mathit{true} \;\middle/\; z = \mathit{max}(x, y) \right] \sqsubseteq C \rightarrow z := x \\ z{:}\left[\mathit{true} \;\middle/\; z = \mathit{max}(x, y) \right] \sqsubseteq D \rightarrow z := y \end{array}}{z{:}\left[\mathit{true} \;\middle/\; z = \mathit{max}(x, y) \right] \quad \sqsubseteq \quad C \rightarrow z := x \;[\!]\; D \rightarrow z := y}$$

needs no further proof, and no more constraints are generated.

Finally, we bind T from our application of `Introduce IF` to this alternation. The constraint is $\neg wp(C \rightarrow z := x \;[\!]\; D \rightarrow z := y,\; \mathit{false})$, which reduces to $(C \vee D)$. We now have the same three constraints as in the earlier treatment of this example, and so can produce the same solution. If we could not find suitable bindings, we could prove further conditional refinements and add the resulting guarded commands to the demonic choice before binding T.

5 Example: Least Common Multiple

Dijkstra [6, p49] gives a simple, elegant modification of Euclid's greatest common divisor algorithm that additionally computes the least common multiple (and sets its proof as an exercise). Proving this four-line algorithm correct, and even getting an intuitive idea of how it works, demands a deep understanding of the factorization properties of the integers involved, and a great deal of insight. Here we present a development of a rather less elegant but more intuitive least common multiple algorithm that well illustrates the metavariables and conditional refinements techniques discussed above.

5.1 Specifying Least Common Multiple

The least common multiple $\mathsf{lcm}(x, y)$ of positive integers x and y is defined by

$$z = \mathsf{lcm}(x, y) \mathrel{\widehat{=}} x \mid z \wedge y \mid z \wedge \neg(\exists z' < z \bullet x \mid z' \wedge y \mid z')$$

That is, $\mathsf{lcm}(x, y)$ is the smallest number that is evenly divisible by both x and y. The notation $a \mid b$ (read 'a divides b') is defined to be true when $a \in \mathbb{N}$, $b \in \mathbb{N}$ and $\exists c \in \mathbb{N} \bullet a = b \times c$. For convenience, we shall use conventional (real-valued) division $\frac{a}{b}$ in the development, which is well-defined wherever $b \neq 0$.

With these definitions in place, we can state the specification of our problem:

$$z{:}\left[x \geq 1 \wedge y \geq 1 \;\middle/\; z = \mathsf{lcm}(x, y)\right]$$

In our refinement, we shall not expand the definition of lcm directly; rather, we appeal to several theorems about the distribution of $\times$ through lcm. The first unconditionally distributes a common factor through both arguments of lcm, while the others are conditional rules that distribute through one argument or the other. These theorems are easily proved from the definition of lcm.

`×-distrib-lcm-both`:	$\mathsf{lcm}(a \times b, a \times c) = a \times \mathsf{lcm}(b, c)$
`×-distrib-lcm-left`:	$\mathsf{lcm}(a \times b, c) = a \times \mathsf{lcm}(b, c)$ if $\mathsf{gcd}(a, c) = 1$
`×-distrib-lcm-right`:	$\mathsf{lcm}(b, a \times c) = a \times \mathsf{lcm}(b, c)$ if $\mathsf{gcd}(a, b) = 1$

5.2 Initial Refinement

We now plan our strategy for refining the specification. We want to enumerate the prime factors of x and y: the product of the bag maximum of the factors is the desired least common multiple[4]. To find the prime factors, we use local variables u and v, whose values are initially x and y, then divide factors from u and v as we find them and accumulate their product in the output z. We shall use a third new variable i to enumerate the potential factors. Rather than generating primes explicitly, we rely on the fact that the smallest prime factor of any number is smaller than any composite factor. This means that having divided out all factors i from u and v, we can simply increment i without concern as to whether the result is prime: if it is composite, it will certainly not be a factor since all primes smaller than it have already been tried.

[4] Bag maximum combines bags B and C such that the number of times any element appears in the maximum is the greater of the number of times it appears in B and C.

We clearly need a loop. We use `Split Specification` with a metavariable *Int* as the intermediate assertion. The first component of the resulting composition will be refined to the initialization, the second to the loop itself.

$$
\begin{array}{lll}
 & & z{:}\left[x \geq 1 \wedge y \geq 1 \;\middle/\; z = \mathsf{lcm}(x,y) \right] \qquad (i) \\
(i) & \sqsubseteq & \texttt{Introduce Variable} \quad (u, v, i : \mathbb{N}) \\
 & & [\![\ \mathbf{var}\ u, v, i : \mathbb{N} \bullet \\
 & & \quad u, v, i, z{:}\left[x \geq 1 \wedge y \geq 1 \;\middle/\; z = \mathsf{lcm}(x,y) \right] \qquad (ii) \\
 & &]\!] \\
(ii) & \sqsubseteq & \texttt{Split Specification} \quad (Int) \\
 & & u, v, i, z{:}\left[x \geq 1 \wedge y \geq 1 \;\middle/\; Int \right] ; \qquad (iii) \\
 & & u, v, i, z{:}\left[Int \;\middle/\; z = \mathsf{lcm}(x,y) \right] \qquad (iv)
\end{array}
$$

5.3 Building the Loop

We want the loop invariant to establish the value of z as the product of the factors found so far. Since we are dividing the factors from u and v, some rearrangement suggests putting $z = \mathsf{lcm}(\frac{x}{u}, \frac{y}{v})$ in the invariant. This will establish the postcondition when both u and v reach 1. The invariant must also ensure that all factors found have been correctly divided out of u and v: $u \mid x \wedge v \mid y \wedge (\forall j \in 2 \,..\, i-1 \bullet j \nmid u \wedge j \nmid v)$ achieves this. Additionally, we shall maintain $u \geq 1 \wedge v \geq 1$, and we only want to try for potential factors greater than one, so $i \geq 2$. Putting this together gives the invariant:

$$
\Omega : \left\{ \begin{array}{l} z = \mathsf{lcm}(\frac{x}{u}, \frac{y}{v}) \wedge u \mid x \wedge v \mid y \wedge \\ (\forall j \in 2 \,..\, i-1 \bullet j \nmid u \wedge j \nmid v) \wedge \\ u \geq 1 \wedge v \geq 1 \wedge i \geq 2 \end{array} \right.
$$

We also require a variant (Gries calls it a bound function) to establish termination; this should be a positive function that decreases as we progress towards termination. If we can decrease either u or v we must certainly eventually terminate, but this is too difficult to establish, since some values for i will not divide u or v evenly. Increasing i also makes some progress towards termination, since eventually we shall find an i that divides one or both of u and v. With this in mind, we postulate the variant $\tau : (u+v) \dot{-} i$, where $a \dot{-} b$ is $a - b$ if $a > b$ and zero otherwise. This variant is indeed strictly decreasing, since $(u+v) - i$ will only become negative at the very last stage of the loop, when $u = v = 1$.

We can now introduce the loop itself, supplying as parameters the invariant, variant and a metavariable S to become the command inside the **do**.

$$
\begin{array}{lll}
(iv) & \sqsubseteq & \texttt{Introduce DO} \quad (\Omega, \tau, S) \\
 & & \mathbf{do}\ S\ \mathbf{od} \qquad (v)
\end{array}
$$

The proof obligations are:

$$
\begin{array}{ll}
C1: & Int \Rightarrow \Omega \\
C2: & u, v, i, z{:}\left[\Omega \wedge \tau = \tau_0 \;\middle/\; \Omega \wedge \tau < \tau_0 \right] \quad \sqsubseteq \quad S \\
C3: & \Omega \wedge wp(S, \mathit{false}) \Rightarrow z = \mathsf{lcm}(x,y)
\end{array}
$$

We shall address constraint $C2$ by further refinements shortly, which will bind the metavariable S; this binding must satisfy $C3$.

$C1$ can most easily be satisfied by putting $Int \mapsto \Omega$. This fully instantiates the initialization specification (iii), which we now refine to an assignment:

$$\begin{array}{ll} (iii) \sqsubseteq & \texttt{Introduce Assignment} \ \ ([u,v,i,z],[x,y,2,1]) \\ & u,v,i,z := x,y,2,1 \end{array}$$

This step requires us to prove that the precondition (of (iii)) implies the substituted postcondition $\Omega[u,v,i,z := x,y,2,1]$:

$$\begin{array}{l} x \geq 1 \wedge y \geq 1 \Rightarrow \\ \quad 1 = \mathsf{lcm}(\frac{x}{x}, \frac{y}{y}) \wedge x \mid x \wedge y \mid y \wedge \\ \quad (\forall j \in 2 \mathinner{..} 1 \bullet \cdots) \wedge x \geq 1 \wedge y \geq 1 \wedge 2 \leq 2 \end{array}$$

The first conjunct of the consequent follows from the definition of lcm; the $\forall j$ conjunct is trivially true since $2 \mathinner{..} 1$ is an empty range; and the other conjuncts are trivial.

5.4 Building the Guarded Commands

We now focus on $C2$, refining $u,v,i,z{:}\left[\Omega \wedge \tau = \tau_0 \;\middle/\; \Omega \wedge \tau < \tau_0\right]$ to a guarded command set S. We build guarded commands one at a time until their nondeterministic composition satisfies $C3$. We do not yet know how many branches S will have, nor what the guards will be. So, we start generating guarded commands using `Introduce Naked Guarded Command` with a metavariable as parameter. The appropriate refinements, then, are of the form

$$\begin{array}{ll} & u,v,i,z{:}\left[\Omega \wedge \tau = \tau_0 \;\middle/\; \Omega \wedge \tau < \tau_0\right] \\ \sqsubseteq & G_k \rightarrow S_k \end{array} \qquad (vi)$$

where each S_k is chosen to decrease the variant τ, while maintaining the invariant Ω under the assumption G_k.

We can reduce τ by decreasing either u or v, or by increasing i. Let us try each of these in turn. To decrease u, but maintain the invariant, we can divide u by i and multiply z by the same thing, and similarly for v:

$$\begin{array}{ll} (vi) \sqsubseteq & \texttt{Introduce Naked Guarded Command} \ \ (G_1) \\ & G_1 \rightarrow \\ & \quad u,v,i,z{:}\left[G_1 \wedge \Omega \wedge \tau = \tau_0 \;\middle/\; \Omega \wedge \tau < \tau_0\right] \qquad (vii) \\ (vii) \sqsubseteq & \texttt{Introduce Assignment} \ \ ([u,z],[\frac{u}{i}, z \times i]) \\ & u,z := \frac{u}{i}, z \times i \end{array}$$

and similarly,

$$\begin{array}{ll} (vi) \sqsubseteq & \texttt{Introduce Naked Guarded Command} \ \ (G_2), \\ & \texttt{Introduce Assignment} \ \ ([v,z],[\frac{v}{i}, z \times i]) \\ & G_2 \rightarrow v,z := \frac{v}{i}, z \times i \end{array}$$

The applicability conditions of the `Introduce Assignment` steps here produce the constraints

$G_1 \wedge \Omega \wedge \tau = \tau_0 \Rightarrow (\Omega \wedge \tau < \tau_0)[u,z := \frac{u}{i}, z \times i]$
and
$G_2 \wedge \Omega \wedge \tau = \tau_0 \Rightarrow (\Omega \wedge \tau < \tau_0)[v,z := \frac{v}{i}, z \times i]$.

Expanding the definitions of Ω and τ on the right-hand side of the first of these constraints yields

$$G_1 \wedge \Omega \wedge \tau = \tau_0 \Rightarrow \left(\begin{array}{l} z \times i = \mathsf{lcm}(x/\frac{u}{i}, \frac{y}{v}) \wedge \\ \frac{u}{i} \mid x \wedge v \mid y \wedge \\ (\forall j \in 2 \mathinner{..} i-1 \bullet j \nmid \frac{u}{i} \wedge j \nmid v) \wedge \\ \frac{u}{i} \geq 1 \wedge v \geq 1 \wedge i \geq 2 \wedge \\ \frac{u}{i} + v \dot{-} i < \tau_0 \end{array} \right)$$

To simplify this, we recall that we defined divisibility so that $u \mid x \Rightarrow \frac{u}{i} \mid x$ only when $\frac{u}{i}$ is an integer; that is, when $i \mid u$. In this case, $x/\frac{u}{i}$ in the *lcm* conjunct on the right-hand side is an integer (so *lcm* is well-defined), and is equal to $\frac{x}{u} \times i$. Further, if $\frac{u}{i} < u$ then $\frac{u}{i} + v \dot{-} i \leq u + v \dot{-} i$, and is strictly less providing $u + v > i$. Also, the theorem `×-distrib-lcm-left` applies to the rewritten *lcm* conjunct providing $\mathsf{gcd}(i, \frac{y}{v}) = 1$. Finally, $j \nmid u \Rightarrow j \nmid \frac{u}{i}$, which means that the $\forall j$ conjunct follows from the corresponding conjunct of Ω on the left-hand side. Thus we simplify further:

$$G_1 \wedge \Omega \Rightarrow v \mid y \wedge u \geq 1 \wedge i \geq 2 \wedge u + v > i \wedge i \mid u \wedge \mathsf{gcd}(i, \frac{y}{v}) = 1$$

The first three conjuncts of the consequent appear directly in Ω in the antecedent. If $i \mid u \wedge u \geq 1$ then $i \leq u$, so if $v \geq 1$ then certainly $u + v > i$. Multiplication distributes through (both sides of) gcd, so $\mathsf{gcd}(i, \frac{y}{v}) = 1$ if and only if $\mathsf{gcd}(v \times i, y) = v$, which in turn holds as long as $\mathsf{gcd}(v, y) = v \wedge i \nmid v$. The first conjunct of this is true since $v \mid y$. We finish with the constraint $G_1 \wedge \Omega \Rightarrow i \mid u \wedge i \nmid v$, which we solve by putting G_1 equal to the right-hand side. A similar line of reasoning is used to solve for G_2. Our first two refinement lemmas are then

$$(vi) \sqsubseteq i \mid u \wedge i \nmid v \rightarrow u, z := \tfrac{u}{i}, z \times i \qquad (viii)$$

and

$$(vi) \sqsubseteq i \nmid u \wedge i \mid v \rightarrow v, z := \tfrac{v}{i}, z \times i \qquad (ix)$$

Our third guarded command increases i:

$$(vi) \sqsubseteq G_3 \rightarrow i := i + 1 \qquad (x)$$

To determine G_3, we need $G_3 \wedge \Omega \wedge \tau = \tau_0 \Rightarrow (\Omega \wedge \tau < \tau_0)[i := i + 1]$. Expanding Ω, substituting, simplifying and removing subsumed conjuncts on the right-hand side yields

$$G_3 \wedge \Omega \wedge \tau = \tau_0 \Rightarrow \left(\begin{array}{l} (\forall j \in 2 \mathinner{..} i \bullet j \nmid u \wedge j \nmid v) \wedge \\ i \geq 1 \wedge \\ (u + v) \dot{-} (i + 1) < u + v \dot{-} i \end{array} \right)$$

The $\forall$ conjunct on the right-hand side requires $i \nmid u \wedge i \nmid v$ in addition to the one in Ω. We do not want to treat $\dot{-}$ as code, so we observe that $a \dot{-} (b+1) < a \dot{-} b$ providing $b < a$, and replace the last conjunct by $i < u + v$. The result suggests the binding $G_3 \mapsto i \nmid u \wedge i \nmid v \wedge i < u + v$.

To satisfy the $\Omega \wedge wp(S, \mathit{false}) \Rightarrow \mathit{Post}$ applicability condition of `Introduce DO`, we are going to need $\Omega \wedge \neg(\bigvee_k G_k) \Rightarrow z = \mathsf{lcm}(x, y)$. The three guards we have so far do not give this, and the calculation suggests we now need to handle the case where i divides both u and v.

$$(vi) \sqsubseteq G_4 \rightarrow u, v, z := \tfrac{u}{i}, \tfrac{v}{i}, z \times i \qquad (xi)$$

Calculating the applicability conditions and simplifying as above (this time using tt ×-distrib-lcm-both) yields the constraint $G_4 \wedge \Omega \Rightarrow i \mid u \wedge i \mid v$, which we solve once again by putting G_4 equal to the right-hand side.

5.5 Combining the Guarded Commands

Now we use `Introduce Demonic Nondeterminism` to combine the guarded commands $(viii)$ ... (xi). The obligations are the four refinement lemmas we just proved, and the result is $(vi) \sqsubseteq (viii) [\!] (ix) [\!] (x) [\!] (xi)$. If we bind S (from `Introduce DO`) to that alternation, we must show the binding satisfies our delayed constraints $C2$ and $C3$. $C2$ is exactly what we proved with `Introduce Demonic Nondeterminism`: $(vi) \sqsubseteq S$. $C3$ is $\Omega \wedge wp(S, \mathit{false}) \Rightarrow z = \mathsf{lcm}(x, y)$:

$$
\begin{array}{ll}
 & wp(S, \mathit{false}) \\
= & \bigwedge_k (G_k \Rightarrow wp(S_k, \mathit{false})) \\
= & \bigwedge_k (\neg G_k) \quad [\text{since each } S_k \text{ is code}] \\
= & i \nmid u \wedge i \nmid v \wedge u + v \leq i
\end{array}
$$

The conjunct $\forall j \in 2 \mathinner{..} i-1 \bullet j \nmid u \wedge j \nmid v$ from Ω means that the only possible factors of u and v are unity and numbers bigger than i. But i is bigger than either u or v, so that option is ruled out; u and v must both be one[5]. And $\Omega \wedge u = 1 \wedge v = 1$ gives $z = \mathsf{lcm}(x, y)$, as required. The completed, collected program is then

$$
\begin{array}{l}
[\![\;\; \mathbf{var}\ u, v, i : \mathbb{N} \bullet \\
\quad u, v, i, z := x, y, 2, 1\ ; \\
\quad \mathbf{do} \;\; \begin{array}{lll}
i \mid u \wedge i \nmid v & \rightarrow & u, z := \frac{u}{i}, z \times i \\
i \nmid u \wedge i \mid v & \rightarrow & v, z := \frac{v}{i}, z \times i \\
i \nmid u \wedge i \nmid v \wedge i < u + v & \rightarrow & i := i + 1 \\
i \mid u \wedge i \mid v & \rightarrow & u, v, z := \frac{u}{i}, \frac{v}{i}, z \times i
\end{array} \\
\quad \mathbf{od} \\
]\!]
\end{array}
$$

6 Conclusions

We have demonstrated two techniques that allow users of the refinement calculus to record developments in a more convenient way. We have also described how supporting these techniques in a tool gives its users the ability to develop programs in a more experimental fashion, by suggesting refinement schemes and having the tool assist with the calculation of appropriate instantiations of those schemes.

Our metavariables technique allows us to select a rule that seems likely to work in a particular situation, but defer specifying exactly what instance of the rule will be used. The applicability condition of the rule will partly constrain the available instantiations, and later refinements will constrain it further. The metavariables can then be instantiated (subject to the accumulated constraints) at any time we find convenient. We have found this technique useful in many of the derivations we have done by hand; for another example, see [10]. We hope to implement an extension to our refinement tool that represents metavariables with Prolog variables and collects the constraints on them using the co-routining mechanisms available in some versions of Prolog [11].

Clement [12] has used metavariables in the context of proof by natural deduction in the μral proof tool [13]. There, too, the benefit is avoiding the need to plan ahead so that steps can be done when they prove necessary, rather than needing to be anticipated. Clement's metavariables can represent predicate symbols (with arguments); our metavariables can stand for complete terms, formulas or commands only.

[5] In fact, this shows that we could weaken the last conjunct of G_3 to $u \neq 1 \vee v \neq 1$ if we so chose.

The conditional refinements technique allows guarded commands to be developed one at a time, hence deferring the decision of how many branches an **if** or **do** command should have. The semantics makes each guarded command alone a 'minor miracle' [14] that behaves magically if executed when its guard is false. Each individual guarded command is then an alternative valid refinement of the original specification. These alternatives can be combined using rules for introducing demonic choice, and feasibility and termination checks can be made by rules that introduce **if** and **do**. Our proposed implementation includes support for recording and combining alternative developments, but we must also weaken our definition of monotonicity and replace the refinement relation by one (actually a family) with respect to which the **if** and **do** constructors are monotonic.

Support for these techniques is useful in a practical tool, since it allows the tool's user to develop a program in a more convenient way, avoiding some of the need to work out parts of the development by hand before performing them with the tool. Supporting the user's thought processes brings us another benefit too: the sequence of operations can be represented more explicitly in the recorded derivation. If all relevant data are stored, we can provide a view of the completed derivation at any desired level, which may differ depending on the reader's aims. If the reader is interested only in obtaining a correct program, we can easily present that. If a demonstration of the correctness of the program is desired, we can present the derivation as a sequence of applications of (instantiated) refinement rules; the order of presentation may or may not be the order in which the rules were actually applied. When the derivation reader requires a more thorough understanding of the algorithm or of how it was actually developed, full details of the sequence of operations and the motivation for particular instantiations may be required.

The theme that unifies the two techniques described here is the benefit arising from deferring decisions about refinement details. There may be other techniques that users of the refinement calculus apply informally when exploring possible derivations. Identifying, categorizing and formalizing these techniques merits further investigation.

References

[1] Carroll C. Morgan and Ken Robinson. Specification statements and refinement. *IBM Journal of Research and Development*, 31(5):546–555, September 1987.

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

[3] Carroll Morgan. *Programming from Specifications.* Prentice Hall, 1990.

[4] Lindsay Groves, Raymond Nickson and Mark Utting. A tactic driven refinement tool. In Cliff B. Jones, Roger C. Shaw and Tim Denvir, editors, *Fifth Refinement Workshop*, Workshops in Computing, pages 272–297. BCS FACS, Springer-Verlag, 1992.

[5] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

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

[7] Lindsay J. Groves and Raymond G. Nickson. An intelligent editor for constructing correct programs. In *Proceedings of the Third New Zealand Conference on Expert Systems.* New Zealand Expert Systems Special Interest Group, 1988.

[8] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[9] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.

[10] Lindsay Groves. Deriving language recognition algorithms: A case study in combining program specialisation and data refinement. In David Till and Roger C. F. Shaw, editors, *Sixth Refinement Workshop*, Workshops in Computing. BCS FACS, Springer-Verlag, 1994.

[11] L. Naish. Negation and quantifiers in NU-Prolog. In *Proceedings of the Third International Conference on Logic Programming*, 1986.

[12] Tim Clement. Using metavariables in natural deduction proofs. In Cliff B. Jones, Roger C. Shaw and Tim Denvir, editors, *Fifth Refinement Workshop*, Workshops in Computing. BCS FACS, Springer-Verlag, 1992.

[13] C. B. Jones, K. D. Jones, P. A. Lindsay and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.

[14] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.