Programming Research Group

PROCEDURES, PARAMETERS, AND SUBSTITUTION IN THE REFINEMENT CALCULUS

Ana Cavalcanti
Augusto Sampaio
Jim Woodcock

PRG-TR-5-97

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Procedures, Parameters, and Substitution in the Refinement Calculus

Ana Cavalcanti   Augusto Sampaio   Jim Woodcock

February 1997
Revised in June 1997

Abstract

Morgan and Back have proposed different formalisations of procedures and parameters in the context of techniques of program development based on refinement. We investigate a surprising and intricate relationship between these works and the substitution operator that renames the free variables of a program, and reveal an inconsistency in Morgan's refinement calculus. Back's formalisation of procedures does not suffer from this inconsistency, but his work is not as appealing to practising programmers as Morgan's calculus, whose distinctive feature is a large number of refinement laws. Here we benefit from both works and use Back's formalism as a model to derive the laws presented in Morgan's calculus.

Keywords: program development, formal methods, refinement calculus, procedures, parameters.

1 Introduction

Inspired by Dijkstra's work on weakest preconditions (*wp*) [5], Back [1, 3], Morgan [12, 11], and Morris [13, 15] have proposed three different formalisations of the stepwise refinement technique of program development. They are all based on a unified language of specification and programming. From the semantic point of view, this unification is achieved by linking the constructors of the language to a single mathematical model: Dijkstra's *wp*.

In this context, specifications are viewed as particular forms of programs, and we actually use the term program to refer to specifications, programs, and designs, where programming structures and specifications are mixed. Furthermore, a specification can be taken as the starting point for the development of a program which is guaranteed, or can be proved, to be correct with respect to that specification. The correctness of the development can be established by showing that the *wp* semantics is preserved.

Morgan's work is a refinement calculus: it distinguishes itself in that an extensive set of refinement laws is presented. They define program transformations that preserve total correctness (the *wp* semantics), and typically yield programs that are better in some sense (either executable, more efficient, or better suited to some other purpose). In [11], the development process consists of calculating programs by repeatedly applying these laws until an adequate result is obtained. The possibility of calculating, as opposed to verifying, programs accounts for developments that can be uniformly presented as sequences of simple refinement steps. Each step can be justified by the application of a refinement law and, possibly, the discharge of corresponding proof-obligations. Moreover, refinement laws provide guidance on the construction of programs.

Back [2], Morgan [8], and Morris [14] have all formalised the use of procedures and parameters. In this paper, we uncover a rather subtle and unexpected relation between these formalisations and the substitution operator that renames the free variables of a program, and show that Morgan's approach presents an inconsistency. The problem with his work (as illustrated by suitable examples later on) is that formal parameters are regarded as global (or free) variables in the procedure body, rather than as local (or bound) variables as is the case in conventional procedural languages.

Analysing Back's formalism, we conclude that it is free from the inconsistency found, essentially because formal parameters are adequately represented by variables which are bound in the procedure body. On the other hand, Back's work is not as attractive to programmers as Morgan's calculus, due to the lack of refinement laws. As there seems to be no obvious way of avoiding the inconsistency using Morgan's approach itself, we use Back's formalism as a consistent model to derive Morgan's refinement laws. Moreover, we investigate how Morgan's approach to the development of recursive (parametrised) procedures can be formalised in this framework. In this way, we compare, contrast, and integrate two of the major approaches to the formalisation of procedures and parameters. Morris's approach is not considered here since it is directly based on that of Back.

In Section 2 we present the approaches to procedures and parameters adopted by Back and Morgan. Section 3 explains their relationship to the substitution operator and the problem in Morgan's approach. In Section 4, we define the semantics of Morgan's constructs related to the treatment of procedures, using Back's formalism. Section 5 presents refinement laws. Section 6 summarises our results. Finally, an appendix presents the derivation of some of the refinement laws.

2 Procedures and Parameters

In both [2] and [11], procedures are declared in a block. In order to illustrate the particular notation we employ to write procedure blocks, we consider the example below.

$$|[\, \mathbf{proc}\ \mathit{Inc} \mathrel{\widehat=} x := x + 1 \bullet \mathit{Inc}\ ;\ \mathit{Inc} \,]| \tag{1}$$

This very simple program uses the procedure Inc to increase the value of x by 2. The program $x := x + 1$ is the body of Inc, and $\mathit{Inc}\ ;\ \mathit{Inc}$ is the main program (the scope of the procedure).

Both [2] and [8] adopt the copy rule of Algol 60 when defining a semantics for non-recursive procedures. According to this rule, a program that contains a procedure call is equivalent to that obtained by substituting the procedure body for the procedure name. Variable capture must be avoided, in order to ensure that the scope of variables is static. The program (1), for instance, is equivalent to $x := x + 1\ ;\ x := x + 1$, as expected.

In order to illustrate the concerns related to the capture of variables, we consider the program below which assigns 2 to a global variable x.

$$|[\, \mathbf{proc}\ \mathit{Ass2tox} \mathrel{\widehat=} x := 2 \bullet |[\, \mathbf{var}\ x \bullet \mathit{Ass2tox} \,]|\ \,]| \tag{2}$$

The program $|[\, \mathbf{var}\ x \bullet \mathit{Ass2tox} \,]|$ is a variable block: it declares a variable x local to its body, $\mathit{Ass2tox}$. Since there is a variable x free in the body of $\mathit{Ass2tox}$, we cannot, as in the previous example, apply the copy rule and substitute $x := 2$ for $\mathit{Ass2tox}$. This substitution would lead to the capture of the global variable x mentioned in the body of $\mathit{Ass2tox}$ by the local declaration of x in the main program and, therefore, would violate the rules of static scope. Before applying the copy rule to (2), we have to rename the local variable x. This is further discussed in Section 3.

In Back's work parametrised procedures can be defined with the use of parametrised statements. They can have the form $(\mathbf{val}\ vl \bullet p)$, $(\mathbf{res}\ vl \bullet p)$, or $(\mathbf{var}\ vl \bullet p)$, which correspond to the traditional mechanisms of parameter passing known as call-by-value, call-by-result, and call-by-reference, respectively. In each case, vl ranges over lists of variables standing for the formal parameters, and p over programs. These conventions are assumed in subsequent definitions, where we also use l to range over lists of variables. Moreover, we use subscripts to extend the set of metavariables, so that vl_1 and vl_2, for instance, also stand for lists of variables.

As opposed to assignments, for instance, parametrised statements are not programs by themselves. However, a parametrised statement (or the name of a procedure whose body is a parametrised statement) can be applied to a suitable list of actual parameters to yield a program. The resulting program acts as that obtained by passing the actual parameters to the program in the body of the parametrised statement. The number of actual parameters must be the same as the number of formal parameters. The correspondence between them is positional.

By way of illustration, we consider the program that assigns 0 to the variables x and y by using a parametrised procedure *Zero*.

$$[\![\, \mathbf{proc}\ \mathit{Zero} \mathrel{\widehat{=}} (\mathbf{res}\ n \bullet n := 0) \bullet \mathit{Zero}(x)\ ;\ \ \mathit{Zero}(y)\,]\!] \tag{3}$$

By the copy rule, this program is equivalent to $(\mathbf{res}\ n \bullet n := 0)(x)\ ;\ (\mathbf{res}\ n \bullet n := 0)(y)$.

Applications of parametrised statements to actual parameters are defined in terms of variable blocks. For example, call-by-result is defined as

$$(\mathbf{res}\ vl_1 \bullet p)(vl_2) = [\![\, \mathbf{var}\ l \bullet p[vl_1 \backslash l]\ ;\ \ vl_2 := l\,]\!] \tag{4}$$

provided l is a list of fresh variables. This variable block implements call-by-result using the well-known technique of assignment from a local variable. The term $p[vl_1 \backslash l]$ denotes the result of substituting the variables of l for every occurrence of the corresponding variables of vl_1 in p.

In Morgan's approach, the use of parametrised procedures is made possible by substitutions which define both the formal and actual parameters of a procedure at the point(s) of call rather than definition. The forms of substitution available correspond to call-by-value, call-by-result, and call-by-value-result.

For example, a substitution by result has the form $p[\mathbf{result}\ vl_1 \backslash vl_2]$, where p is the program to which it applies, vl_1, the list of formal parameters, and vl_2, the list of actual parameters. In order to write a program equivalent to (3), also using a procedure *Zero*, we employ substitutions by result.

$$[\![\, \mathbf{proc}\ \mathit{Zero} \mathrel{\widehat{=}} n := 0 \bullet \mathit{Zero}[\mathbf{result}\ n \backslash x]\ ;\ \ \mathit{Zero}[\mathbf{result}\ n \backslash y]\ \,]\!]$$

Morgan [8] provides a weakest precondition semantics for substitutions. Nevertheless, they can also be defined in terms of variable blocks. For instance, we can derive from the weakest precondition of a substitution by result that

$$p[\mathbf{result}\ vl_1 \backslash vl_2] = [\![\, \mathbf{var}\ l \bullet p[vl_1 \backslash l]\ ;\ \ vl_2 := l\,]\!] \tag{5}$$

for a list l of fresh variables. This is the definition actually adopted in [11]. The right-hand side of this equation is identical to that of (4).

3 Exploring the Effect of Substitution

As we have mentioned previously, since the main program of the procedure block (2) redeclares x, the copy rule cannot be applied to remove the procedure call. The standard way to overcome this difficulty is to rename this local variable.

As a variable block binds the variables that it declares, it is well-known that these variables can be renamed without changing the meaning of the program, in the same way that the variables bound by quantifiers can be renamed in the predicate calculus. The relevant rule is as follows: for every program p and all lists of variables vl_1 and vl_2, $|[\,\mathbf{var}\ vl_1 \bullet p\,]| = |[\,\mathbf{var}\ vl_2 \bullet p[vl_1 \backslash vl_2]\,]|$ provided the variables of vl_2 are not free in p. By applying this result, we conclude that, since z is not free in $Ass2tox$, (2) is equivalent to $|[\,\mathbf{proc}\ Ass2tox \mathrel{\widehat=} x := 2 \bullet |[\,\mathbf{var}\ z \bullet Ass2tox[x \backslash z]\,]|\,]|$. At this point, our main concern is the result of $Ass2tox[x \backslash z]$.

There seems to be two acceptable possibilities: $Ass2tox$ and $z := 2$. In the first case, the substitution operator acts on the name $Ass2tox$ and, since x is clearly not free in this program, $Ass2tox$ is itself the result: substitution is a syntactic operator; this will be referred to as syntactic substitution. In the second case, the substitution operator acts on the body of $Ass2tox$ and yields the result of substituting z for x in that program: the behaviour of substitution is dependent on the context in which it is applied; we will refer to this as context dependent substitution.

Both forms of substitution can be defined by recursion in the usual way. The interesting parts of their definitions are those concerned with the application of substitution to a procedure name. In the case of syntactic substitution, we have that $pn[vl_1 \backslash vl_2] = pn$, where pn is a metavariable that ranges over procedure names. For context dependent substitution, if p is the body of the procedure pn, then $pn[vl_1 \backslash vl_2] = p[vl_1 \backslash vl_2]$. This somewhat unusual form of substitution is adopted in [16] and, as we explain later on, is part of a possible solution to the problems we uncover here. The main purpose of this section is to show that either definition of substitution leads to inconsistency in Morgan's formalisation of procedures and parameters. Moreover, we show that Back's formalisation presents no problems, provided we adopt syntactic substitution.

If we assume that $Ass2tox[x \backslash z]$ is $z := 2$ (context dependent substitution), we can deduce that $|[\,\mathbf{var}\ z \bullet Ass2tox[x \backslash z]\,]|$ is equivalent to $|[\,\mathbf{var}\ z \bullet z := 2\,]|$. This program, however, can be shown to be equivalent to **skip**: it does not change any variable other than the local variable that it introduces, and always terminates. Overall, we can prove that (2), which is supposed to assign 2 to x, is equivalent to **skip**.

In [16], Sampaio avoids this problem by restricting the application of the law that accounts for the renaming of the variables declared by a variable block. He defines the notion of contiguous scope and establishes that the renaming is possible only if the variables have a contiguous scope in the body of the variable block.

A variable is said to have a contiguous scope in a program if either this program does not contain free procedure names (procedure calls) or the variable is not free in the bodies of the procedures that are called. In the above example, since x does not have a contiguous scope in $Ass2tox$, because x is free in the body of this procedure, we cannot deduce, according to [16], that $|[\,\mathbf{var}\ x \bullet Ass2tox\,]| = |[\,\mathbf{var}\ z \bullet Ass2tox[x \backslash z]\,]|$. Consequently, the undesired deduction that we have presented cannot be carried out.

Although it might be considered a solution to the problem, we cannot adopt this approach if we accept the usual weakest precondition semantics of variable blocks proposed in [2, 12, 11]. In all these models, the equality $|[\,\mathbf{var}\ x \bullet Ass2tox\,]| = |[\,\mathbf{var}\ z \bullet Ass2tox[x \backslash z]\,]|$ can be deduced, provided z is a fresh variable. More generally, the conventional law for renaming bound variables can be derived in all these models. As a consequence, if we assume context dependent substitution, the undesired deduction can be carried out in these models.

Sampaio gives an algebraic semantics for the language introduced in [16]. In this semantics, the restricted renaming law that he proposes (based on the notion of contiguous scope) is regarded as an axiom and no model is presented for the language.

In summary, if we discard the possibility of changing the semantics of variable blocks, we have to assume that $Ass2tox[x \backslash z]$ is equal to $Ass2tox$, or, in more general terms, that, substitution is a syntactic operator. This is the case in both [8] and [2].

While this decision avoids the problem discussed above, we show below that it leads to another problem in Morgan's formalisation of parameters [8, 11]. Again, the problem is illustrated by means of an example. We consider the program that assigns 2 to a variable z using the procedure $Ass2tox$ of (2).

$$|[\, \textbf{proc } Ass2tox \mathrel{\widehat{=}} x := 2 \bullet Ass2tox[\textbf{result } x \backslash z] \,]| \qquad (6)$$

In view of our previous comments about procedures and result substitutions, and assuming syntactic substitution, we can get to an unexpected conclusion: this procedure block is equivalent to $|[\,\textbf{var } l \bullet x := 2 \; ; \; z := l\,]|$, a program that assigns 2 to x and an arbitrary value (that assigned to l upon declaration) to z.

$$\begin{array}{ll}
|[\, \textbf{proc } Ass2tox \mathrel{\widehat{=}} x := 2 \bullet Ass2tox[\textbf{result } x \backslash z] \,]| & \\
= |[\, \textbf{proc } Ass2tox \mathrel{\widehat{=}} x := 2 \bullet |[\,\textbf{var } l \bullet Ass2tox[x \backslash l] \; ; \; z := l\,]| \,]| & [\text{by } (5)] \\
= |[\, \textbf{proc } Ass2tox \mathrel{\widehat{=}} x := 2 \bullet |[\,\textbf{var } l \bullet Ass2tox \; ; \; z := l\,]| \,]| & [\text{by } Ass2tox[x \backslash l] = Ass2tox] \\
= |[\,\textbf{var } l \bullet x := 2 \; ; \; z := l\,]| & [\text{by the copy rule}]
\end{array}$$

It might appear that an immediate solution to this problem is to adopt context dependent rather than syntactic substitution. In this case, $Ass2tox[x \backslash l]$, in the second line of the above derivation, would be replaced by $l := 2$ (rather than by $Ass2tox$), and the overall result of the derivation would be $z := 2$, as expected. Nevertheless, we have just concluded that substitution must be regarded as a syntactic operator, since otherwise we run into the problem raised earlier in this section.

If we apply the copy rule at an earlier stage, before replacing the result substitution by the variable block defined by (5), the resulting program assigns 2 to z as well. The development is shown below.

$$\begin{array}{ll}
|[\, \textbf{proc } Ass2tox \mathrel{\widehat{=}} x := 2 \bullet Ass2tox[\textbf{result } x \backslash z] \,]| & \\
= x := 2[\textbf{result } x \backslash z] & [\text{by the copy rule}] \\
= |[\,\textbf{var } l \bullet (x := 2)[x \backslash l] \; ; \; z := l\,]| & [\text{by } (5)] \\
= |[\,\textbf{var } l \bullet l := 2 \; ; \; z := l\,]| & [\text{by a property of substitution}] \\
= z := 2 & [\text{by laws of assignment and declaration (see [16], for instance)}]
\end{array}$$

In conclusion, the order in which the laws are applied influences the result.

In [11], Morgan uses the strategy illustrated by our second development above. However, the laws that can be derived from the model of procedures and substitutions provided in [8, 11] do not enforce the application of this strategy: the order of application used in the first development is also supported by this model.

Altogether, whatever definition we adopt for the substitution operator, we run into problems. If we assume that $Ass2tox[x \backslash z]$ is $z := 2$, in other words, context dependent substitution, we run into the problem illustrated by our first example. Alternatively, if we assume that $Ass2tox[x \backslash z]$ is $Ass2tox$ (syntactic substitution), the problem posed by our second example comes about. This problem is not specific to result substitutions: similar inconsistencies would arise if we had used value or value-result substitutions.

Back's work does not present this problem if we define that substitution is syntactic. In this approach, the program (6) is written $|[\,\textbf{proc } Ass2tox \mathrel{\widehat{=}} (\textbf{res } x \bullet x := 2) \bullet Ass2tox(z)\,]|$. Since the result of applying a procedure name to an actual parameter cannot be established without investigating the body of the procedure, when reasoning about $Ass2tox(z)$, the only way to reduce it to a variable block is by first applying the copy rule. Consequently, within Back's framework, the unwanted deduction that could be carried out using Morgan's approach cannot arise.

As already explained, Sampaio avoids the unwanted deductions by adopting context dependent substitution and introducing a notion of contiguous scope which is used to restrict the application of the law that renames local variables. If we assume that variables cannot be redeclared, or in other words, if we rule out the possibility of using nested scope, the variables will always have a contiguous scope. In this case, the usual law that renames local variables can be applied without further constraints. The restriction over variable declarations, however, is generally too severe. Moreover, as Sampaio's formalisation of procedures and parameters is essentially the same as that of Back, he could have defined substitution as a syntactic operator, and then avoided the restriction imposed on the renaming law.

A negative aspect of Back's work is the introduction of an additional construct: the parametrised statement. Refinement has to be defined for these statements as well, and properties of the new refinement relation have to be proved before we can proceed with the derivation of laws. As a matter of fact, [2] does not present any refinement law, but only rules to prove the correctness of (recursive) procedures. In what follows, we show how we can derive refinement laws that correspond to those in Morgan's calculus and formalise his approach to the development of recursive (parametrised) procedures using Back's formalism.

4 Semantics

The forms of parametrised statement that we consider correspond to call-by-value, call-by-result, and call-by-value-result, instead of call-by-reference as in [2]. The semantics of these statements, or more precisely, of the programs obtained by applying them to lists of actual parameters is not surprising. The definition of call-by-value is as shown below.

Call-by-value $(\mathbf{val}\ vl \bullet p)(el) = [\![\, \mathbf{var}\ l \bullet l := el\ ;\ p[vl \backslash l]\ ]\!]$ provided the variables of l are fresh: not free in p or el, and not in vl.

In this definition el is a metavariable that ranges over lists of expressions.

The variable block that corresponds to a call-by-result has already been defined in Section 3 by equation (4). For call-by-value-result, we have the definition that follows.

Call-by-value-result $(\mathbf{val\text{-}res}\ vl_1 \bullet p)(vl_2) = [\![\, \mathbf{var}\ l \bullet l := vl_2\ ;\ p[vl_1 \backslash l]\ ;\ vl_2 := l\,]\!]$ provided the variables of l are not free in p, and are not in vl_1 or vl_2.

We assume that in the case of a call-by-result or a call-by-value-result, the list of actual parameters is a duplicate-free list of variables.

Parametrised statements that combine different forms of parameter transmission are defined below by composition. The metavariables **par** and *fpd* used in this definition range over mechanisms of parameter passing and formal parameter declarations (either ordinary declarations or declarations that combine different forms of parameter transmission), respectively.

Multiple parametrisation mechanisms

$$(\mathbf{par}\ vl_1;\ fpd \bullet p)(el_1, el_2) = (\mathbf{par}\ vl_1 \bullet (fpd \bullet p)(el_2))(el_1)$$

provided the variables of vl_1 are not free in el_2.

This definition expresses in a general way the definitions of [2].

A refinement relation between programs, $\sqsubseteq$, is defined in both [2] and [8] in terms of weakest preconditions. We rephrase these definitions below, where ψ is a metavariable that stands for predicates and the application of wp to a program p and a postcondition ψ is denoted $wp.p.\psi$.

Definition 1 *For programs p_1 and p_2, $p_1 \sqsubseteq p_2$ if and only if $wp.p_1.\psi \Rightarrow wp.p_2.\psi$, for all postconditions ψ.*

The function wp determines, when applied to a program p and to a postcondition ψ, the weakest precondition that guarantees that p terminates in a state that satisfies ψ. In other words, wp determines the predicate that identifies all states in which the execution of p will lead to a state that satisfies ψ. Intuitively, $p_1 \sqsubseteq p_2$ holds exactly when p_2 terminates whenever p_1 does, and produces only results that are acceptable to p_1. Therefore, if $p_1 \sqsubseteq p_2$, then p_2 is always satisfactory as a substitute for p_1. A more formal justification of Definition 1 can be found in [3, 9].

For parametrised statements, the definition of refinement that we adopt is that of [2].

Definition 2 *For parametrised statements* $(fpd \bullet p_1)$ *and* $(fpd \bullet p_2)$*, with the same formal parameter declaration,* $(fpd \bullet p_1) \sqsubseteq (fpd \bullet p_2)$ *if and only if, for all lists al of actual parameters,* $(fpd \bullet p_1)(al) \sqsubseteq (fpd \bullet p_2)(al)$*.*

Surprisingly, maybe, $p_1 \sqsubseteq p_2$ is not equivalent to $(fpd \bullet p_1) \sqsubseteq (fpd \bullet p_2)$. As established in [2], parametrised statements are monotonic with respect to $\sqsubseteq$, so that $p_1 \sqsubseteq p_2$ implies $(fpd \bullet p_1) \sqsubseteq (fpd \bullet p_2)$. Nevertheless, there are cases in which $(fpd \bullet p_1) \sqsubseteq (fpd \bullet p_2)$, but $p_1 \sqsubseteq p_2$ does not hold. For instance, by applying the definition of call-by-value, it is not difficult to prove that both $(\mathbf{val}\ n \bullet n := n+1)(m)$ and $(\mathbf{val}\ n := n+2)(m)$ are equivalent to **skip**. Therefore, since m is arbitrary, we deduce that $(\mathbf{val}\ n \bullet n := n+1) \sqsubseteq (\mathbf{val}\ n \bullet n := n+2)$. Nevertheless, $n := n+1$ is not refined by $n := n+2$.

In order to simplify the presentation and derivation of refinement laws, we establish that a parametrised statement may declare no formal parameters. These special parametrised statements are actually programs themselves and we define that, for every program p, $(\bullet\, p) = p$.

The main program of a procedure block and, in the case of a recursive procedure, its body as well, may refer to the procedure name, so they are not conventional programs. From the semantic point of view, they are contexts: functions from programs (or parametrised statements) to programs (or parametrised statements). If c is the main program of a procedure block or the body of a recursive procedure, then it corresponds to the function that, when applied to a program (or parametrised statement) p, yields the result of substituting p for the free occurrences of the procedure name in c, where c is a metavariable ranging over programs and parametrised statements.

The semantics of a procedure block, which we define below, is based on the copy rule of Algol 60 and on least fixed points, as adopted, for example, in [2, 12, 11].

Procedure block $\;[\![\, \mathbf{proc}\ pn \mathrel{\widehat{=}} (fpd \bullet p_1)(pn) \bullet p_2(pn)\,]\!] = p_2(\mu(fpd \bullet p_1))$

Since, as we have already said, programs can be seen as special parametrised statements, this definition contemplates procedure blocks that declare either parametrised or non-parametrised procedures. It states that a procedure block is equivalent to the program obtained by applying its main program to the least fixed point of the procedure body. A more usual notation for the least fixed point of a function $(fpd \bullet p_1)$ is $\mu\, pn \bullet (fpd \bullet p_1)(pn)$, but for the sake of brevity, we will adopt the more concise notation $\mu(fpd \bullet p_1)$.

The existence of $\mu(fpd \bullet p_1)$ has to be justified. According to Knaster-Tarski [17], we can establish that $\mu(fpd \bullet p_1)$ exists by showing that the set of programs and the sets of parametrised statements with the same formal parameter declaration are complete lattices, and that $(fpd \bullet p_1)$ is monotonic. As expected, the partial order of interest is refinement.

In the unified languages of [2, 12] programs are modelled as monotonic predicate transformers, which, as shown in [4], form a complete lattice. The bottom element of the set of parametrised statements with formal parameter declaration fpd is $(fpd \bullet \mathbf{abort})$. Moreover, the least upper bound of a set of parametrised statements $\{i \bullet (fpd \bullet p_i)\}$ can be defined as $(\sqcup\{i \bullet (fpd \bullet p_i)\})(al) = \sqcup\{i \bullet (fpd \bullet pi)(al)\}$, where al is a metavariable that stands for lists of actual parameters. Finally, the program constructors are monotonic with respect to $\sqsubseteq$ [2, 12] and, consequently, $(fpd \bullet p_1)$ is a monotonic function, as required.

In the case where $(fpd \bullet p_1)$ does not contain free occurrences of pn or, in other words, pn is not a recursive procedure, $\mu(fpd \bullet p_1)$ is $(fpd \bullet p_1)$. Therefore, $|[\,\mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1) \bullet p_2(pn)\,]|$ is the program obtained by substituting $(fpd \bullet p_1)$ for the calls to pn in p_2, $p_2(fpd \bullet p_1)$, as expected.

The development of recursive procedures requires the use of variants. Recursion may be used if the refinement of a program (parametrised statement) p leads to another program (parametrised statement) that contains p itself as a component. Due to termination concerns, however, the introduction of a recursive procedure requires the definition of a variant: an integer expression whose value must be decreased by each recursive call, but cannot assume negative values. As a matter of fact, from a theoretical point of view, the type of a variant can be any well-founded set, but in practice it is enough to consider that the variant is an integer bounded below by 0.

These expressions should be declared, as suggested in [11], in a variant block. This novel kind of block is a procedure block that also introduces a variant. The form of a variant block is $|[\,\mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2\,]|$, where n and e range over variant names and integer expressions, respectively. As an example, we consider the development of the very simple program that assigns to the variable x the factorial of y.

In Morgan's refinement calculus, specifications are written using specification statements. These have the form $w : [pre, post]$, where w (the frame) ranges over lists of variables, and pre (the precondition) and $post$ (the postcondition) over predicates. This program can change only the value of the variables in w and, when executed from a state that satisfies pre, terminates in a state that satisfies $post$. The program that assigns the factorial of y to x can be written as $x : [\mathrm{true}, x = y!\,]$.

If, when refining this program, we decide that we want to develop a recursive procedure *Fact* that implements the factorial function, we have to introduce the variant block below.

$$
\begin{array}{l}
|[\,\mathbf{proc}\ \mathit{Fact} \mathrel{\widehat=} (\mathbf{val}\ n : \mathbb{N} \bullet \{N = n\}\ x : [\mathrm{true}, x = n!\,])\ \ \mathbf{variant}\ N\ \mathbf{is}\ n \bullet \\
\qquad x : [\mathrm{true}, x = y!\,] \\
]|
\end{array}
$$

At this point, we can refine the body of *Fact* to obtain an implementation for this procedure and refine the main program to introduce the appropriate procedure call. Actually, this approach to stepwise refinement is valid only if all the program constructors are monotonic with respect to the refinement relation. As remarked above, this is being assumed here, since any sensible approach to program development will satisfy this requirement.

In the body of *Fact*, $\{N = n\}\ x : [\mathrm{true}, x = n!\,]$, N is a logical constant that records the initial value of the variant. Following Morgan [11], we call the program $\{N = n\}$ an assumption: $\{N = n\}\ x : [\mathrm{true}, x = n!\,]$ aborts if $N = n$ does not hold and, otherwise, behaves like $x : [\mathrm{true}, x = n!\,]$. Recursive calls may be introduced only at points where N has been strictly decreased.

More generally, the variant name introduced by a variant block is a logical constant whose scope is restricted to the procedure body. This logical constant is supposed to assume ever decreasing values in the successive recursive procedure calls. Accordingly, the semantics of a variant block is as follows.

Variant block

$$
|[\,\mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1)(pn)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2(pn)\,]| = p_2(\mu(fpd \bullet |[\,\mathbf{con}\ n : \mathbb{Z} \bullet p_1\,]|))
$$

The fact that variant names are logical constants implies that variant blocks are not executable and have to be refined away.

In the next section, we present some refinement laws. There we consider again the program that assigns to x the factorial of y and develop an implementation for it formally.

5 Refinement Laws

In view of the semantics of procedure blocks, variant blocks, and parametrised statements, and of the definition of refinement, we can derive a set of refinement laws to support the development of procedures. In particular, we propose laws that can be used to introduce procedure and variant blocks, and procedure calls. These laws have no counterpart in [11] and support Morgan's technique of procedure development. Furthermore, we present refinement laws that can be used to introduce parametrised statements. These correspond to the laws of [10] that introduce substitutions.

5.1 Procedures and Recursion

The refinement law that can be employed to introduce a procedure block is presented below.

Law 1 Introduce procedure block.

$$\begin{array}{l} p_2 \\ = \\ |[\,\mathbf{proc}\ pn \mathrel{\widehat=} (\mathit{fpd} \bullet p_1) \bullet p_2\,]| \end{array}$$

provided pn is not free in p_2.

This refinement law allows any program p_2 to be transformed into a procedure block whose main program is p_2. Calls to pn can be introduced subsequently in p_2 using the Law 4 presented later on. As mentioned in the previous section, a parametrised statement may have an empty formal parameter declaration. Therefore, Law 1 can also be used to introduce a block that declares a non-parametrised procedure. This also holds for the other laws that we present below.

The introduction of a variant block can be achieved by the following refinement law.

Law 2 Introduce a variant block.

$$\begin{array}{l} p_2 \\ = \\ |[\,\mathbf{proc}\ pn \mathrel{\widehat=} (\mathit{fpd} \bullet \{n = e\}\ p_1)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2\,]| \end{array}$$

provided

- pn and n are not free in e or in p_2;
- n is different from pn.

Recursion can be introduced by refining $(\mathit{fpd} \bullet \{n = e\}\ p_1)$ and subsequently replacing occurrences of $(\mathit{fpd} \bullet \{0 \leq n < e\}\ p_1)$ in the resulting program by procedure calls.

The introduction of procedure calls in the main program of a procedure block that declares a non-recursive procedure can be achieved by applying the very simple refinement law below.

Law 3 Introduce call to a non-recursive procedure in the main program of a procedure block.

$$\begin{array}{l} |[\,\mathbf{proc}\ pn \mathrel{\widehat=} (\mathit{fpd} \bullet p_1) \bullet p_2[(\mathit{fpd} \bullet p_1)]\,]| \\ = \\ |[\,\mathbf{proc}\ pn \mathrel{\widehat=} (\mathit{fpd} \bullet p_1) \bullet p_2[pn]\,]| \end{array}$$

provided pn is not recursive.

We identify an occurrence of a program (parametrised statement) p_1 in a context c by writing $c[p_1]$. Subsequent references to $c[p_2]$ denote the context obtained by substituting p_2 for that particular occurrence of p_1 in c. The program $c[p_2]$ should not be confused with $c(p_2)$. As already explained, the latter, is the result of substituting p_2 for the free occurrences of the procedure name, as opposed to a particular occurrence of p_1, in c.

The Laws 1 – 3 are simple consequences of the definitions of procedure and variant blocks. For conciseness, we do not present a derivation for them here. In the appendix, we derive the Laws 4, 5, 7, and 8. A derivation for Law 6 is presented in the next section as an example.

The introduction of a procedure call in the main program of a variant block can be accomplished by the following law.

Law 4 Introduce call to a procedure in the main program of a variant block.

$$\begin{array}{l} \quad |[\,\mathbf{proc}\ pn \mathrel{\widehat{=}} (\mathit{fpd} \bullet p_1)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2[(\mathit{fpd} \bullet p_3)]\,]| \\ \sqsubseteq \\ \quad |[\,\mathbf{proc}\ pn \mathrel{\widehat{=}} (\mathit{fpd} \bullet p_1)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2[pn]\,]| \end{array}$$

provided

- $\{n = e\}\ p_3 \ \sqsubseteq\ p_1$;
- pn is not recursive;
- n is not free in p_3 and $p_1[pn]$.

Recursive calls can be introduced with the use of Law 5 below.

Law 5 Introduce recursive call.

$$\begin{array}{l} \quad |[\,\mathbf{proc}\ pn \mathrel{\widehat{=}} (\mathit{fpd} \bullet p_1[(\mathit{fpd} \bullet \{0 \le e < n\}\ p_3)])\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2\,]| \\ \sqsubseteq \\ \quad |[\,\mathbf{proc}\ pn \mathrel{\widehat{=}} (\mathit{fpd} \bullet p_1[pn]) \bullet p_2\,]| \end{array}$$

provided

- $\{n = e\}\ p_3 \ \sqsubseteq\ p_1[(\mathit{fpd} \bullet \{0 \le e < n\}\ p_3)]$;
- n is not free in p_3 and $p_1[pn]$.

In the next section, we present an example that illustrates the application of this and other refinement laws we have presented before.

5.2 Parametrised Statements

The Laws 3, 4 and 5 allow the substitution of parametrised procedure names for appropriate parametrised statements. We still need, however, laws that introduce parametrised statements. In this section, we present three such laws, which account for value, result, and value-result parametrised statements. They correspond to the laws of [10] that introduce substitutions that apply to specification statements. The laws of [11] combine an application of these simpler laws with an application of an appropriate law that introduces procedure calls.

The law that introduces a call-by-value is as follows.

Law 6 Call-by-value.

$$\begin{array}{l} \quad w : [pre[vl \backslash el], post[vl \backslash el]\ ] \\ = \\ \quad (\mathbf{val}\ vl \bullet w : [pre, post])(el) \end{array}$$

provided the variables of vl are not in w, and the variables of w are not free in el.

In order to exemplify the derivation of refinement laws, we present a proof for this particular law. As mentioned before, the derivation of some other laws can be found in the appendix. There, we also present the wp definitions used in these derivations.

$$
\begin{array}{ll}
wp.w : [pre[vl \backslash el], post[vl \backslash el]\ ].\psi & \\
= pre[vl \backslash el] \wedge (\forall\, w \bullet post[vl \backslash el] \Rightarrow \psi) & \text{[by definition of } wp] \\
= pre[vl \backslash l][l \backslash el] \wedge (\forall\, w \bullet post[vl \backslash l][l \backslash el] \Rightarrow \psi) & \text{[by the variables of } l \text{ are fresh]} \\
= (pre[vl \backslash l] \wedge (\forall\, w \bullet post[vl \backslash l] \Rightarrow \psi))[l \backslash el] & \\
\qquad \text{[by the variables of } l \text{ are fresh and the variables of } w \text{ are not free in } el] & \\
= \forall\, l \bullet (pre[vl \backslash l] \wedge (\forall\, w \bullet post[vl \backslash l] \Rightarrow \psi))[l \backslash el] & \text{[by predicate calculus]} \\
= \forall\, l \bullet (wp.(w : [pre, post])[vl \backslash l].\psi)[l \backslash el] & \text{[by the variables of } vl \text{ are not in } w] \\
= wp.\ |[\ \mathbf{var}\ l \bullet l := el\ ;\ (w : [pre, post])[vl \backslash l]\ \ ]|\ .\psi & \text{[by definition of } wp] \\
= wp.(\mathbf{val}\ vl \bullet w : [pre, post])(el).\psi & \text{[by definition]}
\end{array}
$$

In the general case, implication rather than equivalence between weakest preconditions is established. In other words, refinement rather than equality between particular forms of programs is established.

The introduction of a call-by-result can be achieved with the refinement law below.

Law 7 Call-by-result.

$$
\begin{array}{l}
w, vl_2 : [pre, post] \\
= \\
(\mathbf{res}\ vl_1 \bullet w, vl_1 : [pre, post[vl_2 \backslash vl_1]\ ])(vl_2)
\end{array}
$$

provided the variables of vl_1 are not in w, and are not free in pre or $post$.

For a call-by-value-result we have the following law.

Law 8 Call-by-value-result.

$$
\begin{array}{l}
w, vl_2 : [pre[vl_1 \backslash vl_2], post] \\
= \\
(\mathbf{val\text{-}res}\ vl_1 \bullet w, vl_1 : [pre, post[vl_2 \backslash vl_1]\ ])(vl_2)
\end{array}
$$

provided the variables of vl_1 are not in w and are not free in $post$.

Morgan [10] also presents laws that transform assignments into substitutions. Since assignments can be written as specification statements, we do not present the corresponding laws here.

As an example, we refine the program $x : [\mathrm{true}, x = y!\,]$. We implement by recursion a procedure *Fact* that assigns to x the factorial of its value parameter. The first step in our development is the introduction of the variant block that declares *Fact*.

$$
\begin{array}{lll}
 & x : [\mathrm{true}, x = y!\,] & \\
= & \text{“by Law 2”} & \\
 & |[\ \mathbf{proc}\ \mathit{Fact} \mathrel{\widehat{=}} (\mathbf{val}\ n \bullet \{N = n\}\ x : [\mathrm{true}, x = n!\,])\ \mathbf{variant}\ N\ \mathbf{is}\ n \bullet & \\
 & \qquad x : [\mathrm{true}, x = y!\,] & \triangleleft \\
 &]| &
\end{array}
$$

The symbol $\triangleleft$ on the right margin indicates the program that is refined subsequently: we have arbitrarily decided to refine the main program, instead of the procedure body, first. Obviously, we want to implement this program by a call to *Fact*. With this purpose we introduce an appropriate parametrised statement.

$$
\begin{array}{ll}
= & \text{“by Law 6”} \\
 & (\mathbf{val}\ n \bullet x : [\mathrm{true}, x = n!\,])(y)
\end{array}
$$

$$
\begin{array}{l}
|[\ \textbf{proc}\ \mathit{Fact} \mathrel{\widehat{=}} (\textbf{val}\ n \bullet\ \ \textbf{if}\ n = 0 \rightarrow x := 1 \\
\qquad\qquad\qquad\qquad\qquad\quad [\!]\ n > 0 \rightarrow \mathit{Fact}(n-1)\ ;\ \ x := x \times n \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{fi}) \bullet \\
\qquad \mathit{Fact}(y) \\
]|
\end{array}
$$

Figure 1: Collected code for the factorial program

At this point we can introduce the procedure call.

$$
\begin{array}{ll}
 & |[\ \textbf{proc}\ \mathit{Fact} \mathrel{\widehat{=}} (\textbf{val}\ n \bullet \{N = n\}\ x : [\mathrm{true}, x = n!\,])\ \textbf{variant}\ N\ \textbf{is}\ n \bullet \\
 & \qquad (\textbf{val}\ n \bullet x : [\mathrm{true}, x = n!\,])(y) \\
 &]| \\
\sqsubseteq & \text{“by Law 4”} \\
 & |[\ \textbf{proc}\ \mathit{Fact} \mathrel{\widehat{=}} (\textbf{val}\ n \bullet \{N = n\}\ x : [\mathrm{true}, x = n!\,])\ \textbf{variant}\ N\ \textbf{is}\ n \bullet \mathit{Fact}(y)\,]|
\end{array}
$$

The proof-obligation that is generated by this application of Law 4 amounts to showing that $\{N = n\}\ x : [\mathrm{true}, x = n!\,] \sqsubseteq \{N = n\}\ x : [\mathrm{true}, x = n!\,]$, which is trivial since the refinement relation is reflexive.

It is not difficult to verify, using the laws of [11], for instance, that the program in the procedure body can be refined to

$$
\begin{array}{l}
\textbf{if}\ n = 0 \rightarrow x := 1 \\
[\!]\ n > 0 \rightarrow x : [n > 0 \wedge N = n, x = (n-1)!\,]\ ; \\
\qquad\qquad x := x \times n \\
\textbf{fi}
\end{array}
\qquad \triangleleft
$$

The remaining specification statement in this program assigns to x the factorial of $n-1$. As we show below, it can be implemented by a recursive call. First we introduce a parametrised statement using Law 6 again.

$$
\begin{array}{ll}
= & \text{“by Law 6”} \\
 & (\textbf{val}\ n \bullet x : [n + 1 > 0 \wedge N = n + 1, x = n!\,])(n-1)
\end{array}
$$

The body of this parametrised statement can be refined to the following program.

$$\{0 \leq n < N\}\ x : [\mathrm{true}, x = n!\,]$$

Now we can apply Law 5 to introduce the recursive call to *Fact*. The resulting procedure block is presented in Figure 1. The proof-obligation that is generated is shown below.

$$
\begin{array}{ll}
 & \{N = n\}\ x : [\mathrm{true}, x = n!\,] \\
\sqsubseteq & \textbf{if}\ n = 0 \rightarrow x := 1 \\
 & [\!]\ n > 0 \rightarrow (\textbf{val}\ n \bullet \{0 \leq n < N\}\ x : [\mathrm{true}, x = n!\,])(n-1)\ ;\ \ x := x \times n \\
 & \textbf{fi}
\end{array}
$$

This is exactly the result that we have obtained when developing the body of *Fact*. Therefore, we do not need to provide any additional justification to discharge this proof-obligation.

In general, if we apply Laws 4 and 5 in the way illustrated above, with p_3 as the program in the original specification of the procedure, the discharge of the proof-obligations generated is trivial. This strategy of refinement produces developments that follow Morgan's approach to recursion.

6 Conclusions

We have examined a subtle interaction between substitution, procedures and parameters. Of particular importance is the definition of the substitution operator when applied to a procedure name. Two alternatives have been analysed: one of them specifies that the procedure name itself is taken into account (syntactic substitution); the other one establishes that the substitution operates on the procedure body (context dependent substitution). Unfortunately, whichever option is chosen, Morgan's approach to procedures and parameters is found to be inconsistent.

To our knowledge, the subtle interaction between procedures, parameters, and substitution discussed in this paper was originally pointed out in [16]. Sampaio's idea of restricting the application of the renaming law can be considered as a solution to the problems found in Morgan's approach, but the restriction turns out to be too severe in practice. Also, Sampaio presented no mathematical model to justify the restricted version of the renaming law.

Back's formalisation of procedures and parameters involves a greater number of definitions and theorems. However, it does not present any of the complications we have uncovered in Morgan's work and imposes no restriction as in the solution proposed by Sampaio.

The problem with Morgan's approach seems to be a consequence of an unfortunate design decision: formal parameters are not regarded as local variables in the procedure body. This decision was perhaps an attempt to avoid parametrised statements, as suggested by Back. Nevertheless, even though parametrised statements do indeed increase the complexity of the formalism, Back's approach seems to be the right direction to follow.

In recognition of the advantages of program calculation, we have proposed and proved refinement laws that correspond to those of Morgan's calculus. Our laws, however, are expressed using Back's formalism. Furthermore, we have proposed additional refinement laws that formalise Morgan's approach to procedures development, including his use of variants and variant declarations.

Another analysis of the usage of procedures in the refinement calculus is presented in [6]. This study, however, concentrates on the methodological aspects of the development of procedures. In [6], the suitability of the refinement laws presented in [11] is discussed and an alternative strategy of program refinement, where (non-recursive) procedures are introduced in the final phase of development, is suggested.

Acknowledgements

The authors are indebted to Steve King and Carroll Morgan for their comments on drafts of this paper. We are also grateful to the anonymous referee who has pointed out the benefits of considering programs as special parametrised statements and has made several comments that helped us to improve the presentation of our paper. Paul Gardiner and Bernard Sufrin have helped us to define the semantics of variant blocks properly. The work of Ana Cavalcanti and Augusto Sampaio are financially supported by CNPq, Brazil, grants 204.527/90-2 and 521.039/95-9, respectively.

References

[1] R. J. R. Back. *On The Correctness of Refinement Steps in Program Development.* PhD thesis, Department of Computer Science, University of Helsinki, 1978. Report A-1978-4.

[2] R. J. R. Back. Procedural Abstraction in the Refinement Calculus. Technical report, Department of Computer Science, Åbo - Finland, 1987. Ser. A No. 55.

[3] R. J. R. Back. A Calculus of Refinements for Program Derivations. *Acta Informatica*, 25:593 – 624, 1988.

[4] R. J. R. Back and J. Wright. Duality in Specification Languages: A Lattice-theoretical Approach. *Acta Informatica*, 27(7):583 – 625, 1990.

[5] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[6] L. Groves. Procedures in the Refinement Calculus: A New Approach? In H. Jifeng, editor, *7th Refinement Workshop*, Bath - UK, July 1996.

[7] S. King and C. Morgan. Exits in the Refinement Calculus. *Formal Aspects of Computing*, 7(1):54 – 76, 1995.

[8] C. C. Morgan. Procedures, parameters, and abstraction: Separate concerns. *Science of Computer Programming*, 11:17 – 27, 1988.

[9] C. C. Morgan. The Specification Statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403 – 419, 1988.

[10] C. C. Morgan. *Programming from Specifications.* Prentice-Hall, 1990.

[11] C. C. Morgan. *Programming from Specifications.* Prentice-Hall, 2nd edition, 1994.

[12] C. C. Morgan, K. Robinson, and P. H. B. Gardiner. On the Refinement Calculus. Technical Monograph TM-PRG-70, Oxford University Computing Laboratory, Oxford - UK, October 1988.

[13] J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287 – 306, 1987.

[14] J. M. Morris. Invariance Theorems for Recursive Procedures. Technical report, Department of Computer Science, University of Glasgow, 1988.

[15] J. M. Morris. Laws of Data Refinement. *Acta Informatica*, 26:287 – 308, 1989.

[16] A. Sampaio. *An Algebraic Approach to Compiler Design.* PhD thesis, Oxford University Computing Laboratory, Oxford - UK, 1993. Technical Monograph TM-PRG-110. Revised version to appear as volume 4 of AMAST (Algebraic Methodology and Software Technology) Series in Computing, World Scientific, 1997 (in press).

[17] A. Tarski. A Lattice Theoretical Fixed Point Theorem and its Applications. *Pacific Journal of Mathematics*, 5, 1955.

A Law Derivations

The law derivation that we have presented in Section 5.2 and the law derivations that we present below rely on the weakest precondition semantics of specification statements, assignments, sequential compositions, and variables blocks. For completeness, we present the relevant definitions here.

$$
\begin{array}{l}
wp.w : [pre, post].\psi = pre \wedge (\forall\, w \bullet post \Rightarrow \psi) \\
wp.vl := el.\psi = \psi[vl \backslash el] \\
wp.(p_1 \;\; ; \;\; p_2).\psi = wp.p_1.(wp.p_2.\psi) \\
wp.\; [\![\, \mathbf{var}\ vl \bullet p \,]\!]\; .\psi = (\forall\, vl \bullet wp.p.\psi) \text{ provided the variables of } vl \text{ are not free in } \psi
\end{array}
$$

These definitions are standard and can also be found in [12, 3], for instance.

Derivation of Law 4

$$
\begin{array}{ll}
|[\ \mathbf{proc}\ pn \mathrel{\widehat=} (\mathit{fpd} \bullet p_1)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2[(\mathit{fpd} \bullet p_3)]\ ]| & \\
= p_2[(\mathit{fpd} \bullet p_3)](\mu(\mathit{fpd} \bullet |[\ \mathbf{con}\ n : \mathbb{Z} \bullet p_1\ ]|)) & \text{[by definition]} \\
= p_2[(\mathit{fpd} \bullet |[\ \mathbf{con}\ n : \mathbb{Z} \bullet \{n = e\}\ p_3\ ]|)](\mu(\mathit{fpd} \bullet |[\ \mathbf{con}\ n : \mathbb{Z} \bullet p_1\ ]|)) & \\
 & \text{[by } n \text{ is not free in } e \text{ and } p_3\text{]} \\
= p_2[(\mathit{fpd} \bullet |[\ \mathbf{con}\ n : \mathbb{Z} \bullet p_1\ ]|)](\mu(\mathit{fpd} \bullet |[\ \mathbf{con}\ n : \mathbb{Z} \bullet p_1\ ]|)) & \text{[by the proviso]} \\
= p_2[(\mathit{fpd} \bullet |[\ \mathbf{con}\ n : \mathbb{Z} \bullet p_1\ ]|)](\mathit{fpd} \bullet |[\ \mathbf{con}\ n : \mathbb{Z} \bullet p_1\ ]|) & \text{[by } pn \text{ is not free in } p_1\text{]} \\
= p_2[pn](\mathit{fpd} \bullet |[\ \mathbf{con}\ n : \mathbb{Z} \bullet p_1\ ]|) & \text{[by a property of substitution]} \\
= p_2[pn](\mu(\mathit{fpd} \bullet |[\ \mathbf{con}\ n : \mathbb{Z} \bullet p_1\ ]|)) & \text{[by } pn \text{ is not free in } p_1\text{]} \\
= |[\ \mathbf{proc}\ pn \mathrel{\widehat=} (\mathit{fpd} \bullet p_1)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2[pn]\ ]| & \text{[by definition]}
\end{array}
$$

□

When deriving Law 5, we rely on Lemma 2, which we present below. In [7, p.73], we can find a more restricted version of this lemma, where just programs (parametrised statements with empty formal parameter declarations) are considered. Since [7] outlines a proof for this special case, for the sake of brevity, we consider just parametrised statements with ordinary (non-empty) formal parameter declarations in the proof of Lemma 2. This proof relies on the following additional lemma.

Lemma 1 *Let a family of programs p_i be such that, for any i, $p_i \sqsubseteq c(\mathit{fpd} \bullet \sqcup\{j \mid j < i \bullet p_j\})$, for a non-empty formal parameter declaration fpd, and monotonic c. Then $p_i \sqsubseteq c(\mu(\mathit{fpd} \bullet c))$, for all i.*

Proof By induction:
(Case $i = 0$)

$$
\begin{array}{ll}
p_0 & \\
\sqsubseteq c(\mathit{fpd} \bullet \sqcup\{j \mid j < 0 \bullet p_j\}) & \text{[by assumption]} \\
= c(\mathit{fpd} \bullet \sqcup\varnothing) & \text{[by a property of numbers]} \\
= c(\mathit{fpd} \bullet \mathbf{abort}) & \text{[by } \mathbf{abort} \text{ is the least refined program]} \\
\sqsubseteq c(\mu(\mathit{fpd} \bullet c)) & \text{[by } (\mathit{fpd} \bullet \mathbf{abort}) \text{ is the least refined parametrised statement]}
\end{array}
$$

(Case $i > 0$)

$$
\begin{array}{ll}
p_i & \\
\sqsubseteq c(\mathit{fpd} \bullet \sqcup\{j \mid j < i \bullet p_j\}) & \text{[by assumption]} \\
\sqsubseteq c(\mathit{fpd} \bullet \sqcup\{j \mid j < i \bullet c(\mu(\mathit{fpd} \bullet c))\}) & \text{[by induction hypothesis]} \\
= c(\mathit{fpd} \bullet c(\mu(\mathit{fpd} \bullet c))) & \text{[by a property of least upper bounds]} \\
= c((\mathit{fpd} \bullet c)(\mu(\mathit{fpd} \bullet c))) & \text{[by a property of contexts]} \\
= c(\mu(\mathit{fpd} \bullet c)) & \text{[by a property of fixed points]}
\end{array}
$$

□

Lemma 2 *If, for an integer constant n, an integer expression e, a formal parameter declaration fpd, and a program p, we have that $\{n = e\}\ p \sqsubseteq c(\mathit{fpd} \bullet \{0 \le e < n\}\ p)$, then we can deduce that $(\mathit{fpd} \bullet p) \sqsubseteq \mu(\mathit{fpd} \bullet c)$, provided c is a monotonic context, and n is not free in p and c.*

Proof The assumption can be written as $\{n = e\}p \sqsubseteq c(\mathit{fpd} \bullet \sqcup\{j \mid j < n \bullet \{j = e\}p)$, as we show below.

$$
\begin{array}{ll}
wp.\{0 \leq e < n\}\ p.\psi & \\
= wp.\{\bigvee\{j \mid j < n \bullet j = e\}\}\ p.\psi & \text{[by predicate calculus]} \\
= \bigvee\{j \mid j < n \bullet j = e\} \wedge wp.p.\psi & \text{[by definition of } wp\text{]} \\
= \bigvee\{j \mid j < n \bullet j = e \wedge wp.p.\psi\} & \text{[by predicate calculus]} \\
= \bigvee\{j \mid j < n \bullet wp.\{j = e\}\ p.\psi\} & \text{[by definition of } wp\text{]} \\
= wp. \sqcup \{j \mid j < n \bullet \{j = e\}\ p\}.\psi & \text{[by a property of } \sqcup\text{]}
\end{array}
$$

So, by Lemma 1, $\{n = e\}\ p \sqsubseteq c(\mu(\mathit{fpd} \bullet c))$, for all n. Consequently, as we prove below, $p \sqsubseteq c(\mu(\mathit{fpd} \bullet c))$.

$$
\begin{array}{ll}
wp.p.\psi & \\
= (\exists\, n \bullet n = e) \wedge wp.p.\psi & \text{[by predicate calculus]} \\
= \exists\, n \bullet n = e \wedge wp.p.\psi & \text{[by } n \text{ is not free in } wp.p.\psi\text{]} \\
= \exists\, n \bullet wp.\{n = e\}\ p.\psi & \text{[by definition of } wp\text{]} \\
\Rightarrow \exists\, n \bullet wp.c(\mu(\mathit{fpd} \bullet c)).\psi & \text{[by the conclusion above]} \\
= wp.c(\mu(\mathit{fpd} \bullet c)).\psi & \text{[by n is not free in } wp.c(\mu(\mathit{fpd} \bullet c)).\psi\text{]}
\end{array}
$$

Using this result, we can get to the required conclusion as follows.

$$
\begin{array}{ll}
(\mathit{fpd} \bullet p) & \\
\sqsubseteq (\mathit{fpd} \bullet c(\mu(\mathit{fpd} \bullet c))) & \text{[by the result above]} \\
= (\mathit{fpd} \bullet c)(\mu(\mathit{fpd} \bullet c)) & \text{[by a property of contexts]} \\
= \mu(\mathit{fpd} \bullet c) & \text{[by a property of fixed points]}
\end{array}
$$

$\square$

The derivation of Law 5 also uses the lemma that follows.

Lemma 3 *For any program context* $c[pn]$, $\mu\, c[\mu\, c[pn]] \sqsubseteq \mu\, c[pn]$.

Proof $\mu\, c[pn]$ is a fixed point of $c[\mu\, c[pn]]$

$$
\begin{array}{ll}
c[\mu\, c[pn]](\mu\, c[pn]) & \\
= c[pn](\mu\, c[pn]) & \text{[by a property of substitution]} \\
= \mu\, c[pn] & \text{[by a property of fixed points]}
\end{array}
$$

Therefore, $\mu\, c[\mu\, c[pn]] \sqsubseteq \mu\, c[pn]$, as required.

$\square$

Derivation of Law 5

$$
\begin{array}{ll}
|[\ \mathbf{proc}\ pn \mathrel{\widehat{=}} (\mathit{fpd} \bullet p_1[(\mathit{fpd} \bullet \{0 \leq e < n\}\ p_3)])\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2\ ]| & \\
= p_2(\mu(\mathit{fpd} \bullet |[\ \mathbf{con}\ n : \mathbb{Z} \bullet p_1[(\mathit{fpd} \bullet \{0 \leq e < n\}\ p_3)]\ ]|)) & \text{[by definition]} \\
\sqsubseteq p_2(\mu(\mathit{fpd} \bullet |[\ \mathbf{con}\ n : \mathbb{Z} \bullet p_1[\mu(\mathit{fpd} \bullet p_1[pn])]\ ]|)) & \text{[by Lemma 2 and the proviso]} \\
\sqsubseteq p_2(\mu(\mathit{fpd} \bullet p_1[\mu(\mathit{fpd} \bullet p_1[pn])])) & \text{[by } n \text{ is not free in } p_1\text{]} \\
\sqsubseteq p_2(\mu(\mathit{fpd} \bullet p_1[pn])) & \text{[by Lemma 3]} \\
= |[\ \mathbf{proc}\ pn \mathrel{\widehat{=}} (\mathit{fpd} \bullet p_1[pn]) \bullet p_2\ ]| & \text{[by definition]}
\end{array}
$$

$\square$

Derivation of Law 8

$$
\begin{aligned}
& wp.w, vl_2 : [pre, post].\psi \\
&= pre \wedge (\forall\, w, vl_2 \bullet post \Rightarrow \psi) && \text{[by definition of } wp\text{]} \\
&= pre \wedge (\forall\, w, l \bullet post[vl_2\backslash] \Rightarrow \psi[vl_2\backslash l]) && \text{[by the variables of } l \text{ are fresh]} \\
&= \forall\, l \bullet pre \wedge (\forall\, w, l \bullet post[vl_2\backslash l] \Rightarrow \psi[vl_2\backslash l]) && \text{[by the variables of } l \text{ are fresh]} \\
&= \forall\, l \bullet pre[vl_1\backslash l] \wedge (\forall\, w, l \bullet post[vl_2\backslash vl_1][vl_1\backslash l] \Rightarrow \psi[vl_2\backslash l]) \\
& && \text{[by the variables of } vl_1 \text{ are not free in } pre \text{ or } post\text{]} \\
&= \forall\, l \bullet wp.(w, vl_1 : [pre, post[vl_2\backslash vl_1]\ ])[vl_1\backslash l].\psi[vl_2\backslash l] && \text{[by the variables of } vl_1 \text{ are not in } w\text{]} \\
&= wp.\ |[\ \mathbf{var}\ l \bullet (w, vl_1 : [pre, post[vl_2\backslash vl_1]\ ])[vl_1\backslash l]\ ;\ \ vl_2 := l\ ]|\ .\psi && \text{[by definition of } wp\text{]} \\
&= wp.(\mathbf{res}\ vl_1 \bullet w, vl_1 : [pre, post[vl_2\backslash vl_1]\ ])(vl_2).\psi && \text{[by definition]}
\end{aligned}
$$

□

Derivation of Law 9

$$
\begin{aligned}
& wp.w, vl_2 : [pre[vl_1\backslash vl_2], post].\psi \\
&= pre[vl_1\backslash vl_2] \wedge (\forall\, w, vl_2 \bullet post \Rightarrow \psi) && \text{[by definition of } wp\text{]} \\
&= pre[vl_1\backslash l][l\backslash vl_2] \wedge (\forall\, w, l \bullet post[vl_2\backslash l] \Rightarrow \psi[vl_2\backslash l]) && \text{[by the variables of } l \text{ are fresh]} \\
&= (pre[vl_1\backslash l] \wedge (\forall\, w, l \bullet post[vl_2\backslash l] \Rightarrow \psi[vl_2\backslash l]))[l\backslash vl_2] && \text{[by the variables of } l \text{ are fresh]} \\
&= (pre[vl_1\backslash l] \wedge (\forall\, w, l \bullet post[vl_2\backslash vl_1][vl_1\backslash l] \Rightarrow \psi[vl_2\backslash l]))[l\backslash vl_2] \\
& && \text{[by the variables of } vl_1 \text{ are not free in } post\text{]} \\
&= \forall\, l \bullet (pre[vl_1\backslash l] \wedge (\forall\, w, l \bullet post[vl_2\backslash vl_1][vl_1\backslash l] \Rightarrow \psi[vl_2\backslash l]))[l\backslash vl_2] && \text{[by predicate calculus]} \\
&= \forall\, l \bullet (wp.(w, vl_1 : [pre, post[vl_2\backslash vl_1]\ ])[vl_1\backslash l].\psi[vl_2\backslash l])[l\backslash vl_2] \\
& && \text{[by the variables of } vl_1 \text{ are not in } w\text{]} \\
&= wp.\ |[\ \mathbf{var}\ l \bullet l := vl_2\ ;\ \ (w, vl_1 : [pre, post[vl_2\backslash vl_1]\ ])[vl_1\backslash l]\ ;\ \ vl_2 := l\ ]|\ .\psi \\
& && \text{[by definition of } wp\text{]} \\
&= wp.(\mathbf{val\text{-}res}\ vl_1 \bullet w, vl_1 : [pre, post[vl_2\backslash vl_1]\ ])(vl_2).\psi && \text{[by definition]}
\end{aligned}
$$

□