ELSEVIER

Science of Computer Programming 33 (1999) 87–96

Science of Computer Programming

An inconsistency in procedures, parameters, and substitution in the refinement calculus

Ana Cavalcanti[a], Augusto Sampaio[b], Jim Woodcock[a]

[a] *Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK*
[b] *Departamento de Informática/UFPE, Caixa Postal 7851, 50732-970 Recife PE, Brazil*

Communicated by R. Bird; received 25 September 1996; received in revised form 5 March 1997

Abstract

Morgan and Back have proposed different formalisations of procedures and parameters in the context of techniques of program development based on refinement. In this paper, we investigate a surprising and intricate relationship between these works and the substitution operator that renames the free variables of a program. In this study, we reveal an inconsistency in Morgan's refinement calculus and show that Back's formalisation of procedures does not have the same problem.

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 [10, 9], and Morris [11, 13] 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.

0167-6423/99/$ – see front matter
PII: S0167-6423(97)00015-4

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 [9], the development process consists of calculating programs by repeatedly applying these laws until an adequate result is obtained.

Back [2], Morgan [7], and Morris [12] 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. 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 proposed by Back and Morgan. Section 3 explains their relationship to the substitution operator and the problem in Morgan's approach. In Section 4 we summarise our results. Finally, Appendix A shows how the refinement laws of Morgan's calculus can be expressed using Back's formalism.

2. Procedures and parameters

In [2, 9], procedures are declared in a block. In order to illustrate the particular notation we employ to write procedure blocks, we consider the example below.

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

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 *Inc*; *Inc* is the main program (the scope of the procedure).

Both [2] and [7] 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.

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

The program $|[\ \mathbf{var}\ x \bullet Ass2tox\ ]|$ is a variable block: it declares a variable x local to its body, *Ass2tox*. Since there is a variable x free in the body of *Ass2tox*, we cannot, as in the previous example, apply the copy rule and substitute $x := 2$ for *Ass2tox*. This substitution would lead to the capture of the global variable x mentioned in the body of *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 using parametrised statements. They can have the form (**val** $vl \bullet p$), (**res** $vl \bullet p$), or (**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 example, 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 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*.

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

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

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

Morgan [7] 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 [9]. 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 [14] 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 [14], 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 [14], 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, 10, 9]. 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 [14]. 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 [7] 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 [7, 9]. 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).

$$|[\,\mathbf{proc}\ Ass2tox \mathrel{\widehat{=}} x := 2 \bullet Ass2tox[\mathbf{result}\ x\backslash z]\,]| \tag{6}$$

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

$$\begin{aligned} &|[\,\mathbf{proc}\ Ass2tox \mathrel{\widehat{=}} x := 2 \bullet Ass2tox[\mathbf{result}\ x\backslash z]\,]| \\ &\quad = |[\,\mathbf{proc}\ Ass2tox \mathrel{\widehat{=}} x := 2 \bullet |[\,\mathbf{var}\ l \bullet Ass2tox[x\backslash l];\ z := l\,]| \qquad \text{[by (5)]} \end{aligned}$$

$$= |[\ \textbf{proc}\ Ass2tox \mathrel{\widehat{=}} x := 2 \bullet |[\ \textbf{var}\ l \bullet Ass2tox;\ z := l\ ]|\ ]|$$

[by $Ass2tox[x\backslash l] = Ass2tox$]

$$= |[\ \textbf{var}\ l \bullet x := 2;\ z := l\ ]|$$ [by the copy rule]

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.

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

$$= x := 2[\textbf{result}\ x\backslash z]$$ [by the copy rule]

$$= |[\ \textbf{var}\ l \bullet (x := 2)[x\backslash l];\ z := l\ ]|$$ [by (5)]

$$= |[\ \textbf{var}\ l \bullet l := 2;\ z := l\ ]|$$ [by a property of substitution]

$$= z := 2$$ [by laws of assignment and declaration (see [14], for instance)]

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

In [9], 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 [7, 9] 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 problems if we define that the substitution operator 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 [14] 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. As we have shown above, Back's approach presents no problems whatsoever if syntactic substitution is adopted.

A negative aspect of Back's work is the introduction of an additional construct: the parametrised statement. In this formalism, an extra refinement relation between parametrised statements has to be defined and its properties explored before strategies of procedure development are proposed. Moreover, Back's work is not as appealing to practising programmers as Morgan's calculus, since Back does not propose refinement laws, but only rules to prove the correctness of (recursive) procedures. 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 the corresponding proof-obligations. Moreover, refinement laws provide guidance on the construction of programs.

Fortunately, Morgan's approach to the development of procedures can be formalised using Back's work. In Appendix A we present three refinement laws that correspond to the main laws of [8] concerned with procedures (substitutions). The laws of [9] combine an application of these simpler laws with the introduction of a procedure call. In [4], we use Back's formalism as a consistent model to derive the three laws in the appendix and others that support the approach to the development of recursive (parametrised) procedures proposed by Morgan in [9].

4. 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 [14]. 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.

Another analysis of the usage of procedures in the refinement calculus is presented in [6]. This study, however, concentrates on the methodological (rather than on the semantical) aspects of the development of procedures. In [6], the suitability of the refinement laws presented in [9] 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. 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.

Appendix A. Refinement laws

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*.

In what follows, we present three laws that can be used to transform specification statements into parametrised statements or, more precisely, applications of parametrised statements to actual parameters. As already mentioned, these laws correspond to laws of [8], namely, those that introduce substitutions that apply to specification statements.

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

Law 1. Call-by-value.

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

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

As we show in [4], this law and the other two that follow can be derived from the *wp* semantics of specification statements, assignments, sequential compositions, and variable blocks [10, 3], and from the definitions of parametrised statements in terms of variable blocks.

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

Law 2. Call-by-result.

$$w, vl_2 : [pre, post]$$

$$=$$

$$(\textbf{res}\ vl_1 \bullet w, vl_1 : [pre, post[vl_2 \backslash vl_1]])(vl_2)$$

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 3. Call-by-value-result.

$$w, vl_2 : [pre[vl_1 \backslash vl_2], post]$$

$$=$$

$$(\textbf{val-res}\ vl_1 \bullet w, vl_1 : [pre, post[vl_2 \backslash vl_1]])(vl_2)$$

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

Morgan [8] also presents laws that transform assignments into substitutions. Since assignments can be written as specification statements, we do not present the corresponding laws here.

References

[1] R.-J.R. Back, On the correctness of refinement steps in program development, Ph.D. 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 Inform. 25 (1988) 593–624.
[4] A.L.C. Cavalcanti, A.C.A. Sampaio, J.C.P. Woodcock, Procedures, parameters, and substitution in the refinement calculus, Technical Report TR-5-97, Oxford University Computing Laboratory, Oxford, UK, 1997.
[5] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, NJ, 1976.
[6] L. Groves, Procedures in the refinement calculus: a new approach?, in: H. Jifeng (Ed.), 7th Refinement Workshop, Bath, UK, 1996.
[7] C. Morgan, Procedures, parameters, and abstraction: separate concerns, Science of Computer Programming 11 (1) (1988) 17–27.
[8] C.C. Morgan, Programming from Specifications, Prentice-Hall, Englewood Cliffs, NJ, 1990.
[9] C.C. Morgan, Programming from Specifications, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1994.
[10] C.C. Morgan, K. Robinson, P.H.B. Gardiner, On the refinement calculus, Technical Monograph TM-PRG-70, Oxford University Computing Laboratory, Oxford, UK, 1988.

[11] J.M. Morris, A theoretical basis for stepwise refinement and the programming calculus, Science of Computer Programming 9 (3) (1987) 287–306.
[12] J.M. Morris, Invariance theorems for recursive procedures, Technical report, Department of Computer Science, University of Glasgow, 1988.
[13] J.M. Morris, Laws of data refinement, Acta Inform. 26 (1989) 287–308.
[14] A. Sampaio, An algebraic approach to compiler design, D.Phil. Thesis, TM-PRG-110, Oxford University Computing Laboratory, Oxford, UK, 1993. Revised version to appear as Vol. 4 of AMAST (Algebraic Methodology and Software Technology) Series in Computing, World Scientific, Singapore, 1997, in press.