Restricted demonic choice for modular probabilistic programs

Daniel Robinson and Carroll Morgan*

December 29, 2000

Abstract

It is argued that one approach to modularity in programs containing both demonic and probabilistic choice is to allow variations on the former: 'restricted demonic choice', written $\sqcap_L$, is not allowed to use the value of variables named in the set L as it resolves its nondeterminism; ordinary demonic choice $\sqcap$ is then just the special case $\sqcap_{\{\}}$ in which the set of hidden variables is empty. The intention is that when variables L are declared inside a module, demonic choices outside the module are decorated $\sqcap_L$ and then cannot 'see' the values of L.

Our contribution is to explain why such an operator might be necessary in the presence of probability (recalling that ordinary sequential semantics does perfectly well without it), and then to expose — unfortunately! — some of its more unruly properties. We hope by doing so to make progress towards a proper theory of modularity in the presence of both demonic and probabilistic choices.

Keywords: Modularity, probabilistic choice, demonic choice, nondeterminism, logics of programs, probabilistic weakest preconditions, probabilistic semantics.

1 Introduction

Recent advances in sequential program semantics have combined probabilistic [10] and demonic [4] choice into a framework [16] that generalises weakest preconditions; only a very modest extra complexity in reasoning is required. There is also a correspondingly generalised relational model [8].

Many of the standard techniques of program development — for example loop invariants [18, 13] and variants [7, 13] — generalise usefully to the new context.[1] It seems however that the conventional treatment of modularity does not, and that is the topic of this note.

Simple[2] modularity is traditionally interpreted via the *Copy Rule* (of Algol-60): the module structure is first used to determine any necessary renamings of bound variables, and then the modules and their procedures are eliminated by copying their variable declarations and procedure bodies to suitable places in the surrounding program. The resulting simpler (though now unstructured) program gives the meaning of the original.

*Both authors are members of the Programming Research Group at Oxford University; Robinson's graduate research is supported by the EPSRC.

[1] The earlier citations in each case refer to works without demonic nondeterminism.

[2] By *simple* modularity we mean a language like Modula-2 in which modules are program structures rather than data types: they cannot as a whole be assigned or copied.

The interaction of demonic and probabilistic choice is to blame for the difficulty with modularity: their mutual subtlety is well known from concurrent probabilistic calculi [19, 3, 11, 9, 14], and we have discovered that even in the 'simpler' sequential case the problems are no less acute.

Put briefly, the difficulty is that in the usual regime demonic choice $P \sqcap Q$ between programs P and Q is resolved in any way whatever, allowing even the value of variables hidden inside modules to influence the decision of whether to choose P or Q. (That is because once the module structure has been 'copied away', all variables are treated equally as far as demonic choice is concerned.) Such influences however are difficult even to discuss in 'normal' sequential semantics because they cannot be detected by running a program only once, and such semantics [4] distinguish programs only if a *single* run (of each) can reveal different behaviour.[3] When probability is present, one is necessarily considering several runs — since otherwise probability distributions over outcomes would have no meaning.

We propose below a restricted form of demonic choice $\underset{L}{\sqcap}$ which 'cannot see' a set of variables L (for 'local') because those variables were hidden inside a module (whose structure was subsequently removed by the Copy Rule). Our first technical contribution will be to show that, under even extremely modest assumptions, the algebra of the new operator is not straightforward; and our second contribution will be to identify two key principles and to sketch very briefly the mathematical models that might result from them.

Our hope is that readers might then be encouraged to contribute to a discussion of which approach, if either, might be the right one.

2 The problem

It is a trivial but important feature of sequential programming that two variable-disjoint program fragments commute, that if fragments P and Q have no free variables in common then

$$P;Q \quad = \quad Q;P \ . \tag{1}$$

That fact is crucial to program modularity — and thus to the practicalities of large program construction — as we now review.

Suppose we have a module M, which exports a procedure P assigning to some global variable[4] g in the surrounding program. Consider first the definition

$$P \quad \widehat{=} \quad g := X \sqcap g := Y \ , \tag{2}$$

in which $\sqcap$ denotes demonic choice. All we can observe in a program using M is that each successive call of P changes g in some unpredictable way.

Now hide a local variable l inside M, and define instead

$$P \quad \widehat{=} \quad g := l; \ \ l := X \sqcap l := Y \ . \tag{3}$$

(We assume l is initialised unpredictably to X or Y.) Although the value of g returned by 'this' call of P was in fact determined in advance by the previous call, in languages with only simple modularity the effect of P has not been altered in any way that can be detected externally by observing its effect on g.

[3] That is why for example Dijkstra-style semantics does not distinguish what is sometimes called 'underspecification' [6] from the usual demonic run-time nondeterminism.

[4] We assume in all our examples that variables take values in the set $\{X, Y\}$ only, for some constant values X and Y.

The conventional method of proving that statement, the equivalence in all contexts of the two versions (call them) M_2 and M_3 of M, is to use data refinement. In particular we use *forward simulation* where one proposes a representation program *rep* that links the two modules; in this case [2, 5] it should have the property

$$P_2; \mathit{rep} \quad \sqsubseteq \quad \mathit{rep}; P_3 \ , \tag{4}$$

where $\sqsubseteq$ is the relation of *program refinement* [17, 12, 1] and the subscripts of P refer to the version of the module in which it is defined. Here we choose $l := X \sqcap l := Y$ for *rep*, whence from (4) we must show

$$\begin{array}{lll} & g := X \sqcap g := Y; & \leftarrow P_2 \\ & l := X \sqcap l := Y & \leftarrow \mathit{rep} \\ \\ \sqsubseteq & l := X \sqcap l := Y; & \leftarrow \mathit{rep} \\ & \left.\begin{array}{l} g := l; \\ l := X \sqcap l := Y \end{array}\right\} & \leftarrow P_3 \end{array}$$

The refinement clearly holds (in fact it is equality), and that means we can replace M_2 by M_3 with no-one externally being the wiser.

Now the same reasoning should apply if we turn from demonic choice $\sqcap$ to probabilistic choice ${}_p\!\oplus$, that chooses its left operand with probability p and its right with probability $1-p$. In particular the two probabilistic modules M_5, M_6 containing respectively

$$P_5 \quad \widehat{=} \quad g := X \ {}_{\frac{1}{2}}\!\oplus\, g := Y \tag{5}$$

$$P_6 \quad \widehat{=} \quad g := l; \ \ l := X \ {}_{\frac{1}{2}}\!\oplus\, l := Y \tag{6}$$

should not be distinguishable by an external observer of g: either way it is assigned X or Y with equal probability. The fact that in M_6 it is chosen in advance is irrelevant — the local variable l is as inaccessible as before.

Yet modules M_5 and M_6 are not equal under the standard 'all-seeing' definition of demonic choice — once the Copy Rule has been applied — and that is the problem. The following thought-experiment shows why: consider the program fragment

$$P; \quad h := X \sqcap h := Y; \quad P \ ,$$

in which h is a second global variable. With what probability does the fragment establish $g = h$ on termination?

For M_5 we must assume that no matter how h is assigned, the second call of P_5 could with probability $1/2$ choose the opposite value for g. (The first call of P_5 is irrelevant in this case.) Thus the probability that $g = h$ finally is no better than $1-1/2 = 1/2$.

For M_6 however it is the demon assigning to h that strives to avoid $g = h$, for the eventual value of g has already been decided by the *first* call of P_6. But once that call is replaced by its text, the h-demon can 'see' the future value of g, in l, and can avoid it every time: the probability that $g = h$ holds finally is no better than 0.

Thus modules M_5 and M_6 are different — but we want them to be equal, for otherwise there is little hope for modular development in the presence of both demonic and probabilistic nondeterminism. What has gone wrong?

In conventional data refinement it is assumed that statements in the surrounding program not referring to the module — the great bulk of them, therefore — can be left untouched if the

module is changed internally. In technical terms that means they are trivially data-refined to themselves, and in the case of our thought-experiment it is $h := X \sqcap h := Y$ that should be data-refined to itself. Referring to (4) we see that in our demonic modules that requires only that

$$\begin{array}{l} h := X \sqcap h := Y\,; \\ l := X \sqcap l := Y \end{array} \quad = \quad \begin{array}{l} l := X \sqcap l := Y; \\ h := X \sqcap h := Y\ , \end{array}$$

which is indeed an instance of the principle (1). But in the probabilistic modules we are requiring instead that

$$\begin{array}{l} h := X \sqcap h := Y\,; \\ l := X \underset{\frac{1}{2}}{\oplus} l := Y \end{array} \quad \stackrel{?}{=} \quad \begin{array}{l} l := X \underset{\frac{1}{2}}{\oplus} l := Y; \\ h := X \sqcap h := Y \end{array} \tag{7}$$

which, although again apparently an instance of Principle (1), is not true because on the right-hand side the h-demon can see the value of l whereas on the left-hand side it cannot.

Thus for modularity we must find a definition of $\sqcap$ so that (7) holds after all — or, we might say, so that (1) *continues* to hold even in our more general context. The technical contribution of our paper concerns that question: by the arguments above, we justify the importance of (7); by an algebraic thought-experiment below we show how even that simple requirement can lead to bizarre inconsistencies; and finally we outline some possible mathematical structures that might serve as a model nevertheless.

3 Restricted nondeterminism

To replace '$\stackrel{?}{=}$' by '$=$' in (7), we need a restricted-demonic choice operator which, when making its decision to execute $h := X$ or $h := Y$, is 'not allowed to see the value' of variable l.

Since we concern ourselves here with just one hidden variable l, we write the new choice operator $\underset{l}{\sqcap}$, insisting therefore on the identity

$$\begin{array}{l} h := X \underset{l}{\sqcap} h := Y\,; \\ l := X \underset{\frac{1}{2}}{\oplus} l := Y \end{array} \quad = \quad \begin{array}{l} l := X \underset{\frac{1}{2}}{\oplus} l := Y; \\ h := X \underset{l}{\sqcap} h := Y\ . \end{array}$$

But what does 'seeing the value' of a variable mean? To define precisely what we want our operator to do, we will eventually have to build a mathematical model for it. And to design the model, we first need to gain more experience of the properties of the proposed operator: rather than 'rushing in' and attempting to define $\underset{l}{\sqcap}$ fully, we therefore begin instead by stating a very few properties which it would be reasonable to expect it to have.

Property 1 Since a restricted-demonic choice 'cannot see' variable l, clearly the branch it chooses cannot depend on the value of l. For example, we cannot have

$$g := X \underset{l}{\sqcap} g := Y \quad \stackrel{?}{\sqsubseteq} \quad g := l\ ,$$

because a program that is to assign a value to g without seeing the value of l cannot be implemented by simply copying l into g!

Property 2 In program fragments not otherwise containing l, it should be possible to treat restricted nondeterminism $\underset{l}{\sqcap}$ as the ordinary nondeterminism $\sqcap$. The reason for that is to

allow local reasoning[5]: the subscripts l are attached to $\sqcap$ simply to label it as being outside a module containing l, so that its algebraic manipulation can properly be influenced by that. But in local reasoning we will not know whether the rest of the program in fact does include a module hiding an l within it — indeed, at the time the reasoning is done it might not. But such a module could be added later, for completely disjoint purposes, and it would be methodologically disastrous if that should affect the validity of reasoning already done.

Property 3 In program fragments not containing $\underset{l}{\sqcap}$, variable l may be treated as an ordinary variable. That seems a reasonable requirement, since it is in the interactions of l and $\underset{l}{\sqcap}$ that we expect to find interesting effects — and manipulation of l on its own would be most convenient if it simply followed the normal rules.

We are thus brought to our first principal contribution, the thought-experiment showing that this plausible operator, with its few reasonable properties above, already leads to an inconsistency. Consider the following program refinement steps:

$$
\begin{array}{lll}
 & g := X \underset{l}{\sqcap} g := Y & \\
= & g := X;\ (g := X \underset{l}{\sqcap} g := Y) & \text{Property 2} \\
= & g := l;\ g := X; \quad (g := X \underset{l}{\sqcap} g := Y) & \text{Property 3} \\
= & g := l; \quad g := X;\ (g := X \underset{l}{\sqcap} g := Y) & \text{associativity of ;} \\
= & g := l;\ (g := X \underset{l}{\sqcap} g := Y) & \text{Property 2} \\
\sqsubseteq & g := l;\ \mathsf{skip} & \text{Property 2} \\
= & g := l\ . & \text{Property 3}
\end{array}
$$

Clearly something is very wrong: we have just 'proved' that

$$g := X \underset{l}{\sqcap} g := Y \quad \sqsubseteq \quad g := l\ ,$$

a blatant violation of Property 1. At least one of our putative properties must give way.

4 Resolving the 'paradox'

We now consider the consequences of relaxing each property in turn.

1. Without Property 1 we are left with a restricted demonic choice that can see what we have explicitly said it cannot, making all restricted-demonic choices equivalent to standard unrestricted-demonic choice. We will not consider that possibility any further.

2. Without Property 2, we are not preserving all identities involving demonic choices: in particular, we do not equate the two program fragments

$$g := X \underset{l}{\sqcap} g := Y \qquad \text{and} \qquad g := X;\ (g := X \underset{l}{\sqcap} g := Y)\ .$$

   Now the only difference between those fragments operationally is the state of g when the restricted-demonic choice is made. That suggests that a model based on Properties 1 and

[5] That is, we should be able to manipulate a program fragment without having explicitly to consider all possible programs in which that fragment might eventually be embedded.

3 would have a demonic choice that could only inspect the *current* values of variables it could see, and would not have access to the earlier values that may have been overwritten by subsequent assignments.

A consequence is that standard demonic choice would change its meaning too: if we define $\sqcap$ to be equivalent to $\underset{\{\}}{\sqcap}$, as would seem sensible, then

$$g := X;\ (g := X \sqcap g := Y) \quad \overset{?}{\sqsubseteq} \quad \mathsf{skip} \tag{8}$$

fails because the original value of g has been overwritten by an assignment: the demonic choice, unable in the new model to see that earlier value, is unable to restore it. But that is what it must do in order to have the effect of skip overall.

A model with those properties could not therefore be a simple extension of the standard probabilistic model, as the meaning of ordinary demonic choice would have changed. It could perhaps be thought of as an 'underspecified' conditional choice that assigns a probability to either branch in an unknown way dependent somehow on the current state. The choice would be refined by specifying the type of conditional choice in more detail, until only one probabilistic-conditional choice is allowed.

The restricted-demonic $P \underset{l}{\sqcap} Q$ could then be represented by the probabilistic-conditional choice which used the 'worst' function f from the state to probability that met the specification "The value of f does not change when l varies". For ordinary $\sqcap$ therefore that f could be any function over all the state variables.[6]

3. Without Property 3, we are not allowed to assume the validity of identities free of demonic choice. In particular, we may not assume that $g := l;\ g := X$ and just $g := X$ are equivalent. This tells us that the demon can somehow inspect past states of variables it is allowed to see.

   The occasions in general when $g := a;\ g := b$ and $g := b$ are not equivalent are when $g := a$ passes information to future restricted-demonic choices, that is when a is hidden but g is not: we cannot assign a hidden variable to a visible variable without possibly affecting restricted-demonic choices executed later in the program.

   A model with those properties (preserving Property 2) could therefore be an extension of the existing probabilistic/demonic model [16]. But if we are to achieve that, still we must have some way of 'hiding' the value of l from the all-seeing demon. One way to do that is to keep hidden variables as probability distributions rather than precise values — then the demon could be allowed to see everything, but in the case of l would see only its distribution. When a globally visible variable is assigned the value of a hidden variable, the possible values the global variable could then take are governed by the local distribution — but at the same time that distribution must 'collapse' to a point, since a second assignment of the hidden variable would have to yield exactly the same result as

[6] With the present apparatus the discussion of (8) may now be clearer. In ordinary sequential semantics [4] refinement (8) does hold; but the semantics based on choice-functions would make the left-hand side effectively

$$(\sqcap f{:}\,\{X, Y\} \to \mathrm{Bool} \cdot \ \ldots; g := X; \mathbf{if}\ f(g)\ \mathbf{then}\ g := X\ \mathbf{else}\ g := Y\ \mathbf{fi})\ ,$$

where the choice $\sqcap$ of f is taken before any other actions '...' of the program.

A moment's thought shows that under such an interpretation the fragment $g := X;\ (g := X \sqcap g := Y)$ cannot behave like skip.

the first. That has exactly the effect of allowing the demon to see the value of a hidden variable, once assigned to a globally visible variable and whether or not the global variable is overwritten subsequently, because the collapsed local distribution persists even if the visible variable is overwritten. That is exactly the property we illustrated above.

The model would therefore have distinct global variables, stored normally as 'precise' values, and hidden variables, stored 'smudged' as distributions.

5 Conclusions

We have already experimented with models for both versions of restricted-demonic choice.

The first model has a non-standard meaning for nondeterminism, although that way of looking at nondeterminism could perhaps be as useful as the standard way. In it, all demonic choices are turned effectively into conditional (or more generally probabilistic) choices where the unknown condition (or probability) is allowed to refer only to variables that are not hidden. The meaning of the program is then given by considering (demonically) all possible condition functions.

In that model even $\sqcap$ has altered its meaning — but that may simply be a consequence of the fact that we are now allowed to test programs by running them more than once.

The second model's restricted-demonic choice in its unrestricted form $\bigsqcap_{\{\}}$ is equivalent to the standard nondeterminism, and the idea of having variables which are always hidden from a restricted-demonic choice suggests a neat modular model with global and local variables, and procedures.

And the way that the second model could be an extension of the standard model is something we always look for; it greatly simplifies reasoning about both models at the same time. The local variables are likely to mean a large change to the model though, and the logic we have developed for that kind of reasoning cannot at this stage be said to be elegant.

We therefore do not yet have any particular reason to prefer one idea more than the other, and both models are interesting to construct and use. Also interesting is the prospect of further algebraic thought-experiments to expose more advantages and/or disadvantages of either approach.

Acknowledgements

The work reported here has been done within the Probabilistic Systems Group at Oxford (thus including also Annabelle McIver and Jeff Sanders).

The Group are grateful for the support of the EPSRC.

References

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

[2] R.-J.R. Back. Data refinement in the refinement calculus. In *Proceedings 22nd Hawaii International Conference of System Sciences*, Kailua-Kona, January 1989.

[3] J. C. M. Baeten and J. A. Bergstra. Process algebra with partial choice. In *CONCUR 94*, number 836 in LNCS, pages 465–480. Springer Verlag, 1994.

[4] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall International, Englewood Cliffs, N.J., 1976.

[5] P. H. B. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991. Reprinted in [15].

[6] RAISE Language Group. *The RAISE Specification Language.* Prentice-Hall, 1992.

[7] S. Hart, M. Sharir, and A. Pnueli. Termination of probabilistic concurrent programs. *ACM Transactions on Programming Languages and Systems*, 5:356–380, 1983.

[8] Jifeng He, K. Seidel, and A. K. McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28:171–192, 1997.

[9] C. Jou and S. A. Smolka. Equivalences, congruences, and complete axiomatizations for probabilistic processes. In *CONCUR 90*, number 458 in LNCS. Springer Verlag, 1990.

[10] D. Kozen. A probabilistic PDL. In *Proceedings of the 15th ACM Symposium on Theory of Computing*, New York, 1983. ACM.

[11] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. In *Proceedings of 16th ACM Symposium on Principles of Programming Languages, Austin, Tex.*, 1989.

[12] C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988. Reprinted in [15].

[13] C. C. Morgan. Proof rules for probabilistic loops. In He Jifeng, John Cooke, and Peter Wallis, editors, *Proceedings of the BCS-FACS 7th Refinement Workshop*, Workshops in Computing. Springer Verlag, July 1996.

[14] C. C. Morgan, A. K. McIver, K. Seidel, and J. W. Sanders. Refinement-oriented probability for CSP. *Formal Aspects of Computing*, 8(6):617–647, 1996. Also technical report PRG-TR-12-94.

[15] C. C. Morgan and T. N. Vickers, editors. *On the Refinement Calculus.* FACIT Series in Computer Science. Springer-Verlag, Berlin, 1994.

[16] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.

[17] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[18] M. Sharir, A. Pnueli, and S. Hart. Verification of probabilistic programs. *SIAM Journal on Computing*, 13(2):292–314, May 1984.

[19] W. Yi and K. G. Larsen. Testing probabilistic and nondeterministic processes. In *Proceedings of 12th IFIP International Symposium on Protocol Specification, Testing and Verification, Florida, USA*, 1992.