

Non-deterministic Expressions and Predicate Transformers

Joseph M. Morris

Dept of Computing Science
University of Glasgow
Glasgow G12 8QQ
Scotland, UK

Abstract. Non-determinacy is important in the formal specification and formal derivation of programs, but non-determinacy within expressions is theoretically problematical. The refinement calculus side-steps the problem by admitting non-determinacy only at the level of statements, leading to a style of programming that favours statements and procedures over expressions and functions. But expressions are easier to manipulate than statements, and the poverty of the expression notation has made the formal derivation of imperative programs tedious. Here we introduce non-deterministic expressions into the refinement calculus by constructing a weakest precondition semantics for imperative specifications and programs that holds good even when expressions may be non-deterministic.

Keywords non-deterministic expressions; weakest preconditions; refinement calculus

1 Introduction

Consider the little problem of making a program to compute the sign ('+' or '-') of an integer n , not caring whether '+' or '-' is associated with zero. Using the notation of guarded commands [2], most programmers would write

$$\mathbf{if} \ n \geq 0 \rightarrow x := '+' \ [] \ n \leq 0 \rightarrow x := '-' \ \mathbf{fi} \tag{1}$$

rather than

$$x := \mathbf{if} \ n \geq 0 \rightarrow '+' \ [] \ n \leq 0 \rightarrow '-' \ \mathbf{fi} \tag{2}$$

because the expression on the right hand side of (2) is non-deterministic, and although non-deterministic expressions are fine in practice, they don't work out in theory. One of the obstacles is the axiom of assignment

$$\text{wp}(x:=E, X) \equiv \Delta E \wedge X(x \leftarrow E) \quad (3)$$

where X stands for any predicate in which the program variables (among them x) may occur, ΔE stands for “ E is well-defined”, and $X(x \leftarrow E)$ denotes X with each occurrence of x replaced by E (with re-naming of bound variables if necessary to avoid variable capture). The axiom of assignment doesn't make sense when E is non-deterministic, and so formalists have chosen to outlaw non-determinacy in expressions. To quote Dijkstra and Scholten in [2]: “We will not complicate the semantics of the assignment by admitting for E partial or multivalued expressions, such as p/q or $\pm p$, respectively, generalisations that can be viewed as a misuse of the functional notation. A major charm of defining program semantics in terms of $wlp.S$ and $wp.S$ is that, even in the presence of nontermination or nondeterminacy, these are (total and unique) functions of the postcondition. We won't allow the expression in the assignment statement to destroy this.” This approach throws out the baby with the bath-water. Intuitively one feels that of the rival styles of programming — let's call that typified by (1) “procedural” and that typified by (2) “functional” — the latter is more elegant. The functional style is textually advantageous in that it avoids many assignments when one will do, and it makes immediately evident without detailed examination of the text that an assignment to just one variable is taking place. Moreover, experience has shown that in reasoning a program into existence the functional style requires less housekeeping — formal manipulation of expressions is easier than formal manipulation of statements (and indeed this observation has fuelled the huge interest in functional programming calculi in recent years).

We show how to construct a weakest precondition semantics that accommodates non-deterministic expressions. This goes some way to establishing a calculus of programming in which refinement at the statement level can coexist with refinement at the expression level. In short, we will seek to make a case that (2) is not inferior to (1). For other approaches to accommodating non-deterministic expressions within programming see [5, 6, 12, 13, 14, 15].

2 Elementary Expressions

Our primary subject of study is the world of typed expressions as they occur in imperative specification/programming languages such as the language of the refinement calculus [7]. We assume we have boolean and integer types; there may be others, but these two will suffice for our purposes. For the booleans we have the values True and False and the usual operators. For the integers we assume, addition, subtraction, division, etc. and the relational operators. Expressions need not be well-defined. When an expression is not well-defined, we say it's outcome is \perp (pronounced “bottom”); outcomes other than \perp are said to be “proper”. Note that \perp is not part of the programming/specification language, but is used only for the purposes of informal explanation. For each type we assume the language includes a boolean prefix operator Δ such that ΔE yields True if expression E is well-defined, and otherwise False. We also assume the availability of a strong equality \equiv such that $E \equiv F$ yields True if expressions E and F are both well-defined and equal, and otherwise False.

The behaviour of the booleans with respect to \perp is as follows: all operations are strict with respect to \perp except for Δ and \equiv , as well as \vee , \wedge , and \Rightarrow in the following cases

$$\begin{array}{ll} \text{True} \vee \perp \equiv \text{True} & \perp \vee \text{True} \equiv \text{True} \\ \text{False} \wedge \perp \equiv \text{False} & \perp \wedge \text{False} \equiv \text{False} \\ \text{False} \Rightarrow \perp \equiv \text{True} & \perp \Rightarrow X \equiv \text{True (for any } X) \end{array}$$

For example, $x=0 \vee x \div x=1$ holds for all integers x , as does $E \neq 0 \Rightarrow E \div E=1$ for all integer expressions E . The language will have to include existential and universal quantifiers, but their behaviour in the presence of undefinedness is not needed for the present paper. For a comprehensive treatment of the boolean laws, presented via axioms and inference rules, see [4, 9]. We may not employ Δ , \Rightarrow , \equiv , or the quantifiers in executable programs, but we may use them in specifications and in reasoning about programs.

In what follows, upper case letters near the end of the alphabet stand for arbitrary boolean expressions, and those near the front of the alphabet stand for arbitrary expressions of any type.

3 Non-determinacy

For each type we introduce the choice operator \square (pronounced “choose”): operationally, $E \square F$ yields an outcome obtained by evaluating E or an outcome obtained by evaluating F . We cannot predict which of the possible outcomes will be delivered, and indeed repeated evaluations may yield different results. We postulate that \square is symmetric, associative, and idempotent, and that all unary and binary operators distribute over \square except Δ and \equiv . For example, $(2 \square 3) = (2 \square 3)$ is equivalent to $\text{True} \square \text{False}$, whereas $(2 \square 3) \equiv (2 \square 3)$ is equivalent to True . For E a possibly non-deterministic expression, we interpret ΔE to mean “it is *guaranteed* that E will deliver a proper outcome”; we express this formally by postulating

$$\Delta(E \square F) \equiv \Delta E \wedge \Delta F.$$

There is one further exception to the general rule of distribution: implication does not distribute over \square on the left-hand side. We define $X \Rightarrow Y$ to be equivalent to $(X \neq \text{True}) \vee Y$, but this is not needed in what follows. The net effect of all this is that $\text{True} \square \text{False}$ is hardly distinguishable from \perp , and indeed the six properties of \perp given above continue to hold when \perp is replaced with $\text{True} \square \text{False}$. For a comprehensive treatment of the booleans in the presence of undefinedness and non-determinacy, see [10]. However, for the purposes of this paper, no further understanding is necessary and indeed the reader should be able to get by fairly comfortably with just an understanding of traditional two-valued logic.

4 Predicate Transformers

To accommodate arbitrary expressions in state-changing statements, we introduce two special boolean operators, wp and wlp , called “predicate transformers”. For E any expression and X any boolean expression possibly containing free variable x , $\text{wlp}(E, (x)X)$ is a boolean expression intuitively equivalent to “every proper outcome k of E satisfies $X(x \leftarrow k)$ ”. Similarly, we introduce $\text{wp}(E, (x)X)$ with the intuitive meaning “ E yields only proper outcomes, and each of its outcomes k satisfies $X(x \leftarrow k)$ ”. These operators are related by

$$\text{wp}(E, (x)X) \equiv \Delta E \wedge \text{wlp}(E, (x)X) \tag{4}$$

The X in $\text{wp}(E, (x)X)$ and $\text{wlp}(E, (x)X)$ is referred to as the “postcondition”.

As an example of the usefulness of predicate transformers, we can encode “neither 0 nor \perp is a possible outcome of F ” as $\text{wp}(F, (x)x \neq 0)$, and so we can express the law governing well-definedness of integer division as

$$\Delta(E \div F) \equiv \Delta E \wedge \text{wp}(F, (x)x \neq 0)$$

Note that it would not suffice to replace $\text{wp}(F, (x)x \neq 0)$ above with $F \neq 0$, for then we could infer an absurdity, as follows. Replacing $\text{wp}(F, (x)x \neq 0)$ above with $F \neq 0$ above, we infer that $\Delta(10 \div (3 \sqcup 0))$ is equivalent to $\Delta 10 \wedge (3 \sqcup 0) \neq 0$, which is equivalent to $3 \neq 0 \sqcup 0 \neq 0$, which is equivalent to $\text{True} \sqcup \text{False}$. On the other hand, by first distributing \div over \sqcup , we deduce that $\Delta(10 \div (3 \sqcup 0))$ is equivalent to False (which is actually the desired result, and is the one given by the law above, as will become clear later).

In the world of predicate transformer semantics [2], postconditions are always assumed to be well-defined. However, we feel it more honest to admit the possibility of undefinedness because it is impossible syntactically to exclude it — any boolean expression P may serve as a postcondition, and P could well be constituted from recursive functions which may not terminate. It is possible syntactically to exclude non-determinacy from postconditions, but the extent to which this would be inconvenient is debatable. In any event, it turns out there is little added complexity in catering for non-determinacy once we have admitted undefinedness, and so we choose not to forbid it. Note that both $\text{wlp}(E, (x)X)$ and $\text{wp}(E, (x)X)$ are always well-defined and deterministic, even if X is not so.

For expressions E which do not employ \sqcup , we postulate

$$\text{wlp}(E, (x)X) \equiv \neg \Delta E \vee (X(x \leftarrow E) \equiv \text{True})$$

The role of “ $\equiv \text{True}$ ” above is to accommodate undefinedness and non-determinacy in postconditions ($P \equiv \text{True}$ is the same as P except it is False if P is undefined or non-deterministic).

For the case of \sqcup , we define

$$\text{wlp}(E \sqcup F, (x)X) \equiv \text{wlp}(E, (x)X) \wedge \text{wlp}(F, (x)X)$$

from which we infer via (4):

$$\text{wp}(E \sqcup F, (x)X) \equiv \text{wp}(E, (x)X) \wedge \text{wp}(F, (x)X)$$

5 Conditional Expressions

In defining $\text{if } P \rightarrow E \sqcup Q \rightarrow F \text{ fi}$ we are immediately faced with the possibility, however undesirable, that P or Q (the “guards”) may be undefined or non-deterministic. It seems obvious that if any guard may yield \perp , then so may the entire $\text{if } \dots \text{ fi}$. Likewise, the outcome of $\text{if } \dots \text{ fi}$ is defined to be \perp if all guards are false. The case of non-deterministic guards is more troublesome. Consider, say, $\text{if } (\text{True} \sqcup \text{False}) \rightarrow 0 \sqcup \text{False} \rightarrow 1 \text{ fi}$. We may expect this to yield 0 (if the first guard yields True), or \perp (if the first guard yields False). However, it turns out that reasoning for such a semantics is cumbersome in the extreme. For example, even if in evaluating $\text{if } P \rightarrow E \sqcup Q \rightarrow F \text{ fi}$ the first guard evaluates to True , it cannot be assumed that the subsequent evaluation of E takes place under the truth of P . Non-deterministic guards have no known use and should never arise in practise, but we cannot outlaw them (of course the presence of \sqcup in a guard is of itself quite reasonable, as long as the non-determinacy is not externally manifest). We choose to neutralise the sting by treating $\text{True} \sqcup \text{False}$ in a guard as being no different from \perp . This may seem counter-intuitive at first sight, but it is not inconsistent with the operational viewpoint. It is only required of the machine charged with implementing a program prog that it behave like some refinement of prog . This means that an implementation need only deliver one of the possible outcomes prescribed by the formal semantics of prog , except that if \perp is a possible outcome the implementation may deliver any result whatsoever or may fail to terminate. Consequently the proposed semantics still gives implementations the freedom to take True or False as the outcome

of $\text{True} \sqcup \text{False}$ — it does not impose the obligation to fail to terminate on encountering $\text{True} \sqcup \text{False}$ in a guard.

We introduce the boolean operator δ with δP having the intuitive meaning “ P is well-defined and deterministic, i.e. P is either True or False ”:

$$\delta P \equiv \neg(P \equiv \neg P)$$

Having already defined \sqcup , it is convenient to define $\text{if } P \rightarrow E \sqcup Q \rightarrow F \text{ fi}$ by defining $P \rightarrow E$ and $\text{if } _ \text{ fi}$ separately. Intuitively, if P is undefined or non-deterministic, then the only outcome of $P \rightarrow E$ is \perp ; if P is True then $P \rightarrow E$ behaves like E ; if P is False then $P \rightarrow E$ has no outcomes whatsoever (such expressions are said to be “miraculous”). The expression $\text{if } E \text{ fi}$ behaves like E if E is not miraculous; otherwise its only outcome is \perp . Formally, we define $P \rightarrow E$ thus:

$$\begin{aligned} \Delta(P \rightarrow E) &\equiv \delta P \wedge (P \Rightarrow \Delta E) \\ \text{wlp}(P \rightarrow E, (x)X) &\equiv \neg \delta P \vee (P \Rightarrow \text{wlp}(E, (x)X)) \end{aligned}$$

from which we conclude

$$\text{wp}(P \rightarrow E, (x)X) \equiv \delta P \wedge (P \Rightarrow \text{wp}(E, (x)X))$$

We define $\text{if } E \text{ fi}$ as follows:

$$\begin{aligned} \Delta(\text{if } E \text{ fi}) &\equiv \Delta E \wedge \neg \text{wlp}(E, (x)\text{False}) \\ \text{wlp}(\text{if } E \text{ fi}, (x)X) &\equiv \text{wlp}(E, (x)X) \end{aligned}$$

from which we conclude

$$\text{wp}(\text{if } E \text{ fi}, (x)X) \equiv \neg \text{wp}(E, (x)\text{False}) \wedge \text{wp}(E, (x)X)$$

We can now calculate $\Delta(\text{if } \dots \text{ fi})$, $\text{wlp}(\text{if } \dots \text{ fi}, (x)X)$, and $\text{wp}(\text{if } \dots \text{ fi}, (x)X)$. In particular:

$$\begin{aligned} \text{wp}(\text{if } P \rightarrow E \sqcup Q \rightarrow F \text{ fi}, (x)X) &\equiv \\ &\delta P \wedge \delta Q \wedge (P \vee Q) \wedge (P \Rightarrow \text{wp}(E, (x)X)) \wedge (Q \Rightarrow \text{wp}(F, (x)X)) \end{aligned}$$

For comparison purposes, recall the weakest precondition of if -statements [3]:

$$\text{wp}(\text{if } P \rightarrow s \sqcup Q \rightarrow t \text{ fi}, X) \equiv \Delta P \wedge \Delta Q \wedge (P \vee Q) \wedge (P \Rightarrow \text{wp}(s, X)) \wedge (Q \Rightarrow \text{wp}(t, X))$$

— clearly the semantics of conditional expressions is no more complex than that of conditional statements.

We allow if -expressions as arguments of any strict operator \otimes , postulating that \otimes distributes over $\text{if } \dots \text{ fi}$ in the sense:

$$E \otimes \text{if } P \rightarrow F \sqcup Q \rightarrow G \text{ fi} = \text{if } P \rightarrow E \otimes F \sqcup Q \rightarrow E \otimes G \text{ fi}$$

and analogously for unary operators.

4 Applications

The axiom of assignment

The assignment statement $x := E$ is given a weakest precondition semantics as follows:

$$\begin{aligned} \text{wlp}(x:=E, X) &\equiv \text{wlp}(E, (x)X) \\ \text{wp}(x:=E, X) &\equiv \text{wp}(E, (x)X) \end{aligned}$$

If we assume that non-determinacy is absent from both E and X , and that X is well-defined for all x , then a small calculation gives us the traditional axiom (3).

Using the axiom of assignment and the semantics of **if**-statements and **if**-expressions, it is easy to infer that (1) and (2) have the same semantics. Thus we have achieved our main goal of giving to statement (2) and its like the theoretical respectability already enjoyed by (1).

Imperative functions

An imperative function has the form

$$f = \mathbf{fun} \ y:T \bullet F$$

where F is typically an “expression block” of the form

$$\llbracket y:T; s \bullet E \rrbracket$$

Here, y is a local variable (there may be others), s is a statement that may assign only to the variables local to the block, and E is an expression in which locals may occur free. Although such functions are common in programming, and are extremely convenient to have, they are absent from the refinement calculus [1, 7, 8] primarily because they provide a mechanism by which non-determinacy at the statement level may leak into expressions. For example, given

$$\text{sign} = \mathbf{fun} \ n:\text{int} \bullet \llbracket c: \mathbf{char}; \mathbf{if} \ n \geq 0 \rightarrow c := '+' \ \square \ n \leq 0 \rightarrow c := '-' \ \mathbf{fi} \bullet c \rrbracket$$

then $\text{sign}(0)$ is non-deterministic, even though \square is used only on statements. Our treatment of non-deterministic expressions means that we can now accommodate expression blocks:

$$\begin{aligned} \Delta \llbracket y:T; s \bullet E \rrbracket &\equiv (\forall y:T \bullet \text{wp}(s, \Delta E)) \\ \text{wlp}(\llbracket y:T; s \bullet E \rrbracket, (x)X) &\equiv (\forall y:T \bullet \text{wlp}(s, \text{wlp}(E, (x)X))) \end{aligned}$$

from which we calculate

$$\text{wp}(\llbracket y:T; s \bullet E \rrbracket, (x)X) \equiv (\forall y:T \bullet \text{wp}(s, \text{wp}(E, (x)X)))$$

Refinement

Statement s is refined by statement t , written $s \sqsubseteq t$, if whenever s is guaranteed to yield only proper outcomes, t is guaranteed to yield only values drawn from those outcomes; formally,

$$s \sqsubseteq t \equiv (\forall X \bullet \text{wp}(s, X) \Rightarrow \text{wp}(t, X))$$

where the quantification is over all postconditions X on the common state of s and t .

We can define expression refinement analogously:

$$E \sqsubseteq F \equiv (\forall X \bullet \text{wp}(E, (x)X) \Rightarrow \text{wp}(F, (x)X))$$

Expression refinement enjoys the important property of monotonic replacement:

$$E \sqsubseteq F \Rightarrow G \sqsubseteq G'$$

where G' is obtained from G by replacing one or more occurrences of E in G with F . (Actually, some restrictions apply: we may not replace occurrences of E in non-distributive positions such as ΔE or $E \equiv F$.) It also allows us to refine statements by refining their constituent expressions; for example:

$$E \sqsubseteq F \Rightarrow x := E \sqsubseteq x := F.$$

This is an important property because refining expressions is generally a less tedious task than refining statements. For example, with variables x of type **int** and s of type **sequence of int**, we can specify “assign to x an index of 0 in s ” by $x := ([k:\mathbf{int} \bullet k \in \text{dom}(s) \wedge s[k]=0])$, where $\text{dom}(s)$ denotes the domain of s . We can refine this by translating the expression on the right-hand side of the assignment to an invocation of a recursive function which we also derive. This is in contrast with the more usual approach of taking the entire assignment statement through the refinement process, leading to an iterative solution. Of course we can still take that route by first re-writing the specification as $x:[\text{True}, x \in \text{dom}(s) \wedge s[x]=0]$.

5 Conclusion

We have presented a predicate transformer semantics for statements in the presence of non-deterministic expressions. Of course, one does not use weakest precondition semantics to reason directly about programming, but uses them to establish a set of refinement laws which govern the translation of specifications into programs. The refinement calculus is a collection of such laws based on the refinement of statements. But formal manipulation of statements is burdensome, and the housekeeping sometimes overwhelms us when we apply it in practice. Manipulating expressions is easier. The semantics of expressions we have presented provides the basis for constructing a calculus of expression refinement, and one that will mesh seamlessly with statement refinement. We surmise that the refinement calculus will become more tractable when more of the refinement is carried out at the level of expressions.

References

- [1] R. J. R. Back, *Correctness preserving program refinements: Proof theory and applications*, Mathematical Centre Tracts 131 (Mathematisch Centrum, Amsterdam, 1980)
- [2] E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics* (Springer-Verlag, New York, 1990)
- [3] D. Gries, *The Science of Programming* (Springer-Verlag, New York, 1981)
- [4] C. B. Jones and C. A. Middelburg, A typed logic of partial functions reconstructed classically, *Acta Informatica* 31 (1994) 399-430
- [5] P. G. Larsen and B. S. Hansen, Semantics of under-determined expressions, *Formal Aspects of Computing*, 8 (1996) 47-66

- [6] L. Meertens, Algorithmics — towards programming as a mathematical discipline, in: J. W. de Bakker et al, eds, *Mathematics and Computer Science I*, (C.W.I., Amsterdam, 1986)
- [7] C. Morgan, *Programming from specifications* (Prentice-Hall, London, 1990)
- [8] J. M. Morris, Programs from specifications, in: E.W. Dijkstra, ed., *Formal Development of Programs and Proofs* (Addison-Wesley, Reading (Mass.), 1990)
- [9] J. M. Morris, Reasoning equationally in the presence of undefinedness, submitted for publication, 1996
- [10] J. M. Morris, Undefinedness and nondeterminacy in program proofs, submitted for publication, 1996
- [11] G. Nelson, A generalisation of Dijkstra's calculus, *ACM Transactions on Programming Languages and Systems* 11 (1989) 517-561
- [12] T. S. Norvell and E. C. R. Hehner, Logical specifications for functional programs, in: R. Bird et al., eds, *Mathematics of Program Construction*, Lecture Notes in Computer Science 669 (1993) 269-290.
- [13] H. A. Partsch, *Specification and Transformation of Programs* (Springer-Verlag, New York, 1990)
- [14] H. S. Søndergaard and P. Sestoft, Non-determinism in functional languages, *The Computer Journal*, 35 (1992) 514-523,
- [15] N. T. E. Ward, *A Refinement Calculus for Nondeterministic Expressions*, Ph.D. Thesis, Dept of Computer Science, University of Queensland (1994)