Inspiring success

Teesside University

This full text version, available on TeesRep, is the post-print (final version prior to publication) of:

Stoddart, W. J., Zeyda, F. and Dunne, S. E. (2010) 'Preference and Non-deterministic Choice', 7th international colloquium, Natal, Rio Grande do Norte, Brazil, September 1-3, 2010, in Cavalcanti, A. et. al. (eds) *Theoretical Aspects of Computing – ICTAC 2010*, Lecture Notes in Computer Science, 6255, pp.137-152.

For details regarding the final published version please click on the following DOI link:
http://dx.doi.org/10.1007/978-3-642-14808-8_10

When citing this source, please use the final published version as above.

This document was downloaded from http://tees.openrepository.com/tees/handle/10149/116087
Please do not use this version for citation purposes.

All items in TeesRep are protected by copyright, with all rights reserved, unless otherwise indicated.

TeesRep: Teesside University's Research Repository http://tees.openrepository.com/tees/

Preference and Non-deterministic Choice

Bill Stoddart[1], Frank Zeyda[2], Steve Dunne[1]

[1] University of Teesside (UK)
[2] University of York (UK)

Abstract. After reviewing a previously reported relational model of reversible computing, in which non-deterministic choice may represent provisional choice subject to revision by backtracking, we narrate our attempts to express preference within choice. We begin our investigations by recalling Nelson's extensions to Dijkstra's calculus, and considering his biased-choice operator as a model of preference. However, we find that this is too short-sighted for our purpose, and we outline the necessity of incorporating a notion of continuation. After considering how this might be achieved in a predicate-transformer approach, we adopt instead a prospective-value semantics that is easily extensible to probabilistic choice; here, we find a clue that helps us to obtain a first formulation of preference. Our formulation, however, takes us into a world of non-monotonic computations, and we are motivated to move on. We look for further inspiration within the execution structures of our reversible virtual machine which provides a construct that records all results of a backtracking search. We add a modified version that records results sequentially, and take its description as the basis for a "temporal order of continuations" semantics, to which we add implementor's choice, which is now quite distinct from provisional choice. We give a refinement relation and prove the monotonicity of the new semantics and its consistency with respect to our previous prospective-values formalism.

Key words: reversible computing, semantics, backtracking, continuations

1 Introduction

The theory of reversible computation seeks to organise computations so as to minimise their necessary energy requirements. The link between power consumption and computation comes from the link between energy and information. This tells us, via the theory of Landauer Erasure, that the destruction of information during computation presents a need to consume energy: energy conserving systems are physically reversible, and do not permit a single present state to be arrived at from multiple past states [6]. Reversible computation seeks to minimise information erasure by avoiding the compression of multiple past states into a single state, such as normally happens on execution of an assignment statement. Indeed, such erasure cannot be totally avoided, but, as the analysis of Bennett has shown [1,3] it can be limited to an initialisation phase. For the architecture we envisage, this initialisation would include the writing of a memory area used as a history stack, h, with zero values. This area is subsequently used

to retain information that would normally be erased. Using this technique we can perform any assignment $x := e$ without erasure (i.e. using only operations which have right inverses) as follows.

assignment	$h(i-1)$	$h(i)$	x
	?	0	x_0
$h(i) := h(i) + e$	?	e	x_0
$x, h(i) := h(i), x$	?	x_0	e
$i := i + 1$	x_0	0	e

The above, as well as demonstrating the possibility of an assignment statement free from data erasure, also affords us the luxury of programming reversible computations with a normal range of assignment statements rather than just those that are intrinsically reversible (e.g. $x := x + e$ where x is not free in e). Indeed we can perform such transformations for all sequential language structures, and retain all our familiar sequential programming constructs [14].

Nevertheless, in order to reuse the history stack locations thus consumed we need to sometimes reverse our computations, and to this end we introduce a new program structure $S \diamond E$ which runs program S, evaluates expression E, then reverses execution. Thus, for deterministic S, $S \diamond E$ performs an evaluation of E after S without incurring any of the state-changing effects of S. Details of how this is arranged on an implementation platform have been reported in [11,12], which are papers describing the Reversible Virtual Machine (RVM) we use as an implementation platform for our experiments in reversible language design. On our execution platform, we interpret non-deterministic choice as a provisional choice, subject to revision if it leads to an infeasible continuation. We interpret the stand-alone guarded command $g \rightarrow S$ as a command that will execute S if g is true, but will reverse if g is false. We thus obtain a sequential programming language with backtracking based on choice and guard. Where S is non-deterministic, it may lead to a number of different prospective values for E, and, using Hehner's Bunch Theory, we interpret $\{S \diamond E\}$ as the set of "prospective values" that E could take after S.

Our approach to reversible computation exploits reversibility to provide new execution structures, and simpler memory management (e.g. garbage collection on reverse execution). In this way we hope to offset the added complexity required by reversible computation, illustrated by the analysis of the reversible assignment given above, and necessarily present at all levels — the logic gates of a reversible computer, for example, must equally function without information erasure, requiring designs such as the Toffoli gate [3].

We can also look to reversibility for some new approaches to program semantics. For example, the rules for $S \diamond E$ give us the same relational semantics of conjunctive computations as the wp calculus with the law of the excluded miracle being revoked. This semantics is not rich enough, however, to express all properties of the underlying computations it models, and these properties are crucially exploited by our execution platform, the RVM. One such property is preference, and others will emerge in the course of our discussions.

The rest of the paper is organised as follows. Section 2 presents mathematical preliminaries. Section 3 considers the biased choice introduced in Nelson's generalisation of Dijkstra's calculus. Section 4 constructs a definition of preferential choice which borrows techniques from the definition of probabilistic choice. Section 5 considers the lack of conjunctivity and monotonicity properties of the new choice operator. Section 6 defines a semantics based on the temporal order of continuations, which provides a distinction between implementor's choice and the provisional choice used in backtracking. We prove monotonicity for the new semantics and establish its consistency with the prospective-value approach.

2 Preliminaries

We employ Hehner's bunch theory to describe our prospective-value (pv) semantics. Hehner points out [5] that set theory, which is foundational in mathematics, gives us, simultaneously, collection and packaging. For modelling purposes it is interesting to treat these independently. To this end we think of the content of a set as having a mathematical rather than a syntactic existence, and we call it a bunch. The content of the set $\{1, 2\}$ is the bunch $1, 2$. The comma in this expression is now an operator, known as bunch union, rather than merely syntax. The content of a set A is written as $\sim A$ where $\sim$ is the unpacking operator. The content of the empty set is known as **null**. More exactly, we follow a multi-sorted approach with a different empty set, and different **null** for each type, and we give such types by subscripting where they cannot be deduced contextually.

The lifting of bunch comma to the status of an operator allows it an ubiquity which, given the other uses of commas in our notations, requires some notational adjustments. We no longer use (a, b) for an ordered pair, using $a \mapsto b$ instead. We retain $f(a, b)$ as a notation for the application of a function to an argument pair, using $g((a, b))$ when we want to apply a function of one argument to a bunch. In this latter case the application is "bunch lifted": $g((a, b)) = g(a), g(b)$. Similarly, we have bunch lifting of infix operators, as in $(1, 2)+(3, 4) = 1+3, 1+4, 2+3, 2+4$. We write $A : B$ to assert that the elements of A are all included in B.

We define the guarded bunch $g \twoheadrightarrow E$ as equal to E when its guard g holds and **null** otherwise. We use an improper bunch $\perp$ (more exactly an improper bunch $\perp_T$ for each type) to represent non-termination in a total-correctness interpretation. We define the preconditioned bunch $P \mid E$ to be equal to E when its precondition P holds, and equal to $\perp$ otherwise. For any bunch A we have $\mathbf{null} : A$ and $A : \perp$. Bunches of a given type form a complete lattice under reverse bunch-inclusion, with **null** as its top and $\perp$ as its bottom element.

The expression $\oint x \bullet E$ represents the bunch of all values E can take as x ranges over its type, which is assumed to be inferred from E. For example, $\oint x \bullet (0 \leqslant x \wedge x < 5) \twoheadrightarrow 2x$ is the bunch of even natural numbers less than ten. Bound variables in our theory range over elements, that is singleton bunches.

A model for bunch theory has been formulated by Morris and Bunkenburg [7]. For constant terms, each value V in our universe of bunches has, as its denotation, the set whose elements are the elements of V. Thus $1, 2$ has denotation

$\{1, 2\}$ and so on.

Using bunch theory, we can express the following rules for $S \diamond E$ as S ranges over the basic syntactic constructs of our language.

$\mathbf{skip} \diamond E$	$=$	E	skip
$x := F \diamond E$	$=$	$E[F/x]$	assignment
$P \mid S \diamond E$	$=$	$P \mathbin{\mathbf{|}} (S \diamond E)$	precondition
$g \rightarrow S \diamond E$	$=$	$g \rightarrowtriangle (S \diamond E)$	guard
$S \ \square \ T \diamond E$	$=$	$(S \diamond E), (T \diamond E)$	choice
$S \,;\, T \diamond E$	$=$	$S \diamond T \diamond E$	sequential composition
$@x \bullet S \diamond E$	$=$	$\oint x \bullet S \diamond E$	unbounded choice

Assuming S operates on a state variable, or variable list, s we can obtain the predicate $\mathbf{prd}(S)$ that relates initial and final values of s (with final values being dashed) as $\mathbf{prd}(S) = s' : S \diamond s$. The set of before states from which S is not guaranteed to terminate is $\mathbf{nonpre}(S) = \{s \mid S \diamond s = \bot\}$, and the relation which maps before states of S to after states is $\mathbf{rel}(S) = \{s, s' \mid \mathbf{prd}(S)\}$.

We thus have the semantics of a relational model. For terminating computations, sequential composition of operations corresponds to composition of their respective relations: $\mathbf{rel}(S_1 \,;S_2) = \mathbf{rel}(S_1) \fatsemi \mathbf{rel}(S_2)$. The effect of a backtracking search is modelled by relational composition discarding partial paths.

This model, however, does not capture all we can usefully know about choice. The RVM provides a provisional choice structure in which the first choice is always tried first. This enables us to order choices according to search heuristics, but the idea of preference inherent in our implementation platform contains more information than can be grafted onto the relational model. Thus the use of search heuristics, and a deterministic description of cut, have been beyond the scope of our formal analysis.

We finish this section with a remark on other notations and precedence. We write $[P]$ to assert that a predicate P is universally true. Our precedence rules give highest priority to expression connectives, followed by logical connectives and then program connectives; in descending order precedence is

$$(* \,/) \ (+ \,-) \ {}_p{+} \ \sim \ \times \ \frown \ \cup \ \cap \ \setminus \ \mapsto \ \rightarrowtriangle \ \mathbin{\mathbf{|}} \ , \ (< \leqslant > \geqslant) \ (: = \neq \in \notin)$$

$$\neg \ \wedge \ \vee \ \Rightarrow \ \Leftrightarrow \ := \ \rhd \ \square \ \sqcup \ {}_p{\oplus} \ \sqcap \ \rightarrow \ ; \ . \ \bullet \ \diamond \ \nabla \ (\mathrel{\widehat{=}} \ = \ : \ \equiv)$$

The large equals and bunch containment symbols have the same meaning as, but lower precedence than, their smaller equivalents.

3 Nelson's Biased Choice

Non-determinism was originally proposed by Floyd [4] to model backtracking search, but has become even more important as an abstraction tool, allowing us to describe *what* a program should do without providing details of *how* it should do it. As an example of a problem in which both uses are significant, consider the Knight's Tour [13]. The specification says the program will return a path that forms a Knight's tour, but does not say which path. This is non-determinism used

as an abstraction tool. Within the implementation, non-determinism represents provisional choice, subject to revision if it leads to infeasibility.

A classic paper that pays some attention to the use of non-deterministic choice to provide backtracking is *An Extension of Dijkstra's Calculus* [8] which is a general correctness treatment of the wp calculus in which Dijkstra's "Law of The Excluded Miracle" is dropped, allowing possibly infeasible programming statements to appear.

Nelson defines a biased choice, which in our notation is expressed as

$$S \boxplus T \;\widehat{=}\; S \,\Box\, \neg\mathbf{fis}(S) \to T$$

This chooses its first operand unless that is infeasible, in which case it chooses the second operand. The feasibility of an operation, which we require for this definition, is defined as the inability of the operation to achieve the impossible:

$$\mathbf{fis}(S) \;\widehat{=}\; \neg \mathrm{wp}(S, \mathit{false})$$

Nelson considers biased choice as a fundamental program connective, and uses it to define a loop structure:

$$\mathbf{do}\ S\ \mathbf{od} \;\widehat{=}\; \mu\, X \bullet (S\,;X) \boxplus \mathbf{skip}$$

and considerations of the monotonicity properties of biased choice reveal that it is the Egli-Milner order, rather than refinement ordering that must be used in the corresponding fixed-point treatment.

Looking for an application of this choice in the paper, we find the following: "As an example of the power of clairvoyant non-determinism, we define a command E that parses simple arithmetic expressions, assuming we are given a procedure *Id* that parses identifiers, and procedures *Oplus* and *Otimes* which parse the tokens for the operators $+$ and $\times$. By parsing a syntactic category we mean to accept from the input the longest legal instance of the category, or failing if no prefix of the input belongs to the category. E is defined recursively:"

$$E \;\widehat{=}\; (E\,;\mathit{Oplus}\,;E) \boxplus (E\,;\mathit{Otimes}\,;E) \boxplus \mathit{Id}$$

Preference is used to make the precedence of addition lower than that of multiplication. However, our intuition is that this example is not a description of code, since execution would choose the left recursion at every stage. Indeed, recalling that general correctness allows the description of a definitively non-terminating program **loop**, which has the properties $\mathbf{loop}\,;S = \mathbf{loop}$ and $\mathbf{loop} \boxplus S = \mathbf{loop}$, we see that we can obtain **loop** as a solution to the defining equation for E. Indeed, as **loop** is the bottom element of the Egli-Milner order, it is *the* solution. For our total-correctness calculus, a similar argument would show the solution of the equation to be **abort**. Nelson comments that "an implementation of clairvoyant non-determinism is required to choose an execution that "succeeds" for some notion of success". Here, that notion must imply the avoidance of choices leading to non-termination. Clairvoyance in our formalism, however, as implemented through reversible computation, is based only on the avoidance of infeasibility: "the demon abhors a miracle". Using this paradigm, backtracking parsers whose structure directly mirrors an underlying grammar in the spirit of

Nelson's example can be written for grammars which are not left-recursive, and they can express the necessary preferences using biased choice.

As far as the more general use of biased choice to express preference is concerned, we see that it has no possibility of reacting to infeasibility that becomes apparent (in operational terms) after the left component of the biased choice has terminated. We illustrate this in the following example.

$$S \mathrel{\widehat{=}} x := 1 \boxplus x := 2 \quad \text{and} \quad T \mathrel{\widehat{=}} x = 2 \to \mathbf{skip}$$

We would like S, executed by itself, to be $x := 1$ but when placed in a context where continued execution based on the choice of S will lead to infeasibility, we require the choice to be revised; thus we require that $S\,;T$ is $x := 2$.

We certainly have S equivalent to $x := 1$ but a simple calculation in our pv calculus reveals that $S\,;T \diamond E = \mathbf{null}$, i.e. that $S\,;T$ has no after states and is thus infeasible.

A final consideration for deciding whether to include biased choice in the repertoire of our RVM is its ease of implementation. We have seen that biased choice does not enable the continuation of the program to exercise any revising control, and we therefore wonder how this particular effect could be implemented. In executing a program of the form $S \boxplus T\,;U$, any infeasibility in S causes backtracking which results in a revised choice of T. However, once S has terminated and U is executing, this behaviour must change, and if execution reverses from within U then a revision of our biased choice must *not* be made.

To achieve this effect we could probe the feasibility of S by evaluating $\{S \diamond_1 x\}$. This yields an empty set if S is infeasible, and otherwise yields a unit set containing the value of x found after the first run through S. A candidate for an *executable* definition of $S \boxplus T$ is

$$S \boxplus T \mathrel{\widehat{=}} \mathbf{if}\ \{S \diamond_1 x\} \neq \emptyset\ \mathbf{then}\ S\ \mathbf{else}\ T$$

Here S is executed in the condition clause to probe its feasibility, then the effect of S is undone by reverse execution, before possibly executing S again. A more efficient implementation might be possible, but at the expense of complicating the internal structure of the RVM. So although biased choice keeps us within the relational model, its implementation imposes some added complexity compared to the provisional non-deterministic choice already implemented on the RVM, and, in addition, its ability to express preference is limited to its use at the top level of a sequential program. These considerations, among others, motivate us to reject it as a candidate for preferential choice, and to continue our search.

4 Preference and Probabilistic Choice

Since the wp calculus has proved an effective tool for supporting program development methodologies, we first look here to see what would be involved in expressing the concept of preference we require. Suppose that, whilst performing an analysis of a term $\mathrm{wp}(S, Q)$, we find that we need to analyse $\mathrm{wp}(T, R)$, where T is some component of S, and that after T has terminated some continuation C will complete the execution of S. Thus $R = \mathrm{wp}(C, Q)$. In deciding whether

to prefer T to some other choice, we need a formulation that will allow us to choose T *unless it leads to an infeasible computation*. Note that it is not just the feasibility of T itself that concerns us, but rather the feasibility of T together with its continuation C. This combination is feasible if it cannot establish *false*, i.e. if $\neg\, \mathrm{wp}(T\,;C, \mathit{false})$. We need different wp analyses to establish whether the required postcondition is met and whether it is met in a non-vacuous (non-magical) way. We would also need to explicitly extract the continuation C. Since this approach appears clumsy, we look at pv semantics as an alternative, as this will provide us with the means to distinguish substantial and vacuous achievements of a postcondition. For example if we numerotize our predicates with the notation $|P| \mathrel{\widehat{=}} \textbf{if}\ \ P\ \ \textbf{then}\ \ 1\ \ \textbf{else}\ \ 0$ then we have $S \diamond |P| = \textbf{null}$ where S is infeasible, and where S is feasible we have $S \diamond |P| = 1$ where S will establish P, $S \diamond |P| = 0$ where S will establish $\neg\, P$, $S \diamond |P| = (0, 1)$ where S will establish either P or $\neg\, P$, and $S \diamond |P| = \bot$ where S is not certain to terminate.

Our pv semantics extends smoothly to a probabilistic calculus. Furthermore, probabilistic choice on the RVM (and in our formalisms) is subject to revision by backtracking, and thus presents itself as a candidate for expressing preference. Our expectation calculus for probabilistic choice (including its combination with non-deterministic choice) is given in [9]; here we resume the details required for the current investigation.

We add to our language a probabilistic choice $S\ {}_p\!\oplus\ T$ which makes a provisional choice of S with probability p and of T with probability $1-p$. If the choice leads to infeasibility, it is revised. This is is a rather operational description, but serves to emphasise that not only the feasibility of S or T matters for revision of a choice to occur, but the feasibility of S or T followed by their continuations.

With the addition of probabilism, $S \diamond E$, considered as a term in an executable language, may take different values on different runs. This is suggestive of the intuitive idea of a random variable (though formally a random variable is a function). We use the non-compositional notation $\varepsilon(S \diamond E)$ to express the expected value of $S \diamond E$. The expectation arising from a probabilistic choice where both choices and their continuations are feasible is

$$\varepsilon(S\ {}_p\!\oplus\ T) = p * \varepsilon(S \diamond T) + (1 - p) * \varepsilon(T \diamond E)$$

but that will not give us the properties associated with revision of choice due to backtracking. To formulate this we call on bunch notation, and first formulate a weighted addition operator which adjusts to null arguments (we recall that normally, **null** acts as an annihilator, with $x + \textbf{null} = \textbf{null}$). We define:

$$E\ {}_p\!+\ F\ \mathrel{\widehat{=}}\ p * E + (1 - p) * F,\ E = \textbf{null} \rightarrowtriangle F,\ F = \textbf{null} \rightarrowtriangle E$$

The RHS of this definition is a bunch consisting of three syntactic items. If neither of E, F is empty the first term gives the value of the bunch; otherwise the first term has a null value, and the value of the expression is determined by the second or third term, at most one of which can be non-null.

We can then give the defining equation for the expectation of a probabilistic choice under the assumption $0 < p < 1$:

$$\varepsilon(S\ {}_p\!\oplus\ T \diamond E) = \varepsilon(S \diamond E)\ {}_p\!+\ \varepsilon(T \diamond E)$$

We will need two further properties of the expectation calculus, the first deals with sequential composition:

$$\varepsilon(S\,;T\diamond E)=\varepsilon(S\diamond\varepsilon(T\diamond E))$$

and the second states that for any program S not involving probabilistic choice we have $\varepsilon(S\diamond E)=S\diamond E$.

Now let us see what happens to our little example when the choice is probabilistic; we have to evaluate

$\varepsilon(x:=1\ {}_p\!\oplus\ x:=2\,;x=2\rightarrow\mathbf{skip}\diamond x)$ $=$ "seq comp rule"
$\varepsilon(x:=1\ {}_p\!\oplus\ x:=2\diamond\varepsilon(x=2\rightarrow\mathbf{skip}\diamond x))$ $=$ "by absence of probabilistic choice in the second argument and application of pv guard and skip rules"
$\varepsilon(x:=1\ {}_p\!\oplus\ x:=2\diamond x=2\twoheadrightarrow x)$ $=$ "prob choice"
$\varepsilon(x:=1\diamond x=2\twoheadrightarrow x)\ {}_p\!+\varepsilon(x:=2\diamond x=2\twoheadrightarrow x)$ $=$ "by absence of prob choice and application of pv assignment and guard rules"
$\mathbf{null}\ {}_p\!+2$ $=$ "by weighted addition with null argument" 2

Now, by setting p close to 1 we can use the probabilistic choice $S\ {}_p\!\oplus\ T$ to show a preference for S, and, as we require, this preference will be revised if the preferred choice is infeasible. However, we cannot just set p to 1 and obtain a preferential-choice operator, since in this case probabilistic choice collapses and we are left with just S [9].

However, it seems we could use the same kind of guarded-bunch formalism in expressing preference as in expressing probabilistic choice, namely by introducing a preferential choice $[\!>$, which prefers its first operand, with the property

$$S\ [\!>\ T\diamond E\ =\ S\diamond E,(S\diamond E=\mathbf{null})\twoheadrightarrow T\diamond E$$

We further see a possibility to simplify this definition by formulating the concept of preference within bunch notation by first defining a bunch preference operator $\gg$ by virtue of $E\gg F\mathrel{\widehat{=}}E,(E=\mathbf{null})\twoheadrightarrow F$, so that the defining rule for preferential choice becomes $S\ [\!>\ T\diamond E=(S\diamond E)\gg(T\diamond E)$.

5 Preference and Monotonicity

Standard calculi for sequential programs, such as Dijkstra's GCL, Abrial's GSL, Hoare-He Designs, and Dunne's Prescriptions, describe *conjunctive* computations. In wp terms this means that $\mathrm{wp}(S,Q\wedge R)\Leftrightarrow\mathrm{wp}(S,Q)\wedge\mathrm{wp}(S,R)$. This property is a specialisation of a more general property, that of monotonicity, which is defined as $[Q\Rightarrow R]\Rightarrow(\mathrm{wp}(S,Q)\Rightarrow\mathrm{wp}(S,R))$. Angelic computations are *disjunctive* rather than conjunctive, and disjunctivity, like conjunctivity, implies monotonicity. Where both angelic and demonic computations are accommodated in a calculus, (requiring multi-relations for the associated model) the resulting computations are still monotonic [2].

It may therefore be surprising, to some readers, to find that not all formal descriptions of code are conjunctive or even monotonic, but such, indeed, is what we claim to be the case for computations involving preference as formulated in

the previous section.

To present these ideas we formulate the pv equivalent of monotonicity as $[E : F] \Rightarrow (S \diamond E) : (S \diamond F)$. We prove monotonicity for standard pv semantics, (i.e. excluding preferential choice). We appeal to the following lemmas.

Lemma 1. $x : S \diamond E \ \equiv \ \neg \mathrm{wp}(S, \neg\, x : E)$

A proof is given in [10].

Lemma 2. $[Q \Rightarrow R] \Rightarrow (\neg \mathrm{wp}(S, \neg\, Q) \Rightarrow \neg \mathrm{wp}(S, \neg\, R))$

Proof. We assume $[Q \Rightarrow R]$ and thus by predicate logic $[\neg\, R \Rightarrow \neg\, Q]$.
Then by wp monotonicity $\mathrm{wp}(S, \neg\, R) \Rightarrow \mathrm{wp}(S, \neg\, Q)$
and again by predicate logic $\neg \mathrm{wp}(S, \neg\, Q) \Rightarrow \neg \mathrm{wp}(S, \neg\, R)$.

Theorem 1. ***Monotonicity of pv semantics***

$[E : F] \Rightarrow (S \diamond E) : (S \diamond F)$

Proof. We assume $[E : F]$ and thus for arbitrary x' that $x' : E \Rightarrow x' : F$. We then show $x' : (S \diamond E) \Rightarrow x' : (S \diamond F)$ as below.

$x' : (S \diamond E) \ = \$ "by Lemma 1"
$\neg \mathrm{wp}(S, \neg\, x' : E) \ \Rightarrow \$ "by Lemma 2"
$\neg \mathrm{wp}(S, \neg\, x' : F) \ = \$ "by Lemma 1"
$x' : (S \diamond F) \quad \square$

However, the counterexample below shows that pv monotonicity does not hold when pv semantics is extended with the preferential choice operator $[>$.

$\mathbf{null} : x$ so by pv monotonicity we expect $(S \diamond \mathbf{null}) : (S \diamond x)$
but for $S \mathrel{\widehat{=}} \mathbf{skip} \ [> \mathbf{abort}$ we have
$S \diamond \mathbf{null} = (\mathbf{skip} \diamond \mathbf{null}) \gg (\mathbf{abort} \diamond \mathbf{null}) \ =$ "evaluating terms"
$\mathbf{null} \gg \bot \ =$ "by definition of bunch preference" $\bot$

whereas

$S \diamond x = (\mathbf{skip} \diamond x) \gg (\mathbf{abort} \diamond x) \ =$
$x \gg \bot \ =$ "by definition of bunch preference" x

To see the scope of the problem posed by losing pv monotonicity, let us recall how a fixed-point semantics for a language defined in terms of pv semantics is established. We need to confirm that recursive programs are soundly based, and we can do this by appealing to fixed-point theory. We need to establish a partial order between programs, and one that will serve our purpose is $S \sqsubseteq_{\mathrm{pv}} T \mathrel{\widehat{=}} (T \diamond E) : (S \diamond E)$ for arbitrary E. Indeed, under this order our programs (excluding the $[>$ operator) form a complete lattice. Any recursive program P then has an associated monotonic functional F such that $P = F(P)$. We call this fixed-point (fp) monotonicity since it is the property needed for us to appeal to the appropriate Knaster-Tarski least fixed-point theorem. To demonstrate fp monotonicity, we appeal to monotonicity of the program connectives; we have to show, e.g. that if $S \sqsubseteq_{\mathrm{pv}} S'$ and $T \sqsubseteq_{\mathrm{pv}} T'$ then $g \rightarrow S \sqsubseteq_{\mathrm{pv}} g \rightarrow S'$,

$S \sqcap T \sqsubseteq_{\mathrm{pv}} S' \sqcap T'$, $S;T \sqsubseteq_{\mathrm{pv}} S';T'$, and so on. By way of example, in establishing $S\,;\,T \sqsubseteq_{\mathrm{pv}} S'\,;\,T'$ we appeal to pv monotonicity as follows.

By the pv rule for sequential composition we have $S';T' \diamond E = S' \diamond T' \diamond E$. Now since $(T' \diamond E) : (T \diamond E)$ from the assumption $T \sqsubseteq_{\mathrm{pv}} T'$ then by pv monotonicity $(S' \diamond T' \diamond E) : (S' \diamond T \diamond E)$. And again, from the assumption $S \sqsubseteq_{\mathrm{pv}} S'$ and pv monotonicity $(S' \diamond T \diamond E) : (S \diamond T \diamond E)$. Thus by transitivity of bunch inclusion and the rule for pv sequential composition $(S'\,;\,T' \diamond E) : (S\,;\,T \diamond E)$, which by the definition of pv refinement gives $S\,;\,T \sqsubseteq_{\mathrm{pv}} S'\,;\,T'$.

The loss of pv monotonicity has serious consequences for the construction of a fixed-point semantics. Also, having gone beyond the normal relational model of total correctness, it is not immediately clear what model to now adopt, or what a suitable ordering relationship would be for refinement or fixed-point semantics. These considerations prompt us to consider an alternative approach. A further motivation is the possibility of obtaining a more complete description of the preference already present in the principal implementation of choice within the Reversible Virtual Machine.

6 Temporal Order of Continuations

In seeking another approach to the semantics of preference, we can look to our implementation of the RVM. We recall that the prospective-value term $S \diamond E$ is both a semantic device and a programming structure within an extended language of expressions, where it can occur, for example, in an assignment such as $x := S \diamond E$. The implementation of prospective-value terms requires the calculation of E after S for each of the possible routes through S that arises from making different non-deterministic choices. Each result is added to the set of results that will comprise the value for $\{S \diamond E\}$. As this set is constructed, information concerning the order of results is being discarded, but we can retain this information by recording the values as a sequence, and this we have now implemented as the "nabla term" $S \nabla E$ where E is a sequence expression. The rules defining $S \nabla E$ are explicit about the order in which provisional choices are taken, allowing the definition of a more concrete provisional-choice operator $S \rhd T$, which tentatively executes S but will backtrack to revise its choice in favour of T if execution of S and its continuation proves infeasible.

We give two small programming examples. The first is a backtracking parser for a simple language of expressions, similar to Nelson's clairvoyant parser but avoiding left recursion (which we do by the simple expedient of using a right-associative grammar). As in Nelson's example, *Id*, *Otimes* and *Oplus* attempt to parse an identifier, a multiply symbol and an addition symbol from the input stream. If this is possible they update the input stream pointer, otherwise they enter reverse execution. T parses expressions that contain no addition symbols, and E is the complete expression parser.

$$T \;\widehat{=}\; (\mathit{Id}\,;\mathit{Otimes}\,;T) \rhd \mathit{Id} \quad \text{and} \quad E \;\widehat{=}\; (T\,;\mathit{Oplus}\,;E) \rhd T$$

As an example of a program that uses a nabla term, consider the calculation of frequencies of possible scores obtained by summing the values of three dice.

We have an initialisation to record the scores associated with each of the 6^3 possible outcomes as a sequence, and an operation to interrogate the sequence and tell us the number of entries in the sequence that correspond to a given score. In this code $:\in$ represents provisional choice from a set and $\rhd$ represents Z range restriction. The initialisation code is:

$$@x_1, x_2, x_3 \bullet \mathit{scores} \ := \ (x_1 :\in \mathit{die};\ x_2 :\in \mathit{die};\ x_3 :\in \mathit{die} \nabla \langle x_1 + x_2 + x_3 \rangle)$$

The operation, giving the frequency of the score n, restricts the range of the sequence of scores to n and takes the cardinality of the resulting set:

$$f \leftarrow \mathit{freq}(n) \ \widehat{=} \ f := \mathrm{card}(\mathit{scores} \rhd \{n\})$$

We turn now to the description of nabla terms. With a view to using them within both specification-level and implementation-level language we give two forms of choice. $S \rhd T$ is a provisional preferential choice which tries S before T. $S \sqcap T$ is implementor's choice, which may be resolved as a refinement step and is *not* subject to revision by backtracking. In the following rules giving the semantics of $S \nabla E$, E is a sequence expression. Implementor's choice is represented by a bunch of possible results, whilst provisional choice is sequenced to express preference.

$\mathbf{skip} \nabla E$	$=$	E	skip
$x := F \nabla E$	$=$	$E[F/x]$	assignment
$P \mid S \nabla E$	$=$	$P \mid (S \nabla E)$	precondition
$g \rightarrow S \nabla E$	$=$	$g \rightarrowtriangle (S \nabla E), \neg g \rightarrowtriangle \langle\rangle$	guard
$S \rhd T \nabla E$	$=$	$(S \nabla E) \frown (T \nabla E)$	preferential choice
$S \sqcap T \nabla E$	$=$	$(S \nabla E), (T \nabla E)$	implementor's choice
$S \,;\, T \nabla E$	$=$	$S \nabla T \nabla E$	sequential composition
$@v \bullet S \nabla E$	$=$	$\oint v \bullet S \nabla E$	unbounded implementor's choice

We also have an associated refinement order $S \sqsubseteq_{\mathrm{toc}} T \mathrel{\widehat{=}} (T \nabla E) : (S \nabla E)$ for all sequence expressions E.

We demonstrate the rules for assignment, preference, sequential composition and guard with the following simple examples, which show how preferential choice acts in the absence and in the presence of backtracking.

$x := 1 \rhd x := 2 \nabla \langle x \rangle \ = \$ "pref choice"
$(x := 1 \nabla \langle x \rangle) \frown (x := 2 \nabla \langle x \rangle) \ = \$ "assignment"
$\langle 1 \rangle \frown \langle 2 \rangle \ = \ \langle 1, 2 \rangle$

$x := 1 \rhd x := 2 \,;\, x = 2 \rightarrow \mathbf{skip} \nabla \langle x \rangle \ = \$ "sequential composition"
$x := 1 \rhd x := 2 \nabla x = 2 \rightarrow \mathbf{skip} \nabla \langle x \rangle \ = \$ "guard and skip"
$x := 1 \rhd x := 2 \nabla x = 2 \rightarrowtriangle \langle x \rangle, \neg x = 2 \rightarrowtriangle \langle\rangle \ = \$ "pref choice"
$(x := 1 \nabla x = 2 \rightarrowtriangle \langle x \rangle, \neg x = 2 \rightarrowtriangle \langle\rangle) \frown$
$(x := 2 \nabla x = 2 \rightarrowtriangle \langle x \rangle, \neg x = 2 \rightarrowtriangle \langle\rangle) \ = \$ "assignment"
$(1 = 2 \rightarrowtriangle \langle 1 \rangle, \neg 1 = 2 \rightarrowtriangle \langle\rangle) \frown (2 = 2 \rightarrowtriangle \langle 2 \rangle, \neg 2 = 2 \rightarrowtriangle \langle\rangle)$
$=$ "properties of bunch guard"

$(\mathbf{null}, \langle\rangle) \frown (\langle 2\rangle, \mathbf{null}) \;=\;$ "bunch union with **null**"
$\langle\rangle \frown \langle 2\rangle \;=\; \langle 2\rangle$

We do not give a rule for unbounded preferential choice. We do, however, implement preferential choice from a set in the RVM, but the set must be finite, and this adds no additional expressive power over binary preferential choice. Another form of choice from a set implemented on the RVM, but beyond the scope of the current article, is to choose elements in a random order.

We no longer have the choice symbol $\Box$ in our repertoire of fundamental program connectives. Within pv semantics, choice plays the dual rôle of representing implementor's choice, to be removed during refinement, and provisional choice, to be resolved by backtracking. Since we have now teased these two concepts apart, we need an element of both in the definition that re-introduces the general notion of choice.

$$S \;\Box\; T \;\mathrel{\widehat{=}}\; (S \rhd T) \sqcap (T \rhd S)$$

We can then directly demonstrate some familiar algebraic properties of nondeterministic choice, e.g. commutativity, associativity, distribution of a guard through choice, distribution of precondition through choice, and having **magic** as a unit. However, we lose idempotence.

Let us now return to the issue of monotonicity. We recall that our first form of preferential choice, $S \,[\!> T$, resulted in a non-monotonic pv semantics. We might wonder whether the inclusion of a preferential choice must necessarily have such an effect, but, in fact, we are able to show that toc semantics *is* monotonic.

Theorem 2. ***Monotonicity of toc semantics***
For any program S and sequence expressions A and B

$$[A : B] \Rightarrow (S \bigtriangledown A) : (S \bigtriangledown B)$$

Proof. We prove $(S \bigtriangledown A) : (S \bigtriangledown B)$ under the assumption $[A : B]$. The proof is by structural induction. We have base cases for skip and assignment:

$\mathbf{skip} \bigtriangledown A \;=\; A$ "by toc semantics of skip".
$A : B$ "by assumption".
$B \;=\; \mathbf{skip} \bigtriangledown B$ "by toc semantics of skip".

and for assignment:

$x := E \bigtriangledown A \;=\; A[E/x]$ "by toc semantics of assignment".
$A[E/x] : B[E/x]$ "by assumption $[A : B]$".
$B[E/x] \;=\; x := E \bigtriangledown B$ "by toc semantics of assignment".

Now the inductive cases. For precondition we have:

$P \mid S \bigtriangledown A \;=\; P \mid (S \bigtriangledown A)$ "by toc precondition rule".
Now, noting the bunch property $E : F \Rightarrow P \mid E : P \mid F$
$P \mid (S \bigtriangledown A) \;:\; P \mid (S \bigtriangledown B)$ "by inductive case and noted property".
$P \mid (S \bigtriangledown B) \;=\; P \mid S \bigtriangledown B$ "toc precondition".

For guard the proof is similar to precondition but appealing to the bunch property $E : F \Rightarrow g \rightarrowtriangle E : g \rightarrowtriangle F$.

For preferential choice:

$S \rhd T \nabla A \;=\; (S \nabla A) \frown (T \nabla A)$ "by toc pref choice".

Now noting that $E : E' \wedge F : F' \Rightarrow E \frown F : E' \frown F'$

$(S \nabla A) \frown (T \nabla A) \;:\; (S \nabla B) \frown (T \nabla B)$ "ind. case and noted property".

$(S \nabla B) \frown (T \nabla B) \;=\; S \rhd T \nabla B$ "by toc pref choice".

For implementor's choice:

$S \sqcap T \nabla A \;=\; (S \nabla A), (T \nabla A)$ "by toc implementor's choice".

Now noting that $E : E' \wedge F : F' \Rightarrow E, F : E', F'$

$(S \nabla A), (T \nabla A) \;:\; (S \nabla B), (T \nabla B)$ "ind. case and noted property".

$(S \nabla B), (T \nabla B) \;=\; S \sqcap T \nabla B$ "by toc implementor's choice".

For unbounded choice:

$@\, v \bullet S \nabla A \;=\; \oint v \bullet (S \nabla A)$ "by toc unbounded choice".

Now noting that $[E : F] \Rightarrow \oint v \bullet E : \oint v \bullet F$

$\oint v \bullet (S \nabla A) \;:\; \oint v \bullet (S \nabla B)$ "by inductive case and noted property".

$\oint v \bullet (S \nabla B) \;=\; @\, v \bullet S \nabla B$ "by toc unbounded choice". □

Corollary 1. ***Sub-conjunctivity of toc semantics***

$(S \nabla A), (S \nabla B) : (S \nabla A, B)$

Proof. Since $[A : A, B]$ we have by toc monotonicity $(S \nabla A) : (S \nabla A, B)$ and similarly $(S \nabla B) : (S \nabla A, B)$. Furthermore, we conclude by the bunch property $E_1 : F \wedge E_2 : F \Rightarrow E_1, E_2 : F$ that $(S \nabla A), (S \nabla B) : (S \nabla A, B)$. □

Observe that toc semantics is not, however, conjunctive, and counterexamples are easy to construct, e.g. $S \mathrel{\widehat{=}} \mathbf{skip} \rhd x := x + 2$, $A \mathrel{\widehat{=}} \langle x \rangle$, and $B \mathrel{\widehat{=}} \langle x + 2 \rangle$.

One further algebraic property of choice in wp and pv semantics is distribution of sequential composition through choice. Demonstration of this property requires conjunctivity; we have only sub-conjunctivity and can demonstrate only a weaker property: $S \,;\, T \;\Box\; U \sqsubseteq_{\mathrm{toc}} (S \,;\, T) \;\Box\; (T \,;\, U)$.

To conclude, toc semantics is more discriminating than pv semantics, but should be consistent with it over the pv program connectives. Since toc semantics captures additional information about the order in which results are produced, but describes the same results as pv semantics, we require that $S \nabla \langle E \rangle$ should produce a sequence whose range is equal to $S \diamond E$. This is easily proved as a corollary of the following more general theorem.

Theorem 3. ***Consistency of pv and toc semantics***

For any program S defined over the connectives described by pv semantics, and for any sequence expression E, we have $S \diamond {\sim}\mathrm{ran}(E) = {\sim}\mathrm{ran}(S \nabla E)$.

Proof. The proof is by structural induction, once again with base cases for skip and assignment:

$\mathbf{skip} \diamond {\sim}\mathrm{ran}(E) \;=$ "pv semantics of skip"

${\sim}\mathrm{ran}(E) \;=$ "toc semantics of skip"

${\sim}\mathrm{ran}(\mathbf{skip} \nabla E)$

and for assignment:

$x := F \diamond {\sim}\mathrm{ran}(E)$ = "pv semantics of assignment"
${\sim}\mathrm{ran}(E)[F/x]$ = "property of substitution"
${\sim}\mathrm{ran}(E[F/x])$ = "toc semantics of assignment"
${\sim}\mathrm{ran}(x := F \bigtriangledown E)$

For the inductive cases we provide proofs only for some example language structures. We choose guard, choice and sequential composition. For guard we have:

$g \to S \diamond {\sim}\mathrm{ran}(E)$ = "pv semantics of guard"
$g \to (S \diamond {\sim}\mathrm{ran}(E))$ = "by appeal to inductive case"
$g \to {\sim}\mathrm{ran}(S \bigtriangledown E)$ = "by case analysis on g"
${\sim}\mathrm{ran}(g \to (S \bigtriangledown E))$ = "by toc semantics of guard"
${\sim}\mathrm{ran}(g \to S \bigtriangledown E)$

For choice we have:

$S \sqcap T \diamond {\sim}\mathrm{ran}(E)$ = "pv semantics of choice"
$(S \diamond {\sim}\mathrm{ran}(E)), (T \diamond {\sim}\mathrm{ran}(E))$ = "by appeal to inductive case"
${\sim}\mathrm{ran}(S \bigtriangledown E), {\sim}\mathrm{ran}(T \bigtriangledown E)$ = "by property ${\sim}A, {\sim}B = {\sim}(A \cup B)$"
${\sim}(\mathrm{ran}(S \bigtriangledown E) \cup \mathrm{ran}(T \bigtriangledown E))$ = "by law $(\mathrm{ran}\, s) \cup (\mathrm{ran}\, t) = \mathrm{ran}(s \frown t)$"
${\sim}\mathrm{ran}((S \bigtriangledown E) \frown (T \bigtriangledown E))$ =
"by property $\mathrm{ran}(s \frown t) = \mathrm{ran}(t \frown s)$ and idempotence of bunch union"
${\sim}(\mathrm{ran}((S \bigtriangledown E) \frown (T \bigtriangledown E), (T \bigtriangledown E) \frown (S \bigtriangledown E)))$ = "by toc pref choice"
${\sim}\mathrm{ran}((S \rhd T \bigtriangledown E), (T \rhd S \bigtriangledown E))$ = "by toc defn of $S \sqcap T$"
${\sim}\mathrm{ran}(S \sqcap T)$

For sequential composition:

$S \,;\, T \diamond {\sim}\mathrm{ran}(E)$ = "pv semantics of sequential composition"
$S \diamond T \diamond {\sim}\mathrm{ran}(E)$ = "inductive case"
$S \diamond {\sim}\mathrm{ran}(T \bigtriangledown E)$ = "inductive case"
${\sim}\mathrm{ran}(S \bigtriangledown T \bigtriangledown E)$ = "toc semantics of sequential composition"
${\sim}\mathrm{ran}(S \,;\, T \bigtriangledown E)$ $\square$

7 Conclusions

We have described our search for a method to express preference in the context of non-deterministic choice. We first considered Nelson's biased choice, but found it was too short-sighted to meet our needs. We then looked to probabilistic choice for inspiration, and we constructed a form of preferential choice. However, on inspection we found this made our calculus non-monotonic. Although this opens interesting perspectives, the loss of monotonicity is not attractive. Finally, we looked for inspiration to our Reversible Virtual Machine. This has a programming structure that will collect all the possible results of a non-deterministic computation. We added a similar structure which records the results of a search as a sequence. The formal description of this structure forms the basis for a calculus which captures preference by representing provisional choice in terms of

sequences of possible expression values to be passed to a continuation. By adding separate formulations for implementor's choice, we obtain descriptions for the essential programming connectives of a reversible guarded command language with preferential choice. Since provisional choice is now ordered and the rôle of continuations is paramount, we call this a "temporal order of continuations" semantics. We give a refinement ordering which allows implementor's choice to be reduced and preconditions to be widened. We showed that toc semantics is monotonic, though only sub-conjunctive, and we established that it is consistent with prospective-value semantics.

Future discussions will consider the partial-order properties of the toc refinement relation and its use in fixed-point treatments. Additional items on the agenda are the description of an associated relational model, the investigation of a probabilistic unification, and elaboration of the proof obligations required to employ toc as a refinement-based development method.

Acknowledgements *We acknowledge the helpful comments of the referees and thank Walter Guttmann for an interesting electronic correspondence.*

References

1. C. H. Bennett. Logical Reversibility of Computation. *IBM Journal of Research and Development*, 17(6):525–532, November 1973.
2. S. E. Dunne. Chorus Angelorum. In *B2007: Formal Specification and Development in B*, volume 4355 of *Lecture Notes in Computer Science*. Springer, 2007.
3. R. P. Feynman. *Lectures on Computation*. Westview Press, 1996.
4. R. W. Floyd. Nondeterministic Algorithms. *J of the ACM*, 14(4):636–664, 1967.
5. E. C. R. Hehner. *A Practical Theory of Programming*. Springer Verlag, 1993. Latest version available on-line at http://www.cs.toronto.edu/~hehner/aPToP/.
6. R. Landauer. Irreversibility and Heat Generated in the Computing Process. *IBM Journal of Research and Development*, 5:183–191, July 1961.
7. J. M. Morris and A. Bunkenburg. A Theory of Bunches. *Acta Informatica*, 37(8):541–561, May 2001.
8. G. Nelson. A Generalization of Dijkstra's Calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.
9. W. J. Stoddart and F. Zeyda. A unification of probabilistic choice within a design-based model of reversible computation. *Formal Aspect of Computing*, August 2007. Published on-line, DOI 10.1007/s00165-007-0048-1.
10. W. J. Stoddart, F. Zeyda, and A. R. Lynas. A Design-based model of reversible computation. In *Unifying Theories of Programming, First International Symposium*, volume 4010 of *Lecture Notes in Computer Science*, pages 63–83, June 2006.
11. W. J. Stoddart, F. Zeyda, and A. R. Lynas. A reversible virtual machine. In *Proceedings of Reversible Computation 2009*, March 2009.
12. W. J. Stoddart, F. Zeyda, and A. R. Lynas. A Virtual Machine for Supporting Reversible Probabilistic Guarded Command Languages. *Electronic Notes in Theoretical Computer Science*, 253(6):33–56, March 2010.
13. F. Zeyda. *Reversible Computations in B*. PhD thesis, University of Teesside, Middlesbrough, TS1 3BA, UK, July 2007.
14. P. Zuliani. Logical reversibility. *IBM Journal of Research and Development*, 45(6):807–818, November 2001.