bGSL: An Imperative Language for Specification and Refinement of Backtracking Programs*

Steve Dunne[1], João F. Ferreira[a], Alexandra Mendes[b], Campbell Ritchie[1], Bill Stoddart[c], Frank Zeyda[1]

[a] *INESC-ID & IST, University of Lisbon, Portugal*
[b] *HASLab / INESC TEC & Faculty of Engineering, University of Porto, Portugal*
[c] *Teesside University, United Kingdom*

Abstract

We present an imperative refinement language for the development of backtracking programs and discuss its semantic foundations. For expressivity, our language includes prospective values and preference — the latter being a variant of Nelson's biased choice that backtracks from infeasibility of a continuation. Our key contribution is to examine feasibility-preserving refinement as a basis for developing backtracking programs, and several key refinement laws that enable compositional refinement in the presence of non-monotonic program combinators.

Keywords: backtracking, semantics, preferential choice, nondeterminism

1. Introduction

Backtracking is a well-established technique to efficiently and elegantly solve a variety of computational problems. For example, the authors have been exploring backtracking in the context of reversible computing (Stoddart et al., 2010). The motivation for reversible computing comes from Landauer's observation that the loss of information in a computing device is inevitably tied to the expenditure of energy (Landauer, 1961). The inevitable cost of $kTln(2)$ units of energy for erasing one bit of information is known as Landauer's limit. For *reversible* processes, however, this limit does not apply, since all information is preserved during the computational process.

Interest in reversible computing emerged from the field of formal methods and verification. Guarded command languages (GCLs) (Dijkstra, 1975) pro-

* Authors sorted alphabetically by surname.
** DOI: https://doi.org/10.1016/j.jlamp.2022.100811
[1] Independent researcher

vide the formal underpinning of some influential verification techniques (Abrial, 1996; Crocker, 2003), and Zuliani showed how a guarded command language (GCL) can be made reversible by introducing auxiliary state to record information about the resolution of nondeterminism (Zuliani, 2001).

Noting that reversible processes can be implemented as *backtracking programs*, one can go one step further by claiming that GCLs are *already* describing reversible computations due to the implicit backtracking that arises from the semantic properties of stand-alone guards and nondeterministic choice. To illustrate this, we consider the abstract program:

$$P \;\widehat{=}\; (x := 1 \sqcap x := 2)\ ;\ x = 2 \longrightarrow \mathbf{skip}$$

Using the weakest-precondition calculus (Dijkstra, 1976) (Appendix A), we can show that this program is equivalent to $x := 2$. Operationally, we imagine that a hypothetical daemon[2] exercises a nondeterministic choice of either $x := 1$ or $x := 2$. The daemon is, however, constrained by feasibility of the subsequent guarded statement $x = 2 \longrightarrow \mathbf{skip}$. (We note that the program **skip** merely terminates without changing the program state.) We observe that an earlier choice of $x := 1$ renders the guard infeasible, and hence our daemon is forced to 'go back' and select the second alternative $x := 2$. This backtracking property of $S \sqcap T$ resulting from the interaction of nondeterminism and (in)feasibility is well understood (Morgan and Vickers, 1992). In Appendix B we show a proof of this equivalence.

When the daemon backtracks, all state is restored. Hence, the daemon's behaviour can be depicted in terms of reversibility: choices are made provisionally, with failed guards causing the execution to reverse. This operational view gives rise to a new programming language suited for the development of backtracking programs. We call this language bGSL (backtracking GSL), since its core constructs are those of the Generalised Substitution Language (GSL) of the B verification method (Abrial, 1996). However, unlike Abrial's GSL, bGSL permits nondeterministic choice and stand-alone guards not only as specification constructs but also as executable commands.

Contributions. Our first contribution is to give a precise definition of the syntax and semantics of bGSL. Alongside the standard GSL combinators described by Abrial (1996), we incorporate two additional constructs: prospective values (Zeyda et al., 2005) and preference (Stoddart et al., 2010). Prospective values have only been studied as a semantic vehicle thus far, and we here formally introduce them as programming constructs and illustrate their usefulness by way of an example (Section 6).

[2]The word 'daemon' emphasises the malevolent nature of the resolving agent.

A challenging issue in bGSL is refinement. Refinement is at the heart of many verification techniques and formally establishes when an abstract specification is (correctly) implemented by a concrete program. Conventionally, the everywhere-infeasible program **magic** $\widehat{=}$ *false* $\longrightarrow$ **skip** refines *any other program* and is therefore sometimes called a miracle; while being useful in modelling, it typically cannot be executed as establishing any specified behaviour vacuously. In bGSL, **magic** is called **reverse** and used to backtrack (i.e., to reverse execution). This poses a problem since there is no meaningful way in which a runtime system can respond if faced with executing **reverse** on its own. An attempt to overcome this issue is feasibility-preserving refinement (Zeyda et al., 2003). Sadly, it turns out that this refinement notion is not monotonic with respect to sequence in the first operand. Further monotonicity issues emerge due to preference, as it takes us out of the familiar territory of sequential and monotonic computations (Back and von Wright, 1992). Stoddart et al. (2010) observe this and investigate an alternative 'temporal order of continuation' semantics that makes implementor's and backtracking choice orthogonal. Though this leads to better monotonicity properties, it results in a less tractable refinement calculus with fewer laws. We here delve deeper into the monotonicity issue and present novel laws that facilitate piecewise development despite the aforementioned loss of monotonicity. We thereby lay the foundations for a compositional verification approach in bGSL; this is the second and central contribution of this paper. The soundness proof of the bGSL refinement calculus is not addressed in this paper; ongoing work on mechanising preferential computations will address this (see the section on future work).

The practical impact of our work is to support refinement-based development of (provably) correct backtracking algorithms with heuristics. This is particularly useful to develop correct software that runs on reversible hardware or on a reversible virtual machine such as RVM (Stoddart et al., 2010), which, at the core, consists of a reversible Forth interpreter and runtime system. The introduction of heuristics can, for instance, be expressed as a replacement of nondeterminism by preference, and refinement proof effort is leveraged by feasibility-preserving refinement, being a strict subset of standard refinement.

Structure of the paper. In Section 2, we review preliminary material: bunch theory — a theory of nondeterministic values. Section 3 discusses the syntax and semantics of bGSL, and Section 4 examines refinement and monotonicity issues. In Section 5, we discuss semantic foundations in more detail, pinning down the model of preferential computations. And in Section 6, we illustrate programming in bGSL by discussing an example. Lastly, Section 7 concludes

and discusses future and related work.

2. Preliminaries: Bunch Theory

To define the semantics of bGSL, we use the mathematical notion of a bunch, as proposed by Hehner (1981). A bunch is a non-repetitive collection of objects, similar to a set, but without structure. For instance, $0, 1, 2$ is a bunch that includes the first three natural numbers. Singleton elements of some type, such as 0 or $\{1, 2\}$, are also bunches; we call these bunches (atomic) values. The empty bunch containing no elements is written as *null*.

Bunches are motivated by the observation that set theory aggregates two concepts simultaneously: collection and packaging. Namely, the content of a set such as $\{0, 1\}$ is the bunch $0, 1$ and ought to exist independently of its packaging via the unary operator $\{\_\}$. Hehner invented bunches as a mechanism for incorporating nondeterminism and underspecification into formal programming. Set theory can be used (and often is), but it requires single-valued expressions to be treated as singleton sets, with the accompanying notational overheads of packaging and unpackaging. As Morris and Bunkenburg (2001) put it, *the theory that is 'just right' is bunch theory because there is no distinction between a value and the singleton bunch consisting of that value.*

The separating comma within set enumerations is interpreted as a binary union operator on bunches. Given two bunches E and F, bunch intersection is denoted as $E \prime F$, and bunch difference as $E \setminus F$. The content of a given set S can be extracted (as a bunch) via $\sim S$. The content of the empty set is the empty bunch *null*. We write $A : B$ to denote that the elements of bunch A are also included in bunch B. We define the guarded bunch expression $g \to E$ to be equal to E when the predicate g holds, and *null* otherwise. In previous work, we extended Hehner's bunch theory to include an additional bunch $\perp$ that is larger than (but not equal to) any other bunch over a type; we call this bunch 'improper' (Zeyda et al., 2005). The preconditioned bunch $p \mid E$ is introduced with the following meaning: it is equal to E when the predicate p holds, and otherwise equal to $\perp$. Whenever we use the term 'bunch' hereafter, we implicitly refer to the type of extended bunches. We lastly introduce bunch preference $E \triangleright F$.

Definition 2.1 (bunch preference). *Let E and F be bunches. Then,*

$$E \triangleright F \;\widehat{=}\; E, (E = \mathit{null} \to F)$$

Intuitively, the right-hand bunch F is only considered when the left-hand bunch E is empty. The motivation for this operator will become clear when we present our semantic model of bGSL in Section 3.

The expression $\S\, x \bullet E$ represents the union of all bunches E where x ranges over elementary values of its type. For example,

$$\S\, x \bullet (1 \leq x \wedge x \leq 3) \rightarrow 2x$$

is the bunch $2, 4, 6$. We note that bound variables in our theory range over atomic values only.

Extended bunches form a complete lattice under bunch inclusion. We use the latter as a basis to define a notion of refinement $E \sqsubseteq_b F$ for bunches.

Definition 2.2 (bunch refinement). *Let E and F be bunches. Then,*

$$E \sqsubseteq_b F \;\; \widehat{=} \;\; F : E$$

We observe that bunches are either atomic, plural, empty or improper. By 'plural' we mean that they contain more than one element, but are strictly smaller than $\perp$. Bunches are useful to represent the outcome of nondeterministic computations. Moreover, *null* can be used to describe the outcome of infeasible programs, and $\perp$ to capture the outcome of abortive ones. Atomic and plural bunches yield the outcome of deterministic and nondeterministic computations.

Note that the use of comma for bunch union raises a conflict with the conventional notation for ordered pairs. We therefore use $a \mapsto b$ for an ordered pair rather than the conventional (a, b). We, however, retain $f(x, y)$ as a notation for function application to multiple arguments. To apply functions to an enumerated bunch, we use a special pair of bunch brackets $(\_)_b$ as in $f(a, b)_b$. Importantly, function application distributes through bunch union and is strict with respect to *null* and $\perp$.

Hehner (1993) developed a complete axiomatisation of his bunch theory, and Zeyda (2007) presents a detailed account of extended bunch theory. A model for bunch theory was formulated by Morris and Bunkenburg (2001).

3. Backtracking GSL: Syntax and Semantics

Standard GSL (Abrial, 1996) extends Dijkstra's guarded command language (Dijkstra, 1976) with stand-alone guards and thereby revokes his 'law of the excluded miracle'. This is by admitting partially-feasible programs as specification constructs. The GSL thereby follows Nelson's generalisation of Dijkstra's calculus (Nelson, 1989). Towards defining bGSL, we extend it further in the following ways:

1. We support nondeterministic choice $S \sqcap T$ and stand-alone guards $g \longrightarrow S$ as *executable* statements, in addition to allowing them in specifications.

Construct	**Syntax**: $\mathcal{C}$	**Semantics**: $\mathcal{C} \diamond E$	**Feasibility**: $\mathit{fis}(\mathcal{C})$
Skip	**skip**	E	true
Assignment	$x := F$	$(\lambda\, x \bullet E)\, F$	$F \neq \mathit{null}$
Sequence	$S\,;\, T$	$S \diamond T \diamond E$	$\mathit{true} : S \diamond \mathit{fis}(T)$
Precondition	$p \mid S$	$p \mid\!\mid (S \diamond E)$	$p \Rightarrow \mathit{fis}(S)$
Guard	$g \longrightarrow S$	$g \rightarrow (S \diamond E)$	$g \wedge \mathit{fis}(S)$
Choice	$S \sqcap T$	$S \diamond E, T \diamond E$	$\mathit{fis}(S) \vee \mathit{fis}(T)$
Preference	$S \gg T$	$(S \diamond E) \triangleright (T \diamond E)$	$\mathit{fis}(S) \vee \mathit{fis}(T)$
Unbounded Choice	$@\, x \bullet S$	$\S\, x \bullet S \diamond E$	$\exists\, x \bullet \mathit{fis}(S)$

Table 1: Prospective-value semantics and feasibility of bGSL, where x is a variable, g and p are predicates, E and F are expressions, and S and T are programs.

2. We add a new program operator $S \gg T$ for preferential choice.

3. We add prospective-value expressions $S \diamond E$ to the expression sublanguage of our programming notation. This is a novelty: Zeyda et al. (2005) used them to reason about programs, but not as part of the expression language used in programs.

The syntax and denotational-style rules for bGSL are listed in Table 1 and must be interpreted in our theory of extended bunches. Whereas GSL defines its rules in terms of weakest-precondition (wp) predicate transformers (see Appendix A), bGSL uses a model based on prospective-value expression transformers (Zeyda et al., 2005). We will examine later on in Section 5 why this is crucial to define a model for preference.

Morris et al. (2009) first proposed expression transformers as an alternative model of nondeterministic computation. Here, the use of extended bunches enables us to describe both nondeterminism and nontermination at the level of expressions. For each construct of bGSL, we have a corresponding rule for its prospective-value (pv) effect in the third column of Table 1. The rule $\mathcal{C} \diamond E$ captures the values that expression E could take were it to be evaluated after the execution of program $\mathcal{C}$.

The precedence of operators from higher to lower is as agreed as follows:

$$x := E \;\rightsquigarrow\; \big(\, p \mid S \text{ and } g \longrightarrow S \,\big) \;\rightsquigarrow\; S \gg T \;\rightsquigarrow\; S \sqcap T \;\rightsquigarrow\; S\,;\, T \;\rightsquigarrow\; @\, x \bullet S$$

Regarding the fourth column of Table 1, the feasibility $\mathit{fis}(S)$ of a computation S can be elegantly calculated in wp semantics using the conjugate

weakest precondition $\overline{wp}(S, true)$[3]. One can give a set of structural rules to evaluate the feasibility of S without having to unfold the definition of $fis(S)$. We included them in Table 1 (fourth column). Feasibility in *pv* semantics is defined as $fis(S) \mathrel{\widehat{=}} \neg\, S \diamond \bot : null$ and equivalent to the *wp* definition. The intuition is that only infeasible programs can transform $\bot$ into *null*.

We discuss the constructs and rules in what follows.

• Skip (**skip**) is the computation that simply terminates without changing the program state. Hence the value of expression E is not changed either.

• The assignment $x := F$ also terminates but changes the value of program variable x to that of expression F. This is captured by replacing x by F within E. For instance, we have $x := 2 \diamond x + 1 = (\lambda\, x \bullet x + 1)\, 2 = 2 + 1 = 3$. This means that the value of the expression $x + 1$ after running the program $x := 2$ would be 3.

We observe that the *pv* rule for assignment uses function application rather than syntactic substitution $E[x \backslash F]$. This is to force assignment to be monotonic in the assigned expression F with respect to bunch refinement. We recall that function application, by definition, distributes through bunch union, and moreover is strict with respect to both *null* and $\bot$. This is not necessarily the case for substitution in a bunch logic (Morris and Bunkenburg, 2002). An additional advantage of using functional application is that it enables elegant rules involving functional application and composition, such as:

$$x := F \diamond f(g(x)) = f(x := F \diamond g(x)) = f(g(x := F \diamond x))$$

Intuitively, if we assign to x a plural bunch E, we expect this to become a nondeterministic assignment for each atomic value of E. Moreover, if E is empty, we expect the assignment to be infeasible; and if E is $\bot$ (the outcome of an abortive computation), the assignment must be abortive too. This is captured by the next law, which follows from the *pv* rule for assignment.

Law 1 (assign-rewrite). *Let E be an arbitrary bunch expression. Then,*

$$x := E \;\; = \;\; E \neq \bot \;\big|\; @z \bullet \;\; z : E \;\; \longrightarrow \;\; x := z$$

• A sequence $S\,;\, T$ executes S first, and then T. The self-flattening property of bunches facilitates a pleasingly simple rule, not requiring any form of Kleisli composition.

[3]The conjugate *wp* is defined as $\overline{wp}(S, Q) \mathrel{\widehat{=}} \neg\, wp(S, \neg\, Q)$. It captures if there is some way in which S can establish Q non-vacuously by a terminating behaviour.

• The precondition construct $p \mid S$ behaves just like S if p holds, and otherwise aborts. This means that outside p, it does not make any guarantees about program behaviour — not even that the computation will terminate. The rule captures this by using a preconditioned bunch, which evaluates to $\bot$ outside p. Since $\bot$ is larger than any other bunch, we also allow for any terminating behaviour in that case; this is consistent with the interpretation that we cannot insist on nontermination. A special computation is

$$\textbf{abort} \;\widehat{=}\; \mathit{false} \mid \textbf{skip}$$

which aborts from all initial states.

• A guarded statement $g \longrightarrow S$ likewise behaves like S if g holds, but otherwise refuses to execute and therefore produces no outcomes. This is captured by our rule yielding an empty bunch of outcomes in that case. Operationally, the construct behaves like a program miracle outside g, achieving any desired result vacuously. Our interpretation is, however, that the execution backtracks, since nondeterminism cannot choose infeasible execution paths as long as feasible ones are available. A special computation is **reverse**[4] which forces backtracking irrespective of the current program state.

Definition 3.1 (Reverse). $\textbf{reverse} \;\widehat{=}\; \mathit{false} \longrightarrow \textbf{skip}$

• The *pv* effect of choice translates into bunch union, as we have to consider the outcomes of both computations. Generally, the construct $S \sqcap T$ captures implementor's choice: we have no control over how or even when it is resolved. Nonetheless, it cannot choose **reverse**. This is formalised by Law 2.

Law 2 (reverse-unit). $(\textbf{reverse} \sqcap S) = S \;\; \mathit{and} \;\; (S \sqcap \textbf{reverse}) = S$

This law establishes that **reverse** is both a left and right unit of choice. We note that a guard $g \longrightarrow S$ can be encoded as $(\neg\, g \longrightarrow \textbf{reverse})\,;\, S$. For a terminating program S, we have that $(S\,;\textbf{reverse}) = \textbf{reverse}$. This, as well as the above law, can be easily proved from our semantic rules in Table 1.

The backtracking property of our language is a consequence of the next law, stating that sequential composition distributes through choice.

Law 3 (seq-distr-choice). $(S \sqcap T)\,;\, U \;=\; (S\,;\, U) \sqcap (T\,;\, U)$

[4] **reverse** is called **magic** in traditional theories of programming (see Section 1).

The proof in *pv* semantics is not difficult and omitted here too.

• The semantics of preference is given in terms of bunch preference (see Definition 2.1). Operationally, in $S \gg T$, the program S is chosen unless it reverses and thus yields *null*. Stoddart et al. (2010) first suggested this programming operator. It is similar to Nelson's biased choice, but differs in that it backtracks from infeasibility of a continuation U. It hence gives rise to a similar distribution law as Law 3. We discuss this issue in more detail in Section 5. We note that unlike choice, preference is perfectly deterministic.

• Unbounded choice captures unbounded nondeterminism of all behaviours of S where the variable x ranges over the atomic bunches of its type. Our rule rewrites it into a bunch comprehension that aggregates all bunches $S \diamond E$.

A note on assignment and prospective-value expressions. As an important novelty, we admit statements of the form $x := S \diamond E$ in bGSL. This is, fundamentally, not a problem since our meta and object term-languages are assumed to be the same. By way of an example, calculating the prospective value $y := ((x := 1 \sqcap x := 2) \diamond x{+}1) \diamond 2{*}y$ yields $y := (2, 3)_b \diamond 2{*}y$ which, by Law 1, reduces to $(y := 2 \sqcap y := 3) \diamond 2{*}y$ and thus is equal to $4, 6$. The only issue that requires more care is that of fixpoint constructions $\mu\, X \bullet F(X)$ where X occurs as part of the prospective value of an expression. For instance, if $x := G[X \diamond E]$ occurs in F, we have to additionally show that $G[E]$ is monotonic in E. (The notation $E[X]$ is used to express that X is a subterm of E.) This is trivially the case for $x := X \diamond E$, but, for instance, not so for $x := \{X \diamond E\}$ since packaging is not monotonic. Finally, we note that monotonicity of assignment in F is important to carry out piecewise refinement of prospective values: it enables us to refine $x := (S \diamond E)$ into $x := (T \diamond E)$, provided that $S \sqsubseteq T$.

4. Refinement in bGSL

Refinement is a relation between programs that makes precise when a concrete program T correctly implements an abstract program or specification S. The main idea behind this technique is to develop a program through a sequence of refinement steps, starting from an abstract specification of the program and ending up with an efficient program meeting the specification. We write $S \sqsubseteq T$ to postulate that all behaviours of T must also be possible behaviours of S.

Refinement is a powerful technique for formal program development since high-level design patterns can be formulated as algebraic laws, verified and applied. It proceeds stepwise by transforming an abstract specification, by

virtue of suitable laws, gradually into a more concrete 'design' of a program, until a description is reached that is concrete enough to be directly executable or translatable into code. Refinement may also proceed piecewise (componentwise) as long as program combinators are monotonic with respect to it. We note that for the GSL and standard refinement in *wp* semantics, this is the case.

In *pv* semantics, we characterise program refinement as bunch refinement: all possible outcomes of T must also be possible outcomes of S.

Definition 4.1 (Refinement). $S \sqsubseteq T \;\widehat{=}\; \forall E \bullet (S \diamond E) \sqsubseteq_b (T \diamond E)$

This definition of refinement is indeed consistent with refinement in *wp* program calculi. Zeyda (2007) proved that it establishes an isomorphic link to *wp* refinement if we restrict ourselves to the GSL fragment of bGSL not using preference. Thus, all GSL refinements remain valid in bGSL. The same is true for refinement laws as far as programs without preference are concerned.

A problem in bGSL with the above definition of refinement is that **magic**, in the guise of **reverse** (Definition 3.1), is, by definition, a permissible refinement of any specification or program. Whilst this makes the task of an implementor easy, it is not what we intuitively desire. In our Reversible Virtual Machine (Stoddart et al., 2010), for instance, running **reverse** at the top level results in a program failure "`ko`", as opposed to the "`ok`" that is typically displayed in Forth-like interactive environments after successful termination of a program or command. Refinement as above with its default (*wp*) semantics is hence not suitable in bGSL.

To overcome this issue, we introduce a new refinement notion $S \sqsubseteq^* T$ with the following guiding properties:

1. As before, the reduction of nondeterminism must be possible.
2. We prohibit the introduction of new infeasibility: T must be feasible in all states where S is feasible.

We thus have the following definition of feasibility-preserving refinement.

Definition 4.2 (*-Refinement). $S \sqsubseteq^* T \;\widehat{=}\; S \sqsubseteq T \wedge [\mathit{fis}(S) \Rightarrow \mathit{fis}(T)]$

This definition effectively strengthens Definition 4.1 by a feasibility caveat that formalises the second guiding property stated above.

To use feasibility-preserving refinement for stepwise and piecewise program development, it has to be a preorder and all constructs of our language (Table 1) have to be monotonic with respect to it. It is easy to prove that $\sqsubseteq^*$ is a preorder, however, it has been shown that not all constructs

of the GSL are monotonic with respect to this notion of refinement (Zeyda et al., 2003). In particular, we do not have monotonicity in the first operator of sequential composition. To illustrate this, we consider

$$x := 1 \sqcap x := 2 \;\; ; \;\; x = 1 \longrightarrow \mathbf{skip}$$

which is equivalent to $x := 1$ because the second choice $x := 2$ cannot be taken due to infeasibility of the sequenced guard $x = 1 \longrightarrow \mathbf{skip}$. We observe that $\mathit{fis}(x := 2)$ is equivalent to *true* because $2 \neq \bot$ (see the corresponding rule in Table 1). We hence have that $x := 2$ is not only a conventional but also a feasibility-preserving refinement of $x := 1 \sqcap x := 2$. Monotonicity of sequence in the first program would accordingly require that from

$$x := 1 \sqcap x := 2 \;\; \sqsubseteq^* \;\; x := 2$$

we can establish the refinement

$$\boxed{x := 1 \sqcap x := 2}; x = 1 \longrightarrow \mathbf{skip} \;\; \sqsubseteq^* \;\; \boxed{x := 2}; x = 1 \longrightarrow \mathbf{skip}$$

This, however, is not the case. Whereas the left-hand computation, as noted, is equivalent to $x := 1$, the right-hand computation reduces to **reverse** since the assignment invalidates the guard. Since $\mathit{fis}(\mathbf{reverse})$ is *false*, we clearly have that $\mathit{fis}(x := 1) \not\Rightarrow \mathit{fis}(\mathbf{reverse})$, therefore the above is not a feasibility-preserving refinement. By narrowing the nondeterministic choice in the first program of a sequential composition, we have inadvertently disabled the second program and thereby introduced new infeasibility into the overall computation.

If we limit ourselves to the GSL subset of bGSL, sequence turns out to be the only case where monotonicity with respect to $\sqsubseteq^*$ breaks down. A proof of monotonicity of the other constructs can be found in a previous report (Zeyda et al., 2003). More recently, we showed that the addition of preference has a more invasive effect (Stoddart et al., 2010): it causes certain operators to be non-monotonic *even with respect to standard refinement* $\sqsubseteq$. These are, in particular, sequence in the second (!) and preference in the first operand. Counterexamples illustrating this are included in Appendix F.

We have additionally developed a mechanised theory (Zeyda, 2022) to investigate and validate monotonicity properties of bGSL, albeit in a different semantic setting that uses three-valued Gödel-Dummet logic rather than *pv* transformers[5]. We could mechanically verify that bGSL is monotonic with

[5]On-going work is to mechanically prove that the extended *wp* model (Zeyda, 2022) using three-valued Gödel logic, and the *pv* model in this paper are isomorphic.

Operator	**Monotonic w.r.t $\sqsubseteq$**	**Monotonic w.r.t $\sqsubseteq^*$**
$p \mid S$	bGSL	bGSL
$g \longrightarrow S$	bGSL	bGSL
$S\,;\,T$	left:bGSL right:**GSL**	left:**no** right:**GSL**
$S \sqcap T$	left:bGSL right:bGSL	left:bGSL right:bGSL
$S \gg T$	left:**GSL** right:bGSL	left:**GSL** right:bGSL
$@\,x \bullet S$	bGSL	bGSL

Table 2: Monotonicity properties of bGSL.

respect to $\sqsubseteq$ in all other operators, and thus monotonic with respect to $\sqsubseteq^*$ in those operators too, except for the aforementioned case of sequence in the first operand. Table 2 summarises the monotonicity properties that have been mechanically verified.

While the above cases of losing monotonicity are challenging for piecewise development, we propose a combination of possible strategies to deal with it.

Strategy 1. We disallow piecewise refinement in non-monotonic operator positions; the same has to apply to fixpoint constructions: any $F(X)$ within $\mu\, X \bullet F(X)$ has to be syntactically restricted so that F remains monotonic.

Strategy 2. We establish the feasibility caveat $\mathit{fis}(S\,;\,U) \Rightarrow \mathit{fis}(T\,;\,U)$ independently, using the structural laws for feasibility in Table 1.

Strategy 3. Where preference is not used, Table 2 suggests that we can make stronger monotonicity assumptions since we remain in the semantic realm of the GSL. We therefore aim to postpone refinement that introduces preference.

Strategy 4. We make use of specialised (compositional) laws to refine non-monotonic operators; their caveats are separately discharged as proof obligations.

While Strategy 1 altogether excludes piecewise refinement of certain programs, the other strategies mitigate the repercussions of non-monotonic constructs. For instance, Strategy 2 enables us to $\sqsubseteq^*$-refine $S\,;\,U$ into $T\,;\,U$ by first proving $S\,;\,U \sqsubseteq T\,;\,U$ and then discharging the feasibility caveat $\mathit{fis}(S\,;\,U) \Rightarrow \mathit{fis}(T\,;\,U)$ as an additional proof obligation. This approach gives rise to a dual refinement regime where $\sqsubseteq$ and $\sqsubseteq^*$ are used in combination. Strategy 3 takes advantage of the structure of the refined program. For example, if there is no preference in the refined program and neither in

the law, we can take advantage of monotonicity of sequence with respect to $\sqsubseteq^*$ in the second operand. Ideally, only at the last stage of the refinement, nondeterministic choice is replaced by preferential choice, and the result is bound to be a feasibility-preserving refinement. We justify this in Section 5.

A downside of Strategy 2 is that the feasibility caveat can become complex, having to consider the entire program context. An alternative approach is to mimic compositionality with specialised laws (Strategy 4). A useful one is the following.

Law 4 (seq-fisref-mono). *Let S, T and U be bGSL programs. Then,*

$$S\,;\,U \sqsubseteq^* T\,;\,U \quad \textit{provided} \quad S \sqsubseteq^* T \;\; \textit{and} \;\; [\mathit{fis}(U)]$$

A proof of this law is in Appendix C. The proof burden in applying this law is lower than the one for Strategy 2 if it is easy to establish $\mathit{fis}(U)$. This, for instance, may be the case if U is syntactically restricted to feasible constructs only. Hence, an equivalent proviso replacing $[\mathit{fis}(U)]$ would be a syntactic restriction that U does not include stand-alone guarded statements and **reverse**. An operational interpretation is that U does not cause any backtracking by itself — reverse execution, if so, is either triggered by S or some computation that succeeds U.

5. Preferential Computations

Choice is symmetric: $S \sqcap T$ is semantically indistinguishable from $T \sqcap S$. An implementation is hence at liberty to decide which of S or T it executes first, and we have no control over this decision. Preference, on the other hand, gives us more control: we know that in $S \gg T$, S is tried first; and only if S fails then do we execute T. We require, however, that preference must backtrack just like choice to be useful for program development.

It turns out that the closest we can get to defining such an operator in the standard model of sequential and monotonic computations (Back and von Wright, 1992) is Nelson's biased choice (Nelson, 1989). A definition of it, using Nelson's syntax, is recaptured below.

Definition 5.1 (biased choice). $S \boxplus T \;\; \widehat{=} \;\; S \sqcap \neg\, \mathit{fis}(S) \longrightarrow T$

The nondeterminism turns into a Hobson's choice when S is feasible, since then the guard in the right-hand program becomes *false*, preventing T from being executed. The operator is hence deterministic and preferring S. Yet, it does not quite fit our desiderata. The reason is that it does not backtrack

from infeasibility of a continuation U in $(S \boxplus T)\,;\,U$. We illustrate this by way of an example. Consider the program

$$x := 1 \boxplus x := 2 \;\;;\;\; x = 2 \longrightarrow \mathbf{skip}$$

Intuitively, we expect the above to be equal to $x := 2$, in analogy to the example shown in the introduction using plain choice (page 2). If we unfold the definition of biased choice and evaluate the feasibility guard, we obtain the following.

$$\begin{array}{lll}
 & x := 1 \boxplus x := 2 \;\;;\;\; x = 2 \longrightarrow \mathbf{skip} & \{\text{unfolding biased choice}\} \\
= & x := 1 \sqcap \neg\, \mathit{fis}(x := 1) \longrightarrow x := 2\,; & \\
 & x = 2 \longrightarrow \mathbf{skip} & \{\text{calculating } \mathit{fis}(S)\} \\
= & x := 1 \sqcap \mathit{false} \longrightarrow x := 2\,; & \\
 & x = 2 \longrightarrow \mathbf{skip} & \{\text{definition of } \mathbf{reverse}\} \\
= & x := 1 \sqcap \mathbf{reverse}\,;\, x = 2 \longrightarrow \mathbf{skip} & \{\mathbf{reverse} \text{ unit law (Law 2)}\} \\
= & x := 1\,;\, x = 2 \longrightarrow \mathbf{skip} &
\end{array}$$

We observe that the second alternative $x := 2$ has disappeared, and this renders the overall computation equivalent to **reverse**. The reason for this is that the feasibility guard $\neg\, \mathit{fis}(x := 1)$ did not take into consideration that the assignment $x := 1$ may cause the continuation $x = 2 \longrightarrow \mathbf{skip}$ to fail. We hence obtain a program that is *less* feasible than what we expected since choices have been prematurely pruned. The preference operator we desire is intuitively sandwiched in between nondeterministic and biased choice within the standard refinement lattice, but equivalent to neither of them:

$$S \sqcap T \;\; \sqsubsetneq \;\; S \gg T \;\; \sqsubsetneq \;\; S \boxplus T \tag{1}$$

A question naturally arises whether our notion of preference can even be defined in standard *wp* semantics. It turns out that this is not the case (see Appendix D). To our rescue comes the theory of prospective values. We already mentioned a link between *pv* and *wp* semantics that shows that *pv* semantics is at least as discriminating as *wp* semantics. A profound insight is that it is also genuinely more expressive since we can define $S \gg T \diamond E$ within it as $(S \diamond E) \triangleright (T \diamond E)$. Unfolding the bunch preference operator and folding the *pv* effect of guards and choice yields the following alternative but equivalent formulation:

$$S \gg T \diamond E \;\; \widehat{=} \;\; (S \sqcap (S \diamond E) = \mathit{null} \longrightarrow T) \diamond E$$

We observe that this is very similar to Nelson's biased choice, however, instead of $\neg\, \mathit{fis}(S)$ we have the predicate $(S \diamond E) = \mathit{null}$. We can think of the latter as a more subtle notion of feasibility that takes into account the

continuation via the expression E. It elucidates that bunches contain more information than predicates. For instance, a *true* (weakest) precondition does not contain information of *how* the postcondition was established — that is, either by a terminating behaviour or vacuously by a miracle. The prospective value gives us this residual bit of information via the test $E = \mathit{null}$, and this facilitates a semantic characterisation of preference that indeed backtracks from infeasibility.

We introduce the synonym $\mathit{fis}^{\#}(S, E) \mathrel{\widehat{=}} (S \diamond E) \neq \mathit{null}$ as a notion of feasibility that is sensitive to the continuation E. With this we have:

$$S \gg T \diamond E \;=\; (S \sqcap \neg\, \mathit{fis}^{\#}(S, E) \longrightarrow T) \diamond E$$

The above is indeed Nelson's definition of $S \boxplus T$ with $\mathit{fis}(S)$ being replaced by $\mathit{fis}^{\#}(S, E)$. By using the rules in Table 1, we can convince ourselves that preference indeed exhibits the desired backtracking.

A downside of preference, as noted in Section 4, is that it is not monotonic with respect to standard refinement in the first operand. This trivially implies that it cannot be monotonic with respect to feasibility-preserving refinement either. To illustrate this, we consider the program $x := 1 \gg x := 2$. The permissible standard refinement of $x := 1$ into **reverse** yields **reverse** $\gg x := 2$, which is equivalent to $x := 2$. Using the *pv* rule for preference, we can show that $x := 1 \gg x := 2 \not\sqsubseteq x := 2$ or, more generally, $(S \gg T) \not\sqsubseteq T$. We thus cannot discard the first choice of a preference. We are, however, entitled to discard the second choice, as per the law $(S \gg T) \sqsubseteq S$. Preference is indeed monotonic with respect to both refinement notions in the second operand, namely $T \sqsubseteq^{[*]} T'$ implies $S \gg T \sqsubseteq^{[*]} S \gg T'$.

A fundamental preference law is that we can refine nondeterministic choice into preferential choice.

Law 5 (pref-intro). *Let S and T be bGSL programs. Then,*

$$S \sqcap T \;\sqsubseteq^{*}\; S \gg T$$

We shall discuss the proof of this key law in detail. First, to discharge the feasibility caveat $[\mathit{fis}(S \sqcap T) \Rightarrow \mathit{fis}(S \gg T)]$, we prove the following structural rule: $\mathit{fis}(S \gg T) = \mathit{fis}(S) \vee \mathit{fis}(T)$.

$$\begin{array}{lll}
 & \mathit{fis}(S \gg T) & \{\text{definition of feasibility}\} \\
= & \neg\,(S \gg T) \diamond \bot = \mathit{null} & \{pv \text{ of preference}\} \\
= & \neg\,(S \diamond \bot) \triangleright (T \diamond \bot) = \mathit{null} & \{\quad E \triangleright F = \mathit{null} \\
 & & \Leftrightarrow E = \mathit{null} \wedge F = \mathit{null}\ \} \\
= & \neg\,(S \diamond \bot = \mathit{null} \wedge T \diamond \bot = \mathit{null}) & \{\text{logic}\} \\
= & \neg\, S \diamond \bot = \mathit{null} \vee \neg\, T \diamond \bot = \mathit{null} & \{\text{definition of feasibility}\} \\
= & \mathit{fis}(S) \vee \mathit{fis}(T) &
\end{array}$$

The feasibility caveat now follows trivially since $\mathit{fis}(S \sqcap T)$ likewise can be shown to evaluate to $\mathit{fis}(S) \vee \mathit{fis}(T)$.

Second, we prove the refinement $S \sqcap T \sqsubseteq S \gg T$.

$$\begin{array}{lll} & S \sqcap T \sqsubseteq S \gg T & \{\text{definition of refinement}\} \\ = & (S \gg T \diamond E) : (S \sqcap T \diamond E) & \{pv \text{ of preference and choice}\} \\ = & (S \diamond E \triangleright T \diamond E) : (S \diamond E, T \diamond E) & \{\text{bunch law: } (E \triangleright F) : E, F\} \\ = & \mathit{true} & \end{array}$$

Above, we make use of two laws for bunch preference. For brevity, we omit their proofs in the paper, as they are altogether not difficult to establish using the definition of bunch preference.

Law 5 is special in that it can be compositionally applied even in non-monotonic operator positions.

6. Programming in bGSL: the Minimax Algorithm

A downside of only having guarded and choice constructs for backtracking is the following limit of expressivity: when backtracking, we inevitably lose any result that has been obtained by the computation up to the point where we encounter an infeasible guard. This makes it difficult to implement solutions for certain kinds of algorithmic problems in our paradigm. In this section, we show how bGSL's prospective values and preference can be used to overcome this limitation.

As an example, we consider the minimax algorithm (Russell and Norvig, 2009), an algorithm for choosing the best next move in an n-player game (usually of two players). Each game position has an associated value (score) for the first player. The idea behind minimax is that the first player should choose a move that maximises that score, assuming that the second player similarly tries to minimise it. A first attempt to design the algorithm in bGSL for a depth of two plies could be

$$\mathit{minimax} \;\widehat{=}\; \left(\begin{array}{l} a\_move_1 \sqcap a\_move_2 \sqcap \ldots ; \\ b\_move_1 \sqcap b\_move_2 \sqcap \ldots ; \\ \mathit{score} := \mathit{eval}(\mathit{state}) \,;\, \textbf{reverse} \end{array} \right)$$

We have a sequence of four statements. The first statement performs a move for player A. The second performs a move for player B. The third statement evaluates the game position and assigns the result to a global variable *score*; this is by virtue of a function $\mathit{eval}(\_)$ on the program state. We then use **reverse** to explore all combinations of moves.

The issue with this design is that whatever the value of *score* may become after we performed a move, it is inevitably restored during backtracking

caused by **reverse**. We therefore cannot carry out the min and max construction that is central to the algorithm, namely to determine the best move for player A.

In general, dynamic programming relies on search procedures propagating intermediate results between different branches of a traversal tree. Those results are used, for instance, to prune the tree — alpha-beta pruning (Russell and Norvig, 2009) achieves this for minimax — or, in some cases, to apply heuristics that control traversal to make it more efficient. We note that this style of programming is not natively possible with guards and choice alone because intermediate results are lost during backtracking, and thus cannot be reused in other branches of the search.

Prospective values. To overcome this limitation, bGSL supports prospective values as *program expressions*. The prospective value $S \diamond E$ does not have a side effect and hence retains referential transparency of the expression language. With prospective values, we can implement a two-ply minimax algorithm as sketched by the program below.

$$
\begin{array}{rcl}
a\_move & \widehat{=} & a\_move_1 \sqcap a\_move_2 \sqcap \ldots ; score := \mathbf{min}\,\{b\_move \diamond score\} \\
b\_move & \widehat{=} & b\_move_1 \sqcap b\_move_2 \sqcap \ldots ; score := eval(state) \\
minimax & \widehat{=} & \left(\begin{array}{l} scores := \{a\_move \diamond (move \mapsto score)\} ; \\ best\_move := \left(\epsilon\ move \bullet \left(\begin{array}{l} move \in \mathrm{dom}(scores) \wedge \\ scores(move) = \mathbf{max}\ \mathrm{ran}(scores) \end{array} \right) \right) \end{array} \right)
\end{array}
$$

The local definition *b_move* performs a move for player B and then assigns the evaluated game position to the variable *score*. The local program *a_move* does so accordingly for player A, while making use of *b_move* within the prospective-value term $\mathbf{min}\,\{b\_move \diamond score\}$. The latter wraps into a set the bunch of results obtained for *score* by executing *b_move*, and then calculates the minimum of that set (via the **min** function). To determine the best move for player A, we first construct a relation *scores* that maps moves of player A to their highest expected scores. This is done with the help of a local variable *move* that we assume to be set appropriately in a_move_1, a_move_2, and so on. Hilbert's ϵ is used to pick a best move amongst those that maximise the score for all moves of player A recorded in the *scores* relation; the possible moves and scores within *scores* can be conveniently obtained via its domain (dom) and range (ran).

Use of preference to encode heuristics. An optimised version of the minimax algorithm above may rank and explore moves of player A according to some heuristic that can identify more promising moves. Looking for the universally best move is often not feasible due to the exponential growth in complexity when increasing the number of plies. For some scenarios, a more efficient strategy may be to dynamically search for a move that has a score above a given threshold. With preference, we can easily design an

implementation for this strategy. The *miniopt* program below captures it.

$$
\begin{array}{lcl}
a\_move & \widehat{=} & a\_move_{\pi(1)} \gg a\_move_{\pi(2)} \gg \cdots ; score := \mathbf{min}\ \{b\_move \diamond score\} \\
b\_move & \widehat{=} & b\_move_1 \sqcap b\_move_2 \sqcap \ldots ; score := eval(state) \\
miniopt & \widehat{=} & \left(@\, score, move \bullet \left(\begin{array}{l} a\_move; \\ score < threshold \longrightarrow \mathbf{reverse}; \\ best\_move := move \end{array} \right) \right) \gg \mathbf{abort}
\end{array}
$$

Here, we use a sequential guard to explore all choices of moves for player A in *a_move*. Choices are explored in a predefined order that is *a priori* fixed here by the permutation π, which acts as a heuristic for ranking moves. The outer preference with **abort** guarantees feasibility even if no permissible move is found — in that case we make no guarantees and results obtained for *best_move* are undefined then. Exploration stops when a move choice passes the guard.

Refinement in bGSL. The starting point for a formal development is the original *minimax* with $scores(move) = \mathbf{max}\ \mathrm{ran}(scores)$ replaced by $scores(move) \geq threshold$ in the epsilon term. We first refine it into the following program which retains nondeterminism in *a_move*. This is mostly using standard GSL laws and a specialised law to eliminate the *pv* assignment $scores := \{a\_move \diamond \ldots\}$ and ϵ term.

$$
\begin{array}{lcl}
a\_move & \widehat{=} & a\_move_1 \sqcap a\_move_2 \sqcap \cdots ; score := \mathbf{min}\ \{b\_move \diamond score\} \\
b\_move & \widehat{=} & b\_move_1 \sqcap b\_move_2 \sqcap \cdots ; score := eval(state) \\
miniopt & \widehat{=} & \left(@\, score, move \bullet \left(\begin{array}{l} a\_move; \\ score \leq threshold \longrightarrow \mathbf{reverse}; \\ best\_move := move \end{array} \right) \right) \gg \mathbf{abort}
\end{array}
$$

The final step of the refinement is the introduction of preference in *a_move*, imposing an order on move choices. This last step is justified by compositional application of Law 5. (We note that *a_move* occurs in a non-monotonic position due to the outer preference.) The feasibility caveat is established independently by the structural rules in Table 1, adopting Strategy 2 on page 12.

7. Conclusion

We have presented the syntax and semantics of bGSL, a refinement language for the development of backtracking programs. Our work links and expands on several earlier ideas that have been investigated to some extent in isolation: feasibility-preserving refinement (Zeyda et al., 2003), prospective values (Zeyda et al., 2005) and preference (Stoddart et al., 2010). A novel contribution here is a detailed analysis of monotonicity issues, a justification of the semantic foundations of bGSL via *pv* transformers, and validated laws for refinement to overcome emerging monotonicity issues. We have thereby

laid the foundations for a refinement theory and calculus for bGSL that support verification of backtracking programs.

Setting out from the basic premise of using demonic choice and program miracles to perform backtracking, we explored how limitations of this approach can be overcome. Most interestingly, we showed that *pv* semantics facilitates a model of computation that is genuinely more expressive than monotonic predicate transformers. Monotonicity was for a long time thought of as fundamental to any useful model of computation; our work here challenges that view.

A related work is Nelson's "Language of the Included Miracle" (LIM) (Nelson, 2005). Nelson starts from a similar premise: using choice and guards to perform backtracking. However, he does not examine refinement issues in detail. Furthermore, he only provides deterministic choice. Stoddart et al. (2010) investigate an alternative 'temporal order of continuation' semantics that makes implementor's and backtracking choice orthogonal. Though this leads to better monotonicity properties, it results in a less tractable refinement calculus with fewer laws. A framework that *does* address refinement in much detail is Hoare and Jifeng's Unifying Theories of Programming (UTP) (Hoare and Jifeng, 1998). However, to the best of our knowledge, existing UTP theories, such as UTP Designs, cannot be used out-of-the-box to model preference, and it is unclear how to combine existing UTP theories towards a predicative model of preferential computations — although this could be an interesting piece of work in its own right. We believe that it is fundamentally feasible to define a UTP-based theory of the language that we present in this paper, but it is unclear how non-deterministic and preferential choice would combine in a denotational semantics that is based on predicates.

Future work. As future work, we plan to encode bGSL's *pv* semantics in a theorem prover. We have already done complementary work that uses a three-valued Gödel-Dummet Logic to define a *wp* calculus that is expressive enough to encode preferential computations (Zeyda, 2022). Various laws have been proved in a mechanised theory within Isabelle/HOL, and we are currently trying to formally establish an isomorphism to the *pv* semantics presented in this paper. This would immediately transfer the validity of all laws, including a Galois connection between GSL and bGSL that justifies compositional refinement of $S \sqcap T$ into $S \gg T$. This work will provide strong evidence for the soundness of the bGSL refinement calculus.

We also intend to develop more case studies demonstrating the value of combining prospective values with preference. For example, we have modelled a Knight's Tour solver that uses preference to encode Warndorf's heuris-

tic (von Warnsdorf, 1823). We plan to encode solutions to the solitaire-like games presented by Backhouse et al. (2010, 2013). It would also be interesting to explore applications of this work in creative AI, such as narrative generation (Martens et al., 2013, 2014). For example, bGSL can be used to encode a narrative generator where preference would allow to prioritize certain sub-plots. Moreover, we plan to elicit refinement patterns and strategies in bGSL for the development of systems that involve autonomous control and decision making; we claim that our language is tailored for such systems as it natively supports the verification for constraint problem solvers. In the area of security, it would be interesting to explore the use of bGSL to concisely express password cracking algorithms: since some password patterns are more likely than others, preference can be used to guide the generation process, thus producing more likely passwords first (Johnson et al., 2020; Grilo et al., 2022). Other opportunities we envisage in the area of security is where information erasure (e.g., through reversibility) may in fact be required to ensure safe and secure operations.

Finally, one of the anonymous reviewers of this paper brought to our attention that Groves (2002) addressed similar monotonicity concerns with respect to refinement regarding the Z schema calculus. Groves's solution was to decompose the standard refinement relation into two simpler (*pre* and *post* refinement) relations that made refinement in Z more mathematically tractable. As future work, it would be interesting to study if our refinement relation admits a similar kind of factorization.

Acknowledgements

The authors would like to thank the anonymous reviewers, whose comments and corrections have led to significant improvements. The authors are also grateful to Teesside University for the Staff Club, where many fruitful and joyful research meetings on this topic were held.

João F. Ferreira and Alexandra Mendes would like to thank Luís Soares Barbosa for showing them the beauty of mathematical approaches to software quality and for encouraging and supporting them in their pursuit of a career in academia. Since their undergraduate studies, Luís has inspired them with his work, passion, and kindness. They will always be grateful for having the privilege of his friendship.

This work was partially funded by the PassCert project, a CMU Portugal Exploratory Project funded by Fundação para a Ciência e a Tecnologia (FCT), with reference CMU/TIC/0006/2019 and supported by national funds through FCT under project UIDB/50021/2020.

References

W. J. Stoddart, A. R. Lynas, F. Zeyda, A Virtual Machine for Supporting Reversible Probabilistic Guarded Command Languages, Electronic Notes in Theoretical Computer Science 253 (2010) 33–56. doi:10.1016/j.entcs.2010.02.005, the RVM is openly available from Sourceforge: https://sourceforge.net/projects/rvm-forth/.

R. Landauer, Irreversibility and Heat Generation in the Computing Process, IBM Journal of Research and Development 5 (1961) 183–191. doi:10.1147/rd.53.0183.

E. W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Communications of the ACM 18 (1975) 453–457.

J.-R. Abrial, The B-Book: Assigning Programs to Meanings, Cambridge University Press, New York, NY, USA, 1996.

D. Crocker, Perfect Developer: A tool for Object-Oriented Formal Specification and Refinement, in: In Tools Exhibition Notes at Formal Methods Europe, 2003.

P. Zuliani, Logical reversibility, IBM Journal of Research and Development 45 (2001) 807–818. doi:10.1147/rd.456.0807.

E. W. Dijkstra, A Discipline of Programming, Prentice Hall, Upper Saddle River, NJ, USA, 1976.

C. C. Morgan, T. Vickers, On the Refinement Calculus, Springer, 1992.

F. Zeyda, W. J. Stoddart, S. E. Dunne, A Prospective-Value Semantics for the GSL, in: ZB 2005: Formal Specification and Development in Z and B, volume 3455 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 187–202. doi:10.1007/11415787_12.

W. J. Stoddart, F. Zeyda, S. E. Dunne, Preference and Non-deterministic Choice, in: Theoretical Aspects of Computing — ICTAC 2010, volume 6255 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 137–152. doi:10.1007/978-3-642-14808-8_10.

F. Zeyda, W. J. Stoddart, S. E. Dunne, The Refinement of Reversible Computations, in: RCS'03: 2nd International Workshop on Refinement of Critical Systems: Methods, Tools and Developments, 2003.

R. J. R. Back, J. von Wright, Combining angels, demons and miracles in program specifications, Theoretical Computer Science 100 (1992) 365–383. doi:10.1016/0304-3975(92)90309-4.

E. C. R. Hehner, Bunch theory: A simple set theory for computer science, Information Processing Letters 12 (1981) 26–30. doi:10.1016/0020-0190(81)90071-5.

J. M. Morris, A. Bunkenburg, A theory of bunches, Acta Informatica 37 (2001) 541–561. doi:10.1007/PL00013316.

E. C. H. Hehner, A Practical Theory of Programming, Monographs in Computer Science, Springer, 1993. Available online at http://www.cs.toronto.edu/~hehner/aPToP/.

F. Zeyda, Reversible Computations in B, Ph.D. thesis, Teesside University, School of Computing, Middlesbrough, TS1 3BA, UK, 2007.

G. Nelson, A Generalization of Dijkstra's Calculus, ACM Transactions on Programming Languages and Systems 11 (1989) 517–561. doi:10.1145/69558.69559.

J. M. Morris, A. Bunkenburg, M. Tyrrell, Term Transformers: A New Approach to State, ACM Transactions on Programming Languages and Systems 31 (2009) 16:1–16:42. doi:10.1145/1516507.1516511.

J. M. Morris, A. Bunkenburg, A source of inconsistency in theories of nondeterministic functions, Science of Computer Programming 43 (2002) 77–89. doi:10.1016/S0167-6423(01)00022-3.

F. Zeyda, An Extended *wp* Calculus for Preferential Computations, Technical Report, Teesside University, Middlesbrough, TS1 3BA, UK, 2022. Available from: https://joaoff.com/publication/2022/JLAMP/WPE-report.pdf.

S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach (3rd Edition), Prentice Hall, Upper Saddle River, NJ, USA, 2009.

G. Nelson, LIM and Nanoweb, Technical Report HPL-2005-41, Imaging Systems Laboratory, HP Laboratories Palo Alto, 2005.

C. A. R. Hoare, H. Jifeng, Unifying theories of programming, volume 14, Prentice Hall Englewood Cliffs, 1998.

H. C. von Warnsdorf, Des Rösselsprunges einfachste und allgemeinste Lösung, Th. G. Fr. Varnhagensehen Buchhandlung (1823).

R. Backhouse, W. Chen, J. F. Ferreira, The algorithmics of solitaire-like games, in: International Conference on Mathematics of Program Construction, Springer, Berlin, Heidelberg, 2010, pp. 1–18.

R. Backhouse, W. Chen, J. F. Ferreira, The algorithmics of solitaire-like games, Science of Computer Programming 78 (2013) 2029–2046.

C. Martens, A.-G. Bosser, J. F. Ferreira, M. Cavazza, Linear logic programming for narrative generation, in: International Conference on Logic Programming and Nonmonotonic Reasoning, Springer, Berlin, Heidelberg, 2013, pp. 427–432.

C. Martens, J. F. Ferreira, A.-G. Bosser, M. Cavazza, Generative story worlds as linear logic programs, in: Seventh Intelligent Narrative Technologies Workshop, 2014.

S. Johnson, J. F. Ferreira, A. Mendes, J. Cordry, Skeptic: Automatic, justified and privacy-preserving password composition policy selection, in: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, 2020, pp. 101–115.

M. Grilo, J. Campos, J. F. Ferreira, J. B. Almeida, A. Mendes, Verified password generation from password composition policies, in: International Conference on Integrated Formal Methods, Springer, 2022, pp. 271–288.

L. Groves, Refinement and the z schema calculus, Electronic Notes in Theoretical Computer Science 70 (2002) 70–93.

Appendix A. The wp semantics and feasibility of the GSL

Construct	**Syntax**: $\mathcal{C}$	**Semantics**: $wp(\mathcal{C}, Q)$	**Feasibility**: $\mathit{fis}(\mathcal{C})$
Skip	**skip**	Q	true
Assignment	$x := E$	$Q[x\backslash E]$	true
Sequence	$S\,;\,T$	$wp(S, wp(T, Q))$	$\overline{wp}(S, \mathit{fis}(T))$
Precondition	$p \mid S$	$p \wedge wp(S, Q)$	$p \Rightarrow \mathit{fis}(S)$
Guard	$g \longrightarrow S$	$g \Rightarrow wp(S, Q)$	$g \wedge \mathit{fis}(S)$
Choice	$S \sqcap T$	$wp(S, Q) \wedge wp(T, Q)$	$\mathit{fis}(S) \vee \mathit{fis}(T)$
Preference	$S \gg T$	**not definable**	$\mathit{fis}(S) \vee \mathit{fis}(T)$
Unbounded Choice	$@\, x \bullet S$	$\forall\, x \bullet wp(S, Q)$	$\exists\, x \bullet \mathit{fis}(S)$
Refinement	$S \sqsubseteq T$	$\forall\, Q \bullet wp(S, Q) \Rightarrow wp(T, Q)$	

Above, x is a variable, g, p and Q are predicates, and S and T are programs. The conjugate weakest-precondition $\overline{wp}(S, Q)$ is defined as $\neg\, wp(S, \neg\, Q)$. The third column follows from the definition of feasibility: $\mathit{fis}(S) \mathrel{\widehat{=}} \overline{wp}(S, \mathit{true})$.

Appendix B. The interplay of guards and choice

The following proof illustrates the backtracking property of guards and nondeterministic choice, using the example on page 1 and wp rules above.

$$
\begin{array}{lll}
 & wp(x := 1 \sqcap x := 2\,;\, x = 2 \longrightarrow \mathbf{skip}, Q) & \{wp \text{ of sequence}\} \\
= & wp(x := 1 \sqcap x := 2, wp(x = 2 \longrightarrow \mathbf{skip}, Q)) & \{wp \text{ of guard}\} \\
= & wp(x := 1 \sqcap x := 2, x = 2 \Rightarrow wp(\mathbf{skip}, Q)) & \{wp \text{ of skip}\} \\
= & wp(x := 1 \sqcap x := 2, x = 2 \Rightarrow Q) & \{wp \text{ of choice}\} \\
= & wp(x := 1, x = 2 \Rightarrow Q) \wedge wp(x := 2, x = 2 \Rightarrow Q) & \{wp \text{ of assignment}\} \\
= & (x = 2 \Rightarrow Q)[x\backslash 1] \wedge (x = 2 \Rightarrow Q)[x\backslash 2] & \{\text{substitution}\} \\
= & (1 = 2 \Rightarrow Q[x\backslash 1]) \wedge (2 = 2 \Rightarrow Q[x\backslash 2]) & \{\text{logic}\} \\
= & Q[x\backslash 2] & \{wp \text{ of assignment}\} \\
= & wp(x := 2, Q) &
\end{array}
$$

This proves the equivalence of the above program to $x := 2$, as it was claimed on page 2 of the introduction. Exploiting it in computations offers new and interesting ways to specify programs that use nondeterminism not merely to characterise "don't care" situations but also to provide, for instance, the alternatives of a search algorithm.

Appendix C. Proof of Law 4 (seq-fisref-mono)

We assume the proviso $[\mathit{fis}(U)]$ and $S \sqsubseteq^* T$. We show that $S\,;\,U \sqsubseteq^* T\,;\,U$.

$$
\begin{array}{lll}
 & S\,;\,U \sqsubseteq^* T\,;\,U & \{\text{definition of } \sqsubseteq^*\} \\
= & \left(\begin{array}{l} S\,;\,U \sqsubseteq T\,;\,U \wedge \\ [\mathit{fis}(S\,;\,U) \Rightarrow \mathit{fis}(T\,;\,U)] \end{array}\right) & \{\text{monotonicity of sequence w.r.t. } \sqsubseteq\} \\
= & \mathit{fis}(S\,;\,U) \Rightarrow \mathit{fis}(T\,;\,U) & \{\text{feasibility of sequence}\} \\
= & \overline{wp}(S, \mathit{fis}(U)) \Rightarrow \overline{wp}(T, \mathit{fis}(U)) & \{\text{assumption } [\mathit{fis}(U)]\} \\
= & \overline{wp}(S, \mathit{true}) \Rightarrow \overline{wp}(T, \mathit{true}) & \{\text{definition of } \mathit{fis}(S)\} \\
= & \mathit{fis}(S) \Rightarrow \mathit{fis}(T) & \{\text{assumption } S \sqsubseteq^* T\}
\end{array}
$$

Appendix D. Preference cannot be defined in *wp* semantics (!)

To show this, we try to elicit the *wp* model of the program $x := 1 \gg x := 2$ to establish $x = 1$ where x ranges over a type containing only two values $\{1, 2\}$. Because $x := 1$ is executed first, we expect $wp(x := 1 \gg x := 2, x = 1)$ to hold ($\bigstar$). Secondly, let us consider the *wp* effect of that program sequenced with $x = 2 \longrightarrow \mathbf{skip}$ to establish the same postcondition.

$$
\begin{array}{lll}
 & wp(x := 1 \gg x := 2\,;\, x = 2 \longrightarrow \mathbf{skip}, x = 1) & \{wp \text{ of sequence}\} \\
= & wp(x := 1 \gg x := 2, wp(x = 2 \longrightarrow \mathbf{skip}, x = 1)) & \{wp \text{ of guard}\} \\
= & wp(x := 1 \gg x := 2, x = 2 \Rightarrow wp(\mathbf{skip}, x = 1)) & \{wp \text{ of } \mathbf{skip}\} \\
= & wp(x := 1 \gg x := 2, x = 2 \Rightarrow x = 1) & \{\text{logic}\} \\
= & wp(x := 1 \gg x := 2, x \neq 2 \vee x = 1) & \{\text{typing}\} \\
= & wp(x := 1 \gg x := 2, x = 1) = \mathit{true} & \{\text{assumption } (\bigstar)\}
\end{array}
$$

This contradicts our intuition of the program, namely we would expect the above *wp* effect to be *false* since the guard $x = 2 \longrightarrow \mathbf{skip}$ is disabled by the preferred choice $x := 1$, and this ought to result in backtracking and the second program $x := 2$ being executed. The latter, clearly, cannot establish $x = 1$.

The assumptions we made are that $wp(x := 1 \gg x := 2, x = 1)$ is equivalent to *true* and that sequence, guards and **skip** have their usual *wp* meanings as in Appendix A. This is enough to derive the result above which contradicts our operational intuition of the operator. There appears to be no sensible way to trade our assumptions. Hence, this suggests that $S \gg T$ cannot be defined as a *wp* predicate transformer with the desired properties, unless perhaps we put into question the core semantics of constructs like sequence and guards.

Appendix E. Backtracking from infeasibility of a continuation

Below is a proof that demonstrates how our *pv* definition of preference has the desired property of backtracking form infeasibility of a continuation.

$$
\begin{array}{lll}
 & x := 1 \gg x := 2; x = 2 \longrightarrow \mathbf{skip} \diamond E & \{pv \text{ of sequence}\} \\
= & x := 1 \gg x := 2 \diamond x = 2 \longrightarrow \mathbf{skip} \diamond E & \{pv \text{ of guard and } \mathbf{skip}\} \\
= & x := 1 \gg x := 2 \diamond x = 2 \to E & \{pv \text{ of preference}\} \\
= & (x := 1 \diamond x = 2 \to E) \triangleright (x := 2 \diamond x = 2 \to E) & \{pv \text{ of assignment}\} \\
= & (1 = 2 \to (\lambda\, x \bullet E)\, 1) \triangleright (2 = 2 \to (\lambda\, x \bullet E)\, 2) & \{\text{simpl. of bunch guards}\} \\
= & \mathit{null} \triangleright (\lambda\, x.E)\, 2 & \{\text{bunch law: } \mathit{null} \triangleright E = E\} \\
= & (\lambda\, x.E)\, 2 & \{pv \text{ of assignments}\} \\
= & x := 2 \diamond E &
\end{array}
$$

This proves the equivalence of $x := 1 \gg x := 2 \;\; ; \;\; x = 2 \longrightarrow \mathbf{skip}$ and $x := 2$ in *pv* semantics. It is instructive to compare this proof to the one in Appendix D.

Appendix F. Monotonicity loss of preferential computations

In this appendix we present two counterexamples that highlight the loss of monotonicity of preferential computations with respect to standard refinement ($\sqsubseteq$).

Sequence in the second operand. First, we have $\mathbf{skip} \sqsubseteq \mathbf{reverse}$ since **reverse**, being synonymous for **magic**, is the top of the refinement lattice. However,

$$(\mathbf{skip} \gg \mathbf{abort}); \boxed{\mathbf{skip}} \;\not\sqsubseteq\; (\mathbf{skip} \gg \mathbf{abort}); \boxed{\mathbf{reverse}}$$

because the right-hand program $(\mathbf{skip} \gg \mathbf{abort}); \mathbf{reverse}$ is actually equivalent to **abort** due to the backtracking caused by **reverse**. And **abort** does not refine **skip** (or indeed any program other than itself).

Preference in the first operand. As before, we assume $\mathbf{skip} \sqsubseteq \mathbf{reverse}$. We have

$$(\boxed{\mathbf{skip}} \gg \mathbf{abort}) \;\not\sqsubseteq\; (\boxed{\mathbf{reverse}} \gg \mathbf{abort})$$

because the right-hand program $(\mathbf{reverse} \gg \mathbf{abort})$ is equivalent **abort** due to the key algebraic law $(\mathbf{reverse} \gg S) = S$ for preference, which can be easily verified using the rules in Table 1. Clearly, the left-hand program $\mathbf{skip} \gg \mathbf{abort}$ is not equivalent to **abort** when the continuation persists to be feasible.