Reasoning about Loops in Total and General Correctness

Steve E. Dunne[1], Ian J. Hayes[2] and Andy J. Galloway[3]

[1] School of Computing
University of Teesside, Middlesbrough, UK
s.e.dunne@tees.ac.uk
[2] School of Information Technology and Electrical Engineering
University of Queensland, Brisbane, Australia
[3] Department of Computer Science
University of York, York, UK

Abstract. We introduce a calculus for reasoning about programs in total correctness which blends UTP designs with von Wright's notion of a demonic refinement algebra. We demonstrate its utility in verifying the familiar loop-invariant rule for refining a total-correctness specification by a while loop. Total correctness equates non-termination with completely chaotic behaviour, with the consequence that any situation which admits non-termination must also admit arbitrary terminating behaviour. General correctness is more discriminating in allowing non-termination to be specified together with more particular terminating behaviour. We therefore introduce an analogous calculus for reasoning about programs in general correctness which blends UTP prescriptions with a demonic refinement algebra. We formulate a loop-invariant rule for refining a general-correctness specification by a while loop, and we use our general-correctness calculus to verify the new rule.

1 Introduction

In this paper we introduce a calculus for reasoning about programs in total correctness which blends UTP designs [15] with von Wright's notion of a demonic refinement algebra [27]. We demonstrate the utility of such a calculus in verifying succinctly the familiar loop-invariant rule for refining a total-correctness specification by a while loop. The rule itself is by no means new, but we believe that both our formulation of it and —even more particularly— our algebraic style of its verification are of interest.

Total correctness equates non-termination with completely chaotic behaviour, with the consequence that any situation which admits non-termination must also admit arbitrary terminating behaviour. In contrast, general correctness is more discriminating in allowing non-termination to be specified together with more particular terminating behaviour, or even without any allowed terminating behaviour at all. We introduce an analogous calculus for reasoning about programs in general correctness which blends UTP prescriptions [9] with a demonic

refinement algebra. We formulate a loop-invariant rule for refining a general-correctness specification by a while loop —the first time, as far as we are aware, that this has been done. We then use our general-correctness calculus to verify the new rule, whose significance is that it completes our general-correctness refinement calculus by allowing us to verify the refinement of a UTP prescription all the way to an actual implementation in executable code.

The rest of the paper is organised as follows. After disposing of certain necessary preliminary issues in Section 2, in Section 3 we develop our total-correctness calculus, then in Section 4 we formulate and prove our version of the classical total-correctness variant-invariant while-loop refinement rule. In Section 5 we develop a corresponding general-correctness calculus based on UTP prescriptions instead of designs, and in Section 6 we formulate and verify our new general-correctness while-loop refinement rule, and illustrate its application on a small example refinement. We conclude Section 6 by deriving an interesting subsidiary general-correctness while-loop refinement rule.

2 Preliminaries

Here we clarify the symbolic conventions we employ throughout the rest of the paper, and we also give a brief historical summary of program algebras.

2.1 Systematic decoration

We note here a typographical convention that we adopt throughout the paper, namely the systematic dash-decoration of variables and metavariables which is usual in UTP. So if b, for example, represents a condition on the plain (*i.e.* undashed) variables of a program's before-state, then b' represents the corresponding condition on the dashed variables of that program's after-state.

2.2 Logical notation

We use $\Rightarrow$ only as the *material implication* connective between propositions in our relational term language. On the other hand, we use $\Rrightarrow$ and its typographical inverse $\Lleftarrow$ in rules and proofs to signify logical entailment between factual assertions *about* relations. Similarly, we use $\equiv$ to assert mutual logical entailment in both directions. However, we employ the ordinary equality symbol $=$ to assert equivalence between individual computation-denoting relations such as UTP designs and prescriptions, because this is more natural when treating such entities algebraically.

Although we use $\wedge$ as the conjunction connective between propositions in our relational term language, we also employ it in rules and proofs to conjoin factual assertions *about* relations. Its latter use is always distinguished by generous white spacing around it.

2.3 About refinement in UTP

The most important single unifying feature of UTP is that refinement is always modelled simply by universally quantified reverse logical implication. In this paper we encounter refinement in three separate semantic contexts: namely, partial correctness, total correctness and general correctness. In each case refinement is denoted by the same symbol $\sqsubseteq$ because this always signifies universally quantified reverse logical implication. All that varies between the three contexts is the family of alphabetised relations being related by $\sqsubseteq$. In the case of partial correctness, the relations concerned are just the binary relations on the alphabet $\{v, v'\}$ where v is the list of program state variables. In the case of total correctness, the relations concerned are designs, which, as we shall see, are a family of binary relations over the alphabet $\{v, ok, v', ok'\}$[4] which are characterised by certain healthiness conditions. Similarly, in the case of general correctness the relations concerned are prescriptions, which are another family of binary relations over the same alphabet $\{v, ok, v', ok'\}$ characterised by different healthiness conditions. In any appearance of $\sqsubseteq$ in a formula within the ensuing text its operands there will enable us to determine whether it signifies partial-, total- or general-correctness refinement in that particular case.

2.4 Precedence

We adopt the conventional order of precedence for the binding powers of our propositional connectives: namely (in descending order), negation $\neg$, conjunction $\wedge$, disjunction $\vee$, material implication $\Rightarrow$ and material bi-implication $\Leftrightarrow$. These all precede our design constructor $\vdash$ and prescription constructor $\Vdash$, which in turn precede the equality operator $=$ and refinement operator $\sqsubseteq$ between computation-denoting relations such as designs and prescriptions. These in turn precede our widely-spaced conjunction " $\wedge$ " between factual assertions about relations. Lowest in precedence we have our logical entailment metasymbols $\Rrightarrow$, $\Lleftarrow$ and $\equiv$. In our program algebra sequential composition, denoted by " ; " or more usually by simple juxtaposition, has a higher precedence than nondeterministic choice $\sqcap$. Where we believe it improves readability, especially in designs and prescriptions, we sometimes use parentheses even when they are superfluous in the light of these precedence rules.

2.5 History of program algebras

Kleene Algebras (KA) were developed first by Conway [5] and later by Kozen [18] in response to the American mathematician Stephen Kleene's challenge to find a complete axiomatisation of regular expressions. As well as binary operators representing choice, sequence, etc, KA has a postfix unary operator "$*$" (star) to represent finite repetition. More recently, KA was extended by Kozen

[4] ok' and ok are auxiliary boolean variables introduced to record the observation of termination respectively of the current program and of its sequential predecessor.

[19, 20], who added tests to provide "Kleene Algebra with Tests" (KAT) for reasoning about programs in partial correctness. Subsequently von Wright [26, 27] developed a variation of KAT called Demonic Refinement Algebra (DRA), intended as a framework for reasoning about programs in total correctness, in which he introduces a second unary postfix operator ω (omega) to represent arbitrary (*i.e.* finite or infinite) repetition. To accommodate the latter he discards the right-zero axiom of KAT. As von Wright points out, DRA is in many ways similar to Cohen's omega algebra [4], save that the latter, being a conservative extension of KAT, requires finite and infinite executions to be reasoned about separately. From DRA von Wright has gone on to develop General Refinement Algebra (GRA), so called because it incorporates a second choice operator to model angelic as well as demonic nondeterminism, but this is beyond the scope of our work here. In particular, the "General" in von Wright's GRA is not to be confused with with the semantics of general correctness which we address in this paper.

3 A Total-correctness Program Calculus

Our program calculus models conjunctive programs, *i.e.* those which may exhibit demonic but not angelic nondeterminism. The basic statements of our calculus are H3-healthy UTP designs [15]. Let v be the list of state variables of the state space and ok be an additional auxiliary boolean variable which when initially true (ok) signifies that, its predecessor having terminated, the present computation has started, and when finally true (ok') signifies the present computation has itself subsequently terminated. Then a design is an alphabetised relation over an alphabet $\{v, ok, v', ok'\}$ which is expressible in the form

$$ok \wedge p \Rightarrow ok' \wedge q$$

where p and q are subsidiary predicates respectively over v and $\{v, v'\}$. We abbreviate the above to $p \vdash q$, which can be informally interpreted as

> If the program starts in circumstances satisfying p, it will terminate in a state satisfying q.

The design $p \vdash q$ is semantically equivalent, for example, to the Morgan specification statement [23]

$$v : [p, q]$$

or the B Abstract Machine Notation (AMN) [1] substitution

$$\mathsf{PRE}\ p\ \mathsf{THEN}\ \mathsf{ANY}\ v'\ \mathsf{WHERE}\ q\ \mathsf{THEN}\ v := v'\ \mathsf{END}\ \mathsf{END}$$

In particular, for convenience the following designs have their own special representations:

skip	$=_{\mathsf{def}}$	$\mathrm{true} \vdash v' = v$	
abort	$=_{\mathsf{def}}$	$\mathrm{false} \vdash \mathrm{true}$	
magic	$=_{\mathsf{def}}$	$\mathrm{true} \vdash \mathrm{false}$	
$[p]$	$=_{\mathsf{def}}$	$\mathrm{true} \vdash p \wedge v' = v$	guard
$\{p\}$	$=_{\mathsf{def}}$	$p \vdash v' = v$	assertion

Our operational intuitions about these primitive statements are as follows:

- skip terminates without changing the state;
- abort is completely chaotic and may even fail to terminate;
- magic behaves everywhere miraculously;
- The guard statement $[p]$ behaves like skip from states where p holds, while from states where p doesn't hold it behaves miraculously. We note that magic could equivalently be expressed as $[\mathrm{false}]$.
- The $\mathsf{assertion}$ statement $\{p\}$ also behaves like skip from states where p holds, but from states where p doesn't hold it behaves like abort. We note that abort could equivalently be expressed as $\{\mathrm{false}\}$.

Our program calculus has the following binary operators:

nondeterministic choice	$s \sqcap t$
sequential composition	$s ; t$

We usually omit the " ; " in a sequential composition when no confusion arises, relying instead on simple juxtaposition. Thus we write, for example, simply $s\,t$ instead of $s ; t$. We give sequential composition a higher precedence than $\sqcap$. The familiar *conditional* construct can be expressed in terms of our primitive constructs as

$$\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t\ \mathsf{end} \quad =_{\mathsf{def}} \quad [b]\,s \sqcap [\neg b]\,t$$

As we have seen, in UTP each conjunctive program is modelled predicatively as an alphabetised binary relation between undashed before-state and dashed after-state variables which is "design-healthy", *i.e.* semantically equivalent to one in the form of a design. In particular $\sqcap$ is modelled by disjunction and sequential composition by alphabetised relational composition. Designs enjoy the following algebraic properties:

$s \sqcap t = t \sqcap s$	commutativity
$(s \sqcap t) \sqcap v = s \sqcap (t \sqcap v)$	associativity
$s \sqcap s = s$	idempotence
$s \sqcap \mathsf{magic} = s = \mathsf{magic} \sqcap s$	unit of choice
$s \sqcap \mathsf{abort} = \mathsf{abort} = \mathsf{abort} \sqcap s$	zero of choice
$(s\,t)\,v = s\,(t\,v)$	associativity
$s\ \mathsf{skip} = s = \mathsf{skip}\ s$	unit of composition
$\mathsf{abort}\ s = \mathsf{abort}$	left zero of composition
$\mathsf{magic}\ s = \mathsf{magic}$	left zero of composition

$$(s \sqcap t)\, v \;=\; s\, v \sqcap t\, v \qquad \text{distributivity}$$
$$s\,(t \sqcap v) \;=\; s\, t \sqcap s\, v \qquad \text{distributivity}$$

In addition we make use of the following specific properties of UTP designs, where a, b, c and p are simple conditions, *i.e.* predicates over the undashed state variables, and q is an alphabetised binary relation, *i.e.* a predicate relating before-states represented by undashed variables to after-states represented by dashed variables:

$$(p \vdash q)\,[b] \;=\; p \vdash q \wedge b' \tag{1}$$
$$\{b\}\,(p \vdash q) \;=\; p \wedge b \vdash q \tag{2}$$

The relationship between the refinement ordering on programs and nondeterministic choice is expressed by the following property:

$$s \sqsubseteq t \;\equiv\; s = s \sqcap t \tag{3}$$

The associativity, commutativity and idempotence of $\sqcap$ respectively guarantee that $\sqsubseteq$ is transitive, antisymmetric and reflexive, so ensuring that it is a partial order. Its bottom is abort and its top is magic:

$$\mathsf{abort} \sqsubseteq s \sqsubseteq \mathsf{magic} \tag{4}$$

It is easy to show that $\sqcap$ is a meet operator for $\sqsubseteq$:

$$(s \sqsubseteq t) \wedge (s \sqsubseteq u) \;\equiv\; s \sqsubseteq t \sqcap u \tag{5}$$

Composition and $\sqcap$ are monotonic with respect to $\sqsubseteq$, and the following refinement properties of designs are easily verified:

$$p \vdash p' \;\sqsubseteq\; \mathsf{skip} \tag{6}$$
$$a \vdash c' \;\sqsubseteq\; (a \vdash b')\ (b \vdash c') \tag{7}$$

This equality follows from the previous two properties and the monotonicity of composition with respect to $\sqsubseteq$:

$$(p \vdash p')\ (p \vdash p') \;=\; p \vdash p' \tag{8}$$

This trading rule captures the duality between assertions and guards:

$$\{p\}\, s \sqsubseteq t \;\equiv\; s \sqsubseteq [p]\, t \tag{9}$$

This rule captures the equivalence between two ways of expressing that a given program s is always guaranteed to terminate:

$$\mathsf{magic} \sqsubseteq s\ \mathsf{magic} \;\equiv\; \mathrm{true} \vdash \mathrm{true} \sqsubseteq s \tag{10}$$

The set of conjunctive programs over the state space spanned by state variable(s) v forms a complete lattice with respect to $\sqsubseteq$.

3.1 Fixed-point definitions in total correctness

We define the following unary operators as fixed points with respect to $\sqsubseteq$, where ν_{ref} and μ_{ref} denote respectively greatest fixed point and least fixed point:

$$s^* \quad =_{\mathsf{def}} \quad \nu_{\mathsf{ref}}\, x \;.\; s\, x \;\sqcap\; \mathsf{skip} \qquad \text{weak iteration} \tag{11}$$

$$s^\infty \quad =_{\mathsf{def}} \quad \mu_{\mathsf{ref}}\, x \;.\; s\, x \qquad \text{infinite repetition} \tag{12}$$

$$s^\omega \quad =_{\mathsf{def}} \quad \mu_{\mathsf{ref}}\, x \;.\; s\, x \;\sqcap\; \mathsf{skip} \qquad \text{strong iteration} \tag{13}$$

The Knaster-Tarski theorem [25] guarantees that the above fixed-point definitions are sound. We give these three unary operators a higher precedence than either sequential composition or $\sqcap$. Our operational intuition of them is as follows:

- s^* signifies zero or finitely more repetitions of s;
- s^∞ signifies infinite repetition of s;
- s^ω signifies arbitrary general (*i.e.* zero, finite or infinite) repetition of s.

The $*$, ∞ and ω operators are themselves monotonic with respect to $\sqsubseteq$. They are related by the property [21, Lemma 13]

$$s^\omega \quad = \quad s^* \;\sqcap\; s^\infty \tag{14}$$

The $*$, ∞ and ω operators enjoy the following standard pre- and post-fixpoint induction properties respectively of greatest and least fixed points *cf* [15, Laws 2.6L1 and 2.7L1]:

$$t \;\sqsubseteq\; s\, t \;\sqcap\; \mathsf{skip} \qquad \Rrightarrow \qquad t \;\sqsubseteq\; s^* \tag{15}$$

$$s\, t \;\sqsubseteq\; t \qquad \Rrightarrow \qquad s^\infty \;\sqsubseteq\; t \tag{16}$$

$$s\, t \;\sqcap\; \mathsf{skip} \;\sqsubseteq\; t \qquad \Rrightarrow \qquad s^\omega \;\sqsubseteq\; t \tag{17}$$

The following property is a consequence of the definition of s^*, see [3, Lemma 21.2]:

$$s^*\, t \qquad = \qquad \nu_{\mathsf{ref}}\, x \;.\; s\, x \;\sqcap\; t \tag{18}$$

3.2 Well-foundedness in total correctness

We describe a program s as *well-founded* if its infinite repetition is everywhere miraculous: that is, $s^\infty = \mathsf{magic}$. Also, if p is a condition on the state we describe a program s as *well-founded on* p if its infinite repetition is miraculous from all states which satisfy p: that is, if $[\,p\,]\, s^\infty = \mathsf{magic}$. Note that $[\,p\,]\, s^\infty$ is not the same as $([\,p\,]\, s)^\infty$, so saying s is well-founded on p is not the same as saying $[\,p\,]\, s$ is well-founded.

4 Loop Refinement in Total Correctness

Before we can formulate any loop refinement rule we must determine the formal meaning of the loop construct in question. The appropriate way to do so is to interpret the construct concerned as a recursive expression obtained by "unfolding" it, whose meaning is then taken as a least fixed-point with respect to total-correctness refinement. In this way, we define our while loop as follows:

$$\mathsf{while}\ b\ \mathsf{do}\ s\ \mathsf{end} \quad =_{\mathsf{def}} \quad \mu_{\mathsf{ref}}\ x\ .\ \mathsf{if}\ b\ \mathsf{then}\ s\,x\ \mathsf{else\ skip\ end}$$

Such a while-loop turns out to be closely related to strong-iteration. This is exposed by the following equality [3, Lemma 21.8]:

$$\mathsf{while}\ b\ \mathsf{do}\ s\ \mathsf{end} \quad = \quad ([\,b\,]\,s)^{\omega}\,[\neg\,b] \tag{19}$$

4.1 A total-correctness loop-refinement rule

Identifying an invariant condition and a variant expression by which to demonstrate that a particular while loop refines a given abstract specification is a familiar and long-established technique in formal development. There are numerous presentations of such a technique: among the very first must be those of Floyd [12] expressed using flowcharts and of Hoare [14] —the latter in partial correctness only, while more recent ones include those in [23, 17, 1]. In our calculus the technique can be expressed succinctly by the following rule:

$p \ \vdash\ p' \wedge \neg\, b' \quad \sqsubseteq \quad \mathsf{while}\ b\ \mathsf{do}\ s\ \mathsf{end}$ TC Loop

provided

1. $p \ \vdash\ p' \ \sqsubseteq\ [\,b\,]\,s$
2. $[\,b\,]\,s$ is well-founded on p.

Proviso 1 requires that, under the assumption of the loop guard b, the loop body, s, preserves the loop invariant p; its purpose is to ensure that if and when the loop terminates the resulting final state will satisfy p. Proviso 2 requires that infinite repetition of s under the assumption of b is impossible starting from a state which satisfies p; its purpose is to guarantee termination from any starting state which satisfies the loop invariant p.

There is no explicit notion of a variant in this formulation of the rule. However, a standard technique for demonstrating well-foundedness of a computation is to formulate a variant expression, *i.e.* a natural-number-valued or similar well-founded-domain-valued expression over the state space which is strictly decreased by that computation. Our formulation of the rule therefore neatly separates our twin concerns of correctness and termination. We verify the rule by the following reasoning in our total-correctness calculus.

Proof:

$p \vdash p' \wedge \neg\, b' \quad \sqsubseteq \quad \mathsf{while}\ b\ \mathsf{do}\ s\ \mathsf{end}$

$\Leftarrow\!\!\!\!\equiv$ { (1), (19) }

$$
\begin{array}{ll}
 & (p \vdash p')\,[\neg\, b] \quad \sqsubseteq \quad ([\,b\,]\,s)^{\omega}\,[\neg\, b] \\
\Leftarrow & \quad \{ \text{ monotonicity of seq comp wrt } \sqsubseteq \} \\
 & p \vdash p' \quad \sqsubseteq \quad ([\,b\,]\,s)^{\omega} \\
\equiv & \quad \{ \text{ (14) } \} \\
 & p \vdash p' \quad \sqsubseteq \quad ([\,b\,]\,s)^{*} \;\sqcap\; ([\,b\,]\,s)^{\infty} \\
\equiv & \quad \{ \text{ (2) } \} \\
 & \{p\}(p \vdash p') \quad \sqsubseteq \quad ([\,b\,]\,s)^{*} \;\sqcap\; ([\,b\,]\,s)^{\infty} \\
\equiv & \quad \{ \text{ (9) } \} \\
 & p \vdash p' \quad \sqsubseteq \quad [p]\,(([\,b\,]\,s)^{*} \;\sqcap\; ([\,b\,]\,s)^{\infty}) \\
\equiv & \quad \{ \text{ distributivity } \} \\
 & p \vdash p' \quad \sqsubseteq \quad [p]\,([\,b\,]\,s)^{*} \;\sqcap\; [p]\,([\,b\,]\,s)^{\infty} \\
\equiv & \quad \{ \text{ Proviso 2 } \} \\
 & p \vdash p' \quad \sqsubseteq \quad [p]\,([\,b\,]\,s)^{*} \;\sqcap\; \mathsf{magic} \\
\equiv & \quad \{ \mathsf{magic} \text{ is unit of } \sqcap \} \\
 & p \vdash p' \quad \sqsubseteq \quad [p]\,([\,b\,]\,s)^{*} \\
\equiv & \quad \{ \text{ (9) } \} \\
 & \{p\}(p \vdash p') \quad \sqsubseteq \quad ([\,b\,]\,s)^{*} \\
\equiv & \quad \{ \text{ (2) } \} \\
 & p \vdash p' \quad \sqsubseteq \quad ([\,b\,]\,s)^{*} \\
\Leftarrow & \quad \{ \text{ (15) } \} \\
 & p \vdash p' \quad \sqsubseteq \quad [\,b\,]s\,(p \vdash p') \;\sqcap\; \mathsf{skip} \\
\equiv & \quad \{ \text{ (5), (6) } \} \\
 & p \vdash p' \quad \sqsubseteq \quad [\,b\,]s\,(p \vdash p') \\
\equiv & \quad \{ \text{ (8) } \} \\
 & (p \vdash p')\,(p \vdash p') \quad \sqsubseteq \quad [\,b\,]s\,(p \vdash p') \\
\Leftarrow & \quad \{ \text{ monotonicity of seq comp wrt } \sqsubseteq \} \\
 & p \vdash p' \quad \sqsubseteq \quad [\,b\,]s \qquad\qquad \{ \text{ Proviso 1 } \} \qquad\qquad \Box
\end{array}
$$

So much for total correctness. In the next sections we move on to consider general correctness.

5 A General-correctness Program Calculus

Whereas the total-correctness semantics of programs is captured in predicate-transformer terms by the weakest-precondition (wp) transformer alone, for general correctness the weakest-liberal-precondition (wlp) transformer is also required [7, 16, 24]. In [9] prescriptions were introduced as the general-correctness counterparts of Hoare and He's total-correctness designs, and their properties have since been further explored in [6] and [13]. Let v be the list of state variables of the state space and ok be an additional auxiliary boolean variable with the same interpretation as that already described in Section 3 for designs. Then a

prescription is an alphabetised relation over (v, ok, v', ok') whose predicate can be expressed in the form

$$(ok \wedge p \;\Rightarrow\; ok') \;\wedge\; (ok' \;\Rightarrow\; q \wedge ok)$$

where p and q are subsidiary predicates not referring to ok or ok'. We abbreviate this as $p \Vdash q$. If p is simply a *condition* —*i.e.* it constrains only the undashed state variables v— then we call $p \Vdash q$ a *normal* prescription. From here on we only consider normal prescriptions. Intuitively, we can then interpret $p \Vdash q$ operationally as follows:

> If the computation starts from an initial state satisfying p it must inevitably terminate; moreover, if it terminates —whether inevitably from an initial state satisfying p or just fortuitously from any other— then q will be satisfied, and the computation must certainly have started.

General-correctness specifications are more expressive than total-correctness ones because among other things they can express non-termination requirements as well as termination requirements. For example, the extreme prescription $\mathsf{false} \Vdash \mathsf{false}$ describes a computation which can start from any initial state but must then never terminate. Our general-correctness calculus like our earlier total-correctness one models conjunctive programs. Its basic statements are prescriptions. In particular, for convenience the following prescriptions have these special representations, where v is the list of all state variables and p is a predicate on the state:

skip	$=_{\mathsf{def}}$	$\mathsf{true} \Vdash v' = v$	
abort	$=_{\mathsf{def}}$	$\mathsf{false} \Vdash \mathsf{true}$	
magic	$=_{\mathsf{def}}$	$\mathsf{true} \Vdash \mathsf{false}$	
$[p]$	$=_{\mathsf{def}}$	$\mathsf{true} \Vdash p \wedge v' = v$	guard
$\{p\}$	$=_{\mathsf{def}}$	$p \Vdash v' = v$	assertion
loop	$=_{\mathsf{def}}$	$\mathsf{false} \Vdash \mathsf{false}$	

Our operational intuitions about the first five of these primitive statements are the same as before. The last one loop is particular to general correctness and represents an infinite loop which never terminates. Our new calculus has the same binary operators $\sqcap$ and " ; " modelling nondeterministic choice and sequential composition, all of whose algebraic properties are the same as before, and its conditional construct is defined in the same way. We define our refinement ordering $\sqsubseteq$ exactly as in total correctness. Properties (4), giving us the bottom and top of our $\sqsubseteq$ ordering, and (9), our trading rule for guards and assertions, remain the same in general correctness as in total correctness. We make use of the following properties of UTP prescriptions, which are the direct counterparts of those we saw earlier for designs:

$$(p \Vdash q) \;;\; [b] \quad = \quad p \Vdash q \wedge b' \tag{20}$$

$$\{b\} \;;\; (p \Vdash q) \quad = \quad p \wedge b \Vdash q \tag{21}$$

$$a \Vdash c' \quad \sqsubseteq \quad (a \Vdash b') \;;\; (b \Vdash c') \tag{22}$$

5.1 Splitting a prescription

A general-correctness specification can be projected without loss of information into its total-correctness and partial-correctness components. We adopt a subscripting convention that, if u denotes a general-correctness specification, we write u_{par} and u_{tot} to denote respectively its partial-correctness and total-correctness components. In terms of prescriptions, the prescription $p \Vdash q$ can be projected into its total-correctness and partial-correctness components as follows:

$$\begin{array}{lll} (p \Vdash q)_{\mathsf{tot}} & =_{\mathsf{def}} & p \vdash q \\ (p \Vdash q)_{\mathsf{par}} & =_{\mathsf{def}} & q \end{array}$$

The following equivalence relates refinement of prescriptions in general correctness to refinement of designs in total correctness and relations in partial correctness:

$$p \Vdash q \;\sqsubseteq\; u \qquad \equiv \qquad (p \vdash \mathrm{true} \;\sqsubseteq\; u_{\mathsf{tot}}) \;\wedge\; (q \;\sqsubseteq\; u_{\mathsf{par}}) \tag{23}$$

It can easily be verified by unpacking the prescription and designs into their underlying predicative forms and interpreting $\sqsubseteq$ as reverse implication.

5.2 Fixed-point definitions in general correctness

We introduce a different ordering with respect to which we make our fixed-point definitions in general correctness, namely the Egli-Milner approximation ordering $\leq_{\mathsf{em}}$. This has long been recognised as the appropriate ordering for the interpretation of recursions in general correctness [24]. We define it as follows:

$$s \leq_{\mathsf{em}} t \quad =_{\mathsf{def}} \quad (s_{\mathsf{tot}} \sqsubseteq t_{\mathsf{tot}}) \;\wedge\; (t_{\mathsf{par}} \sqsubseteq s_{\mathsf{par}}) \tag{24}$$

Expressing this equivalently in terms of prescriptions we have

$$p_1 \Vdash q_1 \;\leq_{\mathsf{em}}\; p_2 \Vdash q_2 \quad \equiv \quad [q_1 \Rightarrow q_2] \wedge [p_1 \Rightarrow p_2] \wedge [p_1 \Rightarrow (q_1 \Leftrightarrow q_2)] \tag{25}$$

where the square brackets [...] on the right-hand side denote universal quantification over the alphabet of variables. The set of conjunctive programs over the state space spanned by state variable(s) v forms a complete partial order (cpo) with respect to $\leq_{\mathsf{em}}$. The bottom of the $\leq_{\mathsf{em}}$ ordering is loop, but we note that $\leq_{\mathsf{em}}$ has no overall top, because any everywhere-terminating program is maximal. Our composition and $\sqcap$ operators are monotonic with respect to $\leq_{\mathsf{em}}$. We define the following unary operators as least fixed points with respect to $\leq_{\mathsf{em}}$:

s^{∞}	$=_{\mathsf{def}}$	$\mu_{\mathsf{em}}\, x \;.\; s\, x$	infinite repetition
s^{ω}	$=_{\mathsf{def}}$	$\mu_{\mathsf{em}}\, x \;.\; s\, x \;\sqcap\; \mathsf{skip}$	strong iteration

Because $\leq_{\mathsf{em}}$ induces a cpo rather than a complete lattice we need the generalisation of Tarski's fixed-point theorem given in [24] to ensure that the above definitions are sound. Our operational intuitions of them remain the same as before, namely that

- s^∞ signifies infinite repetition of s;
- s^ω signifies arbitrary general (*i.e.* finite or infinite) repetition of s.

However, these operational interpretations must be made in the context of general correctness rather than total correctness. This means, for example, than skip^∞ is loop rather than abort as it would be in total correctness. Our use of the Egli-Milner ordering $\leq_{\mathsf{em}}$ rather than the refinement ordering $\sqsubseteq$ in the formal definitions of these fixed points reflects this.

Our ∞ and ω operators here also again enjoy the standard post-fixpoint induction property of least fixed points, although the ordering in question is now $\leq_{\mathsf{em}}$ rather than $\sqsubseteq$:

$$s\,t \quad \leq_{\mathsf{em}} \quad t \qquad \Rrightarrow \qquad s^\infty \quad \leq_{\mathsf{em}} \quad t \tag{26}$$

$$s\,t \;\sqcap\; \mathsf{skip} \quad \leq_{\mathsf{em}} \quad t \qquad \Rrightarrow \qquad s^\omega \quad \leq_{\mathsf{em}} \quad t \tag{27}$$

5.3 Well-foundedness in general correctness

For general correctness we again describe a program s as *well-founded* if its infinite repetition is everywhere miraculous: that is, $s^\infty = \mathsf{magic}$. Also, if p is a condition on the state we describe a program s as *well-founded on* p if its infinite repetition is miraculous from all states which satisfy p: that is, $[\,p\,]\,s^\infty = \mathsf{magic}$. In fact, a general-correctness program is well-founded exactly when its projection in total correctness is well-founded, so there is no need in practice to distinguish well-foundedness in general correctness from that in total correctness.

5.4 Partitioning the Egli-Milner ordering

Let $E(x)$ be any expression in our general-correctness calculus built using any of its primitive statements and its binary operators of $\sqcap$ and ";" as well as the variable x denoting any general-correctness computation. Then $\lambda\, x\,.\,E(x)$ is monotonic with respect to $\leq_{\mathsf{em}}$ and therefore has a well-defined $\leq_{\mathsf{em}}$-least fixed point $\mu_{\mathsf{em}}\, x\,.\,E(x)$. Moreover, the definition in (24) of the Egli-Milner ordering ensures that a sufficient condition for a general-correctness computation u to be such a fixed point is that $u_{\mathsf{tot}} = \mu_{\mathsf{ref}}\, x\,.\,E(x)$ and $u_{\mathsf{par}} = \nu_{\mathsf{ref}}\, x\,.\,E(x)$. That is to say, we have that

$$u_{\mathsf{tot}} = \mu_{\mathsf{ref}}\, x\,.\,E(x) \;\wedge\; u_{\mathsf{par}} = \nu_{\mathsf{ref}}\, x\,.\,E(x) \qquad \Rrightarrow \qquad u = \mu_{\mathsf{em}}\, x\,.\,E(x) \tag{28}$$

If we now apply property (28) above to the case where $E(x)$ is $([\,b\,]\,s\,x \sqcap \mathsf{skip})$ and apply the definitions of strong and weak iteration, we obtain

$$u_{\mathsf{tot}} = ([\,b\,]\,s)^\omega \;\wedge\; u_{\mathsf{par}} = ([\,b\,]\,s)^* \qquad \Rrightarrow \qquad u = ([\,b\,]\,s)^\omega \tag{29}$$

Notice that because u_{tot} is a total-correctness computation the first occurrence of $([\,b\,]\,s)^\omega$ in (29) above is interpreted in total correctness, whereas its second occurrence is interpreted in general correctness because u is a general-correctness computation.

6 Loop Refinement in General Correctness

As in the case of total correctness, in general correctness we again interpret a while loop by "unfolding" it, and its meaning is then again taken as a least fixed-point, although "least" now means least with respect to the Egli-Milner ordering $\leq_{\mathsf{em}}$:

$$\mathsf{while}\ b\ \mathsf{do}\ s\ \mathsf{end} \quad =_{\mathsf{def}} \quad \mu_{\mathsf{em}}\, x\ .\ \mathsf{if}\ b\ \mathsf{then}\ s\, x\ \mathsf{else\ skip\ end}$$

Again our while-loop turns out to be intimately related to our strong-iteration construct via the following equality, which is analogous to property (19) for the total-correctness case:

$$\mathsf{while}\ b\ \mathsf{do}\ s\ \mathsf{end} \quad = \quad ([\,b\,]\,s)^\omega\,[\neg\, b] \tag{30}$$

We must remember, though, that both the while loop and the strong iteration in (30) have different meanings from their total-correctness counterparts in (19), because they are defined with respect to $\leq_{\mathsf{em}}$ rather than $\sqsubseteq$. Nevertheless property (30) can be proved in a similar manner to (19) in [3, Lemma 21.8]. This requires only that the Fusion Theorem cited in the associated [3, Lemma 21.2] is replaced by the Transfer Lemma[5] of [2].

6.1 A general-correctness loop-refinement rule

One might reasonably ask why we need a new loop-refinement rule for general correctness at all. After all, can we not simply translate the existing total-correctness rule directly into a general-correctness setting? Indeed we can, giving us the following rule:

$$p \Vdash (p \Rightarrow p' \wedge \neg\, b') \quad \sqsubseteq \quad \mathsf{while}\ b\ \mathsf{do}\ s\ \mathsf{end}$$

provided

1. $(p \Rightarrow p') \ \sqsubseteq\ [\,b\,]\,s$
2. $[\,b\,]\,s$ is well-founded on p.

However, such a rule is inadequate for proving any refinements involving required non-termination. The simplest of these, remembering that loop is the prescription false $\Vdash$ false, would be

[5] The proof also relies on the fact that the relevant "transfer" function used in [3, Lemma 21.2], namely $(\lambda\, X\, .\, X\,;\, T)$, is continuous with respect to $\leq_{\mathsf{em}}$. Fortunately, this *is* so, because, as observed in [24], sequential composition is indeed continuous with respect to $\leq_{\mathsf{em}}$ in its left argument.

$$\text{false} \Vdash \text{false} \quad \sqsubseteq \quad \textsf{while true do skip end}$$

Any putative general-correctness loop-refinement rule that cannot be applied to verify even so simple a refinement as this is hardly worth our consideration at all. We therefore propose instead the following general-correctness while-loop refinement rule:

$$p \Vdash (j \Rightarrow j' \wedge \neg\, b') \quad \sqsubseteq \quad \textsf{while } b \textsf{ do } s \textsf{ end} \qquad \textsf{GC Loop 1}$$

provided

1. $(j \Rightarrow j') \quad \sqsubseteq \quad [\, b \,]\, s$
2. $[\, b \,]\, s$ is well-founded on p.

We note that Proviso 1 involves only partial-correctness refinement because the relation $j \Rightarrow j'$ on its left-hand side is simply a before-after relation on the state space. As far as we are aware no such practical general-correctness loop refinement rule has been formulated before. An interesting feature of the rule is that the loop invariant j is entirely separate from the termination precondition p. We verify the rule by the following reasoning in our general-correctness calculus.

Proof:

$$\begin{array}{ll}
 & p \Vdash (j \Rightarrow j' \wedge \neg\, b') \quad \sqsubseteq \quad \textsf{while } b \textsf{ do } s \textsf{ end} \\
\equiv & \quad \{\ (30)\ \} \\
 & p \Vdash (j \Rightarrow j' \wedge \neg\, b') \quad \sqsubseteq \quad ([\, b \,]\, s)^{\omega}\, [\neg\, b] \\
\Leftarrow & \quad \{\ \text{relax postcondition}\ \} \\
 & p \Vdash (j \Rightarrow j') \wedge \neg\, b' \quad \sqsubseteq \quad ([\, b \,]\, s)^{\omega}\, [\neg\, b] \\
\equiv & \quad \{\ (20)\ \} \\
 & (p \Vdash j \Rightarrow j')\, [\neg\, b] \quad \sqsubseteq \quad ([\, b \,]\, s)^{\omega}\, [\neg\, b] \\
\Leftarrow & \quad \{\ \text{monotonicity of seq comp wrt } \sqsubseteq\ \} \\
 & p \Vdash (j \Rightarrow j') \quad \sqsubseteq \quad ([\, b \,]\, s)^{\omega} \\
\Leftarrow & \quad \{\ (23)\ \} \\
 & p \vdash \text{true} \ \sqsubseteq \ ([\, b \,]\, s)^{\omega} \quad \wedge \quad (j \Rightarrow j') \ \sqsubseteq \ (([\, b \,]\, s)^{\omega})_{\textsf{par}} \\
\equiv & \quad \{\ (29)\ \} \\
 & p \vdash \text{true} \ \sqsubseteq \ ([\, b \,]\, s)^{\omega} \quad \wedge \quad (j \Rightarrow j') \ \sqsubseteq \ ([\, b \,]\, s)^{*} \qquad \textsf{A}
\end{array}$$

We now prove each of the two conjuncts of A above separately. First, we prove the right-hand conjunct:

$$\begin{array}{ll}
 & (j \Rightarrow j') \ \sqsubseteq \ ([\, b \,]\, s)^{*} \\
\Leftarrow & \quad \{\ (15)\ \} \\
 & (j \Rightarrow j') \ \sqsubseteq \ [\, b \,]\, s\, (j \Rightarrow j') \sqcap \textsf{skip} \\
\Leftarrow & \quad \{\ j \Rightarrow j' \ \sqsubseteq \ \textsf{skip}\ \} \\
 & (j \Rightarrow j') \ \sqsubseteq \ [\, b \,]\, s\, (j \Rightarrow j') \\
\equiv & \quad \{\ (j \Rightarrow j')(j \Rightarrow j') \ = \ (j \Rightarrow j')\ \} \\
 & (j \Rightarrow j')(j \Rightarrow j') \ \sqsubseteq \ [\, b \,]\, s\, (j \Rightarrow j') \\
\Leftarrow & \quad \{\ \text{monotonicity of seq comp wrt } \sqsubseteq\ \}
\end{array}$$

$(j \Rightarrow j') \sqsubseteq [b]\,s$ { Proviso 1 }

Secondly, we prove the left-hand conjunct of A:

$p \vdash \text{true} \sqsubseteq ([b]\,s)^\omega$

$\equiv$ { (14) }

$p \vdash \text{true} \sqsubseteq ([b]\,s)^* \sqcap ([b]\,s)^\infty$

$\equiv$ { (2), noting that $p \vdash \text{true}$ is $p \wedge \text{true} \vdash \text{true}$ }

$\{p\}(\text{true} \vdash \text{true}) \sqsubseteq ([b]\,s)^* \sqcap ([b]\,s)^\infty$

$\equiv$ { (9) }

$\text{true} \vdash \text{true} \sqsubseteq [p](([b]\,s)^* \sqcap ([b]\,s)^\infty)$

$\equiv$ { distributivity }

$\text{true} \vdash \text{true} \sqsubseteq [p]([b]\,s)^* \sqcap [p]([b]\,s)^\infty$

$\equiv$ { Proviso 2 }

$\text{true} \vdash \text{true} \sqsubseteq [p]([b]\,s)^* \sqcap \mathsf{magic}$

$\equiv$ { magic is unit of $\sqcap$ }

$\text{true} \vdash \text{true} \sqsubseteq [p]([b]\,s)^*$

$\equiv$ { (10) }

$\mathsf{magic} \sqsubseteq [p]([b]\,s)^*\,\mathsf{magic}$

$\equiv$ { (9) }

$\{p\}\,\mathsf{magic} \sqsubseteq ([b]\,s)^*\,\mathsf{magic}$

$\equiv$ { (18) }

$\{p\}\,\mathsf{magic} \sqsubseteq \nu_{\mathsf{ref}}\,x \,.\, [b]\,s\,x \sqcap \mathsf{magic}$

$\equiv$ { magic is unit of $\sqcap$ }

$\{p\}\,\mathsf{magic} \sqsubseteq \nu_{\mathsf{ref}}\,x \,.\, [b]\,s\,x$

$\Longleftarrow$ { $\mu f \sqsubseteq \nu f$, transitivity of $\sqsubseteq$ }

$\{p\}\,\mathsf{magic} \sqsubseteq \mu_{\mathsf{ref}}\,x \,.\, [b]\,s\,x$

$\equiv$ { (12) }

$\{p\}\,\mathsf{magic} \sqsubseteq ([b]\,s)^\infty$

$\equiv$ { (9) }

$\mathsf{magic} \sqsubseteq [p]([b]\,s)^\infty$

$\equiv$ { (4) }

$[p]([b]\,s)^\infty = \mathsf{magic}$ { Proviso 2 } □

6.2 An application of the general-correctness loop rule

Concert In previous works such as [11, 8, 10] Dunne *et al.* have already described combining two general-correctness computations "in concert" under a termination pact by which the overall result, if any, of their parallel executions on separate copies of the state space is determined entirely by whichever of them happens to terminate first. Such a concert operator # can be simply defined in terms of prescriptions by

$$(p_1 \Vdash q_1) \;\#\; (p_2 \Vdash q_2) \;=_{\mathsf{def}}\; (p_1 \vee p_2) \Vdash (q_1 \vee q_2) \tag{31}$$

What may at first sight seem surprising about the definition of concert here in (31) is that the preconditions of the two prescriptions on the left-hand side appear in the right-hand prescription to have lost their particular association with their respective postconditions. This is indeed so, and reflects the fact that even if the concerted execution takes place from an initial state where only, say, its first component $p_1 \Vdash q_1$ is guaranteed to terminate, it is still possible that its other component $p_2 \Vdash q_2$ will terminate first entirely fortuitously. In such a case the result delivered by $p_2 \Vdash q_2$ must under general correctness still satisfy q_2 despite its termination being fortuitous rather than guaranteed. This is in contrast to the analogous situation in total correctness where the result delivered by any fortuitous termination of a design $p \vdash q$ from an initial state outside its precondition p is unconstrained by its postcondition q.

Our concert operator $\#$ is both well-defined (because $p \Vdash q$ is a canonical form for prescriptions[6]) and monotonic on the refinement ordering $\sqsubseteq$ on prescriptions.

An example refinement using concert To illustrate the use of our concert operator $\#$ we consider an impoverished computing environment in which values can be tested only for equality (or inequality) with zero, and where variables can only be modified by incrementing or decrementing by one. In such an austere environment even something as simple as setting an integer variable to zero, as specified by the prescription $\text{true} \Vdash x' = 0$, poses a considerable programming challenge. However, rising to that challenge we observe that

$$\begin{array}{ll} & \text{true} \Vdash x' = 0 \\ = & \{ \text{ integer property } \} \\ & (x \leq 0 \lor x \geq 0) \Vdash x' = 0 \\ = & \{ \text{ defn of } \# \ \} \\ & (x \leq 0 \Vdash x' = 0) \quad \# \quad (x \geq 0 \Vdash x' = 0) \end{array}$$

Interestingly, the two concerted specifications above can each be implemented by a while loop within our austere computing environment. Intuitively

$$x \leq 0 \Vdash x' = 0 \quad \sqsubseteq \quad \mathsf{while}\ x \neq 0\ \mathsf{do}\ x := x + 1\ \mathsf{end} \tag{32}$$

$$x \geq 0 \Vdash x' = 0 \quad \sqsubseteq \quad \mathsf{while}\ x \neq 0\ \mathsf{do}\ x := x - 1\ \mathsf{end} \tag{33}$$

This means that we can fulfil our original requirement to set x to zero within the constraints imposed by our impoverished computing environment by executing the two loops above in concert on separate copies of the state space. But how do we verify these putative refinements (32) and (33) formally? In the case of (32) we do so by applying GC Loop 1 with p as $x \leq 0$, j as true, b as $x \neq 0$ and s as $x := x + 1$. This gives us an obligation to discharge the provisos

[6] That is to say, two prescriptions $p_1 \Vdash q_1$ and $p_2 \Vdash q_2$ are equal if and only if $p_1 = p_2$ and $q_1 = q_2$, as shown in [9].

1. $(\text{true} \Rightarrow \text{true}) \sqsubseteq [x \neq 0]\, x := x + 1$

and

2. $[x \neq 0]\, x := x + 1$ is well-founded on $x \leq 0$,

which are both trivial. In the case of (33), on the other hand, we again apply GC Loop 1 but this time with p as $x \geq 0$, j as true, b as $x \neq 0$ and s as $x := x - 1$. This then gives us an obligation to discharge the provisos

1. $(\text{true} \Rightarrow \text{true}) \sqsubseteq [x \neq 0]\, x := x - 1$

and

2. $[x \neq 0]\, x := x - 1$ is well-founded on $x \geq 0$,

which again are both trivial.

What the example shows We would stress that the purpose of the above example is not the refinement *per se*, which is —in operational terms at least— obviously quite trivial. Rather, it lies in the refinement's formal verification. This is simply not possible within the confines of total correctness. Only with our new general-correctness loop-refinement rule GC Loop 1 can we establish the refinement's correctness. The point of the example is therefore to illustrate the necessity of such a rule in verifying even such "obvious" refinements.

6.3 Another general-correctness loop rule

We can derive more specialised general-correctness loop rules from our primary rule GC Loop 1. For example if we simply re-write it with $\neg\, j$ replacing j we obtain this version of the rule:

$p \Vdash (\neg\, j \Rightarrow \neg\, j' \wedge \neg\, b') \quad \sqsubseteq \quad \textsf{while } b \textsf{ do } s \textsf{ end}$ GC Loop 1a

provided

1. $(\neg\, j \Rightarrow \neg\, j') \sqsubseteq [b]\, s$
2. $[b]\, s$ is well-founded on p.

Now combining GC Loop 1 and GC Loop 1a we obtain this further version:

$p \Vdash (j \Leftrightarrow j') \wedge \neg\, b' \quad \sqsubseteq \quad \textsf{while } b \textsf{ do } s \textsf{ end}$ GC Loop 1b

provided

1. $(j \Leftrightarrow j') \sqsubseteq [b]\, s$
2. $[b]\, s$ is well-founded on p.

If we then strengthen our provisos with the requirement that $\neg\, j \Rightarrow b$, from which it follows immediately that $\neg\, j' \Rightarrow b'$, we can then further simplify the postcondition of the prescription on the left-hand side of GC Loop 1b to obtain the following rule:

$p \Vdash j \wedge \neg\, b' \quad \sqsubseteq \quad \textsf{while } b \textsf{ do } s \textsf{ end}$ GC Loop 2

provided

1. $\neg\, j \Rightarrow b$
2. $(j \Leftrightarrow j') \quad \sqsubseteq \quad [\, b \,]\, s$
3. $[\, b \,]\, s$ is well-founded on p.

In the next subsection we illustrate the use of this rule by applying it to verify another "intuitively obvious" general-correctness refinement.

6.4 An application of the GC Loop 2 rule

A prescription of the particular form $p \Vdash (p \wedge q)$, where as usual p is a condition on the the initial state v and q is a binary relation on $\{v, v'\}$, has a commitment which demands that the initial state of any execution which terminates must have satisfied p. It therefore has the following interesting operational interpretation:

> From any initial state which satisfies p the program must terminate in a final state which satisfies q, whereas from any other initial state the program **must not terminate**.

From our operational intuition it therefore seems obvious that the while loop

$$\mathsf{while}\ \ x \neq 0\ \ \mathsf{do}\ \ x := x - 1\ \ \mathsf{end}$$

which we saw earlier will terminate in a final state with $x = 0$ when started from any initial state where $x \geq 0$, whereas it will fail to terminate from any other initial state. In other words, it implements the prescription

$$x \geq 0 \ \Vdash\ (x \geq 0 \wedge x' = 0)\,.$$

Yet our GC Loop 1 rule cannot be applied directly to verify such a refinement because the above prescription doesn't match the form $p \Vdash (j \Rightarrow j' \wedge \neg\, b')$. On the other hand, by setting p and j both to $x \geq 0$ and b to $x \neq 0$ it does match the form $p \Vdash j \wedge \neg\, b'$ of our GC Loop 2 rule. Moreover, all three of this rule's provisos, namely

1. $\neg\, (x \geq 0) \Rightarrow x \neq 0$,
2. $(x \geq 0 \Leftrightarrow x' \geq 0) \quad \sqsubseteq \quad [x \neq 0]\, x := x - 1$

and

3. $[x \neq 0]\, x := x - 1$ is well-founded on $x \geq 0$,

are then satisfied. Hence we can apply GC Loop 2 to verify this implementation.

7 Conclusion

We have presented a calculus for reasoning about programs in total correctness, and demonstrated its utility in verifying succinctly the familiar loop-invariant rule for refining a specification in total correctness by a while loop. We have also presented an analogous calculus for reasoning about programs in general correctness, which we then used to verify our new loop-invariant rule for refining

a specification in general correctness by a while loop. We believe our verification proofs of our rules demonstrate that our algebraically-inspired calculi provide an apt framework for reasoning about such rules.

Finally, it is perhaps worth noting that we are not the only ones to have espoused an algebraic style in reasoning about general correctness. In [22], for example, Möller and Struth apply a notably abstract algebraic approach to reasoning about wp and wlp. It would certainly be interesting to explore further the relationship of their work to ours.

Acknowledgements

We are grateful to Walter Guttmann for feedback on the work-in-progress extended abstract which preceded this paper, to Roland Backhouse for directing us to Apt and Plotkin's Transfer Lemma in [2], and to the referees of the original review draft of this paper whose insightful comments we have endeavoured to address in this final version. The second author's research was supported by Australian Research Council (ARC) Discovery Grant DP0558408, *Analysing and generating fault-tolerant real-time systems* and the EPSRC-funded Trustworthy Ambient Systems (TrAmS) Platform Project.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. K.R. Apt and G.D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, 1986.
3. R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag New York, 1998.
4. E. Cohen. Separation and reduction. In R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 45–59. Springer, 2000.
5. J.H. Conway. *Regular Algebra and Finite Machines.* Chapman & Hall, 1971.
6. M. Deutsch and M.C. Henson. A relational investigation of UTP designs and prescriptions. In S.E. Dunne and W.J. Stoddart, editors, *Unifying Theories of Programming: First International Symposium, UTP 2006*, volume 4010 of *Lecture Notes in Computer Science*, pages 101–122. Springer Verlag, 2006.
7. E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer Berlin, 1990.
8. S.E. Dunne. Abstract commands: a uniform notation for specifications and implementations. In C.J. Fidge, editor, *Computing: The Australasian Theory Symposium 2001*, volume 42 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 2001. http://www.elsevier.nl/locate/entcs.
9. S.E. Dunne. Recasting Hoare and He's unifying theory of programs in the context of general correctness. In A. Butterfield, G. Strong, and C. Pahl, editors, *Proceedings of the 5th Irish Workshop in Formal Methods, IWFM 2001*, Workshops in Computing. British Computer Society, 2001. http://ewic.bcs.org/conferences/2001/5thformal/papers.

10. S.E. Dunne. Junctive compositions of specifications in Total and General Correctness. In J. Derrick, E. Boiten, J.C.P. Woodcock, and J. von Wright, editors, *Refine 2002: The BCS FACS Refinement Workshop*, volume 70(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science BV, 2002. http://www.elsevier.nl/locate/entcs.
11. S.E. Dunne, W.J. Stoddart, and A.J. Galloway. Specification and refinement in general correctness. In A. Evans, D. Duke, and A. Clark, editors, *Proceedings of the 3rd Northern Formal Methods Workshop*. BCS Electronic Workshops in Computing, 1998. http://www.ewic.org.uk/ewic/workshop/view.cfm/NFM-98.
12. R.W. Floyd. Assigning meanings to programs. *Proceedings of Symposia in Applied Mathematics*, 19:19–32, 1967.
13. W. Guttmann and B. Möller. Modal design algebra. In S.E. Dunne and W.J. Stoddart, editors, *Unifying Theories of Programming: First International Symposium, UTP 2006*, volume 4010 of *Lecture Notes in Computer Science*, pages 236–256. Springer Verlag, 2006.
14. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.
15. C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.
16. D. Jacobs and D. Gries. General correctness: a unification of partial and total correctness. *Acta Informatica*, 22:67–83, 1985.
17. C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, second edition, 1990.
18. D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110:366–390, 1994.
19. D. Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems*, 19:427–443, 1999.
20. D. Kozen. On Kleene algebras and closed semirings. In B. Rovan, editor, *Proc. Mathematical Foundations of Computer Science*, volume 452 of *Lecture Notes in Computer Science*, pages 26–47. Springer, 2000.
21. L. Meinicke and I.J. Hayes. Algebraic reasoning for probabilistic action systems and while-loops. *Acta Informatica*, 45(5):321–382, 2008.
22. B. Möller and G. Struth. wp is wlp. In I. Düntsch, W. MacCaull, and M. Winter, editors, *Relational Methods in Computer Science*, volume 3929 of *Lecture Notes in Computer Science*, pages 855–874. Springer, 2005.
23. C.C. Morgan. *Programming from Specifications*. Prentice Hall International, second edition, 1994.
24. G. Nelson. A generalisation of Dijkstra's calculus. *ACM Transactions on Programmg Languages and Systems*, 11(4), 1989.
25. Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:2:285–309, 1955.
26. J. von Wright. From Kleene algebra to refinement algebra. In B. Möller and E. Boiten, editors, *Mathematics of Program Construction, 6th International Conference, MPC 2002*, volume 2386 of *Lecture Notes in Computer Science*, pages 233–262. Springer-Verlag, 2002.
27. J. von Wright. Towards a refinement algebra. *Science of Computer Programming*, 51:23–45, 2004.