Specification and Refinement in General Correctness

Steve Dunne\textsuperscript{1}, Andy Galloway\textsuperscript{2}, and Bill Stockart\textsuperscript{1}

\textsuperscript{1} School of Computing and Mathematics, University of Teesside
\textsuperscript{2} High Integrity Systems Engineering Group, University of York

Abstract. We augment B's existing total-correctness semantics of \textit{weakest precondition} (wp) with a partial-correctness semantics of \textit{weakest liberal precondition} (wlp). By so doing we achieve a general-correctness semantics for B operations which not only accords more fully with our natural computational intuition, but also extends the essential expressive capability of B's Generalised Substitution Language (GSL) to embrace a whole new class of operations called \textit{semi-decidable}, whose behaviour cannot be characterised in terms of total correctness alone. The ability to specify semi-decidable operations is important because a desired conventional operation may lend itself to implementation as a concurrent federation of semi-decidable operations co-operating under a mutual "termination pact". Indeed, computational constraints may render this the only viable implementation strategy. We call a generalised substitution invested with our general-correctness semantics an \textit{abstract command}. Our Abstract Command Language (ACL) is thus syntactically indistinguishable from the GSL, save for the introduction of one new composition operator, \textit{concert}, expressing a "termination pact" between two concurrent abstract commands.

1 Introduction

In [7] Jacobs and Gries tell us that neither total correctness nor partial correctness is on its own adequate to distinguish certain obviously different programs. In particular, total correctness perceives two programs to be equivalent if they are guaranteed to terminate over the same set of start-states and can produce similar sets of results from each such terminating start-state; partial correctness, on the other hand, perceives two programs as equivalent if their respective sets of possible results from each start-state coincide, regardless of whether or not termination is guaranteed. Programs can be simultaneously equivalent under total correctness and distinguishable under partial correctness, and \textit{vice versa}. For example, under total correctness the demonic choice program

\texttt{skip \texttt{[]} WHILE true DO skip END}

is distinguishable from \texttt{skip} but indistinguishable from

\texttt{WHILE true DO skip END}
Conversely, under partial correctness the same program is distinguishable from the latter but indistinguishable from \textit{skip}.

Jacobs and Gries go on in [7] to introduce the notion of general correctness which subsumes both total and partial correctness, two programs being equivalent under general correctness if they are distinguishable neither by total nor partial correctness. While general correctness is now widely acknowledged as an appropriate framework for expressing program semantics, as for example in [6] and [12], current abstract specification methods like B [2], Z [13], VDM [8] and the Refinement Calculus [10] remain confined within the realm of total correctness. This is understandable given the historical simple sequential programming context from which all these specification methods have emerged, but for today's distributed environments with their innate parallelisms total-correctness specifications are no longer always adequate.

In this paper we show how one such abstract specification notation, the B method’s Generalised Substitution Language, can readily accommodate an appropriate semantic enhancement which will enable us to express abstract specifications in general correctness. Moreover, we achieve this without inflicting any syntactic extensions on the language. We will refer to generalised substitutions after investing them with our proposed new general-correctness semantics as \textit{abstract commands}.

In Section 2 we describe a small motivating problem, and in the remainder of the paper we develop our new theory of abstract commands which will enable us to solve it. We make no apologies at all for the small scale and admittedly somewhat artificially contrived context of our “toy” problem. On the contrary, that even such a toy example defeats the capabilities of conventional total-correctness refinement only emphasises all the more, we would maintain, the need for an extension to that refinement theory. Another advantage of our toy specification is that the correctness of its proposed implementation is intuitively readily apparent, even though that correctness cannot be formally demonstrated within any classical (i.e. total-correctness) refinement theory.

2 A Toy Programming Problem

Suppose we wish to develop a program to determine whether a given natural number (known to be within our computer's register capacity) is even or odd. The snag is that the target computer on which we want to execute the program offers an extremely impoverished computing environment: there is one hardware register and only the values zero and one may be loaded into it; thereafter the register can only be modified by means of an “increment-by-two” instruction. In case such an increment would exceed the register’s capacity its effect then instead is to set the register to an indeterminate value within its capacity, subject only to preserving the parity of its prior content. A simple test instruction is available to compare the register’s content for equality with that of a specified memory location.
Unsurprisingly in such a drastically restricted computing environment, it is impossible to formulate a single sequential program to accomplish the required task. Fortunately, we can replicate the computing environment at negligible cost, so we conceive the notion of writing two separate programs to run in parallel. One program will attempt to determine if the given number is even while the other program will attempt to determine whether it is odd. Even in our impoverished environment the two subprograms can be respectively implemented as

\[
\text{ProgramEven} \triangleq x := 0; \\
\quad \text{WHILE } x \neq \text{the_value} \text{ DO } x := x + 2 \text{ END; } \\
\quad \text{answer} := \text{even}
\]

and

\[
\text{ProgramOdd} \triangleq x := 1; \\
\quad \text{WHILE } x \neq \text{the_value} \text{ DO } x := x + 2 \text{ END; } \\
\quad \text{answer} := \text{odd}
\]

Each subprogram can be depended on to give the correct answer if that lies within its competence and otherwise to give no answer at all since it will then never terminate. So we propose to run both of them in concert side by side. We will obtain our desired answer from whichever of them terminates, as one of them always surely must.

Our overall program requirement can easily be specified in conventional total-correctness terms. For instance, taking a slight liberty with B’s GSL we might write

\[
\text{answer} \leftrightarrow \text{Program}(\text{the_value}) \triangleq \\
\quad \text{the_value} \in \text{Evens} \implies \text{answer} := \text{even} \\
\quad \text{the_value} \notin \text{Evens} \implies \text{answer} := \text{odd}
\]

What is interesting, though, is that the two subprograms we have conceived to implement it cannot themselves be adequately specified just in total-correctness terms. Nor does conventional refinement theory provide any means of formally verifying our proposed implementation of the overall program by the simultaneous concerted execution of these two subprograms. Our aim is to rectify this deficiency by developing a general-correctness refinement theory in terms of which our implementation can be properly formulated and verified.

The sceptical reader may think we are being somewhat precious in attributing such significance to this rather arcane shortcoming in classical refinement: surely, he might well muse, in any genuinely credible computing context, without the contrived restrictions of our little toy example, the need for such a parallel-subprograms implementation could never plausibly arise. To counter such a view we will merely mention here two more recognisably useful computing scenarios where exactly this mode of implementation is needed. First, in programming we are often required to evaluate a conjunction or disjunction of logical operands where, notwithstanding that because of undefined terms evaluation of some operands will not terminate, the overall logical expression can
still be evaluated. Such a philosophy inspires VDM's Logic of Partial Functions (LPF): Jones [9] says the various logical operators exhibited in his LPF truth tables

... can be thought of as evaluating their operands in parallel and delivering a result as soon as enough information is available.

Our second example of a parallel execution scenario is drawn from automated proof-searching, where it might well be appropriate to seek to resolve some goal \( G \) one way or the other by embarking simultaneously on proofs of both \( G \) and \( \neg G \). We assume, of course, that our proof-searchers can be relied on to terminate only after finding their proofs and otherwise to carry on searching indefinitely.

2.1 Semi-decidable programs

Technically, we will describe subprograms such as \text{ProgramEven} and \text{ProgramOdd} as semi-decidable programs. We cannot be certain in every circumstance that a semi-decidable program will terminate, but whenever it does so it will yield a “correct” result. We have borrowed the term from computability, where it is synonymous with others such as partially decidable, partially solvable, semi-computable and recursively enumerable, see [4]. There the term is applied to sets \( S \) for which no full decision procedure exists for the predicate \( x \in S \), but only a partial one—that is, one which will terminate when \( x \in S \) is true but may not when it is false.

3 Refinement

In [3] Back and Butler give this cogent description of the formal development of sequential programs through refinement:

In the refinement calculus, the required behaviour of the program is specified as an abstract, possibly non-executable, program which is then refined by a series of correctness-preserving transformations into an efficient, executable program. The notion of correctness-preserving transformation is modelled by a refinement relation between programs which is transitive, thus supporting stepwise refinement, and is monotonic with respect to program constructors, thus supporting piecewise refinement.

In designing an abstract programming language for program development we should therefore choose its constructors carefully, to ensure that they are monotonic with respect to whatever correctness-preserving refinement ordering we wish to uphold in transforming our specifications piece by piece into executable programs.
3.1 About wp and wlp

Each well-formed generalised substitution of the GSL defines a predicate transformer. The formal semantics of a B operation is given by interpreting its characterising generalised substitution as a weakest precondition (wp) predicate transformer. Thus, just as the Refinement Calculus does, B extends to the realm of specification Dijkstra’s original concept of interpreting executable programs in this way [5]. By itself wp provides a semantics of total correctness, which only enables us to reason about outcomes which a program is guaranteed to provide.

It was actually in [5] too, that Dijkstra also introduced a second predicate transformer, the weakest liberal precondition (wlp) transformer, but he didn’t utilise it much there. Only later, in [6], was wlp given equal prominence alongside wp in expressing the meanings of programs, wlp being used to express the partial-correctness behaviour of a program, as wp is used to express its total-correctness behaviour. Nelson [12] extended Dijkstra’s calculus further by introducing partial commands, again employing both wp and wlp to express program meanings. More recently Morgan and McIver [11] have unified both predicate transformers within a single mathematical artifact called an extended weakest precondition (ewp) transformer which acts on three-valued entities they call extended predicates.

3.2 On wlp in specification

While wlp is now securely established alongside wp as a vital mathematical tool for the complete interpretation of the formal meaning of an executable program, abstract specifications are still conventionally understood as articulating only the required total-correctness behaviour of their subjects. The very term refinement has been till now virtually synonymous with the preservation of total correctness.

Like Back and Butler in [3], Morgan [10] convincingly advocates the discarding of any conceptual distinction between programs and specifications: all are programs, he says, though distributed over a wide refinement continuum; some programs are more abstract while others are more concrete, and of the latter some are sufficiently concrete to be directly executable. We wholly subscribe to this uniform view of specifications and programs. Indeed, we would maintain that by instilling general correctness into abstract specifications we are merely propagating throughout the rest of Morgan’s continuum the same finer grain of behavioural description which has already been well established by Dijkstra and others at its executable extremity.

3.3 Strong refinement

In conjunction with our introduction of such “strong” specifications, we need a stronger notion of refinement, one which conceives refinement as the preservation of general correctness rather than just total correctness. A strong specification A will be (strongly) refined by another strong specification B, written $A \sqsubseteq_s B$, provided that for every postcondition $Q$ the following pair of conditions are
satisfied, where \( \text{wp}(A, Q) \) means the weakest precondition under which \( A \) is certain to establish \( Q \), and \( \text{wl} \text{p}(A, Q) \) means the weakest precondition under which \( A \) if it terminates at all must establish \( Q \):

\[
\begin{align*}
\text{wp}(A, Q) & \Rightarrow \text{wp}(B, Q) \\
\text{wl} \text{p}(A, Q) & \Rightarrow \text{wl} \text{p}(B, Q)
\end{align*}
\]

The general-correctness refinement ordering defined above is a natural extension of the classical total-correctness one involving just \( \text{wp} \), and like that one it is transitive. Intuitively, it captures the notion of refinement as the developer’s contractual commitment that his concrete artifact will be a generally-correct realisation of the abstract one. Our refinement ordering is thus quite distinct from the so-called Egli-Milner ordering “\( A \) approximates \( B \)” discussed in [12] and [11], which is defined by

\[
\begin{align*}
\text{wp}(A, Q) & \Rightarrow \text{wp}(B, Q) \\
\text{wl} \text{p}(B, Q) & \Rightarrow \text{wl} \text{p}(A, Q)
\end{align*}
\]

This is not a refinement relation at all in any usual contractual sense, but rather an approximation ordering on programs used to give a fixed-point semantics to iteration, as we will see later.

4 Abstract Commands

An abstract command is syntactically indistinguishable from a generalised substitution. However, whereas a generalised substitution is interpreted in \( B \) simply as a \( \text{wp} \) predicate transformer, an abstract command bears a dual interpretation as a \( \text{wp} \) predicate transformer and, simultaneously, as a \( \text{wl} \text{p} \) predicate transformer. The \( \text{wp} \) interpretation of an abstract command coincides almost exactly with that of its generalised-substitution alter ego, only when the abstract command involves assignment of an undefined term is there a slight difference, as we shall see in the next subsection.

First, we need to fix some notation. We write \( Q(E/x) \) to signify the systematic syntactic replacement of all free occurrences of \( x \) within the predicate \( Q \) by the expression \( E \). We write \( \delta E \) for the predicate which tests whether the expression \( E \) is well-defined: thus, for example, \( \delta(2/1) \) is \text{true}, \( \delta(1/0) \) is \text{false} and \( \delta(1/x) \) is \( x \neq 0 \). If \( A \) is an abstract command and \( Q \) a postcondition, then, in conformance with the usual notation for generalised substitutions, we write \( [A]Q \) to denote the weakest precondition under which \( A \) is guaranteed to establish \( Q \). But we need to extend this notation to represent the \( \text{wl} \text{p} \) effect of
so we will write $[A]^{o}Q$ to denote the weakest precondition under which $A$ “liberally” establishes $Q$, that is to say, the weakest precondition under which $A$ is guaranteed not to establish $\neg Q$.

### 4.1 Formal semantics of abstract commands

In the following table we give the formal wp semantics and wlp semantics of all the basic syntactic constructs of the Abstract Command Language. It will hold no surprises for anyone already familiar with generalised substitutions.

<table>
<thead>
<tr>
<th>$A$</th>
<th>$[A]Q$</th>
<th>$[A]^{o}Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>$Q$</td>
<td>$Q$</td>
</tr>
<tr>
<td>$x:=E$</td>
<td>$\delta E \land Q(E/x)$</td>
<td>$\delta E \Rightarrow Q(E/x)$</td>
</tr>
<tr>
<td>$P \mid B$</td>
<td>$P \land [B]Q$</td>
<td>$[B]^{o}Q$</td>
</tr>
<tr>
<td>$P \quad \Rightarrow B$</td>
<td>$P \Rightarrow [B]Q$</td>
<td>$[B]^{o}Q$</td>
</tr>
<tr>
<td>$B\parallel C$</td>
<td>$[B]Q \land [C]Q$</td>
<td>$[B]^{o}Q \land [C]^{o}Q$</td>
</tr>
<tr>
<td>$q \cdot y \cdot B$</td>
<td>$\forall y \cdot [B]Q$</td>
<td>$\forall y \cdot [B]^{o}Q$</td>
</tr>
<tr>
<td>$B: C$</td>
<td>$[B][(C]Q)$</td>
<td>$[B]^{o}([C]^{o}Q)$</td>
</tr>
</tbody>
</table>

The wlp definition of the preconditioned command $P \mid B$ is of particular interest: the precondition $P$ is simply discarded. Among all the ACL constructs exhibited in the above table, differences in respective wp and wlp definitions occur only in this case of a preconditioned command, and also in simple assignment in the rather special case of an undefined term. These differences may seem trifling, but nonetheless we will see that they provide us with precisely that enriching of semantic context which we need.

### 4.2 Healthiness of wp and wlp

In [5] Dijkstra prescribed several “healthiness” (well-formedness) properties which wp predicate transformers for executable programs were obliged to exhibit. Among these, interesting here in retrospect, were his so-called Property of Continuity forbidding unbounded non-determinism, and the more celebrated Law of the Excluded Miracle forbidding partial commands $\Leftrightarrow$ both now largely defunct, at least in the context of non-executable specifications.

In [6] Dijkstra completely reformulated his healthiness properties so as to encompass wlp as well as wp. The reformulated properties comprised the positive conjunctivity of wp (conjunctivity over all non-empty families of predicates) and

3rd Northern Formal Methods Workshop, 1998 7
the universal conjunctivity of wlp (conjunctivity over all families of predicates, including the empty family). A corollary of the latter is that for any command $A$ the predicate $[A]^\circ \text{true}$ is identically true. There is also a necessary relationship between wp and wlp, expressed for any command $A$ and any postcondition $Q$ in the following logical equivalence:

$$[A] Q \iff ([A] \text{true}) \land [A]^\circ Q$$

The wp and wlp predicate transformer effects we have defined for all our abstract command constructs ensure that any abstract command admitted by our ACL syntax will conform with both of Dijkstra’s conjunctivity healthiness conditions.

### 4.3 Some extreme abstract commands

The following extreme commands are worth particular consideration. They illustrate the expressive power of our abstract command language:

**magic**: expressed in our ACL as $\text{false} \Rightarrow \text{skip}$, magic can do the impossible, even establish the postcondition false. It cannot therefore feasibly be implemented. It is the top of our strong refinement ordering on abstract commands, since it refines anything.

**never**: expressed in our ACL as $\text{false} \land \text{false} \Rightarrow \text{skip}$, never is guaranteed never to terminate successfully. We can think of it as a never-ending loop, or a program which can only terminate improperly by aborting. (Indeed, some authors call it abort: to avoid confusion we prefer not to use this name in general-correctness, since it so often appears in a total-correctness context where it is interpreted like our anarchy below.) Our never is the bottom of the Egli-Milner approximation ordering on abstract commands, since it “approximates” anything.

**perhaps**: expressed in our ACL as $\text{false} \land \text{skip}$, or equivalently as $\text{never} \land \text{skip}$, perhaps is never guaranteed to terminate successfully. We can be sure, though, that if it does terminate it will behave like skip and so have no effect.

**anarchy**: expressed in our ACL as $\text{false} \land x \in X$, where $x \in X$ means the non-deterministic assigning to $x$ of any value from its type-set $X$, anarchy is our least predictable command. We never know whether it will terminate successfully, and even if it does it may deliver any result at all. It is the bottom of our strong refinement ordering on abstract commands, since it is refined by anything.

The last three commands nicely illustrate the more finely-grained semantics of the ACL. As generalised substitutions anarchy, never and perhaps cannot be distinguished, since $\text{false} \land x \in X$, $\text{false} \land \text{false} \Rightarrow \text{skip}$ and $\text{false} \land \text{skip}$ are all synonymous in the GSL. Having anarchy and magic respectively as its
bottom and top ensures that our strong refinement ordering induces a full lattice structure on abstract commands. The Egli-Milner ordering, on the other hand, has no unique top command, since every always-terminating command is maximal, approximating only itself.

4.4 Termination, feasibility and non-termination

For any abstract command $A$ we define in the following table the predicates $\text{trm}(A)$ and $\text{fis}(A)$, characterising respectively the realms of guaranteed termination and of feasibility of $A$. They are defined identically to the corresponding ones for generalised substitutions in [2]. We also define the further predicate $\text{nev}(A)$ which characterises the realm from where $A$ is guaranteed never to terminate.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{trm}(A)$</td>
<td>$[A] \text{true}$</td>
</tr>
<tr>
<td>$\text{fis}(A)$</td>
<td>$\neg [A] \text{false}$</td>
</tr>
<tr>
<td>$\text{nev}(A)$</td>
<td>$([A]^0 \text{false}) \land \neg [A] \text{true}$</td>
</tr>
</tbody>
</table>

4.5 Undefined terms in assignment

Simple substitutions like $x := 1/0$ involving an undefined term are problematic in ordinary B. When transcribed directly as executable code in a target language such as C, the code inevitably aborts with an execution error. Apologists for the status quo in B assert that this direct code transcription is just invalid, in the sense that it doesn’t comprise valid code in the target language. But such an assertion about the C code is unconvincing in the face of the C compiler’s demonstrable quiescence in compiling it. Moreover, B’s ordinary wp calculus gives $\text{trm}(x := 1/0)$ and $\text{fis}(x := 1/0)$ as each identically true –indicating both that the substitution is feasible to implement and that such an implementation will terminate successfully when executed! These indications cannot both be valid. Either we must regard the substitution as infeasible to implement, or else we must accept that its implementation cannot terminate successfully. Our ACL treatment of an assignment like $x := 1/0$ is more satisfactory. Because $1/0$ is an undefined term the assignment $x = 1/0$ has the following characteristics:

| $\text{trm}(x := 1/0)$ | $\text{false}$ |
| $\text{fis}(x := 1/0)$ | $\text{true}$ |
| $\text{nev}(x := 1/0)$ | $\text{true}$ |
In other words our ACL semantics indicates that $x := 1/0$ is never, for in fact never is the only ACL program with the above characteristics. This is entirely consistent with the widely accepted computational interpretation of undefinedness in terms of non-termination or other abortive behaviour, and matches exactly our intuitively expected behaviour of an attempted execution of $x := 1/0$.

5 A Set-theoretic Model

In [2] Abrial gives generalised substitutions a set-theoretic characterisation which he uses both to develop a mathematical theory of their properties and to formulate further GSL constructs, notably iteration. We can propose a corresponding characterisation of abstract commands for similar ends. In several respects the mathematical theory of abstract commands which thus emerges turns out to be simpler than that of generalised substitutions. This should not surprise us: Jacobs and Gries in [7] comment that healthiness properties for their general-correctness predicate transformers are “more uniform” than those for partial and total correctness, which can both be naturally recovered by restricting the general-correctness ones. It supports our own intuition that abstract commands are in some sense a very natural “completion” of generalised substitutions.

5.1 The pre-rel characterisation of an abstract command

We will assume from now on that $A$ is an abstract command working with a variable $x$ spanning a state space $X$. We make the following two definitions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{pre}(A)$</td>
<td>${ x \mid x \in X \land \text{brm}(A) }$</td>
</tr>
<tr>
<td>$\text{rel}(A)$</td>
<td>${ x, x' \mid x, x' \in X \times X \land \neg [A]^\circ(x \neq x') }$</td>
</tr>
</tbody>
</table>

Notice that the definition of $\text{rel}(A)$ given above features $[A]^\circ$ rather than $[A]$; in this way it differs from Abrial’s corresponding definition for generalised substitutions in [2]. Like a generalised substitution, an abstract command is always wholly characterised by its $\text{pre}$ and $\text{rel}$. In the following table we have calculated $\text{pre}(A)$ and $\text{rel}(A)$ for some special commands:
A generalised substitution $S$ acting on a state-space $X$ has a necessary well-formedness constraint that

$$\text{pre}(S) \times X \subseteq \text{rel}(S)$$

There is no corresponding constraint on an abstract command. Any subset of $X$ with any subset of $X \times X$ comprises the $\text{pre}$ and $\text{rel}$ of a valid abstract command. Thus, termination and partial correctness are quite orthogonal features of any abstract command. Indeed, we might say this orthogonality is the very essence of our notion of an abstract command.

## 6 Iteration

We characterise iteration indirectly in our ACL in terms of a more fundamental construct which we introduce in the next subsection. First we must fix some basic mathematical notation for relations. We use the conventional mathematical notation $R^*$ for the reflexive transitive closure of a homogeneous relation $R$. We also denote the well-founded source of $R$ — the largest subset of the source of $R$ upon which $R$ is free of infinite chains or cycles — by $\text{wf}(R)$. A formal definition of $\text{wf}(R)$ is given in [2].

### 6.1 Reflexive transitive closure of an abstract command

We denote the reflexive transitive closure of an abstract command $A$ by $A^*$. It can be characterised as a least fixed-point: specifically, $A^*$ is the Egli-Milner-least solution $X$ of the equation

$$X = (A ; X) [] \text{skip}$$

Since $\text{never}$ is the Egli-Milner bottom abstract command, $A^*$ can be characterised as the limit as $n \leftrightarrow \infty$ of the following sequence of abstract commands $X_0, X_1, X_2, \ldots X_n, \ldots$ derived by taking $\text{never}$ as our initial approximation $X_0$. 

<table>
<thead>
<tr>
<th>$A$</th>
<th>$\text{pre}(A)$</th>
<th>$\text{rel}(A)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>$X$</td>
<td>$\text{Id}_X$</td>
</tr>
<tr>
<td>magic</td>
<td>$X$</td>
<td>${}$</td>
</tr>
<tr>
<td>anarchy</td>
<td>${}$</td>
<td>$X \times X$</td>
</tr>
<tr>
<td>never</td>
<td>${}$</td>
<td>${}$</td>
</tr>
<tr>
<td>perhaps</td>
<td>${}$</td>
<td>$\text{Id}_X$</td>
</tr>
</tbody>
</table>
then re-applying it in the right-hand side of the above equation to obtain our next approximation $X_1$, and so on:

$$
\begin{align*}
X_0 &= \text{never} \\
X_1 &= (\text{fis}(A) \implies \text{never})[\text{skip}] \\
X_2 &= (\text{fis}(A^2) \implies \text{never})[\text{skip}][A] \\
X_3 &= (\text{fis}(A^3) \implies \text{never})[\text{skip}][A][A^2] \\
X_4 &= (\text{fis}(A^4) \implies \text{never})[\text{skip}][A][A^2][A^3]
\end{align*}
$$

For example, $\text{skip}^*$ is perhaps (i.e. $\text{never}[\text{skip}]$) because $\text{skip}$ can be executed indefinitely without becoming infeasible, while $\text{magic}^*$ is $\text{skip}$ because any positive iteration of $\text{magic}$ is infeasible.

### 6.2 A set-theoretic definition of $A^*$

Equivalently, we can define $A^*$ via our set-theoretic model for abstract commands, in terms of $\text{pre}(A^*)$ and $\text{rel}(A^*)$:

$$
\begin{array}{|c|c|}
\hline
\text{pre}(A^*) & \overline{\text{rel}(A)^*}^{-1}[\text{pre}(A)] \cap \text{wlf}(\text{rel}(A)) \\
\text{rel}(A^*) & \text{rel}(A)^* \\
\hline
\end{array}
$$

The definition of $\text{pre}(A^*)$ above is complex enough to warrant some explanation. Its first intersecting term

$$
\overline{\text{rel}(A)^*}^{-1}[\text{pre}(A)]
$$

is the strong inverse image of $\text{pre}(A)$ under the reflexive transitive closure of $\text{rel}(A)$. It expresses that part of the state space from where in any multiple iteration of $A$ each individual execution of $A$ is sure to terminate. The other intersecting term

$$
\text{wlf}(\text{rel}(A))
$$

represents that part of the state space from where indefinite iteration of $A$ is precluded, because after a finite number of executions we are sure to reach a state where $A$ is no longer feasible. In his terminology of [6] Dijkstra would say the first intersecting term addresses our “inner eternal” termination concern, while the second addresses our “outer eternal” one.

### 6.3 Monotonicity with respect to refinement

Although we characterised it as a least fixed-point with respect to the Egli-Milner ordering, our reflexive transitive closure of an abstract command also
enjoys the very important property of being monotonic with respect to our strong refinement ordering: that is to say, if \( A \) and \( B \) are abstract commands then
\[
A \subseteq B \quad \Rightarrow \quad A^* \subseteq B^*
\]
This will be of significance for any subsequent construct we may define in terms of reflexive transitive closure.

### 6.4 A while-loop construct

Having formally defined the reflexive transitive closure of an abstract command we can now define our ACL while-loop construct:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>while ( G ) do ( A ) end ( (G \Rightarrow A)^* ; \neg G \Rightarrow \text{skip} )</td>
<td></td>
</tr>
</tbody>
</table>

The \( \neg G \Rightarrow \text{skip} \) following the \( (G \Rightarrow A)^* \) can be thought of as a “feasibility filter” inhibiting all outcomes from \( (G \Rightarrow A)^* \) which fail to satisfy \( \neg G \).

### 6.5 Monotonicity of while-do-end

Our while-loop construct inherits from \( A^* \) the important property of being monotonic with respect to strong refinement, so that
\[
A \subseteq B \quad \Rightarrow \quad \text{while } G \text{ do } A \text{ end } \subseteq \quad \text{while } G \text{ do } B \text{ end}
\]
Such a property allows us to employ our while-loop construct at any stage in the piecewise refinement of an ACL program. This is in contrast to Nelson’s guarded command language [12], whose iteration construct \textbf{do} \( A \textbf{od} \) lacks this piecewise refinement property, although it is monotonic with respect to the Egli-Milner ordering. We have only to translate Nelson’s iteration into its equivalent abstract command
\[
\text{while } \neg \text{fis}(A) \text{ do } A \text{ end}
\]
and thence into
\[
A^* \text{; } \neg \text{fis}(A) \Rightarrow \text{skip}
\]
for the reason for its failure to be monotonic with respect to refinement to become at once apparent: while \( A^* \) is certainly refinement-monotonic, \( \neg \text{fis}(A) \Rightarrow \text{skip} \) is actually anti-monotonic in \( A \). We note in passing that Nelson’s alternation construct \textbf{if} \( A \textbf{fi} \) similarly fails to be refinement-monotonic. Again, we can see this best when we translate it into its ACL equivalent \( \text{fis}(A) \mid A \), revealing how Nelson’s alternation “inverts” \( A \)'s own intrinsic feasibility guard into a precondition.
6.6 Comparison with Abrial’s treatment of iteration

Our set-theoretic treatment of iteration for abstract commands echoes Abrial’s for generalised substitutions in [2]. However, ours only utilises the relatively familiar mathematical construct of the reflexive transitive closure of a relation, whereas Abrial has to devise the novel mathematical notion of the opening $R^*$ of a homogeneous relation $R$ in order to express his opening $S^*$ of a generalised substitution $S$.

6.7 Two practical strong refinement rules for while-loops

The classical B method has the following traditional while-loop refinement rule which utilises a loop invariant $Inv$ and variant $V$ for proving correctness of GSL implementations employing its while-do-end construct. $B$ is the loop body, $G$ is the loop guard, $z$ represents the variables whose values are altered in the loop, and $A$ initially establishes the loop invariant before the loop is entered.

$$\begin{align*}
[A] & Inv \\
\forall z \cdot (Inv \land G \Rightarrow [B] Inv) \\
\forall z \cdot (Inv \land G \Rightarrow [n := V][B] V < n) \\
\forall z \cdot (Inv \land \neg G \Rightarrow Q) \\
\Rightarrow \\
[A; \text{while } G \text{ do } B \text{ invariant } Inv \text{ variant } V \text{ end} ] Q
\end{align*}$$

The traditional rule is not adequate by itself for strong-refinement purposes since it addresses only total correctness. We must therefore augment it with the following second rule, which addresses the partial-correctness characteristics of while-loops in our ACL:

$$\begin{align*}
\forall z \cdot (Inv \land G \Rightarrow [B]^o Inv) \\
\forall z \cdot (\neg Inv \land G \Rightarrow [B]^o (\neg Inv \land G)) \\
\forall z \cdot (Inv \land \neg G \Rightarrow Q) \\
\Rightarrow \\
[\text{while } G \text{ do } B \text{ invariant } Inv \text{ variant } V \text{ end} ]^o Q
\end{align*}$$

The second rule does not address termination, because termination is not a partial-correctness concern. It demands that, in the context of the guard $G$, the loop-body $B$ preserves not only the invariant $Inv$ but also its converse $\neg Inv$ and in that case the guard $G$ too. This ensures that the while-loop has appropriate guaranteed non-terminating behaviour as well as delivering a correct result where it is allowed to terminate.
7 Concert

Now we have the necessary semantic apparatus of general correctness we can
define a new parallel composition operator for abstract commands. If \( A \) and
\( B \) are abstract commands, we write \( A \# B \) —read as “\( A \) in concert with \( B \)”— to
denote the effect of executing \( A \) and \( B \) concurrently until either one of them
terminates. We can formalise our new ACL construct directly in terms of its \( \wp \)
and \( \wp_{lp} \) effects on any postcondition \( Q \) as follows:

\[
\begin{array}{c|c|c}
\text{\( [A \# B]^\circ Q \)} & \text{\( [A]^\circ Q \land [B]^\circ Q \)} \\
\text{\( [A \# B] \)} & \text{\( ([A]Q \land [B]^\circ Q) \lor ([B]Q \land [A]^\circ Q) \)}
\end{array}
\]

Equivalently, using our set-theoretic model of abstract commands we can define
it in terms of \( \text{pre} \) and \( \text{rel} \):

\[
\begin{array}{c|c|c}
\text{\( \text{pre}(A \# B) \)} & \text{\( \text{pre}(A) \cup \text{pre}(B) \)} \\
\text{\( \text{rel}(A \# B) \)} & \text{\( \text{rel}(A) \cup \text{rel}(B) \)}
\end{array}
\]

Our new construct is clearly symmetrical, as one would expect. From its def-
inition the following rather pleasing distributive properties of concert can also
immediately be inferred:

\[
(P \mid A) \# (Q \mid A) = (P \lor Q) \mid A
\]

\[
(P \mid A) \# (P \mid B) = P \mid (A \# B)
\]

That we can define an abstract command construct \( A \# B \) which captures the
behaviour we wish to describe of executing \( A \) and \( B \) simultaneously until either
one of them terminates, is not of itself particularly significant. We could after
all have done something similar just with generalised substitutions. If \( S \) and \( T \)
are generalised substitutions we could have defined their “concert” to be

\[
\text{trm}(S) \lor \text{trm}(T) \mid \text{trm}(S) \Rightarrow S \mid \text{trm}(T) \Rightarrow T
\]

This certainly expresses a generalised substitution which is guaranteed to termi-
nate whenever either of \( S \) and \( T \) is, delivering a result compatible with at least
one of them. The flaw here is that such an operator would not be monotonic with
respect to (total-correctness) refinement, so would be useless for the piecewise program development which is our whole goal.

In contrast to such a discredited generalised-substitution “concert”, our abstract command concert operator does have the crucial property of monotonicity with respect to strong refinement. That is to say, for any abstract commands \(A, B\) and \(C\) we have:

\[
A \sqsubseteq_s B \Rightarrow A\#C \sqsubseteq_s B\#C
\]

This is the real significance of our abstract command language. Indeed, we could almost characterise our strong-refinement ACL as the least completion of the total-correctness GSL which renders concert monotonic.

### 7.1 A refinement rule introducing concert

The following refinement rule is immediately deducible from our definitions of strong refinement and concert:

\[
(P \lor Q) | (A\|B) \sqsubseteq_s (P | A) \# (Q | B)
\]

Its practical significance is that it allows us formally to develop programs in a piecewise way in which this form of concurrency can be introduced at any stage of refinement. We will utilise it in the next section.

### 8 Our Toy Programming Problem Revisited

We return at last to our toy programming problem of Section 2. We will now sketch how our parallel-subprograms implementation can be formally verified through strong refinement. The original specification can be expressed as

\[
\text{even}(y) \Rightarrow r := \text{even} || \text{odd}(y) \Rightarrow r := \text{odd}
\]

which since \(y\) is either even or odd is refined under our new concert-introduction rule by

\[
\text{even}(y) \mid \text{even}(y) \Rightarrow r := \text{even} \# \text{odd}(y) \mid \text{odd}(y) \Rightarrow r := \text{odd}
\]

We can now refine each of the two above concerted subprograms separately. First we consider

\[
\text{even}(y) \mid \text{even}(y) \Rightarrow r := \text{even}
\]

Introducing a local variable \(x\) we will refine this subprogram by
\begin{verbatim}
  x := 0;
  similar(x, y) | similar(x, y) \implies x := y;
  r := even
\end{verbatim}

where \(similar(x, y)\) is a predicate expressing that \(x\) and \(y\) are both odd or both even. Using our two strong loop refinement rules the middle command can now be refined by a while-loop with guard \(x \neq y\), invariant \(similar(x, y) \land x \leq y\) and variant \(y \leftrightarrow z\). A suitable loop-body is \(x := x + 2\). (Admittedly, we are here skating precariously over some proof obligations which arise from this loop-refinement claim, but their proofs are quite trivial.) What is important is that our loop refinement rules enable us to formulate those proof obligations precisely. Thus our subprogram is refined by

\begin{verbatim}
x := 0;
while x \neq y do x := x + 2 end;
r := even
\end{verbatim}

The other subprogram can be correspondingly refined by

\begin{verbatim}
x := 1;
while x \neq y do x := x + 2 end;
r := odd
\end{verbatim}

In this way we can formally vindicate our earlier implementation intuition. We needed the new general-correctness semantic machinery of our ACL to do so, the total-correctness semantics of ordinary generalised substitutions being inadequate for our purpose.

\section{Conclusion}

We have shown how the total-correctness semantics of B can be enhanced to encompass general correctness by defining a weakest liberal precondition predicate-transformer effect for each generalised substitution to complement its recognised (strict) weakest precondition one. In thus transforming generalised substitutions into what we have called abstract commands we have discovered a rich new specification language with a very intuitive underlying semantic calculus. We have explained that, by virtue of the crucial property of monotonicity with respect to refinement of all its constructs, our language supports piecewise refinement. In this, and in other respects such as arbitrary preconditioning of commands, it improves on Nelson’s generalisation of Dijkstra’s original guarded command language.

We can specify semi-decidable programs in our abstract command language, something beyond the capability of B’s generalised substitution language. Concurrent semi-decidable programs can co-operate to implement a decidable program. We have proposed our concurrent operator \emph{concert} to formalise such cooperation, and we have given it a semantics \emph{within} our general-correctness model.
Abstract commands are a very natural semantic extension of B's generalised substitutions, according completely with our computational intuition. Refinement in general correctness with abstract commands closely mirrors refinement in total correctness undertaken in ordinary B. The only significant addition is an extra rule governing partial correctness in while-loops.

We believe there are no fundamental technical barriers which would prevent the adoption and support of our general-correctness calculus by the existing B computer-based development support tools. Indeed, we think this would need only an incremental enhancement of such tools. On the other hand, if the potentiality of concert is to be properly exploited new target-language compilers are needed which can effectively realise the concurrency which the new operator expresses.

Of course, we make no claim to have established a new general concurrency semantics. Far from it. The particular co-operation characterised by our concert is about as rudimentary as could be conceived between two concurrent agents. Nevertheless, it is undeniably concurrent co-operation. This inevitably raises the question of whether the more complex concurrent interactions typically occurring in concurrent software systems might also in principle lend themselves to similar characterisations within a uniform general-correctness semantics like ours. Certainly, the uniform formal verification of an entire concurrent software system within such a single semantic framework is a tantalising prospect to contemplate, however tentatively from here.

References

1. CADE-13 Workshop on Mechanisation of Partial Functions, held at Rutgers University, NJ USA, July 1996.