Formalising General Correctness

Jeremy E. Dawson [1,2]

NICTA, RSISE,
Australian National University, Canberra ACT 0200, Australia

Abstract

We consider the abstract command language of Dunne, and his account of general correctness. We provide an operational interpretation of his abstract commands, and use the automated theorem proving system Isabelle to prove that this operational interpretation leads to Dunne's semantics. We consider the difficulties in precisely formalising some formulae found in the literature.

Keywords: general correctness, abstract commands.

1 Introduction

General correctness was introduced as an alternative to partial correctness and total correctness by Jacobs & Gries (1985) [5], see also Nelson (1989) [8]. Jacobs & Gries use a relational model, representing a program as a relation between initial states and final states: their space of final states includes $\perp$, representing non-termination. In this way they can distinguish when a program guarantees termination, guarantees non-termination, or neither. Neither partial correctness nor total correctness (alone) can do this.

In [1] and [2], Dunne gives an account of general correctness, in which he gives a set of "abstract commands", with associated semantics. For each abstract command, Dunne gives its semantics in terms of its termination condition, its weakest liberal precondition predicate and its *frame*, which is (loosely) the set of program variables which might be altered by the command. From these one can derive total-, partial- *and* general correctness semantics.

We describe the abstract commands in terms of an operational interpretation similar to that of Jacobs & Gries. We then use the automated prover Isabelle to show that this interpretation implies the semantics given by Dunne. We also use Isabelle to prove some of his more difficult results. Use of an automated theorem prover helped ensure the correct statement of precise details

[1] Supported by an Australian Research Council Large Grant

[2] `jeremy@csl.anu.edu.au`, `http://csl.anu.edu.au/~jeremy/`

which can easily be overlooked otherwise, and forced us to address the distinction between program variables and logical variables (which are easily confused in an informal treatment). We refer to results proved in Isabelle – the code is available via the author's home page.

In [3], Gordon provided an operational interpretation of programs (commands), and used the HOL theorem prover to verify the axioms (rules) of Hoare logic. He explains in detail certain problematic aspects of such work, which we will allude to briefly.

In [4], Harrison formalized Dijkstra's program logic in the HOL theorem prover, using a relation between states and outcomes to model commands.

In [6], Lermer, Fidge & Hayes considered the semantics of program execution, focussing on control-flow paths. They consider weakest liberal preconditions, and also discuss strongest postconditions. Their conditions are more complex than those discussed by Dunne and in this paper, since they consider typed local variables and execution of a command within a "context".

2 Modelling Commands and Conditions

Commands

Typically one models a command (or program) as a function acting on the machine state. A deterministic command which must terminate can be modelled as a function returning simply a single new machine state. A deterministic command which may or may not terminate could be modelled as a function which returns either a new state or nothing, representing the idea that a non-terminating command returns no result. However if we represent a non-deterministic program as a function which returns a set of new states, then this leaves us without a way of representing non-termination as one of several possible outcomes.

We also want to represent commands which are infeasible. (These are a useful building-block, even if you don't want to write such programs, as Dunne discusses). In fact this, rather than non-termination, is naturally represented by a command returning no new state.

The solution (Plotkin [9], also used by Harrison [4]) is to consider command *outcomes*, where an outcome is either termination in a new state or non-termination.

Conditions

Boolean expressions, or conditions, on the machine state, occur in work such as this in two contexts. Firstly, many commands (such as if–then–else, or while–do) incorporate conditions on the state. A state is typically represented as a function from the set of variable names to their values. The condition in such a command will most naturally be represented as text in the programming language, or as an abstract syntax tree, but as it will be capable of being evaluated in any machine state, we might well think of it as a function of type *state* $\to$ *bool* (and we could treat the notion of state as an abstract entity).

Secondly, a condition Q can appear in an expression $wlp(C, Q)$ (where wlp means weakest liberal precondition), or in $\{P\}C\{Q\}$ (Hoare logic). It may be most natural to think of these as predicates on states (or functions of type *state* $\to$ *bool*). However the rule for wlp, and a related rule in Hoare logic, are

$$wlp(x := E, Q) = Q[x := E] \qquad \{Q[x := E]\}\ (x := E)\ \{Q\}$$

By $Q[x := E]$ we mean Q with occurrences of x replaced by E; various other authors write this as $Q[x/E]$, $Q[E/x]$, $Q_{E \to x}$, $Q\langle E/x\rangle$, $\{E/x\}Q$, with, confusingly, both the first two being popular. The notion of substitution in these rules is meaningless when Q is an arbitrary predicate on states; they require Q to be an expression written in the command language, or something like it, or as, say, an abstract syntax tree, containing literal program variable names. Note that the language for such predicates must not be able to express a condition like "no two different variables may have the same value" (for, then, what would $Q[x := E]$ mean?) However, Q *may* also contain logical variables, as in the following Hoare logic example (taken from Gordon[3, §5.0], where X, Y, Z denote program variables and x, y denote logical variables)

$$\{X = x \land Y = y\}(Z := X; X := Y; Y := Z)\{X = y \land Y = x\}$$

It is also worth noting at this point that where boolean expressions are used in abstract commands (such as the guarded command $P \to A$ and the preconditioned command $P|A$) the boolean P is not treated as a fragment of code but rather as an arbitrary predicate on the state. Thus, as is clear from Dunne's treatment of these commands, the possibility of P looping or producing other than a single answer is not considered.

Gordon [3] discusses these issues. What this means for us now is that our analysis of many commands (*not* including assignment) can be performed at the level of abstraction where a boolean expression is modelled as a predicate on states, and a command is modelled as a function from states to sets of outcomes. The next section contains the analysis at that level.

Frames

Dunne has also defined that each abstract command has a *frame*. Loosely, this is the set of variables which "might" be affected. Note, however, that *frame*$(x := x) = \{x\}$. Also, from any command a new command may be defined which has an enlarged frame but is otherwise the same.

Stating the frame of a command does not contribute to a description of what the command does, so we can show, for example, that two commands behave the same way, without considering their frames. The work in this section proceeds on this basis. Note that the results are therefore subject to the proviso that two abstract commands are in fact distinct if their frames differ. We think the relevant proofs about frames would be quite straightforward.

Consideration of literal commands and expressions, of the frames of commands and of the assignment command, is deferred to the following section.

3 Commands as transformations of state

3.1 Monadic Types

As mentioned, we model a command as a function from states to sets of outcomes. Here is the formal definition of the type *outcome*.

```
datatype outcome = NonTerm | Term state
```

So when we model sequencing of two commands A and B, we first apply A to a given state, obtaining a set of outcomes, and we must then apply B, a function of type $\mathit{state} \to \mathit{outcome\ set}$, to the set of outcomes obtained from A. We can think of this as "extending" the function B to a function $\mathit{ext}\ B$ of type $\mathit{outcome\ set} \to \mathit{outcome\ set}$. When this can be done in a way that satisfies certain conditions, we call the relationship between the types a "monad". See Wadler [10] for further information on monads.

In fact, this is an example of a *compound monad*. The type *outcome*, relative to the type *state*, is a monad, where the extension function, of type $(\mathit{state} \to \mathit{outcome}) \to (\mathit{outcome} \to \mathit{outcome})$ would be given by

$$\mathit{exto}\ f\ \texttt{NonTerm} = \texttt{NonTerm}$$
$$\mathit{exto}\ f\ (\texttt{Term}\ s) = f s$$

The type constructor *set*, which for any type α gives the type $\alpha\ \mathit{set}$ (the type of sets of things of type α) also gives a monad, whose extension function, of type $(\alpha \to \alpha\ \mathit{set}) \to (\alpha\ \mathit{set} \to \alpha\ \mathit{set})$, is given by

$$\mathit{exts}\ f\ os = \bigcup_{o \in os} f\ o$$

Apart from the extension function, specification of a monad includes a *unit* function, which converts a value of the "base" type, usually in a rather natural way, to a value of the monadic type. For the two monads mentioned, we have

$$\begin{array}{ll} \mathit{unito} : \mathit{state} \to \mathit{outcome} & \mathit{units} : \alpha \to \alpha\ \mathit{set} \\ \mathit{unito}\ s = \texttt{Term}\ s & \mathit{units}\ e = \{e\} \end{array}$$

Note also that the extension function is often called *bind* and written in infix format (as in [10]), so $\mathit{ext}\ f\ s = s\ \mathit{bind}\ f$.

Two monads cannot in general be composed to form another monad, but the first monad mentioned above can in general be composed with any other monad to give a compound monad (see [7, §7.3]). The formulae for the extension function, both generally (in terms of *units* and *exts*) and for our specific choice of *units* and *exts*, are given below. In the specific case, *extos* has type $(\mathit{state} \to \mathit{outcome\ set}) \to (\mathit{outcome\ set} \to \mathit{outcome\ set})$.

$$\begin{array}{ll} \textit{extos}\ f\ \textit{os}\ = & \textit{extos}\ f\ \textit{os}\ = \\ \quad \textit{let}\ f'\ (\texttt{Term}\ s) = f\ s & \quad \textit{let}\ f'\ (\texttt{Term}\ s) = f\ s \\ \quad\quad f'\ \texttt{NonTerm} = \textit{units}\ \texttt{NonTerm} & \quad\quad f'\ \texttt{NonTerm} = \{\texttt{NonTerm}\} \\ \quad \textit{in}\ \textit{exts}\ f'\ \textit{os} & \quad \textit{in}\ \bigcup_{o \in os} f'\ o \end{array}$$

As mentioned above, a monad includes functions *unit* and *ext* (of appropriate types), which must satisfy certain conditions, as follows:

$$\begin{aligned} \textit{ext}\ k \circ \textit{unit} &= k && \text{(Left Unit)} \\ \textit{ext}\ \textit{unit} &= \textit{id} && \text{(Right Unit)} \\ \textit{ext}\ (\textit{ext}\ h \circ k) &= \textit{ext}\ h \circ \textit{ext}\ k && \text{(Assoc)} \end{aligned}$$

Let $seq\ A\ B$ denote the sequencing of commands A and B (where A, B and $seq\ A\ B$ are of type $state \to outcome\ set$). As noted, we want to first apply A to the given state, obtaining a set of outcomes; we must then apply the extension of B (of type $outcome\ set \to outcome\ set$) to that set of outcomes. That is, $seq\ A\ B = extos\ B \circ A$. Then we can prove the associativity of *seq*:

$$\begin{aligned} seq\ A(seq\ B\ C) &= extos\ (seq\ B\ C) \circ A \\ &= extos\ (extos\ C \circ B) \circ A \\ &= extos\ C \circ extos\ B \circ A \qquad \text{by the monad rule (Assoc)} \\ &= extos\ C \circ seq\ A\ B \\ &= seq\ (seq\ A\ B)\ C \end{aligned}$$

This is proved in Isabelle as `seq_assoc`. Dunne [1, §7] uses ‘;’ for sequential composition, so he writes $seq\ A\ B$ as $A; B$.

The *unit* function, of type $state \to outcome\ set$, of the compound monad is given by

$$unitos\ s = units\ (\texttt{Term}\ s) = \{\texttt{Term}\ s\}$$

This represents the command *skip*, which always terminates in its initial state.

3.2 Refinement

As we will often just give Isabelle code, we mention some less obvious Isabelle notation. The “?” indicates a variable for which anything (of a suitable type) may be substituted. Logical equivalence of booleans is denoted by the equality symbol `=` or `==`. By convention in this paper, results we have proved as theorems are given in quotes (often preceded by the theorem’s name), whereas definitions are given without quotes.

Some set and function notation (mathematical and Isabelle equivalents) follows:

$a \notin \bigcup_{b \in C} D$	`a ~: UN b:C. D`
$\{a\} \cup (C \cup D) \subseteq E \cap F \backslash G$	`insert a (C Un D) <= E Int F - G`
$\lambda x.E$	`(%x. E)`
$A \to B$ (implication)	`A ==> B` or `A --> B`

We define functions corresponding to *wlp*, *trm*, and *wp* of [1, §2].

```
wlpm ?C ?P ?state == ALL nst. Term nst : ?C ?state --> ?P nst
trmm ?C ?state     == NonTerm ~: ?C ?state
wpm ?C ?P          == wlpm ?C ?P && trmm ?C
```

Here `&&` and `||` lift conjunction and disjunction over states, and `--->` is the "is stronger" relation between predicates, so, where `P` and `Q` are predicates on states,

```
?P ---> ?Q      ==  ALL s. ?P s --> ?Q s
(?P && ?Q) ?s   ==  ?P ?s & ?Q ?s
(?P || ?Q) ?s   ==  ?P ?s | ?Q ?s
```

These definitions work with commands (A, B, C) as functions of type *state* → *outcome set* and conditions (P, Q) as functions of type *state* → *bool*. We note that a command (as such a function) is uniquely determined by its *wlp* and termination conditions. This is proved in Isabelle as `unique`. Later we will introduce corresponding (differently named) functions which take abstract syntax trees as arguments.

In [1, §5] Dunne discusses several notions of refinement, including general-, total- and partial-correctness refinement. The second equivalent definition of `gencref` is derived from Dunne's (Gcref2) ([2, §2.1]). Again, (A, B, C) are commands, of type *state* → *outcome set*.

```
totcref ?A ?B   ==  ALL Q. wpm ?A Q ---> wpm ?B Q
partcref ?A ?B ==  ALL Q. wlpm ?A Q ---> wlpm ?B Q
gencref ?A ?B   ==  partcref ?A ?B & totcref ?A ?B
gencref ?A ?B   ==  partcref ?A ?B & (trmm ?A ---> trmm ?B)
```

From these definitions, we proved the following more direct characterizations of these three notions of refinement. It is worth noting that the characterization for general correctness is simpler than the other two although it is defined in terms of both of them; this no doubt explains how general correctness semantics often seems simpler than either partial or total correctness semantics. Also, the general correctness relation is anti-symmetric, unlike either total or partial correctness.

```
"totcref ?A ?B  =  (ALL st. ?B st <= ?A st | NonTerm : ?A st)"
"partcref ?A ?B =  (ALL st. ?B st <= insert NonTerm (?A st))"
"gencref ?A ?B  =  (ALL state. ?B state <= ?A state)"
gencref_antisym = "[| gencref ?A ?B; gencref ?B ?A |] ==> ?A = ?B"
```

3.3 Strongest Postconditions

Strongest postconditions have been discussed by a number of authors, cited in [6]. In [6], Lermer, Fidge & Hayes give the strongest postcondition semantics, as well as the weakest liberal precondition semantics, of the commands they define, and we will refer to some of these later. Meanwhile we give its definition in Isabelle, and that of what they call “sp-refinement”.

```
"slpm ?C ?P ?state == EX sb. ?P sb & Term ?state : ?C sb"
"slpref ?A ?B == ALL Q. slpm ?B Q ---> slpm ?A Q"
```

That is, supposing precondition `B` to be satisfied, `slpm C` holds (only) for those states which are possible (terminating) outcomes of `C`. This is clearly analogous to a weakest *liberal* precondition (hence our name, `slpm`), in that if a command is changed to allow (or disallow) the additional possibility of non-termination, the strongest postcondition remains unchanged. In fact it was trivial to prove (as `wlp_slp_ref`) that strongest post-condition refinement is equivalent to partial-correctness refinement ([6, Thm 6.1(vii)]). This is readily explained by the Galois connection between the weakest liberal precondition and the strongest postcondition functions (`wlp_slp_gal`, of which Theorem 6.1(v) and (vi) of [6] are easy corollaries).

```
wlp_slp_ref = "partcref = slpref"
wlp_slp_gal = "(?Q ---> wlpm ?C ?R) = (slpm ?C ?Q ---> ?R)"
```

We may well ask whether the weakest precondition function wp has a Galois dual sp, but the answer is in the negative. For that would require $sp\ C\ Q \longrightarrow true$ (which always holds) being equivalent to $Q \longrightarrow wp\ C\ true$ (which does not hold at states which satisfy Q but from which C may not terminate). However we can define functions `spom` and `wpom` (strongest post- and weakest pre-conditions based on outcomes) which are Galois duals and are neatly related to general correctness refinement. Note that R is a predicate on outcomes, Q on states.

$$\begin{aligned}
spom &: (state \to outcome\ set) \to (state \to bool) \to outcome \to bool\\
wpom &: (state \to outcome\ set) \to (outcome \to bool) \to state \to bool\\
spom\ C\ Q\ c &= \exists s.\ Q\ s \land c \in C\ s\\
wpom\ C\ R\ s &= \forall c \in C\ s.\ R\ c
\end{aligned}$$

```
wpo_spo_gal = "(?Q ---> wpom ?C ?R) = (spom ?C ?Q ---> ?R)"
gencref_wpo = "gencref ?A ?B = (ALL Q. wpom ?A Q ---> wpom ?B Q)"
gencref_spo = "gencref ?A ?B = (ALL Q. spom ?B Q ---> spom ?A Q)"
```

3.4 Meaning of Commands

skip, perhaps, magic, abort

[1, §7] *skip* is the command which is feasible, terminates and does nothing to the state. It is exactly the function *unitos*. It follows immediately from the (Left Unit) and (Right Unit) monad laws that *skip* is an identity (left and right) for the binary function *seq*. These are proved in Isabelle as `seq_unitL` and `seq_unitR`. We also define

```
perhaps ?st  ==  {Term ?st, NonTerm}
magic ?st    ==  {}
abort ?st    ==  {NonTerm}
```

preconditioned command
[1, §7] The command $P|A$ is the same as A except that, if P does not hold, then $P|A$ *may* fail to terminate.

```
precon ?P ?C ?state ==
   if ?P ?state then ?C ?state else insert NonTerm (?C ?state)
```

guarded command
[1, §7] The command $P \longrightarrow A$ is the same as A if P holds, but is *infeasible* (the outcome set is empty) if P does not hold.

```
guard ?P ?C ?state == if ?P ?state then ?C ?state else {}
```

A command has a "natural" guard and precondition. Here `fis` A means A is *feasible*, that is, its outcome set is non-empty. We have proved

```
fis_guard  =  "guard (fis ?A) ?A = ?A"
pc_trm     =  "precon (trmm ?A) ?A = ?A"
```

choice
In [1, §7] Dunne defines a binary operator, $A \Box B$, for *bounded choice*: $A \Box B$ is a command which can choose between two commands A and B. This is a special case of choice among an arbitrary set of commands, defined by

$$\mathit{choice}\ C\ s = \bigcup_{c \in C} c\ s \qquad \texttt{choice ?Cs ?state == UN C:?Cs. C ?state}$$

From these, we prove the definitions, and other results, of Dunne [2, §5,6].

```
perhaps_alt = "perhaps = precon (%st. False) unitos"
magic_alt = "magic = guard (%st. False) ?A"
abort_alt = "abort = precon (%st. False) (guard (%st. False) ?A)"
pma = "seq perhaps magic = abort"
asp = "choice {abort, unitos} = perhaps"
```

concert
[1, §12] The command $A \# B$ represents A and B executing independently, on separate copies of the state: whichever of A or B terminates first gives the effect of $A \# B$. Thus the possible outcomes of $A \# B$ are:

- `Term` s, if it is an outcome of A,
- `Term` s, if it is an outcome of B,

- `NonTerm`, if it is an outcome of *both* A and B.

```
conc ?A ?B ?state == concrs (?A ?state) (?B ?state)
concrs ?cr1 ?cr2 ==
   ?cr1 Un ?cr2 - {NonTerm} Un {NonTerm} Int ?cr1 Int ?cr2
```

Interestingly, this means that if B is *magic* (everywhere infeasible), then $A\#B$ is just A with any possibility of non-termination removed (difficult though it is to see from the first sentence above!). This is proved in Isabelle as `conc_magic`.

The *wlp* and termination conditions for these commands, which are used by Dunne to *define* these commands, are proved in Isabelle from our definitions, as `precon_trm`, `precon_wlp`, `guard_trm`, `guard_wlp`, `seq_trm`, `seq_wlp`, `choice_trm`, `choice_wlp`, `conc_trm` and `conc_wlp`. Dunne's results Xpre, Xguard, Xassump and Xassert [2, §4,5] are also proved in Isabelle.

```
Xpre = "seq (precon ?P ?A) ?B = precon ?P (seq ?A ?B)"
Xguard = "seq (guard ?P ?A) ?B = guard ?P (seq ?A ?B)"
Xassump = "guard ?P ?A = seq (guard ?P unitos) ?A"
Xassert = "precon ?P (guard ?P ?A) =
    seq (precon ?P (guard ?P unitos)) ?A"
```

3.5 Repetition and Iteration

finite repetition
[1, §7] Dunne defines $A^0 = \mathit{skip}$ and $A^{n+1} = A; A^n$. A very convenient result which we proved, called `rep_Suc'`, is that $A^{n+1} = A^n; A$.

repetitive closure
[1, §12] We also defined $\mathit{repall}\ c\ s = \bigcup_n \mathit{rep}\ n\ c\ s$, ie,

```
repall ?C ?state == UN n. rep n ?C ?state
```

that is, `repall` A is the (unbounded) choice of any number of repetitions of A. The termination condition for `repall` A is that for every n, A^n terminates (proved as `repall_term`).

The *repetitive closure* of A is A^*, where the outcomes of A^* are those of `repall`, augmented by `NonTerm` in the case where it is feasible to execute A infinitely many times sequentially (we call this an "infinite chain"). It is much easier to define this concept operationally than in terms of *wlp* and *trm*. The definition of an infinite chain asserts an infinite sequence of states, of which each is reachable from the previous one. We omit the Isabelle definition.

$$\mathit{infch}\ A\ s \equiv \exists f.\ f\ 0 = s \wedge (\forall n.\ \mathtt{Term}\ (f\ (n+1)) \in A\ (f\ n))$$

Thus we have the definition

```
repstar ?C ?state == repall ?C ?state Un
    (if infch ?C ?state then {NonTerm} else {})
```

It may be noted that in [1, §10], Dunne defined a predicate *cic* ("cycles and infinite chains"), with the intended meaning that *cic* A s be true if perpetual repetition of A is feasible, (ie, an infinite chain of executions of A is possible). However the definition made *cic* A s true in the situation where A could be executed any given n times sequentially, which is *not* sufficient to ensure an infinite chain of executions. (It would be sufficient under an assumption of bounded non-determinacy, see [4, §3]). As is a common experience, we did not discover this discrepancy until trying to perform Isabelle proofs based on the definition in question.

We have proved some useful results, such as

`wlpca` **:** $wlp\ (A^*, \_) = wlp$ (`repall` $A, \_$) (since they differ only in that A^* has an additional possibility of non-termination)

`seq_repstar` **:** $A^*; A = A; A^*$

In [1, §12] Dunne mentions that repetitive closure could be defined using *Egli-Milner* approximation [1, §6].

$$A \leq_{em} A' \equiv A \sqsubseteq_{tot} A' \wedge A' \sqsubseteq_{par} A$$

where $\sqsubseteq_{tot}$ and $\sqsubseteq_{par}$ denote respectively total- and partial-correctness refinement. Then A^* is a least fixpoint under the ordering $\leq_{em}$:

$$A^* \equiv \mu_{em} X.(A; X) \Box skip$$

We show in Isabelle that our definition of A^* implies this result. Here `fprep_alt2` is a paraphrase of our definition of `fprep` (`fprep` A X means $X = (A; X)\Box skip$), `repstar_isfp` says that A^* is a fixpoint, and `repstar_is_lfp` says that A^* is less than or equal to, in the Egli-Milner ordering, any given fixpoint Y. We also have that the Egli-Milner ordering is anti-symmetric, so a least fixpoint is unique.

```
fprep_alt2 = "fprep ?A ?X = (?X = choice {seq ?A ?X, unitos})"
repstar_isfp = "fprep ?A (repstar ?A)"
repstar_is_lfp = "fprep ?A ?Y ==> egMil (repstar ?A) ?Y"
egMil_antisym = "[| egMil ?A ?B; egMil ?B ?A |] ==> ?A = ?B"
```

Dunne (pers. comm.) also defines $trm(A^*)$ and $wlp(A^*, Q)$ as fixpoints:

$$trm(A^*) = \nu Y.wp(A, Y) \qquad wlp(A^*, Q) = \mu Y.wlp(A, Y) \wedge Q$$

where μ and ν denote the least and greatest fixpoints, that is the weakest and strongest (respectively) fixpoints. We also prove these results in Isabelle, based on our definition of A^*. `trfp` and `wrfp` say that $trm(A^*)$ and $wlp(A^*, Q)$ are fixpoints of the respective functions. `trsfp` says that $trm(A^*)$ is equal or weaker than any given fixpoint Y, and similarly for `wrwfp`.

```
trfp = "trmm (repstar ?A) = wpm ?A (trmm (repstar ?A))"
trsfp = "?Y = wpm ?A ?Y ==> trmm (repstar ?A) ---> ?Y"
```

```
wrfp = "let wlpstar = wlpm (repstar ?A) ?Q
    in wlpstar = (wlpm ?A wlpstar && ?Q)"
wrwfp = "?Y = (wlpm ?A ?Y && ?Q) ==>
    ?Y ---> wlpm (repstar ?A) ?Q"
```

We can get a similar set of results for `repall`. Firstly, `repall` A is a *greatest* fixpoint under the general correctness refinement ordering $\sqsubseteq_{gen}$, and secondly, the termination condition for `repall` A is a *least (weakest)* fixpoint. Recall that wlp (`repall` $A, Q) = wlp(A^*, Q)$.

$$\texttt{repall}\ A = \nu_{gen} X.(A; X) \Box skip \qquad trm(A^*) = \mu Y. wp(A, Y)$$

```
repall_isfp = "fprep ?A (repall ?A)"
repall_is_gfp = "fprep ?A ?X ==> gencref ?X (repall ?A)"
trallfp = "trmm (repall ?A) = wpm ?A (trmm (repall ?A))"
trallwfp = "?Y = wpm ?A ?Y ==> ?Y ---> trmm (repall ?A)"
```

It follows that `repall` A and A^* are both greatest fixpoints under $\sqsubseteq_{par}$ of $\lambda X.\ (A; X) \Box skip$, reflecting that such fixpoints need not be unique, since $\sqsubseteq_{par}$ is not anti-symmetric.

3.6 Monotonicity

For developing a program by starting with an abstract expression (of requirements), and progressively refining it to a concrete program, it is important that the abstract commands constructors are monotonic with respect to general-correctness refinement ($\sqsubseteq_{gen}$).

Given our characterization of $A \sqsubseteq_{gen} B$ as ($\forall state.\ B\ state \subseteq A\ state$), and our operational definition of commands in terms of outcome sets, it is easy to see that all the constructors mentioned are monotonic. In any event, they are proved in Isabelle as (for example) `seq_ref_mono`, `repstar_ref_mono`, etc.

3.7 The while loop

In [1, §7, §12] Dunne defines

$$\begin{aligned} if\ G\ then\ A\ end &\equiv (G \to A) \Box (\neg G \to skip) \\ while\ G\ do\ A\ end &\equiv (G \to A)^*; \neg G \to skip \end{aligned}$$

The definition of *while* which is intuitive to programmers is

$$while\ G\ do\ A\ end \equiv if\ G\ then\ A; while\ G\ do\ A\ end\ end$$

We cannot use this as a definition in Isabelle since it is recursive – as it stands it is non-terminating, and when applied to a particular state, may not terminate. So in Isabelle we have defined *while* as does Dunne, and have proved that it satisfies the "intuitive" definition.

```
while_prog = "while ?G ?A = ifthen ?G (seq ?A (while ?G ?A))"
```

4 Frames and Variable Names

In §3, we viewed a command as a function from a state to a set of outcomes, and a condition as a predicate on states. In this treatment, the view of a state was abstract. As discussed in §2, there are various ways in which a full treatment needs to be more concrete, namely

- referring to program variables
- having conditions in a form allowing substitution for a program variable
- specifying a frame for a command

In this section we discuss those abstract command constructors which require us to address these issues.

In our Isabelle model, the program variable names are of type `'n` (eg, strings) and they take values of type `'v`, where `'n` and `'v` are Isabelle type variables. As a state is an assignment of variables to values, we have the type definition $state = name \rightarrow value$, or, in Isabelle, `state = "'n => 'v"`.

indeterminate assignment
[1, §12] Where x is a (list of) variables, and P is a predicate, the command $x : P$ assigns values to the variable(s) in x in any way such that the change of state satisfies P. More precisely, if α is the "current alphabet" (the set of variables whose names are currently "in scope"), and x_0 is the set of variable names in x, but with subscript 0 added, then P is a predicate on $\alpha \cup x_0$. (The paper [1] says $\alpha \cup \alpha_0$ – we comment on this below). The subscripted variable names represent the values of those variables before the command is executed. We model such a P as a relation on states, so we define this command as

```
"indetass ?vars ?P ?state ==
  Term ` ((?P Int chst ?vars) `` {?state})"
```

where `chst ?vars` means the set of pairs of states which differ only in the variables `?vars`, `` f ` X `` means the image of the set `X` under the function `f`, and ``` r `` X ``` means the "image" of `X` under the relation `r`, ie, $\{y \mid (x, y) \in r \text{ for some } x \in X\}$.

Although Dunne does not give the *wlp* for indeterminate assignment, Lermer, Fidge & Hayes [6] (who use the notation $x : [P]$ for Dunne's $x : P$) give *wlp* and *sp* for it. When we omit reference to contexts, their formulae are as follows, where x' refers to the value(s) of variable(s) x after the execution of a command.

$$wlp(x : Q, R) = \forall x'.Q \rightarrow R[x := x'] \qquad sp(x : Q, R) = (\exists x.R \wedge Q)[x' := x]$$

These are difficult to express in Isabelle. This is because $\exists x. \ldots$ means "for some values x, which, when taken to be the values of variables $x, \ldots$". This is, in a sense, an overloading of the symbol x, which complicates the formalisation. However, we derived the following results for the *wlp* and *sp* of indeterminate assignment, which are in a form fairly close to the expressions above. We use a function `setvars`, of type *name set* $\rightarrow$ *state* $\rightarrow$ *state* $\rightarrow$ *state*, which resets the values of variables in the given set, and we refer to a second state `primed` to refer to the values in x'.

```
setvars ?vars ?primed ?state ?str ==
   if ?str : ?vars then ?primed ?str else ?state ?str
indetass_wlp' = "wlpm (indetass ?x ?Q) ?R ?state =
  (ALL primed. (?state, setvars ?x primed ?state) : ?Q -->
    ?R (setvars ?x primed ?state))"
indetass_slp' = "slpm (indetass ?x ?Q) ?R ?primed =
  (EX state. (setvars ?x state ?primed, ?primed) : ?Q &
    ?R (setvars ?x state ?primed))"
```

prd

[1, §10] The "before-after" predicate *prd* specifies conditions under which the command *may* terminate in a state where variables have certain given values. Dunne defines this as

$$\mathit{prd}\ (A) \equiv \neg \mathit{wlp}\ (A, x \neq x')$$

where x' are new (logical) variables corresponding to the program variables x. We define `prds` and `prdm`, as

```
prds ?vars ?primed ?A ==
   Not o wlpm ?A (%st. EX str:?vars. st str ~= ?primed str)
prdm ?primed ?A == Not o wlpm ?A (%st. st ~= ?primed)
```

where `?primed`, of type *state*, represents the values x', and `prdm` is a simpler version of `prds` for use when x can be taken to be all variable names. Dunne also gives *wlp* in terms of *prd* as $\mathit{wlp}\ (A, Q) = \forall x'. \mathit{prd}\ (A) \Rightarrow Q[x := x']$ which we prove as

```
wlp_prd = "wlpm ?A ?Q ?state =
  (ALL primed. prdm primed ?A ?state --> ?Q primed)"
```

In [2, §9] Dunne states the result $\mathit{prd}\,(x : P) = P[x_0, x := x, x']$. We proved a corresponding result for the special case where x represents all variable names

```
indetass_prd = "prdm ?primed (indetass UNIV ?P) ?state =
     ((?state, ?primed) : ?P)"
```

but found that we could not prove the stated result generally. It turned out that Dunne's result requires that P be a predicate on $\alpha \cup x_0$, not on $\alpha \cup \alpha_0$ (as stated in the paper). This is another example of the common situation that attempting to prove such results formally detects points such as this which can easily be overlooked in an informal treatment.

unbounded choice
[1, §7] The command ($@z.A$) means that variable z is to be set to any value and then A is to be executed. But z is to be a "local" variable in A; if, for example, Q contains z, then it is a *different* z from that in A. In other words, the notation correctly reflects that z behaves as normal for a bound variable (it can be α-converted with no change in meaning).

So we model this command as follows:

- set variables z to arbitrary values, using `setvars`
- execute A
- reset variables z to their initial values, using `revert`

```
revert ?vars ?A ?initst ==
   mapos (setvars ?vars ?initst) (?A ?initst)
at ?vars ?A ?initst ==
   let initptf = %primed. setvars ?vars primed ?initst;
      initptc = %x. UNION UNIV (?A o initptf)
   in revert ?vars initptc ?initst
```

Here, `UNION UNIV` $F = \bigcup_x F\ x$, and `mapos` is the monadic "map" function:

$$
\begin{aligned}
mapos\ f\ ocset &= \{mapo\ f\ s \mid s \in ocset\}\\
mapo\ f\ (\texttt{Term}\ s) &= \texttt{Term}\ (f\ s)\\
mapo\ f\ \texttt{NonTerm} &= \texttt{NonTerm}
\end{aligned}
$$

We then proved

```
at_trm = "trmm (at ?vars ?A) = allvars ?vars (trmm ?A)"
```

where `allvars vars` $B\ s$ means that for any other state s' obtained by taking s and setting the variables `vars` to any values, $B\ s'$ holds. Since Dunne gives $wlp\ (@z.A, Q) = \forall z.\ wlp\ (A, Q)$, we tried to prove

```
wlpm (at ?vars ?A) ?Q = allvars ?vars (wlpm ?A ?Q)
```

but could not. This reflected the fact that the formula for $wlp\ (@z.A, Q)$ given by Dunne assumes that Q does not involve z. (As noted above, the α-convertibility of z in $@z.A$ means that we can sensibly assume this). In fact we proved the more complex results

```
at_wlp = "wlpm (at ?vars ?A) ?Q ?st = (ALL nst.
  wlpm ?A (?Q o setvars ?vars ?st) (setvars ?vars nst ?st))"
at_slp = "slpm (at ?vars ?A) ?Q ?st = (EX nst.
  slpm ?A (?Q o setvars ?vars ?st) (setvars ?vars nst ?st))"
```

Lermer, Fidge & Hayes [6] give a similar command (using '**var**' rather than '@'). Their framework involves typed variables and execution of a command in a "context", but if these aspects are removed, their expressions amount to

$$
\begin{aligned}
wlp(@z.A, Q) &= (\forall z.wlp\ (A, Q[z := y]))[y := z]\\
sp(@z.A, Q) &= (\exists z.sp\ (A, Q[z := y]))[y := z]
\end{aligned}
$$

where y denotes new variables which do not appear (free) in A or Q.

We now compare these results with ours.

(i) Firstly, while $Q[z := y]$ refers to the textual substitution of variable(s) y for variable(s) z, we can interpret it as follows: $Q[z := y]\ s$ means take state s, assign the *values* y (ie, treating y as *logical* variables) to the *program* variables z, and then apply the predicate Q to the resulting state. Thus we can say $Q[z := y] = Q \circ \mathit{setvars}\ z\ y$, noting that the y on the right-hand side is an assignment of values to *all* variables – the y-values outside z are simply not used.

(ii) Next we interpret $(\forall z.P)\ s$, where P is a predicate on states. This means that $P\ s'$ holds for every state s' which is obtained from s by setting the program variables z to other values x. That is, $(\forall z.P)\ s = \forall x.\ (P \circ \mathit{setvars}\ z\ x)\ s$.

(iii) Finally, we need to interpret $R[y := z]s$, for R a predicate containing the logical variable(s) y. This requires replacing occurrences of y in R with the values, in the state s, of the program variables z. Fortunately, we find that y only appears in the sub-expression $\mathit{setvars}\ z\ y$ (the function which resets z to y in a state); in this particular expression, replacing y by the values of z in s gives simply $\mathit{setvars}\ z\ s$. We could also say we get $\mathit{setvars}\ z\ (\mathit{setvars}\ z\ s\ y)$, which we have proved (as `setvars_set`) equal to $\mathit{setvars}\ z\ s$.

Thus the following shows the equivalence of our results to those in [6].

$$
\begin{aligned}
&(\forall z.\mathit{wlp}\ (A, Q[z := y]))[y := z]s \\
&= (\forall z.\ \mathit{wlp}\ (A, Q \circ \mathit{setvars}\ z\ y))[y := z]s && \text{(by (i))}\\
&= \forall x.\ (\mathit{wlp}\ (A, Q \circ \mathit{setvars}\ z\ y) \circ \mathit{setvars}\ z\ x)[y := z]s && \text{(by (ii))}\\
&= \forall x.\ \mathit{wlp}\ (A, Q \circ \mathit{setvars}\ z\ s)(\mathit{setvars}\ z\ x\ s) && \text{(by (iii))}
\end{aligned}
$$

4.1 Assignment; the Syntactic View

As noted in §2, $\mathit{wlp}(x := E, Q) = Q[x := E]$, which is only meaningful when Q is some structure in which we can define substitutions (although we have imitated $Q[x := E]s$ by Qs' where s' is the state which is the same as s except that the variable s is assigned the value of E). So we have defined types for the abstract-syntax-tree version of expressions (`exp`) and boolean expressions (`bexp`), thus:

```
datatype ('n,'v) exp = Val 'v
                     | Op "'v list => 'v" "('n,'v) exp list"
                     | Var 'n

datatype ('n,'v) bexp = Rel "'v list => bool" "('n,'v) exp list"
          | BRel "bool list => bool" "('n,'v) bexp list"
```

Thus an expression is a simple value, or an operation on values together with an argument list, or a program variable. A boolean expression is a relation on values together with an argument list of value expressions, or a boolean function together with an argument list of boolean expressions.

We defined substitution functions, of the following types

```
expSub   :: "'n => ('n,'v) exp => ('n,'v) exp => ('n,'v) exp"
bexpSub  :: "'n => ('n,'v) exp => ('n,'v) bexp => ('n,'v) bexp"
```

where (for example) `expSub` $x\ E\ M$ means $M[x := E]$. We also defined functions `expMng` and `bexpMng` to translate an expression (type *exp* or *bexp* – which we will call a **syntactic** expression) to the corresponding function of type $state \rightarrow value$ or $state \rightarrow bool$ (which we will call a **semantic** expression, calling it the "meaning" of the syntactic expression). Obviously, distinct syntactic expressions may have the same meaning, and therefore the "=" symbol in a proposition of the form "$wlp\ (A, Q) = \ldots$" can only be sensibly interpreted as equality of semantic expressions, notwithstanding that in "$wlp(x := E, Q) = Q[x := E]$", the right-hand side is only meaningful as a syntactic expression. We can also refer to syntactic and semantic *commands*.

```
expMng   :: "('n,'v) exp => ('n,'v) state => 'v"
bexpMng   :: "('n,'v) bexp => ('n,'v) state => bool"
```

We can then prove the following results, and corresponding ones for boolean expressions.

```
subLemma = "expMng (expSub ?x ?E ?Q) ?state =
  expMng ?Q (?state(?x := expMng ?E ?state))"
sub_equiv = "expMng ?Q = expMng ?R -->
  expMng (expSub ?x ?E ?Q) = expMng (expSub ?x ?E ?R)"
```

Here $f(x{:=}E)$ is Isabelle notation for the function that is like f except that its value at argument x is E. The first of these results relates substitution for a variable in an expression to assignment to that variable in the state. The second expresses that if two syntactic expressions have the same meaning, then the results of making the same substitution in the two of them also have the same meaning. (Thanks to Dunne for pointing out the need for this result).

We are now in a position to define assignment and prove its properties. We define `assignv` and `assigne` for the assignment, to a variable, of a value and a (semantic) expression respectively. We also define `assignvs` for the assignment of values to a set of variables.

```
assignv ?var ?n ?state == {Term (?state(?var := ?n))}
assigne ?var ?E ?state == assignv ?var (?E ?state) ?state
assignvs ?vars ?primed ?state ==
   {Term (setvars ?vars ?primed ?state)}
```

We can then prove `ass_trm` (which is trivial – an assignment terminates), and `ass_wlpl`, which says $wlp(x := E, Q) = Q[x := E]$. Although we can prove `ass_wlp`, which is at the level of semantic expressions and conditions, this is trivial, as it follows directly from the definitions.

```
ass_wlp = "wlpm (assigne ?x ?E) ?Q ?st = ?Q (?st(?x := ?E ?st))"
ass_wlpl = "wlpm (assigne ?x (expMng ?E)) (bexpMng ?Q) =
   bexpMng (bexpSub ?x ?E ?Q)"
```

4.2 Normal Form

In [2, §7.1] Dunne gives the following result, giving a "normal form" for an abstract command A.

$$A = \mathit{trm}\ (A) \mid @x'.\mathit{prd}\ (A) \to x := x'$$

Here x is the frame of A (which we first take to be the entire current alphabet of variable names), and x' is a corresponding set of logical variables, with names primed. For this purpose we want somewhat different definitions of @ and of A, involving a set of logical variables x', one for each program variable. So again we use a second state function `primed`, which gives the values of these logical variables, and A will depend on both state functions.

```
atd ?Ad ?state == UN primed. ?Ad primed ?state
```

Here `?Ad` is not a semantic command, but a function which, given a "primed" state as argument, returns a semantic command. Then also the assignment $x := x'$ (where x represents *all* variables) becomes the replacing of state x by "state" x'. Thus we prove the following corresponding result.

```
ACNF = "?A = precon (trmm ?A)
   (atd (%primed. guard (prdm primed ?A) (%st. {Term primed})))"
```

We also proved a corresponding result for the case where x is a proper subset of all variables. Here Dunne's result requires that A does not change variables outside the set x. Rather than specify this requirement as such, we proved a result whose left-hand-side means "A restricted to x", that is, as though you executed A and then reset the variables outside x to their original values.

```
ACNFs = "revert (- ?x) ?A = precon (trmm ?A)
   (atd (%primed. guard (prds ?x primed ?A) (assignvs ?x primed)))"
```

4.3 Frames

In Dunne's formulation [1, §7], each abstract command comes decorated with a frame, and the frame of the new command is defined individually for each abstract command constructor: for example

$$\mathit{frame}\ (A \Box B) = \mathit{frame}\ (A \# B) = \mathit{frame}(A) \cup \mathit{frame}(B)$$

However we are unable to give an exact semantic meaning to the frame in a similar sense to the meaning we have given to commands so far. The frame may be thought of as a set of variables "potentially" set by a command, but it can be larger than the set of variables actually set by the command. The frame may be smaller than the set of variables read by the command, and two commands which have the same semantic meaning can have different frames. Accordingly we could not attempt to prove the statements about frames given by Dunne in his definitions of abstract commands from our operational model, in the way we have done for their *wlp* and *trm* conditions. The best one could do is to attempt to prove that for any abstract command the frame of the result *contains* the set of variables which are changed by the command. However this does not look at all difficult in any case, and so we have not included frames in our model.

parallel composition
[1, §12] This is the only abstract command operator whose meaning depends on the frames of its operands. The command $A||B$ executes A and B, independently, each on its own copy of the variables in its frame, and waits until both have terminated. (Thus, non-termination is a possible outcome of $A||B$ if it is possible for either A or B). We say a new state resulting from A is *compatible* with a new state resulting from B if these new states agree on the values they give to the variables in $\mathit{frame}(A) \cap \mathit{frame}(B)$. Then, for each (s_A, s_B), where s_A and s_B are compatible new states resulting from A and B respectively, there is an outcome `Term` s_{AB} of $A||B$, where s_{AB} is given by:

- the new values of variables in $\mathit{frame}(A) \cap \mathit{frame}(B)$ are as in s_A (or s_B),
- the new values of variables in $\mathit{frame}(A) \backslash \mathit{frame}(B)$ are as in s_A, and
- the new values of variables in $\mathit{frame}(B) \backslash \mathit{frame}(A)$ are as in s_B.

Dunne defines $A||B$ by

$$\begin{aligned} \mathit{trm}(A||B) &= \mathit{trm}(A) \wedge \mathit{trm}(B) \\ \mathit{prd}(A||B) &= \mathit{prd}(A) \wedge \mathit{prd}(B) \end{aligned}$$

but the latter formula contains an implicit reference to the frames of the commands. It is interesting to note that if A is infeasible, and B is feasible but does not terminate, then $A||B$ is feasible but does not terminate.

We consider first a version of this command for which the frame is the entire set of variables, defined by `pcomp_def` and `pcomprs_def` ; for these, we prove the formulae just mentioned, as `pcomp_prd` and `pcomp_trm`. We also prove as, `pcomp_wlp`, a result (communicated by Dunne)

$$\mathit{wlp}\ (A||B)\ Q\ s = \exists Q_1\ Q_2. (\forall t. Q_1\ t \wedge Q_2\ t \Rightarrow Q\ t) \wedge \mathit{wlp}(A, Q_1)\ s \wedge \mathit{wlp}(B, Q_2)\ s$$

Unusually, we have explicitly referred to states s and t in this statement of the result to emphasize that the choice of Q_1 and Q_2 depends on the state s.

The following definition of $A||B$ takes into account the frames of A and B. Firstly, `pccomb` combines two states (resulting from A and B) if they are compatible.

```
"pccomb ?frA ?frB ?initst (?stA, ?stB) =
   (let compat = ALL str:?frA Int ?frB. ?stA str = ?stB str;
    combst = %str.
       if str : ?frA then ?stA str
       else if str : ?frB then ?stB str else ?initst str
    in if compat then {Term combst} else {})"

"pcompfr ?frA ?A ?frB ?B ?state ==
   let tsA = {st. Term st : ?A ?state};
     tsB = {st. Term st : ?B ?state};
     nont = {NonTerm} Int (?A ?state Un ?B ?state)
   in nont Un UNION (tsA <*> tsB) (pccomb ?frA ?frB ?state)"
```

Here `(tsA <*> tsB)` means the set product of `tsA` and `tsB`. We have a result `pcomp_chk` which is a sanity check that, where the frames of A and B are the set of all variable names, this definition is equivalent to the one mentioned in the previous paragraph (a useful check, since our first attempt at the definition above was erroneous). Noting that Dunne's formula $\mathit{prd}(A||B) = \mathit{prd}(A) \wedge \mathit{prd}(B)$ implicitly refers to the frames of the commands, we prove it as `pcompfr_prd`, as follows:

```
pcompfr_prd = "prds (?fA Un ?fB) ?primed (pcompfr ?fA ?A ?fB ?B) =
   (prds ?fA ?primed ?A && prds ?fB ?primed ?B)"
```

5 Conclusion

We have provided an operational model for Dunne's abstract commands and their operators, except that our model does not provide any information about the frame of a command. Based upon this model, we have been able to prove, using the automated prover Isabelle, Dunne's definitions of the abstract command operators, except their frames. That is, we have shown that they follow from our operational model — equivalently, that our commands defined operationally satisfy the definitions given by Dunne.

We have discussed the problems in including the frame of a command in this work. Briefly, while the frame of a command might be thought of as the set of variables which "might" be set by the command, commands such as $x := x$ (whose frame is $\{x\}$) prevent us from defining the command's frame from its behaviour. We could have tried to show that the frame of a command (as defined by Dunne) conforms to a rule that the frame contains any variable which can be changed by the command, but this generally seems obvious.

Formalising the various definitions for use in the mechanised prover has highlighted aspects of the specification of commands which need to be considered, but are easily overlooked until one formalises them. Examples of this appear in our discussions about "syntactic" and "semantic" expressions and commands, about the language used to form "syntactic" expressions, and about the meaning of quantifying over a set x of program variables.

Acknowledgement.
We wish to thank Steve Dunne for his very great assistance in some lengthy discussions on the topic.

References

[1] Dunne, Steve, *Abstract Commands: A Uniform Notation for Specifications and Implementations*, In Computing: The Australasian Theory Symposium (2001), ENTCS **42** (2001), 104–123. http://www.elsevier.nl/gej-ng/31/29/23/68/22/show/Products/notes/index.htt#008

[2] Dunne, Steve, A Case for General Correctness, submitted.

[3] Gordon, Michael J. C., Mechanizing Programming Logics in Higher Order Logic. In G. Birtwistle and P. A. Subrahmanyam (editors), Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989.

[4] Harrison, John, Formalizing Dijkstra. In Jim Grundy, Malcolm C. Newey (Eds.): Theorem Proving in Higher Order Logics, (TPHOLS'98), Lecture Notes in Computer Science, Vol. 1479, Springer, 1998, 171-188.

[5] Jacobs, Dean and David Gries. *General Correctness: A Unification of Partial and Total Correctness*. Acta Informatica **22** (1985), 67-83.

[6] Lermer, Karl, Colin Fidge and Ian Hayes. Formal Semantics for Program Paths. In Computing: The Australasian Theory Symposium (2003), ENTCS **78** (2003), 54–77.

[7] Liang, Sheng, Paul Hudak, and Mark P Jones. Monad Transformers and Modular Interpreters. In Symposium on Principles of Programming Languages (POPL'95), 1995, 333–343.

[8] Nelson, Greg, *A generalization of Dijkstra's calculus*. ACM Transactions on Programming Languages and Systems, 11 (1989), 517-61. Or see DEC (now Compaq) SRC Research Report 16, http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-016.html

[9] Plotkin, Gordon D., *A Power-domain construction*. SIAM J. Comput. **5** (1976), 452–487.

[10] Wadler, Philip, The Essence of Functional Programming. In Symposium on Principles of Programming Languages (POPL'92), 1992, 1–14.