Formalising Generalised Substitutions

Jeremy E. Dawson[1,2]

[1] Logic and Computation Program, NICTA *
[2] Automated Reasoning Group,
Australian National University, Canberra, ACT 0200, Australia
http://users.rsise.anu.edu.au/~jeremy/

Abstract. We use the theorem prover Isabelle to formalise and machine-check results of the theory of generalised substitutions given by Dunne and used in the B method. We describe the model of computation implicit in this theory and show how this is based on a compound monad, and we contrast this model of computation and monad with those implicit in Dunne's theory of abstract commands. Subject to a qualification concerning frames, we prove, using the Isabelle/HOL theorem prover, that Dunne's results about generalised substitutions follow from the model of computation which we describe.
Keywords: general correctness, generalised substitution

1 Introduction

In [7] Dunne gave an account of general correctness, which combines the concepts of partial correctness and total correctness, arguing for its utility in analysing the behaviour of programs. He defined a language of "abstract commands", giving several basic abstract commands and operators for joining them, for which he gave rules in terms of weakest liberal preconditions and termination conditions. In [6] we considered this abstract command language and described the operational interpretation of the abstract commands, showing how it is based on a compound monad. We used the automated theorem proving system Isabelle to prove that the operational interpretation implies the rules given by Dunne.

In [3] Chartier formalised the operations of the B method of Abrial [1] in Isabelle/HOL. He formalised generalised substitutions as an abstract type comprising the *trm* and *prd* functions and a list of variables involved in the substitution. Defining the generalised substitution operations in terms of *trm* and *prd*, he proved that these definitions are equivalent to those of Abrial's definitions in terms of the weakest precondition functions.

In [8] Dunne considered the generalised substitutions used in the B method. He developed the notion of the frame of a substitution, the variables "involved" in it, and defined generalised substitution operations in terms of frames and

* National ICT Australia is funded by the Australian Government's Dept of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Centre of Excellence program.

weakest preconditions. He then proved a number of properties of these generalised substitutions. In contrast to [7], this theory is based on total correctness.

In this paper we formalise this theory of generalised substitutions, using a similar approach to our work on abstract commands in [6], and using some of its results. That is, we develop a model of computation and define the theory in terms of it. We then find that this enables us to derive the previous definitions and results as consequences of our formulation. These results are proved using the Isabelle/HOL theorem prover, see [5]. We have also performed, in Isabelle, proofs of other results of Dunne in [8], of which details are in the Appendix.

We find that the model of computation we use is also based on a compound monad, and provides an interesting example of a distributive law for monads. Furthermore, this distributive law is also a monad morphism from the monad of [6] to the monad described in this paper.

2 The Operational Models

In [7] Dunne argued that general correctness provides a better framework for program refinement than either total or partial correctness, and its relative simplicity is supported by the results at the end of [6, §3.2]. However the theory of generalised substitutions as used in the B method is based on total correctness, so that when two generalised substitutions with the same frame are equivalent in total correctness, they are regarded as the same, although they may not be equivalent in terms of general correctness. So we need to model program (statements) in such a way that two such generalised substitutions are equal.

So we describe the two operational models. Firstly, we review the operational model of [6], which we used for *abstract commands*, based on *general correctness*. Then we describe the model, on which this paper is based, which fits the *total correctness* framework of *generalised substitutions*. We describe these models, at first without reference to frames, which we discuss in §4. Furthermore, where we state that we have proved a result of Dunne [8], this will usually refer to the result as modified by deleting reference to the frames of the substitutions.

2.1 The General Correctness Operational Model

To express that a command can either terminate in a new state or fail to terminate, in [6] we considered command *outcomes*, where an outcome is either termination in a new state or non-termination. Then we model a command as a function, of type *state* $\rightarrow$ *outcome set*, from states to sets of outcomes. Commands are equal if the corresponding functions are equal: that is, we have an extensional definition of equality. This model can distinguish between a command which (when executed in a particular given state) must fail to terminate from one which may or may not fail to terminate. Since Dunne's treatment of abstract commands [7] distinguished between two such commands, this model was effective in considering abstract commands. However a theory of total correctness, using weakest preconditions (which are satisfied only when a command is guaranteed to terminate), does not distinguish between two such commands.

2.2 The Total Correctness Operational Model

For the total correctness model, we model a generalised substitution as a function returning either the tag `NonTerm`, indicating possible non-termination, or `Term` S, indicating guaranteed termination in one of the states contained in the set S. Note that this implies an extensionsal definition of equality: substitutions are equal iff they are equal considered as functions of the appropriate type.

To do this we declare the Isabelle datatype

$$\texttt{datatype } \sigma \texttt{ TorN = NonTerm | Term } \sigma$$

where σ is a type variable. This means that a value of the type σ `TorN` is either the tag `NonTerm` or a member of the type σ, tagged with the tag `Term`. (Thus the type *outcome* of [6] is *state* `TorN`). We then define the type *tcres* to be *state set* `TorN`, that is, either non-termination, or termination in (one of) a set of states. (As in Isabelle, we write a type constructor after the type.)

Then we model a generalised substitution as a function of type *state* $\rightarrow$ *tcres*. For a generalised substitution C, we define $[C]$ (or $wp\, C$, the weakest precondition of C) for post-condition Q and initial state s, as follows. Then, from it, we define total correctness refinement $\sqsubseteq_{tc}$. By $P \longrightarrow Q$ we mean $\forall s.\ P\ s \rightarrow Q\ s$.

$$[C]\ Q\ s = \exists S.\ (\forall x \in S.\ Q\ x) \wedge C\ s = \texttt{Term}\ S$$
$$A \sqsubseteq_{tc} B = \forall Q.\ [A]\ Q \longrightarrow [B]\ Q$$

```
"wp_tc C Q s == EX S<=Collect Q. C s = Term S"
"ref_tc A B == ALL Q. wp_tc A Q ---> wp_tc B Q"
```

The definition makes it clear that refinement is a preorder. We then obtain a direct characterisation of refinement, and, from it, we show (as `ref_tc_antisym`) that two generalised substitutions A, B (of type *state* $\rightarrow$ *tcres*) are refinement-equivalent if and only if they are equal in our operational model. Thus refinement is a partial order. This confirms that the operational model above is appropriate for total correctness refinement and equivalence. The related result `wp_tc_inj` is useful for proving the equality of two generalised substitutions.

```
ref_tc_alt = "ref_tc A B ==
  ALL s SA. A s = Term SA --> (EX SB<=SA. B s = Term SB)"
ref_tc_antisym = "[| ref_tc A B; ref_tc B A |] ==> A = B"
wp_tc_inj = "wp_tc A = wp_tc B ==> A = B"
```

2.3 The Total Correctness Compound Monad

In [6, §3.1] we showed that the general correctness operational monad gave a monad, which we will call the *outcome set* monad. See [6, §3.1] for a brief discussion of monads, or Wadler [10] for further information. We now find that the type *tcres*, relative to the type *state*, is a monad, the *total correctness* monad.

To define a monad M, we need to define the unit and extension functions, of the types shown. The unit function models the command which does nothing (*skip*) and the extension function is used to model sequencing of commands since *ext A* models the action of command A on the output of a previous command. We then need to show that the unit and extension functions satisfy the following rules required for a monad.

$$
\begin{aligned}
\mathit{unit} &: \alpha \to \alpha M \\
\mathit{ext} &: (\alpha \to \beta M) \to (\alpha M \to \beta M)
\end{aligned}
$$

$$\mathit{ext}\ f \circ \mathit{unit} = f \tag{1}$$
$$\mathit{ext}\ \mathit{unit} = \mathit{unit} \tag{2}$$
$$\mathit{ext}\ (\mathit{ext}\ g \circ f) = \mathit{ext}\ g \circ \mathit{ext}\ f \tag{3}$$

As a standard result (see [10]), a monad can be characterised either by the three functions *unit*, *map* and *join*, and seven axioms involving these functions, or the functions *unit* and *ext* and the three axioms shown above. Rule (3) lets us define sequencing of commands, $A; B$ (or *seq A B*) $= \mathit{ext}\ B \circ A$, and, as in [6, §3.1], the associativity of *seq* (which obviously ought to hold!) follows from (3).

We have that $\mathit{tcres} = \mathit{state}\ \mathit{set}$ `TorN`. Each of the type constructors *set* and `TorN` with their associated unit and extension functions, is a monad. It does not follow, however, that *tcres* (relative to *state*) is a monad.

To prove that the total correctness monad is in fact a monad, we used the results of Dawson [4], which develop those of Jones & Duponcheel [9]. As in [9], we consider the composition of two monads M and N, but as in Isabelle, we write a type constructor after the type, so the compound monadic type is αNM. We write ext_{NM}, ext_M, ext_N for the extension functions of NM, M, N.

To get a compound monad, we need the function ext_{NM}, which "extends" a function f from a "smaller" domain, α, to a "larger" one, αNM. Consider, therefore, a "partial extension" function which does part of this job:

$$
\begin{aligned}
\mathit{ext}_{NM} &: (\alpha \to \beta NM) \to (\alpha NM \to \beta NM) \\
\mathit{pext} &: (\alpha \to \beta NM) \to (\alpha N \to \beta NM)
\end{aligned}
$$

The following rules and definitions are sufficient to define a compound monad using such a function *pext*.

$$\mathit{pext}\ f \circ \mathit{unit}_N = f \tag{4}$$
$$\mathit{pext}\ \mathit{unit}_{NM} = \mathit{unit}_M \tag{5}$$
$$\mathit{pext}\ (\mathit{ext}_{NM}\ g \circ f) = \mathit{ext}_{NM}\ g \circ \mathit{pext}\ f \tag{6}$$
$$\mathit{ext}_{NM}\ g = \mathit{ext}_M\ (\mathit{pext}\ g) \tag{7}$$
$$\mathit{unit}_{NM} = \mathit{unit}_M \circ \mathit{unit}_N \tag{8}$$

We now give the definitions for our particular monads $N = \mathit{set}$ and $M =$ `TorN`, the suffix *_tc* indicating the total correctness monad.

$$
\begin{aligned}
\mathit{unit\_tc} &: \mathit{state} \to \mathit{tcres} \\
\mathit{prod\_tc} &: \mathit{tcres}\ \mathit{set} \to \mathit{tcres} \\
\mathit{pext\_tc} &: (\mathit{state} \to \mathit{tcres}) \to \mathit{state}\ \mathit{set} \to \mathit{tcres} \\
\mathit{ext\_tc} &: (\mathit{state} \to \mathit{tcres}) \to \mathit{tcres} \to \mathit{tcres}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{unit\_tc}\ s &= \mathtt{Term}\ \{s\} && && (9)\\
\mathit{prod\_tc}\ S &= \mathtt{NonTerm} && \text{if } \mathtt{NonTerm} \in S && (10)\\
\mathit{prod\_tc}\ (\mathtt{Term}`S) &= \mathtt{Term}\ (\bigcup S) && && (11)\\
\mathit{pext\_tc}\ A\ S &= \mathit{prod\_tc}\ (A`S) && && (12)\\
\mathit{ext\_tc}\ A\ S &= \mathit{ext\_o}\ (\mathit{pext\_tc}\ A)\ S && && (13)
\end{aligned}
$$

where $\mathit{ext\_o}$ is the extension function of the `TorN` monad (see [6]), given by

$$
\begin{aligned}
\mathit{ext\_o}\ f\ \mathtt{NonTerm} &= \mathtt{NonTerm} && (14)\\
\mathit{ext\_o}\ f\ (\mathtt{Term}\ s) &= f\ s && (15)
\end{aligned}
$$

and $f`S$ is Isabelle notation for $\{f\ s \mid s \in S\}$.

We have proved, in Isabelle, the following result. We did this by proving rules (4) to (6), noting that (7) and (8) follow directly from our definitions.

Theorem 1. *σ set TorN is a compound monad.*

2.4 Relation to the outcome set monad

Jones & Duponcheel [9] also use a function $\mathit{swap} : \alpha M N \to \alpha N M$ to define a compound monad. As they show, when such a function *swap* can be defined, satisfying certain conditions S(1) to S(4), then the compound monad $\alpha N M$ can be constructed. Equivalently, the function *swap* is a distributive law for monads, see Barr & Wells [2, §9.2]. Jones & Duponcheel also use the function *prod* as above, and give conditions for defining a compound monad in terms of *prod*.

In fact the total correctness monad can be defined using *swap*. In this case M is the outcome monad, and N is the set monad, and *swap* is a function

$$
\begin{aligned}
&\mathit{swap\_tc} : \sigma\ \mathtt{TorN}\ \mathit{set} \to \sigma\ \mathit{set}\ \mathtt{TorN} && &&\\
\mathit{swap\_tc}\ S &= \mathtt{NonTerm} && \text{if } \mathtt{NonTerm} \in S && (16)\\
\mathit{swap\_tc}\ (\mathtt{Term}`S) &= \mathtt{Term}\ S && && (17)
\end{aligned}
$$

Here, definition (16) reflects the fact that, in total correctness, a command that *may* fail to terminate is equivalent to one which *will* fail to terminate.

Our Isabelle proofs included the conditions S(1) to S(4) of [9], and so we also have shown that *swap_tc* is a distributive law for the monads.

The function $\mathit{swap\_tc} : \sigma\ \mathtt{TorN}\ \mathit{set} \to \sigma\ \mathit{set}\ \mathtt{TorN}$ is also a *monad morphism* from the *outcome set* monad to the *total correctness* monad. We have proved the following theorems (which characterise a monad morphism), where *unit_os* and *ext_os* are the unit and extension functions for the outcome set monad, as defined in [6, §3.1]:

$$
\begin{aligned}
\mathit{unit\_os} &: \mathit{state} \to \mathit{outcome\ set}\\
\mathit{ext\_os} &: (\mathit{state} \to \mathit{outcome\ set}) \to \mathit{outcome\ set} \to \mathit{outcome\ set}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{unit\_tc}\ a &= \mathit{swap\_tc}\ (\mathit{unit\_os}\ a) && (18)\\
\mathit{ext\_tc}\ (\mathit{swap\_tc}\ \circ f)\ (\mathit{swap\_tc}\ x) &= \mathit{swap\_tc}\ (\mathit{ext\_os}\ f\ x) && (19)
\end{aligned}
$$

Since this monad morphism is surjective, we could use the fact that the outcome set monad satisfies the monad axioms to give an alternative proof to show that the total correctness monad also satisfies them.

Often, where two monads can be composed to form another monad, the construction depends on one of them, and the other may be arbitrary. Thus, as discussed in [6, §3.1], the `TorN` monad can be composed with any other monad M to give a compound monad, which gave the *outcome set* monad. In this case we have that the type σ *set* `TorN` is a monad (relative to σ), but it does not seem to be an example of a more general construction, in that for an arbitrary monad M, neither σ M `TorN` nor σ *set* M is in general a monad.

We also proved the following theorem about abstract commands. Dunne's treatment of general correctness in [7] includes a definition of total correctness refinement, which is referred to as `totcref` in the result below. This result also confirms that the operational model of §2.2 is appropriate for total correctness. For two abstract commands A, B (of type *state* $\rightarrow$ *outcome set*) the left-hand side says $A \sqsubseteq B$ according to the total-correctness refinement relation for abstract commands, as defined and discussed in [7] and [6, §3.2]. The right-hand side says that the refinement relation for generalised substitutions holds of the projections of A and B into the type *state* $\rightarrow$ *tcres*.

```
ref_tc_swap = "totcref A B = ref_tc (swap_tc o A) (swap_tc o B)"
```

Often in this work, we drew upon [6], using the fact that if two abstract commands are equal, then so are their projections into the total correctness monad.

3 The Generalised Substitutions

Frames Dunne has also defined that each substitution 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, considered in isolation, does. Thus when we show, for example, that two commands behave the same way, we do so without considering their frames. Indeed, the substitution *skip* and $x := x$ behave the same way, but have different frames. Likewise, it is impossible to deduce the frame of a substitution from its behaviour. The work in this section proceeds on this basis. Therefore the results are therefore subject to the proviso that two generalised substitutions are in fact distinct if their frames differ.

On the other hand, the specified frame of a substitution does have an effect in that the parallel composition $S \| T$ depends on the frames of S and T — that is, if the frames of S or T are extended, then that changes $S \| T$.

Variables Indeed, for many generalised substitution operations, we do not need to consider variables at all: rather, we can consider a machine state abstractly.

For others, we need to consider a state as a map from variables (variable names) to values. As discussed at greater length in [6], where Q is a predicate on states, we may use the notation $Q[x := E]$ to mean Q, with occurrences of x replaced by E, when Q is written in the command language, or some similar notation. This is found in the *wp* rule for assignment, and in the related Hoare logic rule.

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

In fact we could take $Q[x := E]$ to be defined as follows. Considering expression E as a function from states to values, $Q[x := E]\ s = Q\ (s[x := E\ s])$ where, for state s, $s[x := E\ s]$ means the function s, changed at the domain point x. (though using this as a *definition* makes the *wp* rule above rather trivial, since assignment will be defined to take state s to state $s[x := E\ s]$).

3.1 Meaning of Commands

skip, magic, abort [8, §3.1, §3.3] *skip* is the command which is feasible, terminates and does nothing to the state. It is exactly the function *unit_tc*. It follows immediately from the monad laws (1) and (2) that *skip* is an identity for the binary function *seq_tc*. These are proved in Isabelle as `seq_tc_unitL/R`.

We *define magic* and *abort* in terms of the operational model, as `magic_tc_def` and `abort_tc_def`; then with these definitions, we then *prove* Dunne's definitions, as `magic_alt` and `abort_alt` (using `precon_tc` and `guard_tc`, see below).

As *abort* always fails to terminate, it fails to satisfy any post-condition. On the other hand, *magic* is always *infeasible*: while not suffering non-termination, it cannot produce any result which fails to satisfy any given post-condition, so it satisfies every post-condition. So we also prove that *magic* and *abort* are the top and bottom members in the lattice of generalised substitutions [8, §7].

```
magic_tc_def = "magic_tc s == Term {}"
abort_tc_def = "abort_tc s == NonTerm"
magic_alt = "magic_tc = guard_tc (%s. False) unit_tc"
abort_alt = "abort_tc = precon_tc (%s. False) unit_tc"
top_magic_tc = "ref_tc C magic_tc"
bot_abort_tc = "ref_tc abort_tc C"
```

preconditioned command, guarded command [8, §3.1] The preconditioned command $P|A$ is the same as A except that, if P does not hold, then $P|A$ need not terminate. The guarded command $P \Longrightarrow A$ is the same as A if P holds, but is *infeasible* (it cannot reach any outcome, that is, it cannot run) if P does not hold. Dunne defines both of these by giving the formula for their weakest precondition. We define them using `precon_tc_def` and `guard_tc_def`, and then prove Dunne's definitions as `precon_wp_tc'` and `guard_wp_tc`.

```
precon_tc_def = "precon_tc P C s == if P s then C s else NonTerm"
guard_tc_def = "guard_tc P C s == if P s then C s else Term {}"
precon_wp_tc' = "wp_tc (precon_tc P C) Q s = (P s & wp_tc C Q s)"
guard_wp_tc = "wp_tc (guard_tc P C) Q s = (P s --> wp_tc C Q s)"
```

termination, feasibility [8, §5] We define

```
"trm_tc C s == C s ~= NonTerm"
"fis_tc C s == C s ~= Term {}"
```

We can then prove Dunne's definition of *trm* and *fis*, and his results in [8, §5]:

```
trm_alt = "trm_tc C = wp_tc C (%s. True)"
fis_alt = "fis_tc C = Not o wp_tc C (%s. False)"
pc_trm_tc = "precon_tc (trm_tc A) A = A"
fis_guard_tc = "guard_tc (fis_tc A) A = A"
strongest_guard = "(guard_tc g A = A) = (fis_tc A ---> g)"
strongest_pc = "(precon_tc pc A = A) = (trm_tc A ---> pc)"
```

sequencing [8, §3.1] As mentioned earlier, we define sequencing of commands using *ext_tc*, as shown in `seq_tc_def`. Dunne defines it by giving the weakest precondition of $A; B$, and we prove this result, as `seq_wp_tc`, from our definition.

```
seq_tc_def = "seq_tc A B == ext_tc B o A"
seq_wp_tc = "wp_tc (seq_tc A B) Q = wp_tc A (wp_tc B Q)"
```

choice In [8, §3.1] 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. Again, Dunne defines this by giving its weakest precondition. This is a special case of choice among an arbitrary set of commands. In the total correctness setting, where *choice_tc* $\mathcal{C}$ can fail to terminate if any $C \in \mathcal{C}$ can fail to terminate, we define *choice_tc* as shown below.

As the definition is rather unintuitive, we show the types of some of its parts. Recall that if the type σ represents the machine state, then a command has type $\sigma \to \sigma$ *set* `TorN`. The definition is unintuitive perhaps because where *pext* is used (indirectly) in defining sequencing of commands, the types α and β are both the state type σ. But in the use of *pext* below, α is the type of commands.

$$\mathit{choice\_tc}\ \mathcal{C}\ s = \mathit{pext}\ (\lambda C.\ C\ s)\ \mathcal{C}$$

$$\begin{aligned}
\mathit{choice_tc} &: (\sigma \to \sigma\ \mathit{set}\ \mathtt{TorN})\ \mathit{set} \to \sigma \to \sigma\ \mathit{set}\ \mathtt{TorN} \\
\mathit{pext} &: (\alpha \to \beta\ \mathit{set}\ \mathtt{TorN}) \to \alpha\ \mathit{set} \to \beta\ \mathit{set}\ \mathtt{TorN} \\
\lambda C.\ C\ s &: (\sigma \to \sigma\ \mathit{set}\ \mathtt{TorN}) \to \sigma\ \mathit{set}\ \mathtt{TorN} \\
\mathit{pext}\ (\lambda C.\ C\ s) &: (\sigma \to \sigma\ \mathit{set}\ \mathtt{TorN})\ \mathit{set} \to \sigma\ \mathit{set}\ \mathtt{TorN}
\end{aligned}$$

When the definition is expanded (`choice_tc_def''` in the Isabelle proofs), it shows that if $\{C\ s \mid C \in \mathcal{C}\}$ contains `NonTerm` then *choice_tc* $\mathcal{C}\ s =$ `NonTerm`; if $\{C\ s \mid C \in \mathcal{C}\} = \{\mathtt{Term}\ S_C \mid C \in \mathcal{C}\}$, then *choice_tc* $\mathcal{C}\ s = \mathtt{Term}(\bigcup_{C \in \mathcal{C}} S_C)$.

We obtained the following results, relating the distribution of sequencing over choice. The theorem `seq_choice_tcL` was obtained as an easy corollary of rule (6), obtained in the course of the proofs about the total correctness monad. We also proved a result giving the weakest precondition of *choice_tc*, which is the generalisation of Dunne's definition of $A \Box B$, and from which it easily follows that *choice_tc* $\mathcal{C}$ is the glb of the set $\mathcal{C}$. Proposition 7 of [8, p288] follows.

```
seq_choice_tcL =
  "seq_tc (choice_tc Cs) B = choice_tc ((%C. seq_tc C B) ` Cs)"
seq_choice_tcR = "Cs ~= {} ==>
  seq_tc A (choice_tc Cs) = choice_tc (seq_tc A ` Cs)"
choice_wp_tc = "wp_tc (choice_tc Cs) Q s = (ALL C:Cs. wp_tc C Q s)"
choice_glb_tc = "ref_tc A (choice_tc Cs) = (ALL C:Cs. ref_tc A C)"
```

We note that for many generalised substitutions, the definition may be obtained by translation from the definitions in [6] for abstract commands. For example, for *choice_tc* we could have defined *choice_tc* $\mathcal{C}$ in terms of *choice* $\mathcal{A}$ for any set $\mathcal{A}$ of abstract commands corresponding to the set $\mathcal{C}$ of generalised substitutions. Alternatively, we can relate our definitions to the definitions of the corresponding abstract commands, where `seq`, `precon`, `guard` and `choice` are the corresponding operations on abstract commands: [6, §3.4].

```
seq_tc = "seq_tc (swap_tc o A) (swap_tc o B) = swap_tc o seq A B"
precon_tc = "precon_tc P (swap_tc o A) = swap_tc o precon P A"
guard_tc = "guard_tc P (swap_tc o A) = swap_tc o guard P A"
choice_tc = "choice_tc (op o swap_tc ` As) = swap_tc o choice As"
```

where `(op o swap_tc ` As)` means $\{swap\_tc \circ A \mid A \in As\}$ (We have similar results for `magic_tc`, `abort_tc`, `trm_tc` and `fis_tc` also). These results enable us to prove many results for generalised substitutions from the corresponding results for abstract commands. In some cases, such as for the theorem `choice_wp_tc`, this provided much simpler proofs in Isabelle.

3.2 Monotonicity

For developing a program by starting with a generalised substitution (expressing a program specification), and progressively refining it to a concrete program, it is important that the generalised substitution constructors are monotonic with respect to the refinement relation ($\sqsubseteq$). All the constructors mentioned are monotonic. We proved these results in Isabelle as (for example)

```
seq_tc_ref_mono = "[| ref_tc A1 B1; ref_tc A2 B2 |] ==>
    ref_tc (seq_tc A1 A2) (seq_tc B1 B2)"
rephat_ref_mono = "ref_tc A B ==> ref_tc (rephat A) (rephat B)"
```

3.3 Repetition and Iteration for the General Correctness Model

In [7, §7] Dunne defined $A^0 = skip$ and $A^{n+1} = A; A^n$, and we proved that $A^{n+1} = A^n; A$. From this we defined *repall* $A\ s = \bigcup_n A^n s$, that is, `repall` A is the (unbounded) choice of any number n of repetitions of A; it terminates iff for every n, A^n terminates (proved as `repall_term`).

In [7, §12] Dunne defined the *repetitive closure* A^* of 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 (calling this an "infinite chain").

Thus, in [6, §3.5] we defined a function *infch*, where *infch* A s means that it is possible to execute A infinitely many times sequentially, starting in state s.

So we had the following definition, from which we proved various results from [7], including characterisations of `repall` and `repstar` as fixpoints.

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

A Coinductive Definition In [6, §3.5] we defined the concept of an infinite chain explicitly, in terms of the existence of an infinite sequence of states through which the executing program can pass. Some of these Isabelle proofs involving this definition were quite difficult.

Subsequently we used Isabelle's coinductive definition facility to give a more elegant definition of an equivalent notion: *infchs* A is the set of states from which it is possible to execute A infinitely many times sequentially. We also defined inductively a set *icnt* A, which we showed is equivalent to $\{s \mid \texttt{NonTerm} \in A^*\ s\}$.

The Isabelle definitions are:

```
coinductive "infchs A"
  intros I : "Term ns : A s ==> ns : infchs A ==> s : infchs A"

coinductive "icnt A"
  intros NTI : "NonTerm : A s ==> s : icnt A"
    icI : "Term ns : A s ==> ns : icnt A ==> s : icnt A"
```

This defines *infchs* A to be the unique maximal set satisfying

$$\mathit{infchs}\ A = \{s \mid \exists s'.\ \texttt{Term}\ s' \in A\ s \wedge s' \in \mathit{infchs}\ A\}$$

Then Isabelle's coinductive definition facility provides a coinduction principle:

$$\frac{a \in X \qquad \forall z.z \in X \Rightarrow \exists s'.\texttt{Term}\ s' \in A\ z \wedge s' \in X \cup \mathit{infchs}\ A}{a \in \mathit{infchs}\ A}$$

and similarly for *icnt*. We then proved that $s \in \mathit{infchs}\ A$ if and only if *infch* A s holds, and that $s \in \mathit{icnt}\ A$ if and only if $\texttt{NonTerm} \in A^*\ s$. This made some other proofs considerably easier than before.

3.4 Repetition and Iteration for the Total Correctness Model

For the total correctness model we used analogous definitions. We used a coinductive definition to define *infchs_tc* C, the set of states from which it is possible to execute C infinitely many times sequentially and an inductive definition for *reach_NT* C, the set of states from which `NonTerm` is reachable.

```
coinductive "infchs_tc C"    intros
"C s = Term S ==> ns : S ==> ns : infchs_tc C ==> s : infchs_tc C"
```

```
inductive "reach_NT C"
 intros "C s = NonTerm ==> s : reach_NT C"
   "C s = Term S ==> ns : S ==> ns : reach_NT C ==> s : reach_NT C"
```

Then we define $\mathit{icnt\_tc}\, C$, using the same introduction rules as for $\mathit{reach\_NT}\, C$, but in a coinductive definition, not an inductive definition. We then prove that $\mathit{icnt\_tc}\ C = \mathit{reach\_NT}\ C \cup \mathit{infchs\_tc}\ C$, and can relate *icnt_tc* to *icnt*.

```
icnt_alt = "icnt_tc C = reach_NT C Un infchs_tc C"
icnt_tc = "icnt_tc (swap_tc o A) = icnt A"
```

Also using an inductive definition, we define `treach A s` to be the set of states reachable from `s` using `A` repeatedly.

In [8, §8.2] Dunne defines the generalised substitution $C^\wedge$. He *defines* it as the least fixed point in the refinement ordering $\mu X.\ (C; X) \Box \mathit{skip}$. For us to define it in that way would require showing that the least fixed-point exists: in [8, §8.1] Dunne discusses why this result holds. Rather than proving this result in Isabelle, we *define* $C^\wedge$ using the operational interpretation (suggested in [8, §8.2]) that the result of $C^\wedge$ is the states reachable by repeating C, but with the result `NonTerm` either if `NonTerm` is reachable by repeating C or if an infinite sequence of executions of C is possible. We then *proved* that $C^\wedge$, defined thus, is in fact the least fixed-point of $\lambda X.\ (C; X) \Box \mathit{skip}$. The proofs were more difficult than the corresponding ones for abstract commands, mentioned in [6], and it was necessary to use a range of lemmas, including some of those mentioned in §2.3. We can relate the total correctness repetition constructs to those for general correctness, using *swap_tc* (for example, `rephat_star` below). We proved Dunne's examples, $\mathit{magic}^\wedge$ and $\mathit{skip}^\wedge$. We also defined a function `repall_tc`, analogously to *repall* (see §3.3), and showed that $C^\wedge$ could instead have been defined, analogously to A^*, using it. Among the following, `rephat_def` and `fprep_tc_def` are definitions.

```
rephat_def = "rephat C state ==
  if state : icnt_tc C then NonTerm else Term (treach C state)"
fprep_tc_def = "fprep_tc A X ==
  X = choice_tc {seq_tc A X, unit_tc}"
rephat_isfp = "fprep_tc A (rephat A)"
rephat_is_lfp = "fprep_tc A Y ==> ref_tc (rephat A) Y"
rephat_star = "rephat (swap_tc o A) s = swap_tc (repstar A s)"
rephat_magic = "rephat magic_tc = unit_tc"
rephat_skip = "rephat unit_tc = abort_tc"
```

Dunne then defines the *if* and *while* constructs as follows [8, §9]:

$$\mathit{if\_tc}\ G\ \mathit{then}\ S\ \mathit{else}\ T\ \mathit{end} \equiv (G \Longrightarrow S) \Box (\neg G \Longrightarrow T)$$
$$\mathit{while\_tc}\ G\ \mathit{do}\ S\ \mathit{end} \equiv (G \Longrightarrow S)^\wedge \,;\, \neg G \Longrightarrow \mathit{skip}$$

From these we are then able to prove an alternative definition for *if* and the usual programming definition for *while*:

```
if_tc_prog = "if_tc G A B s = (if G s then A s else B s)"
ifthen_tc_prog =
  "ifthen_tc G A s = (if G s then A s else Term {s})"
while_tc_prog =
  "while_tc G A = ifthen_tc G (seq_tc A (while_tc G A))"
```

3.5 The *prd* predicate

[8, §5] The "before-after" predicate *prd* relates the values of the variables in the frame before execution of the command to their values after the command. It is defined in [8, §5] as $prd\ (S) \equiv \neg[S](s \neq s')$ where $s = frame(S)$ and s' are new (logical) variables corresponding to the program variables s. Implicitly, $prd\ (S)$ depends on s'; $(s \neq s')$ is a post-condition on the values of s after executing S.

First we define `prd_tc` which assumes that the frame consists of all variables. Then, for the analysis of `prd_tc`, it is possible to treat the state as abstract.

```
"prd_tc s' C == Not o wp_tc C (%s. s ~= s')
```

where `s'`, of type *state*, represents the values s'. We note a difference between the definitions of [8, §6] and [7, §10]: in the former, if S does not terminate from state s, then $prd(S)s$ *does* hold.

As a sort of inverse to this definition, we derived `wp_prd_tc`, an expression for *wp* in terms of *prd*, namely $[S]Q = trm\ S \wedge \forall s'.\ prd(S) \rightarrow Q$. This was suggested by a similar result in [7, §10].

We also derived [8, §13, p287, Proposition 7] as `ref_tc_prdt`.

```
wp_prd_tc = "wp_tc A Q s =
  (trm_tc A s & (ALL s'. prd_tc s' A s --> Q s'))"
ref_tc_prdt = "ref_tc A B = ((trm_tc A ---> trm_tc B) &
  (ALL s'. prd_tc s' B ---> prd_tc s' A))"
```

3.6 The least upper bound

[8, §7] In §3.1 we showed that, in the refinement ordering, the greatest lower bound of a set of commands is given by the *choice* command. To describe the least upper bound *lub* S T of S and T is more difficult. We might expect that $[lub\ S\ T]\ Q = [S]Q \vee [T]Q$ but this is not so. For if $S\ x = \texttt{Term}\ \{y_S\}$ and $T\ x = \texttt{Term}\ \{y_T\}$, where $y_S \neq y_T$, then $(lub\ S\ T)\ x = \texttt{Term}\ \{\}$ and so $[lub\ S\ T]\ Q\ x$ holds. But if neither $Q\ y_S$ nor $Q\ y_T$ hold, then $([S]Q \vee [T]Q)\ x$ does not hold.

However in [8, §7] Dunne gives a characterisation of the least upper bound of the set of generalised substitutions on a given frame: we prove this result, applied to a set of abstract states, as `lub_tc`. As a corollary of this general result, we get `ACNF_tc`, giving Dunne's normal form, [8, §11, Proposition 4], again in terms of abstract states. These results use the functions `at_tc` and `pc_aux`. The function `at_tc` is analogous to `atd` of [6, §4.2], and is used to express Dunne's unbounded choice over a set of logical variables. The function `pc_aux_tc` is used here and in §7, and $pc\_aux\ \mathcal{C}\ s = \texttt{Term}\ S$, where S consists of those states which every $C \in \mathcal{C}$ can reach from initial state s (we proved this in Isabelle as `pc_aux_alt2`).

```
at_tc_def = "at_tc Ad == choice_tc (range Ad)"
pc_aux_def = "pc_aux Cs == at_tc
  (%s'. guard_tc (%s. ALL C:Cs. prd_tc s' C s) (%s. Term {s'}))"
lub_tc =
  "ref_tc (precon_tc (%s. EX C:Cs. trm_tc C s) (pc_aux Cs)) A =
    (ALL C:Cs. ref_tc C A)"
ACNF_tc = "A = precon_tc (trm_tc A)
  (at_tc (%s'. guard_tc (prd_tc s' A) (%s. Term {s'})))"
```

Having found the least upper bound of a set of generalised substitutions using the function pc_aux, we now derive an expresson for weakest precondition of pc_aux $\mathcal{C}$ in terms of $\{[C] \mid C \in \mathcal{C}\}$, using a function we called lub_pt. We then proved that lub_pt gives the least upper bound of a set of predicate transformers, provided that they are conjunctive (and so monotonic). This is not enough to deduce that pc_aux $\mathcal{C}$ is the least upper bound of $\mathcal{C}$ since weakest precondition functions fail to be conjunctive, where non-termination is involved. Thus the theorem lub_tc contains the termination precondition. In §3.8 we discuss how generalised substitutions correspond to predicate transformers satisfying the non-empty conjunctivity condition.

Here mono_pt T means that T is monotonic. and conj_pt T $\mathcal{Q}$ s means that T is conjunctive in relation to a given set $\mathcal{Q}$ of predicates and state s. The results lub_pt_lub, lub_pt_ub and lub_pt_is_conj, together show that, among conjunctive predicate transformers, lub_pt gives the least upper bound.

```
pc_aux_wp = "wp_tc (pc_aux Cs) = lub_pt (wp_tc ‘ Cs)"
lub_pt_def = "lub_pt Ts Q x == ALL Q':Qcs Q. EX T:Ts. T Q' x"
Qcs_def = "Qcs Q == (%y x. x ~= y) ‘ (- Collect Q)"
lub_pt_lub = "[| ALL Qs. conj_pt T Qs s;
  ALL U:Us. ALL Q. U Q s --> T Q s; lub_pt Us Q s |] ==> T Q s"
lub_pt_ub = "[| mono_pt T; T : Ts |] ==> T Q ---> lub_pt Ts Q"
lub_pt_is_conj = "conj_pt (lub_pt Ts) Qs s"
conj_pt_def = "conj_pt T Qs s ==
  T (%s. ALL Q:Qs. Q s) s = (ALL Q:Qs. T Q s)"
```

3.7 Parallel Composition

[8, §6] Here we describe this in the case where the frames of the commands are the set of all program variables. This enables us to treat the machine state abstractly. We define the parallel composition according to Dunne's informal description: $S||T$ can terminate in a state s only if both S and T can terminate in s (but we define parallel composition of an arbitrary set of generalised substitutions). The "else" part of pcomprs_tc amounts to pcomprs_tc (Term ‘$\mathcal{S}$) = Term ($\bigcap$ $\mathcal{S}$). From this we then derive Dunne's definition, pcomp_tc_alt. We then derive the further results given in [8, §6], and also prd_pcomp_tc, a variant of his expression for $prd(S||T)$ which generalises directly to give the parallel composition of an arbitrary set of generalised substitutions.

```
pcomp_tc_def = "pcomp_tc Cs s == pcomprs_tc ((%C. C s) ` Cs)"
pcomprs_tc_def = "pcomprs_tc rs ==
  if NonTerm : rs then NonTerm else Term (Inter (sts_of_ocs rs))"
pcomp_tc_alt = "pcomp_tc Cs ==
  precon_tc (%s. ALL C:Cs. trm_tc C s) (pc_aux Cs)"
trm_pcomp_tc = "trm_tc (pcomp_tc Cs) s = (ALL C:Cs. trm_tc C s)"
prd_pcomp_tc = "prd_tc s' (pcomp_tc Cs) s =
  (trm_tc (pcomp_tc Cs) s --> (ALL C:Cs. prd_tc s' C s))"
prd_pcomp_tc2 = "prd_tc s' (pcomp_tc {A, B}) s =
  ((trm_tc B s --> prd_tc s' A s) &
    (trm_tc A s --> prd_tc s' B s))"
pcomp_choice_tc = "pcomp_tc {B, choice_tc (insert A As)} =
  choice_tc ((%a. pcomp_tc {B, a}) ` insert A As)"
```

3.8 Healthiness Conditions and positive conjunctivity

[8, §10.1] Dunne asks whether any choice of frame and predicate transformer gives a generalised substitution. He gives three necessary conditions, that is, properties of generalised substitutions. Of these, (GS1) is relevant only when the definitions of *frame*(S) and $[S]$ are given at a syntactic level — they must then be well-defined at the semantic level. Here we are working at the semantic level. Condition (GS3) in effect says that a generalised substitution has no effect outside its frame. However, we look at condition (GS2) which says that a generalised substitution $[S]$ distributes through all non-empty conjunctions (of postconditions): we prove this as `wp_tc_gen_conj`. Given a predicate transformer T, the result `ACNF_tc` enables us to determine the generalised substitution C such that, if T is a weakest precondition, then $T = [C]$. This gives us the definition `gs_of_pt_def` and the theorem `gs_of_pt`. We then prove (as `pt_gc_gs`) that if T is any predicate transformer satisfying the non-empty conjunctivity condition, then $T =$ [`gs_of_pt` T]. That is, the generalised substitutions correspond to the predicate transformers satisfying the non-empty conjunctivity condition.

```
wp_tc_gen_conj = "Q : Qs ==>
  wp_tc C (%s. ALL Q:Qs. Q s) s = (ALL Q:Qs. wp_tc C Q s)"
gs_of_pt_def = "gs_of_pt T == precon_tc (T (%s. True))
  (at_tc (%s'. guard_tc (Not o T (%st. st ~= s')) (%s. Term {s'})))"
gs_of_pt = "T = wp_tc C ==> C = gs_of_pt T"
pt_gc_gs = "[| ALL Q Qs s.  Q : Qs --> conj_pt T Qs s;
  C = gs_of_pt T |] ==> T = wp_tc C"
```

3.9 Properties of Substitutions

We proved the results of Proposition 1 of [8, §10.2], and of Proposition 2 (shown).

```
trm_or_prd_tc = "trm_tc S s | prd_tc s' S s"
prd_tc_imp_fis = "prd_tc s' S ---> fis_tc S"
```

```
fis_or_wp_tc = "fis_tc S s | wp_tc S Q s"
wp_imp_trm_tc = "wp_tc S Q ---> trm_tc S"
```

Our Isabelle proofs of other results of Dunne [8] are listed in the Appendix.

4 Frames and Variable Names

So far, we have viewed a command as a function from a state to either `NonTerm` or a set of new states, and a condition as a predicate on states. In this treatment, the view of a state was abstract. However, as in [6, §4], we also need to discuss frames, and the values of program variables.

In our Isabelle model, as in [6, §4], 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 $\mathit{state} = \mathit{name} \rightarrow \mathit{value}$, or, in Isabelle, `state = "'n => 'v"`.

In Dunne's formulation [7, §7], each generalised substitution comes decorated with a frame, and the frame of the new command is defined individually for each generalised substitution 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 operational behaviour can have different frames. This means that whereas we can deduce the weakest precondition of a generalised substitution from its operational behaviour, we cannot deduce its frame. (We could confirm that it does not change variables outside its defined frame, but this seems straightforward, and we have not done it in Isabelle).

Certain of Dunne's results involving frames can be seen to follow easily from the results earlier which treat the state abstractly. Thus, to see Proposition 4 in its full generality, you consider the state as consisting of only the variables in the frame, and apply the theorem `ACNF_tc`. Similarly to get Dunne's characterisation of the lub of two generalised substitutions with the same frame u, you just apply `lub_tc` to the state consisting only of the variables in u.

We now describe how to express some of Dunne's other results involving frames, and prove them. The condition $x \setminus Q$ is defined to mean that no variable of x appears free in Q. Since we view Q semantically rather than syntactically, we defined `indep` $x\ Q$ to be the condition that changing the value, in a state s, of a variable in x does not change $Q\ s$. In proving these results we also rely on the fact that a generalised substitution S does not change variables outside $\mathit{frame}(S)$. So we defined `frame_tc` $S\ F$ to mean that F *could* be the frame of S, ie, that S does not change variables outside F. In this way we proved the frame circumscription result [8, §10.1, (GS3)], and [8, Proposition 3].

Note that $(Q$ `V` $R)\ s \equiv Q\ s \vee R\ s$ and $(Q$ `&&` $R)\ s \equiv Q\ s \wedge R\ s$.

```
GS3 = "[| frame_tc S F; indep F Q |] ==>
  wp_tc S Q s = (trm_tc S s & (fis_tc S s --> Q s))"
prop3 = "[| frame_tc S F; indep F R |] ==>
  wp_tc S (Q V R) = (wp_tc S Q V trm_tc S && R)"
```

5 Conclusion

We have formalised a computational model suggested by the notion of total correctness underlying the B method, and have proposed definitions for the generalised substitution operators in terms of this model. We have shown that this model and these definitions do in fact imply the characterisations of these operators in terms of weakest preconditions. We have proved these results in Isabelle [5], and have also proved the other results of Dunne in [8] (although not dealing explicitly with frames). Thus we have used formal verification to confirm a body of theory underlying a widely used program development methodology.

We have shown how the computational model is derived from a compound monad, and have compared this model and monad with those arising from Dunne's theory of abstract commands in [7]. We have shown how the monad arises from a distributive law, which is in fact a monad morphism.

Acknowledgements Finally, I would like to thank Steve Dunne and some anonymous referees for some very helpful comments.

References

1. J-R Abrial. The B-Book: Assigning Programs to Meanings. CUP, Cambridge, 1996.
2. Michael Barr and Charles Wells. Toposes, Triples and Theories. Springer-Verlag, 1983, or see `http://www.cwru.edu/artsci/math/wells/pub/ttt.html`
3. Pierre Chartier. Formalisation of B in Isabelle/HOL. In Recent Advances in the Development and Use of the B Method, Second International B Conference (B'98), Lecture Notes in Computer Science 1393, Springer 1998, 66–83.
4. Jeremy E Dawson. Compound Monads and the Kleisli Category. Unpublished note. `http://users.rsise.anu.edu.au/~jeremy/pubs/cmkc/`
5. Jeremy E Dawson. Isabelle files, at `http://users.rsise.anu.edu.au/~jeremy/isabelle/fgc/`
6. Jeremy E Dawson. Formalising General Correctness. In Computing: The Australasian Theory Symposium (2004), ENTCS **91** (2001), 21–42. `http://www.elsevier.com/locate/entcs`
7. Steve Dunne. Abstract Commands: A Uniform Notation for Specifications and Implementations. In Computing: The Australasian Theory Symposium (2001), ENTCS **42** (2001), 104–123. `http://www.elsevier.com/locate/entcs`
8. Steve Dunne. A Theory of Generalised Substitutions. In Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users, Grenoble, 2002 (ZB 2002), Lecture Notes in Computer Science 2272, Springer 2002, 270–290.
9. Mark P Jones & Luc Duponcheel. Composing Monads. Research Report YALEU/DCS/RR-1004, Yale University, December 1993
10. Wadler, Philip, The Essence of Functional Programming. In Symposium on Principles of Programming Languages (POPL'92), 1992, 1–14.