Compound Monads in Specification Languages

Jeremy E. Dawson

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

Abstract

We consider the language of "extended subsitutions" involving both angelic and demonic choice. For other related languages expressing program semantics the implicit model of computation is based on a combination of monads by a distributive law. We show how the model of computation underlying extended subsitutions is based on a monad which, while not being a compound monad, has strong similarities to a compound monad based on a distributive law. We discuss these compound monads and monad morphisms between them. We have used the theorem prover Isabelle to formalise and machine-check our results.

Categories and Subject Descriptors F.3.1 [*LOGICS AND MEANINGS OF PROGRAMS*]: Specifying and Verifying and Reasoning about Programs—Logics of programs, Pre- and post-conditions; D.2.1 [*SOFTWARE ENGINEERING*]: Requirements/Specifications—Languages

General Terms Languages, Theory, Verification

Keywords specification languages, extended substitutions, compound monads, distributive law for monads, generalised substitutions, demonic choice, angelic choice

1. Introduction

In several papers Dunne has proposed specification languages for analysing (variously named) "computations", given rules for combining these computations in various ways, and stated and proved results about them. These languages contain some constructs similar to those found in Dijkstra's guarded command language (Dijkstra 1976). Underlying each of these languages, whether explicitly or implicitly, is a notion of what a computation is, and for some of these we have used the Isabelle theorem prover to define Dunne's various operations in terms of the identified underlying notion of computation, and have proved that defining the operations in this way implies the definitions and results given by Dunne. In this work we used the fact that a compound monad underlies the model of computation involved.

* 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.

PLPV'07, October 5, 2007, Freiburg, Germany.

Dunne (2001) gave an account of general correctness (Jacobs and Gries 1985), which combines the concepts of partial correctness and total correctness, arguing for its utility in analysing the behaviour of programs. Calling the computations "abstract commands", he gave several basic abstract commands and operators for joining them, for which he gave rules in terms of weakest liberal preconditions and termination conditions, and many results about these operators. In (Dawson 2004) we described an operational interpretation of abstract commands equivalent to that of Jacobs and Gries (1985), which used the construction of Plotkin (1976).

Similarly, Dunne (2002) considered the generalised substitutions used in the B method, which is based on total correctness. In (Dawson 2007b) we formalised the theory he gave, and the operational model underlying it, using a similar approach to, and in some cases the results of, (Dawson 2004). We compared this operational model with that of the theory of general correctness.

For both general correctness and total correctness we used the Isabelle/HOL theorem prover to define an operational model and prove that it implied the rules and results given by Dunne. In both cases the operational model is based on a compound monad and in both cases this compound monad can be explained (*inter alia*) in terms of a distributive law for combining two monads (Barr and Wells 1983, §9.2).

Semantic models combining angelic and demonic non-determinism have been developed by Back and von Wright (1998), and Rewitzky (2003), who described a model using binary multirelations, which is developed further in (Martin et al. 2007), for computations involving angelic and demonic non-determinism. In (Back and von Wright 1998) and (Martin et al. 2007) these computations are described in terms of a game between the angel and the demon, where the angel and the demon make "moves" to try to ensure that the postconditon is or is not satisfied. Recently, Dunne (2007) defined an "extended substitution" language, which encompasses the generalised substitutions, allowing for angelic choice as well as demonic choice, ie, choices which will be made to ensure, if possible, that the postcondition is, or is not, satisfied. As explained in §3.1 we would expect to see a monad underlying the model of computation for this language. In this paper we show how the operational model we use (equivalent to the multirelation model) for this language is in fact based on a monad. We found that while it is not a compound monad, it can be described using the concepts and results about joining two monads by a distributive law. Indeed, in proving that it is a monad, we were helped significantly by copying and adapting the proofs of results relating to compound monads based on a distributive law.

In this paper we describe this construction after surveying the role of compound monads based on distributive laws in the operational models for general and total correctness.

In §2 we discuss the operational models underlying each of these theories. In §3 we briefly introduce monads, and compound monads. We then show how the theories of general correctness and

total correctness are each based on a compound monad which is based on a distributive law, and that one of these distributive laws is also a monad morphism between these two monads.

In §4 we describe the construction of the monad underlying the extended substitutions. This is not actually a compound monad but it is based on equivalence classes of a structure rather like a compound monad, and we describe how we used the results on combining two monads by a distributive law to show that this construction gives a compound monad.

We use the interactive theorem prover Isabelle/HOL. Isabelle is a framework based on intuitionistic typed higher-order logic, on which a variety of object logics can be built. We use the HOL ("higher-order logic") object logic, which is a classical strongly typed higher-order logic, giving an object language which has strong similarities to the functional programming languages Standard ML and Haskell. In the logic, functions are total, function equality is defined extensionally, and functions are not necessarily computable. All results referred to in this paper have been proved using Isabelle.

2. The Operational Models

We describe the operational models. Firstly, we make a caveat concerning the role of frames. In each case Dunne's definition of a computation involves a *frame*, which, loosely, is the set of variables which "might" be affected. In this paper we ignore frames (equivalently, we assume the frame to be the set of all variables in the machine state). In fact this enables us to treat the machine state entirely as an abstract type.

In each case, a computation is modelled by a function from the state type to a result type, which is different for each model. Having described the operational model, we showed in each case that two computations are refinement-equivalent (as defined by Dunne, but ignoring frames) if and only if they correspond to the same function in the operational model.

2.1 The General Correctness Operational Model

Firstly, we describe the operational model of (Dawson 2004), used to describe "abstract commands" and general correctness. Jacobs and Gries (1985) (see also (Dunne 2001)) explain how, in the case of a non-deterministic computation, weakest preconditions and total correctness do not distinguish a computation which simply fails to terminate from one which (on a given input) may terminate in a particular final state or may fail to terminate. On the other hand, weakest liberal preconditions and partial correctness fail to distinguish a computation which terminates in a given final state from one which may do that or may fail to terminate. Accordingly, they develop the theory of general correctness, which does distinguish each of these pairs of computations, and in which refinement (as defined in (Dunne 2001)) means partial correctness refinement *and* total correctness refinement. The computational model that we use underlying this theory is the following, suggested by Jacobs and Gries (1985): each computation (on a given initial state) results in (one of) a set of *outcomes*, where each outcome is either non-termination, or termination in a particular final state. Then we proved that two abstract commands are equivalent in general correctness refinement if and only if they are the same function in this model, from states to sets of outcomes.

So we let an outcome be either the tag `NonTerm`, indicating non-termination, or `Term` s, indicating termination in the state s, as expressed in Isabelle by

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

where σ is a type variable, and, in Isabelle, a type constructor follows its argument. 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`. But we will generally write a type constructor such as `TorN` or *set before* its type argument. Thus the type *outcome* is `TorN` *state*, and the type of computations in this model is *state* $\rightarrow$ *set* (`TorN` *state*). Thus a computation which (from a given initial state) can terminate in either state s_1 or s_2 is represented by the outcome set $\{\texttt{Term}\ s_1, \texttt{Term}\ s_2\}$; if the computation can also fail to terminate, it is represented by $\{\texttt{Term}\ s_1, \texttt{Term}\ s_2, \texttt{NonTerm}\}$.

2.2 The Total Correctness Operational Model

Subsequently, Dunne (2002) described generalised substitutions, based on the B method, in which only total correctness is of interest. So we want to consider computations equal if they are equivalent in total correctness refinement: we do not want to distinguish a computation which may terminate or not (on a particular initial state) from one which will not terminate. So here the model of computation we want is that a computation is a function which, given an initial state, returns either possible non-termination, or guaranteed termination, in (one of) a set of final states. We then define the type *tcres* ("total correctness result") to be `TorN` (*set state*), that is, either `NonTerm`, meaning possible non-termination, or `Term` S, termination in (one of) a set S of states. So we define the weakest precondition function $[C]$, where $[C]\ Q\ s$ is the condition that command C, when executed in state s, *will* terminate in a state satisfying the predicate Q.

$$[C]\ Q\ s \equiv (\exists S.\ (\forall x \in S.\ Q\ x) \wedge (C\ s = \texttt{Term}\ S))$$

Then we define the total correctness refinement relation $\sqsubseteq_{tc}$ accordingly, where $[A]\ Q \longrightarrow [B]\ Q$ is an abbreviation for $\forall s.\ [A]\ Q\ s \Rightarrow [B]\ Q\ s$.

$$A \sqsubseteq_{tc} B \equiv (\forall Q.\ [A]\ Q \longrightarrow [B]\ Q)$$

This definition of refinement $\sqsubseteq_{tc}$ is as in (Dunne 2002, §7), but ignoring frames. Then we proved that two generalised substitutions are equivalent in total correctness refinement that is, they have the same weakest precondition function, if and only if they are the same when considered as functions in this computational model. Equivalently, the weakest precondition function is injective, that is, if $[C]\ Q = [C']\ Q$, then $C = C'$.

In this model the two computations mentioned at the end of §2.1 are represented by the results $\texttt{Term}\{s_1, s_2\}$ and `NonTerm`. For the second of these, the representation `NonTerm` reflects the fact that, in total correctness, the computation is indistinguishable from one which must fail to terminate: they both fail to satisfy any postcondition.

2.3 The Chorus Angelorum Operational Model

Dunne (2007) notes that conventional program analysis involves demonic choice: we want a non-deterministic program to satisfy its postcondition, regardless of what choice the demon (who is trying to ensure that the postcondition is not satisfied) makes. His framework of computations called "extended substitutions" involving both demonic and angelic choice, is based on the underlying computational model of Rewitzky (2003) which is expressed in terms of up-closed binary multirelations. We describe this in an equivalent way: a computation is a function which, given an initial state, returns a set $\mathcal{S}$ of sets of final states. The meaning of this is that the angel will choose one, $S \in \mathcal{S}$, of these sets, and the demon will choose one final state s from that chosen set S. We define the weakest precondition function accordingly:

$$[C]\ Q\ s \equiv (\exists U \in Cs.\ (\forall u \in U.\ Q\ u))$$

This reflects that the angel tries to make a choice U which defeats the demon, who in turn tries to make a choice u from U which fails the postcondition Q.

But considering a result $\mathcal{S}$: *set set state*, if $S' \supseteq S$ for $S', S \in \mathcal{S}$, then the angel could always choose S rather than S' to limit the demon's choice. So it is enough to consider only results $\mathcal{S}$ which, as sets of sets, are "up-closed", that is, where, if $S' \supseteq S$ and $S \in \mathcal{S}$ then $S' \in \mathcal{S}$; under this restriction we then find that two extended substitions which are refinement-equivalent (ie, have the same weakest precondition function) are represented by the same function in this computational model.

3. The Monads used in these Models

3.1 Monads

As discovered by Moggi (1989), monads (long known in category theory) are useful for representing the structure of a computation.

To define a monad M, we need to define the unit and extension functions *unit* and *ext*, of the types shown, and show that they satisfy the following rules required for a monad, where M is a type constructor, eg, *set* or `TorN`.

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

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

We define $B \odot A = \mathit{ext}\ B \circ A$, and this represents $(A\,;B)$, the sequencing of computations B and A, since *ext* B models the action of computation B on the result of a previous computation. The unit function models the computation which does nothing (*skip*). So then rules (1) to (3) give us the properties (4) to (6). We would expect these properties to hold in a model of computation, since they say that the sequencing operation is associative, and that the *skip* computation is its identity. These properties show that we have a category, in which the objects are types, an arrow from α to β is a function of type $\alpha \to M\beta$, the identity arrow for object α is the function $\mathit{unit} : \alpha \to M\alpha$ and composition is given by $\odot$. This category is called the Kleisli category of M, $\mathcal{K}(M)$. We can write (3) as (7), and (2) and (7) also show that *ext* is a functor from $\mathcal{K}(M)$ to the basic category of types and functions.

$$f \odot \mathit{unit} = f \tag{4}$$
$$\mathit{unit} \odot f = f \tag{5}$$
$$h \odot (g \odot f) = (h \odot g) \odot f \tag{6}$$
$$\mathit{ext}\ (g \odot f) = \mathit{ext}\ g \circ \mathit{ext}\ f \tag{7}$$

Two well-known examples of monads used in this paper are the "non-termination" monad (known under various names such as "error monad" or "exception monad"), and the set monad. In each case the unit function turns a simple state into the form of a computation result, and the extension function turns a computation acting on a simple initial state into a computation acting on the result of a previous computation. Also associated with each monad are functions *map* and *join* of the types shown, and in fact a monad can alternatively be characterised by seven rules involving the functions *unit*, *map* and *join* (see (Wadler 1992)).

$$\begin{aligned} \mathit{join} &: MM\alpha \to M\alpha \\ \mathit{map} &: (\alpha \to \beta) \to (M\alpha \to M\beta) \\ \mathit{ext}\ f &= \mathit{join} \circ \mathit{map}\ f \end{aligned}$$

The non-termination monad has unit, map, join and extension functions:

$$\begin{aligned} \mathit{unit\_nt}\ s &= \mathtt{Term}\ s \\ \mathit{map\_nt}\ f\ \mathtt{NonTerm} &= \mathtt{NonTerm} \\ \mathit{map\_nt}\ f\ (\mathtt{Term}\ s) &= \mathtt{Term}\ (f\ s) \\ \mathit{join\_nt}\ (\mathtt{Term}\ \mathtt{NonTerm}) &= \mathit{join\_nt}\ \mathtt{NonTerm} = \mathtt{NonTerm} \\ \mathit{join\_nt}\ (\mathtt{Term}\ (\mathtt{Term}\ f)) &= \mathtt{Term}\ f \\ \mathit{ext\_nt}\ f\ \mathtt{NonTerm} &= \mathtt{NonTerm} \\ \mathit{ext\_nt}\ f\ (\mathtt{Term}\ s) &= f\ s \end{aligned}$$

The set monad has unit, map, join and extension functions

$$\mathit{unit\_s}\ s = \{s\} \qquad \mathit{join\_s}\ \mathcal{A} = \bigcup \mathcal{A}$$
$$\mathit{map\_s}\ f\ S = \{f\ s \mid s \in S\} \qquad \mathit{ext\_s}\ f\ S = \bigcup_{s \in S} f\ s$$

Thus the non-termination monad gives a model where a computation either terminates in a new state, or fails to terminate, and the set monad models non-deterministic (but necessarily terminating) computations.

3.2 Compound Monads

Given that each of the type constructors M and N, with their unit and extension functions, is a monad, it does not follow, however, that $MN\alpha$ (relative to α) is a monad. Yet, in §2.1 and §2.2, we have found such combination of type constructors arise naturally in modelling programs. There are several seemingly distinct approaches to constructing a monad out of two simpler monads; see for example (Liang et al. 1995, §7.3), (Hyland et al. 2006). In some cases, several of these constructions may be applicable. This point is discussed further in §5. We use our results in (Dawson 2007a), which develop those of Jones and Duponcheel (1993), and are closely related to the distributive law of Barr and Wells (1983, §9.2). We describe how those results show that the non-termination monad and the set monad can be composed in both ways, ie, *set* $(\mathtt{TorN}\ \alpha)$ and $\mathtt{TorN}\ (\mathit{set}\ \alpha)$ to form compound monads.

As in (Jones and Duponcheel 1993), we consider the composition of two monads M and N, so the compound monadic type is $MN\alpha$. We write ext_{MN}, ext_M and ext_N for the extension functions of MN, M and N, etc.

To define the compound monad, we need the function ext_{MN}, which "extends" a function f from a "smaller" domain, α, to a "larger" one, $MN\alpha$. So we use a "partial extension" function which does part of this job:

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

The following rules and definitions are sufficient to define a compound monad using such a function *pext*. Note that in view of (12), (10) and (11) are equivalent, and that (9) and (11) give that *pext* is a functor from $\mathcal{K}(MN)$ to $\mathcal{K}(M)$. (In fact, unit_{MN} and *pext* are the unit and extension functions of a monad *in* the category $\mathcal{K}(M)$, whose Kleisli category is also $\mathcal{K}(MN)$, see (Dawson 2007a)).

$$\mathit{pext}\ f \circ \mathit{unit}_N = f \tag{8}$$
$$\mathit{pext}\ \mathit{unit}_{MN} = \mathit{unit}_M \tag{9}$$
$$\mathit{pext}\ (\mathit{ext}_{MN}\ g \circ f) = \mathit{ext}_{MN}\ g \circ \mathit{pext}\ f \tag{10}$$
$$\mathit{pext}\ (g \odot_{MN} f) = \mathit{pext}\ g \odot_M \mathit{pext}\ f \tag{11}$$
$$\mathit{ext}_{MN}\ g = \mathit{ext}_M\ (\mathit{pext}\ g) \tag{12}$$
$$\mathit{unit}_{MN} = \mathit{unit}_M \circ \mathit{unit}_N \tag{13}$$

Relation to distributive laws and compatible monads

Here we briefly outline the relationships between our sufficient conditions for compound monads and other work in the literature.

Jones and Duponcheel (1993) give two conditions, J(1) and J(2), which compound monads may satisfy. Under the reasonable assumptions that $unit_{MN} = unit_M \circ unit_N$ and $map_{MN} = map_M \circ map_N$, the compound monads that arise from a function *pext* are those that satisfy J(1). Jones & Duponcheel use a function *prod*, where $pext\ f = prod \circ map_N\ f$, and give conditions for defining a compound monad using *prod*. The following are equivalent to J(1) and J(2) respectively.

$$ext_M\ join_{MN} = join_{MN} \circ join_M \qquad \text{J(1)}'$$
$$ext_{MN}\ (map_M\ join_N) = map_M\ join_N \circ join_{MN} \qquad \text{J(2)}'$$

The compound monads that satisfy both J(1) and J(2) are those which arise from a function *swap* : $NM\alpha \to MN\alpha$ satisfying conditions S(1) to S(4) of (Jones and Duponcheel 1993), shown below. Compound monads do not necessarily arise from such a function *swap*, or from a function *pext* as above, though it seems that in most cases they do so. If so, $swap = pext\ (map_M\ unit_N)$.

Conditions S(1) to S(4) of (Jones and Duponcheel 1993), shown below, on a function *swap* : $NM\alpha \to MN\alpha$ are necessary and sufficient to define a compound monad in terms of *swap*. The statement of S(4) uses two further functions *prod* and *dorp* defined in terms of *swap*.

$$prod : N\ M\ N\ \alpha \to M\ N\ \alpha$$
$$dorp : M\ N\ M\ \alpha \to M\ N\ \alpha$$
$$prod = map_M\ join_N \circ swap$$
$$dorp = ext_M\ swap$$
$$swap \circ map_N\ (map_M\ f) = map_{MN}\ f \circ swap \qquad \text{S(1)}$$
$$swap \circ unit_N = map_M\ unit_N \qquad \text{S(2)}$$
$$swap \circ map_N\ unit_M = unit_M \qquad \text{S(3)}$$
$$prod \circ map_N\ dorp = dorp \circ prod \qquad \text{S(4)}$$

In fact the function *swap* of (Jones and Duponcheel 1993) is the distributive law λ of (Barr and Wells 1983, §9.2). In this paper we use the terminology and results of Jones and Duponcheel (1993) rather than the text (Barr and Wells 1983) because Jones and Duponcheel (1993) also describe *prod* and *dorp*, which we use. We now relate (Jones and Duponcheel 1993) to (Barr and Wells 1983, §9.2), but this is not used in the remainder of this paper.

We have shown in (Dawson 2007a) that conditions J(1) and J(2) match conditions (C1) to (C5) of (Barr and Wells 1983, §9.2), for compatibility of monads N, M and MN. These are the conditions on a compound monad which are necessary and sufficient for it to be definable by a distributive law, that is, by a *swap* function. Conditions (C2) and (C5) (which are in fact equivalent) are equivalent to J(2) of (Jones and Duponcheel 1993), and conditions (C3) and (C4) (which are also equivalent) are equivalent to J(1) of (Jones and Duponcheel 1993). Finally, condition (C1) is (13) above.

We have also shown in (Dawson 2007a) that conditions S(1) to S(4) correspond to conditions (D1) to (D4) of (Barr and Wells 1983, §9.2) defining a distributive law. S(1) simply says that *swap* is a natural transformation, S(2) and S(3) are (D2) and (D1), and, in the presence of these, S(4) is equivalent to (D3) and (D4).

3.3 The General Correctness Compound Monad

Here we need to show that *set* (TorN α) is a monad; in fact, for any monad M, M (TorN α) is a monad (as is well-known, see, eg, (Liang et al. 1995, §7.3), (Hyland et al. 2006)). For an arbitrary monad M we define the compound monad M (TorN α) by

$$pext\ f\ (\texttt{Term}\ a) = f\ a \qquad (14)$$
$$pext\ f\ \texttt{NonTerm} = unit_M\ \texttt{NonTerm} \qquad (15)$$

We also define *unit* and *ext* for the compound monad by (13) and (12). Then we prove the *pext* rules (8) to (10) as follows. (8) is just (14). From (14) and (15), we get (9) using (13), and (10) using (1) for M and (12).

In fact the general correctness monad can be defined using *swap*: that is, it satisfies J(1) and J(2). Then the function *swap* is given by the rule $swap = pext\ (map_M\ unit_N)$, so with M the *set* monad we have

$$swap\ \texttt{NonTerm} = unit_M\ \texttt{NonTerm} = \{\texttt{NonTerm}\}$$
$$swap\ (\texttt{Term}\ S) = map_M\ \texttt{Term}\ S = \{\texttt{Term}\ s \mid s \in S\}$$

We proved in Isabelle that *pext* defined in this way satisfies rules (8) to (10) as outlined above, and we also proved that *swap* satisfies the conditions S(1) to S(4), and so *swap* is a distributive law for these monads. These Isabelle proofs are available at `http://users.rsise.anu.edu.au/~jeremy/isabelle/fgc/Dmng.{thy,ML}`. Henceforth we will refer to the functions defined for the general correctness monad as *pext_gc*, *swap_gc*, *ext_gc*, and so on.

3.4 The Total Correctness Compound Monad

The type *tcres* = TorN (*set state*), relative to the type *state*, with the unit and extension functions defined below, is also a monad, the *total correctness* monad.

To prove that the total correctness monad is in fact a monad, we now give the definitions for our particular monads $N = set$ and $M = \texttt{TorN}$.

$$unit\_tc : state \to tcres$$
$$prod\_tc : set\ tcres \to tcres$$
$$pext\_tc : (state \to tcres) \to set\ state \to tcres$$
$$ext\_tc : (state \to tcres) \to tcres \to tcres$$
$$unit\_tc\ s = \texttt{Term}\ \{s\}$$
$$prod\_tc\ \mathcal{T} = \texttt{NonTerm} \qquad \text{if } \texttt{NonTerm} \in \mathcal{T}$$
$$prod\_tc\ \{\texttt{Term}\ s \mid s \in \mathcal{S}\} = \texttt{Term}\ (\bigcup \mathcal{S})$$
$$pext\_tc\ A\ S = prod\_tc\ \{A\ s \mid s \in S\}$$
$$ext\_tc\ A\ T = ext\_nt\ (pext\_tc\ A)\ T$$

where *ext_nt* is the extension function of the TorN monad (see §3.1).

We then proved, in Isabelle, that TorN (*set* α) is a compound monad, by proving rules (8) to (10), noting that (12) and (13) follow directly from our definitions.

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

$$swap\_tc : set\ (\texttt{TorN}\ \alpha) \to \texttt{TorN}\ (set\ \alpha)$$
$$swap\_tc\ S = \texttt{NonTerm} \qquad \text{if } \texttt{NonTerm} \in S$$
$$swap\_tc\ \{\texttt{Term}\ s \mid s \in S\} = \texttt{Term}\ S$$

Our Isabelle proofs included the conditions S(1) to S(4), and so we have also shown that *swap_tc* is a distributive law for the monads. These Isabelle proofs are available at `http://users.rsise.anu.edu.au/~jeremy/isabelle/fgc/Dtc.{thy,ML}`.

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 §3.3, the TorN monad can be composed with any other monad M to give a compound monad M (TorN α), which gave the *outcome set* monad. In this case we have shown that the type TorN (*set* α), with *unit_tc* and *ext_tc* as defined, is a monad, but we have not been able to give a more general construction for unit and extension functions to exhibit, for an arbitrary monad M, either TorN (M α) or M (*set* α) as a monad.

3.5 Relating the general correctness and total correctness monads

The function *swap_tc* : *set* (TorN σ) $\rightarrow$ TorN (*set* σ) is also a *monad morphism* from the general correctness monad to the total correctness monad. We have the following results, which characterise a monad morphism:

$$\begin{aligned} \textit{unit\_tc}\ a &= \textit{swap\_tc}\ (\textit{unit\_gc}\ a) \\ \textit{ext\_tc}\ (\textit{swap\_tc} \circ f)\ (\textit{swap\_tc}\ x) &= \textit{swap\_tc}\ (\textit{ext\_gc}\ f\ x) \end{aligned}$$

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

The definition of *swap_tc* reflects the fact that in the general correctness framework (the abstract command language) a computation (on a particular given input) may non-deteministically either terminate or not; in total correctness such a computation (on that input) does not satisfy any postcondition, and so is equivalent to a computation which simply does not terminate.

We note that the definitions of many generalised substitution operations may be obtained by translation from the definitions in (Dawson 2004) for abstract commands. Indeed, for the *choice_tc* function (see below), some results were more easily proved using the following translation from *choice_gc* of the general correctness model: $\textit{choice\_tc}\ \mathcal{C}\ s = \textit{swap\_tc}\ (\textit{choice\_gc}\ \mathcal{A}\ s)$, for any set $\mathcal{A}$ of abstract commands which gives the generalised substitutions $\mathcal{C}$, ie, such that $\mathcal{C} = \{\textit{swap\_tc}\ A \mid A \in \mathcal{A}\}$.

3.6 Definition of Commands

Dunne (2002) then defines a number of substitutions and operations on them, by giving their weakest preconditions and frames.

In (Dawson 2007b) we give operational definitions for the generalised substitutions defined by Dunne (2002), and show that these definitions correspond. For example, sequencing, defined by Dunne as $[A; B]\ Q = [B]\ ([A]Q)$, is represented by the composition $\odot$ in the Kleisli category. Then Dunne's definition gives an alternative proof (once we have proved that the weakest precondition function is injective) of the associativity of sequencing, and so of $\odot$, one of the key properties of monads.

choice

Dunne (2002, §3.1) defines a binary operator, $\Box$: $A \Box B$ is a computation which can choose between two computations A and B. Again, Dunne defines this by giving its weakest precondition, $[A \Box B]Q = [A]Q \wedge [B]Q$. This is a special case of choice among an arbitrary set of commands. In the general correctness setting this was easy to define, $\textit{choice\_gc}\ \mathcal{C}\ s = \bigcup_{C \in \mathcal{C}} C\ s$. 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* by: $\textit{choice\_tc}\ \mathcal{C}\ s = \textit{pext\_tc}\ (\lambda C.\ C\ s)\ \mathcal{C}$.

Note how this definition uses polymorphism: when we use *pext_tc* (indirectly) in defining sequencing of computations, the types α and β are both the state type σ. But in this use of *pext*, α is the type of computations.

$$\textit{pext} : (\alpha \rightarrow \texttt{TorN}\ (\textit{set}\ \beta)) \rightarrow \textit{set}\ \alpha \rightarrow \texttt{TorN}\ (\textit{set}\ \beta)$$

Expanding the definition shows that:

$$\begin{aligned} &\text{if } \texttt{NonTerm} \in \{C\ s \mid C \in \mathcal{C}\} \text{ then } \textit{choice\_tc}\ \mathcal{C}\ s = \texttt{NonTerm} \\ &\text{if } \{C\ s \mid C \in \mathcal{C}\} = \{\texttt{Term}\ S_C \mid C \in \mathcal{C}\}, \\ &\text{then } \textit{choice\_tc}\ \mathcal{C}\ s = \texttt{Term}\ (\textstyle\bigcup_{C \in \mathcal{C}} S_C) \end{aligned}$$

4. The Chorus Angelorum Monad

In §2.3 we described a model of the computation as a function returning an up-closed set of sets of final states. As Dunne (2007, §5.2) explains, there is an alternative model, where a computation returns a set of sets of states, of which the demon first chooses one set of states, from which the angel chooses one state.

We define a function *swap_uc* which turns an angel-chooses-first result into the corresponding demon-chooses-first result; it swaps the order of the choices of the demon and the angel. As we shall see, it also has a role similar to that of the *swap* function (the distributive law) discussed in §3.2. We also define the *up-closure* of a set of sets.

$$\begin{aligned} \textit{swap\_uc}\ \mathcal{A} &= \{B \mid \forall A \in \mathcal{A}.\ B \cap A \neq \{\}\} \\ \textit{up\_cl}\ \mathcal{A} &= \{A' \mid \exists A \in \mathcal{A}.\ A \subseteq A'\} \end{aligned}$$

We then have the following results, which suggest that we should work on equivalence classes of sets of sets of states, where $\mathcal{A} \equiv \mathcal{A}'$ iff $\textit{up\_cl}\ \mathcal{A} = \textit{up\_cl}\ \mathcal{A}'$, and each equivalence class has exactly one up-closed member.

LEMMA 1. *With up_cl and swap_uc as defined above,*

$$\begin{aligned} \textit{up\_cl}\ (\textit{up\_cl}\ \mathcal{A}) &= \textit{up\_cl}\ \mathcal{A} \\ \textit{swap\_uc}\ (\textit{swap\_uc}\ \mathcal{A}) &= \textit{up\_cl}\ \mathcal{A} \\ \textit{swap\_uc}\ (\textit{up\_cl}\ \mathcal{A}) &= \textit{swap\_uc}\ \mathcal{A} \\ \textit{up\_cl}\ (\textit{swap\_uc}\ \mathcal{A}) &= \textit{swap\_uc}\ \mathcal{A} \end{aligned}$$

We cannot define a monad based on the *set* (*set* _) type constructor, but we can define a monad on this set of equivalence classes. As such it is not a compound monad, but in obtaining it we proceed much as though we were combining the *set* monad with itself using *swap_uc* as a distributive law. The results described in this section have been proved in Isabelle, `http://users.rsise.anu.edu.au/~jeremy/isabelle/monad/Chorus.{thy,ML}` and `http://users.rsise.anu.edu.au/~jeremy/isabelle/fgc/{Dch,Dch_tc}.{thy,ML}`.

Firstly we list some functions and their types, as used in (Jones and Duponcheel 1993).

$$\begin{aligned} &\textit{join} : M\ N\ M\ N\ \alpha \rightarrow M\ N\ \alpha \qquad \textit{prod} : N\ M\ N\ \alpha \rightarrow M\ N\ \alpha \\ &\textit{dorp} : M\ N\ M\ \alpha \rightarrow M\ N\ \alpha \qquad \textit{swap} : N\ M\ \alpha \rightarrow M\ N\ \alpha \end{aligned}$$

Here both the type constructors M and N are *set*, but we can intuitively understand these functions, and some of the results about them, by thinking of the type constructor M as representing a set from which the angel is to choose, and N as representing a set from which the demon is to choose. The final result is to be a set of sets from which the angel is to choose first, and then the demon.

Imitating the procedure for defining *set* (*set* α) as a compound monad using the function *swap_uc*, we try to prove S(1) to S(4): we cannot, but we can prove them modulo up-closure. We proved the following results, where *dorp_uc* and *prod_uc* are defined from *swap_uc*, following (Jones and Duponcheel 1993):

$$\begin{aligned} \textit{dorp\_uc} &= \textit{join\_s} \circ \textit{map\_s}\ \textit{swap\_uc} \\ \textit{prod\_uc} &= \textit{map\_s}\ \textit{join\_s} \circ \textit{swap\_uc} \end{aligned}$$

LEMMA 2. *With dorp_uc and prod_uc defined from swap_uc as shown,*

$$\begin{aligned} \textit{swap\_uc} \circ \textit{map\_s}\ (\textit{map\_s}\ f) &= \\ \textit{up\_cl} \circ \textit{map\_s}\ (\textit{map\_s}\ f) \circ \textit{swap\_uc} &\quad \text{S(1)}' \\ \textit{swap\_uc}\ A = \textit{up\_cl}\ (\textit{map\_s}\ \textit{unit\_s}\ A) &\quad \text{S(2)}' \\ \textit{swap\_uc}\ (\textit{map\_s}\ \textit{unit\_s}\ A) = \textit{up\_cl}\ A &\quad \text{S(3)}' \\ \textit{prod\_uc} \circ \textit{map\_s}\ \textit{dorp\_uc} = \textit{dorp\_uc} \circ \textit{prod\_uc} &\quad \text{S(4)} \\ \textit{swap\_uc} \circ \textit{map\_s}\ \textit{join\_s} = \textit{dorp\_uc} \circ \textit{swap\_uc} &\quad \text{(D3)} \\ \textit{swap\_uc} \circ \textit{join\_s} = \textit{prod\_uc} \circ \textit{map\_s}\ \textit{swap\_uc} &\quad \text{(D4)} \end{aligned}$$

That is, we have proved S(1) to S(3) modulo up-closure, and S(4). The proof of S(4) is difficult, and uses (equivalents of) (D3) and (D4) of Barr and Wells (1983, §9.2).

Then, as in (Jones and Duponcheel 1993), we define *join_uc* by (17), or equivalently (18). Defining *map_uc* in the obvious way by (16), we define *ext_uc* and $\odot_{uc}$ from *join_uc* in the usual way by (19) and (21), and also *pext_uc* from *prod_uc* by (22). We can then prove (20) and (23). We also give the equivalent literal expressions for *ext_uc* and *pext_uc*.

$$\mathit{map\_uc}\ f = \mathit{map\_s}\ (\mathit{map\_s}\ f) \tag{16}$$
$$\mathit{join\_uc} = \mathit{join\_s} \circ \mathit{map\_s}\ \mathit{prod\_uc} \tag{17}$$
$$\mathit{join\_uc} = \mathit{map\_s}\ \mathit{join\_s} \circ \mathit{dorp\_uc} \tag{18}$$
$$\mathit{ext\_uc}\ f = \mathit{join\_uc} \circ \mathit{map\_uc}\ f \tag{19}$$
$$\mathit{ext\_uc} = \mathit{ext\_s} \circ \mathit{pext\_uc} \tag{20}$$
$$g \odot_{uc} f = \mathit{ext\_uc}\ g \circ f \tag{21}$$
$$\mathit{pext\_uc}\ f = \mathit{prod\_uc} \circ \mathit{map\_s}\ f \tag{22}$$
$$\mathit{prod\_uc}\ \mathcal{A} = \bigcap\{\mathit{map\_s}\ \mathit{up\_cl}\ \mathcal{A}\} \tag{23}$$
$$\mathit{pext\_uc}\ f\ A = \{B \mid \forall a \in A.\ \exists B' \in f\ a.\ B' \subseteq B\} \tag{24}$$
$$\mathit{ext\_uc}\ f\ \mathcal{A} = \{B \mid \exists A \in \mathcal{A}.\ \forall a \in A.\ \exists B' \in f\ a.\ B' \subseteq B\} \tag{25}$$

The proofs of the monad rules for *set* $(\mathit{set}\ \alpha)$ (again, some equalities only modulo up-closure: compare the results below with (1) to (3))) proceed as normal from S(1) to S(4) (see, eg, (Jones and Duponcheel 1993) or (Dawson 2007a)).

LEMMA 3. *With join_uc defined by* (17) *or equivalently* (18)*, and ext_uc by* (19)

$$\mathit{ext\_uc}\ f\ \{\{x\}\} = \mathit{up\_cl}\ (f\ \{\{x\}\}) \tag{26}$$
$$\mathit{ext\_uc}\ (\lambda x.\ \{\{x\}\}) = \mathit{up\_cl} \tag{27}$$
$$\mathit{ext\_uc}\ (\mathit{ext\_uc}\ g \circ f) = \mathit{ext\_uc}\ g \circ \mathit{ext\_uc}\ f \tag{28}$$

The Monad Functions on *Up-closed* Sets of Sets

Finally we need to show that these results give a monad on the set of equivalence classes, which requires several lemmas concerning the behaviour of *ext_uc* and other functions on different but "equivalent" arguments. Isabelle's *type definition* facility was useful here, not least in ensuring that no necessary part of the proof was overlooked. We define the type *uca* α ("up-closed abstract") as the type of up-closed sets of sets (a representative of each of the equivalence classes). The set of members of the new type is isomorphic to, but distinct from, the set of up-closed sets of sets.

In Isabelle, defining a new type σ (the "abstract" type) isomorphic to a set S : *set* ρ causes the creation of an abstraction function *Abs* : $\rho \to \sigma$ and a representation function *Rep* : $\sigma \to \rho$, such that *Abs* and *Rep* are mutually inverse bijections between S and the set of all values of type σ. Note that the domain of *Abs* is the type ρ, but that nothing is said about the values it takes outside the set S. Thus we get abstraction and representation functions *Rep_uca* : *uca* $\alpha \to$ *set* (*set* α) and *Abs_uca* : *set* (*set* α) $\to$ *uca* α. We define an alternative abstraction function *uc_Abs* of the same type as *Abs_uca* but whose action is specified, in the natural way, on non-upclosed sets: *uc_Abs* $\mathcal{A} \equiv$ *Abs_uca* (*up_cl* $\mathcal{A}$).

Then we define the monad functions *unit_uca* and *ext_uca* for the *uca* α type; we also define *swap* on this type.

DEFINITION 4.

$$\mathit{unit\_uca}\ a = \mathit{uc\_Abs}\ \{\{a\}\} \tag{29}$$
$$\mathit{ext\_uca}\ f\ \mathcal{A} = \mathit{uc\_Abs}\ (\mathit{ext\_uc}\ (\mathit{Rep\_uca}\ \circ f)\ (\mathit{Rep\_uca}\ \mathcal{A})) \tag{30}$$
$$\mathit{swap\_uca}\ \mathcal{A} = \mathit{uc\_Abs}\ (\mathit{swap\_uc}\ (\mathit{Rep\_uca}\ \mathcal{A})) \tag{31}$$

We then used (26) to (28) about the type *set* $(\mathit{set}\ \alpha)$ to prove the monad rules (1) to (3) for the type *uca* α.

THEOREM 1. *The type constructor uca, with unit and extension functions unit_uca and ext_uca, is a monad.*

Changing the Interpretation ; the Conjugate of a Substitution

The result of a computation might equally well be modelled as a set of sets from which the demon is to choose first, using the same monad, but translating the result by *swap_uc*. We showed that the function *swap_uca* (Definition 4, (31)) is a monad morphism. We also proved some results such as *swap_uc* $\circ$ *prod_uc* $=$ *dorp_uc* which would involve a type error if M and N were not the same monad.

Dunne also defines the conjugate of a substitution (Dunne 2007, §3.4): $[T^o]Q = \neg[T]\neg Q$. This also amounts to switching the interpretation of the set-of-sets result from angel-chooses-first to demon-chooses-first. Thus we can obtain it by applying *swap_uc* to the result of a computation, so we proved $T^o\ s = \mathit{swap\_uca}\ (T\ s)$.

A Link to the Continuation Monad

We can obtain an interesting link to the continuation monad (Wadler 1992, §3.1). We define an "evaluation function" *eval_uc* : *set* $(\mathit{set}\ \alpha) \to (\alpha \to \mathit{bool}) \to \mathit{bool}$, where *eval_uc* $\mathcal{A}\ P$ tells whether the postcondition P is satisfied when angel and demon have made their choices from $\mathcal{A}$. (The weakest precondition function could then be defined in terms of *eval_uc*). Naturally it is defined by *eval_uc* $\mathcal{B}\ P \equiv \exists B \in \mathcal{B}.\ \forall b \in B.\ P\ b$, and so has the property that *eval_uc* $\mathcal{B}$ $=$ *eval_uc* (*up_cl* $\mathcal{B}$). Thus also, if $\mathcal{B}$ is up-closed and P' is the set corresponding to predicate P, then we have *eval_uc* $\mathcal{B}\ P' = P \in \mathcal{B}$. This shows that the model we describe is also essentially that of the choice semantics of Back and von Wright (1998, Ch 15).

Now the type $K\ \alpha = (\alpha \to \mathit{bool}) \to \mathit{bool}$ is the type of the continuation monad K (Wadler 1992, §3.1), when the fixed "answer" type is *bool*. The composition $\odot_K$ in the Kleisli category for K can be given simply in terms of the C combinator:

$$C\ f\ x\ y = f\ y\ x \qquad g \odot_K f = C\ (C\ f \circ C\ g)$$

The functions *Ball* and *Bex*, of type *set* $\alpha \to K\ \alpha$, used in Isabelle to express quantification over a given set: *Ball* $S\ P \equiv \forall s \in S.\ P\ s$, can be used to express the way the functions above involve the demonic and angelic choices. We find that *eval_uc* $=$ *Ball* $\odot_K$ *Bex* and *eval_uc* $\circ$ *swap_uc* $=$ *Bex* $\odot_K$ *Ball*. Further, there is an obvious isomorphism $K\ \alpha \to$ *set* (*set* α), which we call *K_to_SS*. Thus *up_cl* $=$ *K_to_SS* $\circ$ *eval_uc*. Then we find these results which represent the sequence of angelic and demonic choices involved in these functions.

$$\mathit{up\_cl} = \mathit{K\_to\_SS} \circ (\mathit{Ball} \odot_K \mathit{Bex})$$
$$\mathit{join\_uc} = \mathit{K\_to\_SS} \circ (\mathit{Ball} \odot_K \mathit{Bex} \odot_K \mathit{Ball} \odot_K \mathit{Bex})$$
$$\mathit{dorp\_uc} = \mathit{K\_to\_SS} \circ (\mathit{Bex} \odot_K \mathit{Ball} \odot_K \mathit{Bex})$$
$$\mathit{prod\_uc} = \mathit{K\_to\_SS} \circ (\mathit{Ball} \odot_K \mathit{Bex} \odot_K \mathit{Ball})$$
$$\mathit{swap\_uc} = \mathit{K\_to\_SS} \circ (\mathit{Bex} \odot_K \mathit{Ball})$$
$$\mathit{ext\_uc}\ f = \mathit{K\_to\_SS} \circ (\mathit{Ball} \odot_K (\mathit{Bex} \circ f) \odot_K \mathit{Ball} \odot_K \mathit{Bex})$$
$$\mathit{pext\_uc}\ f = \mathit{K\_to\_SS} \circ (\mathit{Ball} \odot_K (\mathit{Bex} \circ f) \odot_K \mathit{Ball})$$

Under the isomorphism *K_to_SS*, up-closed sets correspond to *monotonic* continuations, which we define by

$$\mathit{mono}_K\ c \equiv \forall P\ Q.\ (\forall s.\ P\ s \Rightarrow Q\ s) \Rightarrow c\ P \Rightarrow c\ Q$$

Then we find that *K_to_SS* and *eval_uc* are mutually inverse bijections between the up-closed families of sets and the monotonic continuations. It follows that there is a bijective correspondence between functions $f : \alpha \to K\ \beta$ which always return monotonic continuations, and functions $S : \alpha \to \mathit{set}\ (\mathit{set}\ \beta)$ which always

return up-closed families. From this we can get the bijective correspondence of Rewitzky (2003) between monotonic predicate transformers $F : (\beta \to \mathit{bool}) \to (\alpha \to \mathit{bool})$ and up-closed binary multirelations.

Trivially, *Ball* S and *Bex* S are monotonic, and also $\odot_K$ preserves monotonicity in this sense:

$$\forall a.\ \mathit{mono}_K\ (f\ a) \wedge \forall b.\ \mathit{mono}_K\ (g\ b) \Rightarrow \forall x.\ \mathit{mono}_K\ ((g \odot_K f)\ x)$$

Similar to the results above is the following:

$$f \odot_{uc} g = \mathit{K\_to\_SS} \circ ((\mathit{eval\_uc} \circ f) \odot_K (\mathit{eval\_uc} \circ g))$$

This leads to an alternative proof of the associativity of $\odot$*_uc*, for

$$\begin{aligned} & f \odot_{uc} (g \odot_{uc} h) \\ & = \mathit{K\_to\_SS} \circ ((\mathit{eval\_uc} \circ f) \odot_K \\ & \qquad (\mathit{eval\_uc} \circ (\mathit{K\_to\_SS} \circ ((\mathit{eval\_uc} \circ g) \odot_K (\mathit{eval\_uc} \circ h))))) \\ & = \mathit{K\_to\_SS} \circ ((\mathit{eval\_uc} \circ f) \odot_K ((\mathit{eval\_uc} \circ g) \odot_K (\mathit{eval\_uc} \circ h))) \end{aligned}$$

where *eval_uc* $\circ$ *K_to_SS* is the identity on its argument, which is monotonic. Thus the associativity of $\odot_{uc}$ follows from that of $\odot_K$.

Angelic and Demonic Choice

Dunne (2007) defines angelic (demonic) choice by giving their weakest preconditions, which are just the disjunctions (conjunctions) of the preconditions of the individual substitutions. In the case of each of these, applying a set of computations to an initial state gives a set of sets of sets of final states, of type *set* $(\mathit{set}\ (\mathit{set}\ \sigma))$. In the case of angelic choice, the angel makes the first two choices and the demon the final choice, so for angelic choice we simply take the union of the results of the individual computations. For demonic choice, the consecutive choices are made by the demon, the angel and the demon again. This is exactly as for the *prod_uc* function. The definitions are as follows, but we omit the functions *uc_Abs* and *Rep_uca* between the *set* $(\mathit{set}\ \alpha)$ and the *uca* α types.

$$\begin{aligned} \mathrm{dem}\ \mathcal{B}\ s &= \mathit{prod\_uc}\ \{B\ s \mid B \in \mathcal{B}\} = \mathit{pext\_uc}\ (\lambda B.\ B\ s)\ \mathcal{B} \\ \mathrm{ang}\ \mathcal{B}\ s &= \bigcup \{B\ s \mid B \in \mathcal{B}\} \end{aligned}$$

So these results are analogous to those above:

$$\begin{aligned} \mathit{up\_cl} \circ \bigcup = \bigcup \circ \mathit{map\_s}\ \mathit{up\_cl} &= \mathit{K\_to\_SS} \circ (\mathit{Ball} \odot_K \mathit{Bex} \odot_K \mathit{Bex}) \\ \bigcap \circ \mathit{map\_s}\ \mathit{up\_cl} &= \mathit{K\_to\_SS} \circ (\mathit{Ball} \odot_K \mathit{Bex} \odot_K \mathit{Ball}) \end{aligned}$$

Sequencing and Choice

We proved the following results about the distribution of demonic or angelic choice over sequencing, where the notation implies demonic or angelic choice of a set of commands.

$$\mathrm{dem}\ \mathcal{B}\ ;\ C = \operatorname*{dem}_{B \in \mathcal{B}} (B\,;C) \qquad \mathrm{ang}\ \mathcal{B}\ ;\ C = \operatorname*{ang}_{B \in \mathcal{B}} (B\,;C)$$

Note that these results hold because in either case there is first a choice of $B \in \mathcal{B}$, and then execution of $(B\,;C)$. The similar results for choice of C, ie, $B\,;(\mathrm{dem}_{C \in \mathcal{C}}\ C) = \mathrm{dem}_{C \in \mathcal{C}} (B\,;C)$ do not hold, since that would involve reordering the demonic choice of C with the choices involved in executing B.

The proofs of these results use (32) and (33) (which is of the form of (10)); note however that in this use of them, g is C, while f is not a command, but rather the function $\lambda B.\ B\ s$, for an initial state s.

$$\mathit{ext\_s}\ (\mathit{ext\_uc}\ g \circ f) = \mathit{ext\_uc}\ g \circ \mathit{ext\_s}\ f \tag{32}$$

$$\mathit{pext\_uc}\ (\mathit{ext\_uc}\ g \circ f) = \mathit{ext\_uc}\ g \circ \mathit{pext\_uc}\ f \tag{33}$$

Since *ext_uc* $=$ *ext_s* $\circ$ *pext_uc*, (32) and (33) combine to give (28), of the form of (3), which is the significant (usually non-trivial) one among the monad rules.

A Monad Morphism from the Total Correctness Monad

As Dunne points out, the language of extended substitutions encompasses that of generalised substitutions.

We defined the inclusion mapping *tc_to_ch*, of type `TorN` $(\mathit{set}\ \alpha) \to$ *uca* α, as shown below (and as suggested by the discussion in (Dunne 2007, §5.1,§5.2))

$$\mathit{tc\_to\_ch}\ (\texttt{Term}\ S) = \mathit{uc\_Abs}\ \{S\} \tag{34}$$

$$\mathit{tc\_to\_ch}\ \texttt{NonTerm} = \mathit{uc\_Abs}\ \{\} \tag{35}$$

For the result `NonTerm`, which fails every postcondition, we get the corresponding result $\{\}$, because from it the angel cannot choose a set of results all of which satisfy even the postcondition *true*. At the other extreme is the result `Term` $\{\}$, which satisfies every postcondition. Corresponding to it we have a result from which the angel can choose $\{\}$, from which the demon cannot choose any which will fail to satisfy even the postcondition *false*.

We proved that this mapping is a monad morphism. Since this mapping is *injective*, it gives an alternative proof that the total correctness monad is in fact a monad. This is an interesting contrast to the point noted in §3.5, where a *surjective* mapping *from* the general correctness monad into the type *tcres* α provided yet another proof that the total correctness monad is in fact a monad.

5. Conclusion; Further Work

We have shown how several models of computation arising naturally in the study of program semantics are based on monads, which can be shown to be monads using results on combining two monads using a distributive law.

For the general correctness monad we used a well-known general construction combining the error (`TorN`) monad with *any* monad M. The total correctness monad used a different construction, which seems not to be similarly generalisable. For the Chorus Angelorum monad, having combined the set monad with itself, we needed to take equivalence classes of the result.

In the first two cases the combined monad arises from a distributive law for monads, and in the third case, the distributive law equations are satisfied modulo the equivalence relation. In this case the proofs that we have a monad were helped significantly by basing our approach on the results of (Barr and Wells 1983) and (Jones and Duponcheel 1993) on compound monads based on a distributive law. (Trying to prove equation (28) using (25) as a definition is not recommended!) We also found some monad morphisms relating these monads: in some cases the distributive laws are themselves monad morphisms.

All the discussion of compound monads in this paper focussed on distributive laws. There are several monad transformers, or constructions which produce compound monads, such as those described by Liang et al. (1995). The theory underlying these has been analysed, for example by Hyland et al. (2006), who give several general constructions, dealing with arbitrary categories (such as $\omega-$**Cpo**) rather than just the category of sets. While all our work (and in particular, our Isabelle proofs) are, following (Jacobs and Gries 1985) and (Dunne 2007), in terms of sets, it would be interesting to investigate whether the Chorus Angelorum operational model and monad can be extended in this way to a more general category than **Set**, and whether any of the constructions for a compound monad, other than the distributive law, can similarly be applied to that situation.

References

Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science, Springer, 1998. URL `http://crest.cs.abo.fi/publications/public/1998/RefinementCalculusBook.pdf`.

Michael Barr and Charles Wells. *Toposes, Triples and Theories*. Springer-Verlag, 1983. URL `http://www.cwru.edu/artsci/math/wells/pub/ttt.html`.

Jeremy E Dawson. Compound monads and the Kleisli category. Unpublished note, 2007a. URL `http://users.rsise.anu.edu.au/~jeremy/pubs/cmkc/`.

Jeremy E Dawson. Formalising general correctness. In *Computing: The Australasian Theory Symposium*, volume ENTCS 91, pages 21–42, 2004. URL `http://www.elsevier.com/locate/entcs`.

Jeremy E Dawson. Formalising generalised substitutions. In *Theorem Proving in Higher-Order Logics*, page to appear, 2007b. URL `http://users.rsise.anu.edu.au/~jeremy/pubs/fgc/fgs/`.

Edsger W Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.

Steve Dunne. Abstract commands: A uniform notation for specifications and implementations. In *Computing: The Australasian Theory Symposium*, volume ENTCS 42, pages 104–123, 2001. URL `http://www.elsevier.com/locate/entcs`.

Steve Dunne. Chorus angelorum. In *B 2007: Formal Specification and Development in B*, volume LNCS 4355, pages 19–33. Springer, 2007.

Steve Dunne. A theory of generalised substitutions. In *Formal Specification and Development in Z and B, (ZB 2002)*, volume LNCS 2272, pages 270–290. Springer, 2002.

Martin Hyland, Gordon D Plotkin, and John A Power. Combining effects: Sum and tensor. *Theor. Comput. Sci.*, 357:70–99, 2006.

Dean Jacobs and David Gries. General correctness: A unification of partial and total correctness. *Acta Informatica*, 22:67–83, 1985.

Mark P Jones and Luc Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Yale University, 1993.

Sheng Liang, Paul Hudak, and Mark P Jones. Monad transformers and modular interpreters. In *Symposium on Principles of Programming Languages (POPL'95)*, pages 333–343, 1995.

Clare E Martin, Sharon A Curtis, and Ingrid Rewitzky. Modelling angelic and demonic nondeterminism with multirelations. *Sci. Comput. Program.*, 65:140–158, 2007.

Eugenio Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science (LICS)*. IEEE, 1989.

Gordon D Plotkin. A powerdomain construction. *SIAM J. Computing*, 5: 452–487, 1976.

Ingrid Rewitzky. Binary multirelations. In *Theory and Applications of Relational Structures as Knowledge Instruments 2003*, volume LNCS 2929, pages 256–271. Springer, 2003.

Philip Wadler. The essence of functional programming. In *Symposium on Principles of Programming Languages (POPL'92)*, pages 1–14, 1992.