Compound Monads and Models of Computation

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 consider the theory of "extended subsitutions" involving both angelic and demonic choice. For other related formal theories describing 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.
Keywords: extended substitutions, compound monads, distributive law for monads, generalised substitution, demonic choice, angelic choice

1 Introduction

In several papers Dunne has proposed formal systems for analysing (variously named) "computations", given rules for combining these computations in various ways, and proved results about them. Underlying each of these theories 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 the defining the operations in this way implies the definitions given by Dunne. In this work we used the fact that a compound monad underlies the model of computation involved.

In [5] Dunne gave an account of general correctness (Jacob & Gries [8]), 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 [3] we described an operational interpretation of abstract commands equivalent to that of Jacob & Gries [8], which used the construction of Plotkin [13].

Similarly, in [6], Dunne considered the generalised substitutions used in the B method. In [4] we formalised the theory he gave, and the operational model

* 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.

underlying it, using a similar approach to, and in some cases the results of, [3]. 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 [1, §9.2].

Recently, in [7] Dunne defined "extended substitutions", which encompass the generalised substitutions, as they allow for angelic choice as well as demonic choice. He described the model for these in terms of "up-closed binary multirelations". In this paper we describe how the operational model we use (equivalent to that of [7]) is 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. 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.

All results referred to in this paper have been proved using the theorem prover 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 [3], used to describe "abstract commands" and general correctness. In [8] Jacobs & Gries (see also [5]) 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 [5]) means partial correctness refinement *and* total correctness refinement. The computational model that we use underlying this theory is the following: 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*.

2.2 The Total Correctness Operational Model

Subsequently, in [6], Dunne 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 and total correctness refinement accordingly:

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

This definition of refinement $\sqsubseteq_{tc}$ is as in [6, §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 $C = C'$.

2.3 The Chorus Angelorum Operational Model

In [7], Dunne notes that conventional program analysis involves demonic choice: we want a non-deterministic program to satisfy its post-condition, regardless of what choices the demon makes. He develops a framework of computations called "extended substitutions" involving both demonic and angelic choice, and describes the underlying computational model 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:

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

But considering a result $\mathcal{S} : \mathit{set\ set\ state}$, 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 [12], 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, of the types shown. We need to show that the unit and extension functions satisfy the following rules required for a monad, where M is a type constructor, eg, *set* or `TorN`.

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

$$\mathit{ext}\ f \circ \mathit{unit} = f \tag{1}$$

$$\mathit{ext}\ \mathit{unit} = \mathit{id} \tag{2}$$

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

As a standard result (see [14]), 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. We define $B \odot A = ext\ B \circ A$, and this represents sequencing of computations B and A, as *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 following expected properties (4) to (6). 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 $unit : \alpha \to M\alpha$ 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.

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

Two examples of monads used in this paper are the "non-termination" monad (well-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 the result of a previous computation. We will also use the *map* function which applies a transformation to each item of the simple type involved in a given value of the monadic type.

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

$$unit_nt\ s = \texttt{Term}\ s$$
$$map_nt\ f\ \texttt{NonTerm} = \texttt{NonTerm} \qquad map_nt\ f\ (\texttt{Term}\ s) = \texttt{Term}\ (f\ s)$$
$$ext_nt\ f\ \texttt{NonTerm} = \texttt{NonTerm} \qquad ext_nt\ f\ (\texttt{Term}\ s) = f\ s$$

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

$$unit_s\ s = \{s\} \qquad join_s\ \mathcal{A} = \bigcup \mathcal{A}$$
$$map_s\ f\ S = \{f\ s \mid s \in S\} \qquad ext_s\ f\ S = \bigcup_{s \in S} f\ s$$

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. There are several seemingly distinct approaches to constructing a

monad out of two simpler monads; see for example [11, §7.3], [9]. In some cases, several of these constructions may be applicable. This point is discussed further in §5. We use our results in [2], which develop those of Jones & Duponcheel [10], and are closely related to the distributive law of Barr & Wells [1, §9.2].

As in [10], 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:

$$ext_{MN} : (\alpha \to MN\beta) \to (MN\alpha \to MN\beta)$$
$$pext : (\alpha \to MN\beta) \to (N\alpha \to MN\beta)$$

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 [2]).

$$pext\ f \circ unit_N = f \tag{8}$$
$$pext\ unit_{MN} = unit_M \tag{9}$$
$$pext\ (ext_{MN}\ g \circ f) = ext_{MN}\ g \circ pext\ f \tag{10}$$
$$pext\ (g \odot_{MN} f) = pext\ g \odot_M pext\ f \tag{11}$$
$$ext_{MN}\ g = ext_M\ (pext\ g) \tag{12}$$
$$unit_{MN} = unit_M \circ unit_N \tag{13}$$

Jones & Duponcheel [10] 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 compound monads that satisfy both J(1) and J(2) are those which arise from a distributive law $\lambda : NM\alpha \to MN\alpha$, see Barr & Wells [1, §9.2]. Compound monads do not necessarily arise from a distributive law, or from a function *pext* as above, though it seems that in most cases they do so. If so, $\lambda = pext\ (map_M\ unit_N)$.

The function *swap* of [10] is the distributive law λ of [1, §9.2]. We have shown in [2] that conditions J(1) and J(2) match conditions (C1) to (C5) of [1, §9.2], for compatibility of monads N, M and MN. Conditions (C2) and (C5) (which are in fact equivalent) are equivalent to J(2) of [10], and conditions (C3) and (C4) (which are also equivalent) are equivalent to J(1) of [10]. Finally, condition (C1) is (13) above.

Likewise, conditions S(1) to S(4) of [10] on the function *swap* correspond to conditions (D1) to (D4) of [1, §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) can be shown equivalent to (D3) and (D4). See [2] for proofs of these.

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, [11, §7.3], [9]). For an arbitrary monad M we define the compound monad M `TorN` α by

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

We also define *unit* and *ext* for the compound monad by (13) and (12). Then we prove the *pext* axioms (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) of [10], and (C1) to (C5) of [1, §9.2]. 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 axioms (8) to (10) as outlined above, and we also proved that *swap* satisfies the conditions S(1) to S(4) of [10], 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 $M =$ `TorN`.

$$\begin{aligned}
unit_tc &: state \to tcres \\
prod_tc &: set\ tcres \to tcres \\
pext_tc &: (state \to tcres) \to set\ state \to tcres \\
ext_tc &: (state \to tcres) \to tcres \to tcres \\
unit_tc\ s &= \texttt{Term}\ \{s\} \\
prod_tc\ \mathcal{T} &= \texttt{NonTerm} \qquad \text{if } \texttt{NonTerm} \in \mathcal{T} \\
prod_tc\ \{\texttt{Term}\ s \mid s \in \mathcal{S}\} &= \texttt{Term}\ (\bigcup \mathcal{S}) \\
pext_tc\ A\ S &= prod_tc\ \{A\ s \mid s \in S\} \\
ext_tc\ A\ T &= ext_nt\ (pext_tc\ A)\ T
\end{aligned}$$

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

$$\begin{aligned} \mathit{swap_tc} &: \mathit{set}\ \mathtt{TorN}\ \alpha \to \mathtt{TorN}\ \mathit{set}\ \alpha \\ \mathit{swap_tc}\ S &= \mathtt{NonTerm} \qquad \text{if } \mathtt{NonTerm} \in S \\ \mathit{swap_tc}\ \{\mathtt{Term}\ s \mid s \in S\} &= \mathtt{Term}\ S \end{aligned}$$

Our Isabelle proofs included the conditions S(1) to S(4) of [10], 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 $\sigma \to$ 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:

$$\begin{aligned} \mathit{unit_tc}\ a &= \mathit{swap_tc}\ (\mathit{unit_gc}\ a) \\ \mathit{ext_tc}\ (\mathit{swap_tc} \circ f)\ (\mathit{swap_tc}\ x) &= \mathit{swap_tc}\ (\mathit{ext_gc}\ f\ x) \end{aligned}$$

Since this monad morphism is surjective, we could use the fact that the general correctness monad satisfies the monad axioms 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 [3] 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: $\mathit{choice_tc}\,\mathcal{C}\ s = \mathit{swap_tc}\,(\mathit{choice_gc}\,\mathcal{A})$, for any set $\mathcal{A}$ of abstract commands which gives the generalised substitutions $\mathcal{C}$, ie, where $\mathcal{C} = \{\mathit{swap_tc}\ A \mid A \in \mathcal{A}\}$.

3.6 Definition of Commands

In [6], Dunne then defines a number of substitutions and operations on them, by giving their weakest preconditions and frames.

In [4] we give operational definitions for the generalised substitutions defined by Dunne in [6], 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 In [6, §3.1] Dunne 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, $\mathit{choice_gc}\ \mathcal{C}\ s = \bigcup_{C \in \mathcal{C}} C\ s$. In the total correctness setting, where $\mathit{choice_tc}\ \mathcal{C}$ can fail to terminate if any $C \in \mathcal{C}$ can fail to terminate, we define *choice_tc* by: $\mathit{choice_tc}\ \mathcal{C}\ s = \mathit{pext_tc}\ (\lambda C.\ C\ s)\ \mathcal{C}$. Expanding this definition shows that:

if $\{C\ s \mid C \in \mathcal{C}\}$ contains `NonTerm` then $\mathit{choice_tc}\ \mathcal{C}\ s = \texttt{NonTerm}$
if $\{C\ s \mid C \in \mathcal{C}\} = \{\texttt{Term}\ S_C \mid C \in \mathcal{C}\}$, then $\mathit{choice_tc}\ \mathcal{C}\ s = \texttt{Term}\ (\bigcup_{C \in \mathcal{C}} S_C)$

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 [7, §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.

$$\mathit{swap_uc}\ \mathcal{A} = \{B \mid \forall A \in \mathcal{A}.\ B \cap A \neq \{\}\}$$
$$\mathit{up_cl}\ \mathcal{A} = \{A' \mid \exists A \in \mathcal{A}.\ A \subseteq A'\}$$

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 $\mathit{up_cl}\ \mathcal{A} = \mathit{up_cl}\ \mathcal{A}'$, and each equivalence class has exactly one up-closed member.

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

$$\mathit{up_cl}\ (\mathit{up_cl}\ \mathcal{A}) = \mathit{up_cl}\ \mathcal{A} \qquad \mathit{swap_uc}\ (\mathit{swap_uc}\ \mathcal{A}) = \mathit{up_cl}\ \mathcal{A}$$
$$\mathit{swap_uc}\ (\mathit{up_cl}\ \mathcal{A}) = \mathit{swap_uc}\ \mathcal{A} \qquad \mathit{up_cl}\ (\mathit{swap_uc}\ \mathcal{A}) = \mathit{swap_uc}\ \mathcal{A}$$

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 [10].

$$\begin{array}{ll} \mathit{join} : M\ N\ M\ N\ \alpha \to M\ N\ \alpha & \mathit{prod} : N\ M\ N\ \alpha \to M\ N\ \alpha \\ \mathit{dorp} : M\ N\ M\ \alpha \to M\ N\ \alpha & \mathit{swap} : N\ M\ \alpha \to M\ N\ \alpha \end{array}$$

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) of [10, §3.4]: 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 [10]:

$$\begin{aligned} \mathit{dorp_uc} &= \mathit{join_s} \circ \mathit{map_s}\ \mathit{swap_uc} \\ \mathit{prod_uc} &= \mathit{map_s}\ \mathit{join_s} \circ \mathit{swap_uc} \end{aligned}$$

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

$$\mathit{swap_uc} \circ \mathit{map_s}\ (\mathit{map_s}\ f) = \mathit{up_cl} \circ \mathit{map_s}\ (\mathit{map_s}\ f) \circ \mathit{swap_uc} \qquad \mathrm{S}(1)'$$

$$\mathit{swap_uc}\ A = \mathit{up_cl}\ (\mathit{map_s}\ \mathit{unit_s}\ A) \qquad \mathrm{S}(2)'$$

$$\mathit{swap_uc}\ (\mathit{map_s}\ \mathit{unit_s}\ A) = \mathit{up_cl}\ A \qquad \mathrm{S}(3)'$$

$$\mathit{prod_uc} \circ \mathit{map_s}\ \mathit{dorp_uc} = \mathit{dorp_uc} \circ \mathit{prod_uc} \qquad \mathrm{S}(4)$$

$$\mathit{swap_uc} \circ \mathit{map_s}\ \mathit{join_s} = \mathit{dorp_uc} \circ \mathit{swap_uc} \qquad \mathrm{(D3)}$$

$$\mathit{swap_uc} \circ \mathit{join_s} = \mathit{prod_uc} \circ \mathit{map_s}\ \mathit{swap_uc} \qquad \mathrm{(D4)}$$

That is, we have 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 & Wells [1, §9.2].

Then, as in [10], we define *join_uc* by (16), or equivalently (17). We then define *ext_uc* and $\odot_{uc}$ from *join_uc* in the usual way by (18) and (20), and also *pext_uc* from *prod_uc* by (21). We can then prove (19) and (22).

$$\mathit{join_uc} = \mathit{join_s} \circ \mathit{map_s}\ \mathit{prod_uc} \qquad (16)$$

$$\mathit{join_uc} = \mathit{map_s}\ \mathit{join_s} \circ \mathit{dorp_uc} \qquad (17)$$

$$\mathit{ext_uc}\ f = \mathit{join_uc} \circ \mathit{map_uc}\ f \qquad (18)$$

$$\mathit{ext_uc} = \mathit{ext_s} \circ \mathit{pext_uc} \qquad (19)$$

$$g \odot_{uc} f = \mathit{ext_uc}\ g \circ f \qquad (20)$$

$$\mathit{pext_uc}\ f = \mathit{prod_uc} \circ \mathit{map_s}\ f \qquad (21)$$

$$\mathit{prod_uc}\ \mathcal{A} = \bigcap\{\mathit{map_s}\ \mathit{up_cl}\ \mathcal{A}\} \qquad (22)$$

The proofs of the monad axioms for *set set* α (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, [10] or [2]).

Lemma 3. *With join_uc defined by* (16) *or equivalently* (17)*, and ext_uc by* (18)

$$ext_uc\ f\ \{\{x\}\} = up_cl\ (f\ \{\{x\}\}) \tag{23}$$
$$ext_uc\ (\lambda x.\ \{\{x\}\}) = up_cl \tag{24}$$
$$ext_uc\ (ext_uc\ g \circ f) = ext_uc\ g \circ ext_uc\ f \tag{25}$$

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 $Rep_uca : uca\ \alpha \to set\ set\ \alpha$ and $Abs_uca : set\ set\ \alpha \to uca\ \alpha$. 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 1.

$$unit_uca\ a = uc_Abs\ \{\{a\}\} \tag{26}$$
$$ext_uca\ f\ \mathcal{A} = uc_Abs\ (ext_uc\ (Rep_uca\ \circ f)\ (Rep_uca\ \mathcal{A})) \tag{27}$$
$$swap_uca\ \mathcal{A} = uc_Abs\ (swap_uc\ (Rep_uca\ \mathcal{A})) \tag{28}$$

We then used (23) to (25) about the type *set set* α 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 1, (28)) 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 [7, §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 $T^o\ s = swap_uca\ (T\ s)$.

A Link to the Continuation Monad We can obtain an interesting link to the continuation monad [14, §3.1]. We define an "evaluation function" $eval_uc : set\ set\ \alpha \to (\alpha \to bool) \to bool$, where $eval_uc\ \mathcal{A}\ P$ tells whether the post-condition 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 $eval_uc\ \mathcal{B} = eval_uc\ (up_cl\ \mathcal{B})$.

Now the type $(\alpha \to bool) \to bool$ is the type of the continuation monad $K\ \alpha$ [14, §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:

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

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 $K\ \alpha \to set\ set\ \alpha$, 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.

$$\begin{aligned}
up_cl &= K_to_SS \circ (Ball \odot_K Bex)\\
join_uc &= K_to_SS \circ (Ball \odot_K Bex \odot_K Ball \odot_K Bex)\\
dorp_uc &= K_to_SS \circ (Bex \odot_K Ball \odot_K Bex)\\
prod_uc &= K_to_SS \circ (Ball \odot_K Bex \odot_K Ball)\\
swap_uc &= K_to_SS \circ (Bex \odot_K Ball)\\
ext_uc\ f &= K_to_SS \circ (Ball \odot_K (Bex \circ f) \odot_K Ball \odot_K Bex)\\
pext_uc\ f &= K_to_SS \circ (Ball \odot_K (Bex \circ f) \odot_K Ball)
\end{aligned}$$

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

$$mono_K\ c \equiv \forall P\ Q.\ (\forall s.\ P\ s \Rightarrow Q\ s) \Rightarrow c\ P \Rightarrow c\ Q$$

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

$$\forall a.\ mono_K\ (f\ a) \wedge \forall b.\ mono_K\ (g\ b) \Rightarrow \forall x.\ mono_K\ ((g \odot_K f)\ x)$$

Similar to the results above is the following:

$$f \odot_{uc} g = K_to_SS \circ ((eval_uc \circ f) \odot_K (eval_uc \circ g))$$

This leads to an alternative proof of the associativity of $\odot_uc$, for

$$\begin{aligned} f \odot_{uc} (g \odot_{uc} h) &= K_to_SS \circ ((eval_uc \circ f) \odot_K \\ &\qquad (eval_uc \circ (K_to_SS \circ ((eval_uc \circ g) \odot_K (eval_uc \circ h))))) \\ &= K_to_SS \circ ((eval_uc \circ f) \odot_K ((eval_uc \circ g) \odot_K (eval_uc \circ h))) \end{aligned}$$

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 In [7] Dunne 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 $set\ set\ set\ \sigma$. 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 $set\ set\ \alpha$ and the $uca\ \alpha$ types.

$$\begin{aligned} \text{dem}\ \ \mathcal{B}\ s &= prod_uc\ \{B\ s \mid B \in \mathcal{B}\} = pext_uc\ (\lambda B.\ B\ s)\ \mathcal{B} \\ \text{ang}\ \ \mathcal{B}\ s &= \bigcup \{B\ s \mid B \in \mathcal{B}\} \end{aligned}$$

So these results are analogous to those above:

$$\begin{aligned} up_cl \circ \bigcup = \bigcup \circ map_s\ up_cl &= K_to_SS \circ (Ball \odot_K Bex \odot_K Bex) \\ \bigcap \circ map_s\ up_cl &= K_to_SS \circ (Ball \odot_K Bex \odot_K Ball) \end{aligned}$$

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.

$$\text{dem}\ \ \mathcal{B}\ ;\ C = \underset{B \in \mathcal{B}}{\text{dem}}\ (B; C) \qquad \text{ang}\ \ \mathcal{B}\ ;\ C = \underset{B \in \mathcal{B}}{\text{ang}}\ (B; C)$$

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, $B; (\text{dem}_{C \in \mathcal{C}}\ C) = \text{dem}_{C \in \mathcal{C}} (B; C)$ 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 (29) and (30) (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.

$$ext_s\ (ext_uc\ g \circ f) = ext_uc\ g \circ ext_s\ f \tag{29}$$
$$pext_uc\ (ext_uc\ g \circ f) = ext_uc\ g \circ pext_uc\ f \tag{30}$$

Since $ext_uc = ext_s \circ pext_uc$, (29) and (30) combine to give (25), of the form of (3), which is the significant (usually non-trivial) one among the monad axioms.

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` $set\ \alpha \to uca\ \alpha$, as shown below (and as suggested by the discussion in [7, §5.1,§5.2]

$$tc_to_ch\ (\texttt{Term}\ S) = uc_Abs\ \{S\} \tag{31}$$
$$tc_to_ch\ \texttt{NonTerm} = uc_Abs\ \{\} \tag{32}$$

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 $tcres\ \alpha$ 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. 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, Hudak and Jones [11]. The theory underlying these has been analysed, for example by Hyland, Plotkin and Power [9], who give several general constructions, dealing with arbitrary categories (such as $\omega-\mathbf{Cpo}$) rather than just the category of sets. While all our work (and in particular, our Isabelle proofs) are, following [8] and [7], 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

1. Michael Barr and Charles Wells. Toposes, Triples and Theories. Springer-Verlag, 1983, or see `http://www.cwru.edu/artsci/math/wells/pub/ttt.html`
2. Jeremy E Dawson. Compound Monads and the Kleisli Category. Unpublished note. `http://users.rsise.anu.edu.au/~jeremy/pubs/cmkc/`
3. Jeremy E Dawson. Formalising General Correctness. In Computing: The Australasian Theory Symposium (2004), ENTCS **91** (2001), 21–42. `http://www.elsevier.com/locate/entcs`
4. Jeremy E Dawson. Formalising General Substitutions. Submitted. `http://users.rsise.anu.edu.au/~jeremy/pubs/fgc/fgs/`
5. 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`
6. 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.
7. Steve Dunne. Chorus Angelorum. In B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, 2007, LNCS 4355, Springer, 2007, 19–33.
8. Dean Jacobs and David Gries. *General Correctness: A Unification of Partial and Total Correctness*. Acta Informatica **22** (1985), 67-83.
9. Martin Hyland, Gordon D Plotkin and John A Power. Combining effects: Sum and tensor. Theor. Comput. Sci. 357(1-3): 70-99, 2006.
10. Mark P Jones & Luc Duponcheel. Composing Monads. Research Report YALEU/DCS/RR-1004, Yale University, December 1993
11. Sheng Liang, Paul Hudak and Mark P Jones. Monad Transformers and Modular Interpreters. In Symposium on Principles of Programming Languages (POPL'95), 1995, 333–343.
12. Eugenio Moggi. Computational lambda-calculus and monads. In IEEE symposium on Logic in Computer Science, IEEE, June 1989.
13. Gordon D Plotkin. A powerdomain construction. SIAM J. Computing. 5 (1976), 452–487.
14. Philip Wadler. The Essence of Functional Programming. In Symposium on Principles of Programming Languages (POPL'92), 1992, 1–14.