

A Monadic Interpretation of Tactics

Andrew Martin and Jeremy Gibbons

Oxford University Software Engineering Centre
Wolfson Building, Parks Road, Oxford OX1 3QD, UK.

Abstract. Many proof tools use ‘tactic languages’ as programs to direct their proofs. We present a simplified idealised tactic language, and describe its denotational semantics. The language has many applications outside theorem-proving activities. The semantics is parametrised by a monad (plus additional structure). By instantiating this in various ways, the core semantics of a number of different tactic languages is obtained.

1 Introduction

The notion of a *tactic* as a program used in the construction of a (machine-assisted) formal proof has become quite widespread. Tactics originate in the work of Gordon et al [GMW79] on Edinburgh LCF. The extent to which other ‘tactic-based’ systems implement essentially the same style of programming facilities varies considerably.

In Edinburgh LCF, a tactic does not itself construct a proof. Rather, it is used in backward reasoning to construct a validation function which may itself prove the desired property. Theoremhood is guarded by use of a ‘safe datatype’, and only sound validation functions may construct elements of this type. In other work, the type of theorems is protected by having the class of tactics itself protected, so that it is impossible to build unsound proofs. The account here tends towards the second view, though the treatment of tactics is actually so abstract that this may not be an impediment to its application in either sense.

Whilst tactics are widespread, tactic programming remains a difficult task. In this paper, we consider abstract descriptions of tactics, with the hope that modern algorithm design techniques, such as those described by Bird and de Moor [BdM97], can be brought to bear on the discipline on tactic programming.

Earlier discussions of tactics in the abstract (without operational bias to any particular proof tool) include those by Schmidt [Sch84] and Milner [Mil84]. The account here builds on the work described in [MGW96]. That paper describes an abstract tactic language, giving a large set of algebraic laws which show how tactics relate to one another, and how tactics may be transformed. A denotational semantics was presented using lazy lists, which gives the language an angelic nondeterminism — or equivalently, backtracking — semantics.

This paper aims to generalise that description by appealing to underlying structure, that of a monad. This permits the description of tactic relationships with even less operational bias, and allows an analysis of the effect of allowing combinations of backtracking, mutable state, and other features. In this way, we

are able to describe the essential features of a large class of different tactic-based systems.

The following section explains the necessary category theory for our task, and sets out the definition of a monad, both categorically and in the context of the Haskell language. Section 3 introduces the *Angel* language from [MGW96], after which Section 4 shows how to interpret that language in a number of different monads. Section 5 considers two ways in which the treatment may be extended to cover more features of the languages in question, and the paper ends with some conclusions.

2 Monads

Monads have been known as a categorical construction for some decades [ML98]. In a seminal paper, Moggi [Mog89] showed how monads can be used to model a number of computational effects in description of the semantics of programming languages. Spivey [Spi90] and Wadler [Wad92] (among others) have popularized this approach, showing how monads can simulate such computational effects within a pure value-oriented (‘functional’) language that does not explicitly provide these features; now monads are a mainstream technique in functional programming and a central feature of the standard lazy functional programming language Haskell [JHA⁺99].

2.1 Categorical perspective

Technically, a *category* \mathcal{C} consists of collections of *objects* $\text{Obj}_{\mathcal{C}}$ and *arrows* $\text{Arr}_{\mathcal{C}}$, such that:

- every arrow $f \in \text{Arr}_{\mathcal{C}}$ has a *source* $\text{src}(f)$ and a *target* $\text{tgt}(f)$, both in $\text{Obj}_{\mathcal{C}}$ — we write ‘ $f : \text{src}(f) \rightarrow \text{tgt}(f)$ ’;
- every object $A \in \text{Obj}_{\mathcal{C}}$ has an *identity* arrow $\text{id}_A : A \rightarrow A$;
- compatible arrows *compose* — two arrows $f : A \rightarrow B$ and $g : B \rightarrow C$ compose to form an arrow $(g \cdot f) : A \rightarrow C$;
- composition is associative, with appropriate identity arrows as units.

When modelling computations categorically, one typically chooses a category with ‘types’ as objects, and ‘programs’ from one type to another as arrows.

A *functor* $\mathcal{F} : \mathcal{C} \rightarrow \mathcal{C}$ on a category \mathcal{C} is a coherent mapping from objects to objects and from arrows to arrows in \mathcal{C} ; that is,

- when $f : A \rightarrow B$, then $\mathcal{F} f : \mathcal{F} A \rightarrow \mathcal{F} B$;
- $\mathcal{F}(\text{id}_A) = \text{id}_{\mathcal{F} A}$;
- $\mathcal{F}(g \cdot f) = \mathcal{F} g \cdot \mathcal{F} f$.

Functors compose: the composition $\mathcal{G} \cdot \mathcal{F}$ (such that $(\mathcal{G} \cdot \mathcal{F}) A = \mathcal{G}(\mathcal{F} A)$ and $(\mathcal{G} \cdot \mathcal{F}) f = \mathcal{G}(\mathcal{F} f)$) is again a functor, as the reader may check. In computational settings, functors typically represent ‘type constructors’; for example, the functor

$\mathcal{P}air$ will take a type A to another type $\mathcal{P}air A$ of pairs with elements drawn from A , and a program $f : A \rightarrow B$ to another program $\mathcal{P}air f : \mathcal{P}air A \rightarrow \mathcal{P}air B$ which applies f pointwise to each element of the argument pair.

One can also define *bifunctors*, which are binary operations on objects and on arrows, functorial in each argument. That is, a bifunctor \oplus satisfies

- when $f : A \rightarrow B$ and $g : C \rightarrow D$, then $(f \oplus g) : (A \oplus C) \rightarrow (B \oplus D)$;
- $\text{id}_A \oplus \text{id}_B = \text{id}_{A \oplus B}$;
- $(g \cdot f) \oplus (k \cdot h) = (g \oplus k) \cdot (f \oplus h)$.

Two bifunctors that will come up later are the *product* bifunctor \times , for which $A \times B$ may be thought of as the type of pairs with left components of type A and right component of type B , and the *disjoint sum* bifunctor $+$, for which $A + B$ may be thought of as the type consisting of the elements of type A together with those of type B . A third operation on objects and arrows is the *function space* operator \Rightarrow , for which $A \Rightarrow B$ may be thought of as the type of functions from A to B ; for technical reasons — contravariance in the left-hand argument — this turns out not to be a bifunctor, but the right-hand argument is well behaved, and the operation $(A \Rightarrow)$ for fixed A taking object B to object $A \Rightarrow B$ and arrow $f : B \rightarrow C$ to arrow $(f \cdot) : (A \Rightarrow B) \rightarrow (A \Rightarrow C)$ is an ordinary functor.

Given two functors $\mathcal{F}, \mathcal{G} : \mathcal{C} \rightarrow \mathcal{C}$, a *natural transformation* $\phi : \mathcal{F} \rightarrow \mathcal{G}$ is a family of arrows ϕ_A indexed by the objects of \mathcal{C} , such that:

- $\phi_A : \mathcal{F} A \rightarrow \mathcal{G} A$;
- $\phi_B \cdot \mathcal{F} f = \mathcal{G} f \cdot \phi_A$ for each $f : A \rightarrow B$.

Interpreted computationally, one can think of a natural transformation $\phi : \mathcal{F} \rightarrow \mathcal{G}$ as a ‘polymorphic function’, of type $\mathcal{F} A \rightarrow \mathcal{G} A$ for any A . The second axiom above guarantees *parametric polymorphism*, that the actions of ϕ for each A are related in a structured way; whatever action ϕ has is unaffected by the elements of type A , so essentially all ϕ can do is to rearrange and discard elements of the data structure. For example, the polymorphic function $\text{fst} : \mathcal{P}air A \rightarrow \mathcal{I}d A$ for any A (where $\mathcal{I}d$ is the identity functor: $\mathcal{I}d A = A$ and $\mathcal{I}d f = f$), satisfies $\text{fst} \cdot \mathcal{P}air f = \mathcal{I}d f \cdot \text{fst}$.

A *monad* is a triple (\mathcal{M}, η, μ) , with a functor \mathcal{M} and natural transformations $\eta : \mathcal{I}d \rightarrow \mathcal{M}$ (the *unit* of the monad) and $\mu : \mathcal{M} \cdot \mathcal{M} \rightarrow \mathcal{M}$ (the *multiplier* of the monad), satisfying the following laws:

- $\mu \cdot \mathcal{M} \mu = \mu \cdot \mu$;
- $\mu \cdot \eta = \text{id}$;
- $\mu \cdot \mathcal{M} \eta = \text{id}$.

For example, consider the functor $\mathcal{L}ist$, for which $\mathcal{L}ist A$ is the type of finite lists with elements drawn from A , and $\mathcal{L}ist f$ applies f pointwise to each element of a list. This functor forms a monad, with unit the natural transformation $\text{wrap} : \mathcal{I}d \rightarrow \mathcal{L}ist$ which turns an element into a singleton list, and multiplier $\text{concat} : \mathcal{L}ist \cdot \mathcal{L}ist \rightarrow \mathcal{L}ist$ which concatenates a list of lists into a single long list. The reader may check the naturality conditions and the extra monad laws

(which state in turn that a list of lists of lists may be concatenated into one list in two equivalent ways, that wrapping a list then concatenating is a null operation, and that wrapping every element of a list then concatenating is another null operation).

2.2 Monads for computational effects

Moggi's [Mog89] seminal contribution was to observe that monads can capture various 'computational effects', such as non-determinism, exceptions and side-effects, in describing the semantics of a programming language. Rather than modelling a program taking a value of type A and returning a value of type B as an arrow $A \rightarrow B$, thereby treating it as a 'pure function', doing no more and no less, Moggi argued for modelling such a program as an arrow $A \rightarrow \mathcal{M} B$ for some monad \mathcal{M} . Intuitively, the object $\mathcal{M} A$ represents a *computation* which, when executed, may have some computational effects as well as yielding results of type A — hence the slogan *a program is a function from values to computations*.

Example 1 (List monad). One might want to model non-deterministic programs, that may give multiple results of type B from the same input of type A . Then one should choose the monad $\mathcal{L}ist$, modelling the program as an arrow $A \rightarrow \mathcal{L}ist B$. (If one chooses the functor of non-empty lists, one gets non-deterministic but never-failing programs; if one chooses possibly-empty lists one also allows the possibility for programs that fail, returning no results at all. Both functors form monads, as the reader may verify.)

Example 2 (Exception monad). If one wants to model possible failure but not non-determinism, one may choose a functor corresponding to 'lists of length at most one', or equivalently, the functor taking A to $1 + A$ where 1 is the unit type with a single element. (The reader may check that this too is a monad.) A failing program is modelled by returning a trivial result in the left of the sum, and a non-failing program by returning a normal result in the right side. This monad may be generalized to take A to $E + A$ for some type E of exceptions.

Example 3 (State monad). Less obviously, a program from A to B which may modify some stored state of type S can be modelled as a function from $A \times S$ to $B \times S$, or equivalently using currying as a function from A to $S \Rightarrow (B \times S)$. The functor $\mathcal{S}tate S$ taking B to $S \Rightarrow (B \times S)$ is another monad, with unit $\lambda b. (\lambda s. (b, s))$ and multiplier $\lambda f. (\lambda s. (\lambda (g, t). g t) (f s))$, as the energetic reader may check.

Example 4 (Continuation monad). Even continuation-passing style, which has been used to model many sophisticated programming language features, from `goto` statements onwards, fits the monadic pattern. In continuation-passing style, a computation taking an argument of type A and yielding results of type B is subject to post-application with a continuation of type $B \Rightarrow R$ for some overall result type R , so may be modelled as a function from A to $(B \Rightarrow R) \Rightarrow R$. It turns out that the functor $\mathcal{C}ps R$ taking B to $(B \Rightarrow R) \Rightarrow R$ is another

monad. (Note that although the argument type B is used here as the left-hand argument to a function space constructor, $\mathit{Cps} R$ is nevertheless covariant: in this case, two wrongs do make a right.) The unit is $\lambda b. (\lambda k. k b)$, and the multiplier $\lambda m. (\lambda c. m (\lambda v. v c))$ — a claim that even the energetic reader may wish to take on trust.

2.3 Combining monads

Of course, one often wants to model not just one kind of computational effect, but several at once. For example, one might want to model programs that both are non-deterministic and modify state. For this, one needs to combine monads. There are no general constructions to achieve this, but in certain special cases one can make progress [JD93,KW93]. We will return to this point later, when we consider the interpretation of tactics under particular monads.

2.4 Monads in Haskell

Moggi's idea was to use monads to simplify and unify disparate constructions in programming language semantics. Wadler and others realized that monads could also be used to simplify and unify disparate constructions in programs in a given language. In many cases, programmers in a pure functional programming language, which by definition does not provide the various computational effects mentioned above, wished to simulate such effects in their programs. Each of the constructions discussed in Section 2.2 was known and used by programmers before Moggi's and Wadler's observations; but their observations revealed that all of these constructions were instances of the same general scheme.

It turns out to be more convenient in programming to use a different, but equivalent, formulation of monads. In this view, a monad (\mathcal{M}, η, μ) is equivalent to a *Kleisli triple* $(\mathcal{M}, \eta, _*)$, where the operator $_*$ lifts an arrow in $A \rightarrow \mathcal{M} B$ to one in $\mathcal{M} A \rightarrow \mathcal{M} B$. In one direction, the equivalence is given by $\mu = \text{id}^*$, and in the other, by $m^* = \mu \cdot \mathcal{M} m$. This view is more convenient for programming because it lives happily with variable binding, whereas the original view leads more towards a pointfree programming style, which is easier for reasoning but harder for practical programming [dMG00].

Haskell, therefore, defines the following type class in its standard prelude:

```
> class Monad m where
>   (>>=) :: m a -> (a -> m b) -> m b
>   return :: a -> m a
```

Clearly, `return` corresponds to η , and (perhaps less clearly) the 'binding' operator `>>=` to $_*$ with the arguments permuted.

Two things are lacking from this declaration. The first is a connection with functors: despite Haskell's standard prelude also having a type class

```
> class Functor f where
>   fmap :: (a -> b) -> (f a -> f b)
```

this has not been required as a superclass of `Monad`. We take this to be an omission, and will pretend from now on that the `Monad` class had been defined

```
> class Functor m => Monad m where ...
```

so that every monad is a functor.

The second omission is any mention of the laws a monad should satisfy: the naturality laws are essentially consequences of the polymorphic types of `>>=` and `return` [Wad89], but the coherence laws cannot be stated in Haskell. The equivalents in the Haskell presentation of the three monad coherence laws are:

- `return v >>= k = k v`;
- `m >>= return = m`;
- `(m >>= k) >>= h = m >>= (\ v -> k v >>= h)`.

The four monads described in Section 2.2 are captured in Haskell as follows. In each case, the reader may wish to check that the monad coherence laws are satisfied.

Example 5 (List monad in Haskell).

```
> instance Monad [ ] where
>   xs >>= f = concat (map f xs)
>   return v = [v]
```

Example 6 (Exception monad in Haskell).

```
> data Maybe a = Just a | Nothing
> instance Monad Maybe where
>   Just v >>= k = k v
>   Nothing >>= k = Nothing
>   return a     = Just a
```

Example 7 (State monad in Haskell).

```
> newtype State s a = St (s -> (a,s))
> unSt (St p) = p
> instance Monad (State s) where
>   return v = St (\ s -> (v,s))
>   St p >>= f = St (\ s -> let (v,s') = p s in unSt (f v) s')
```

Example 8 (Continuation monad in Haskell).

```
> data CPS r a = CPS ((a -> r) -> r)
> unCPS (CPS p) = p
> instance Monad (CPS r) where
>   return v = CPS (\ c -> c v)
>   CPS p >>= k = CPS (\ c -> p (\ v -> unCPS (k v) c))
```

One of the benefits of unifying these monadic notions is that it allows one to define language support for them. Haskell provides a `do` notation, an extension of the list comprehension syntax to arbitrary monads. For example, consider the problem of taking a pair of monadic values and returning a monad of pairs, with all possible combinations of a result from the first computation and a result from the second:

```
> allPairs :: Monad m => (m a, m b) -> m (a,b)
> allPairs m n = m >>= (\ v -> (n >>= (\ w -> return (v,w))))
```

(We'll want to do something very like this in Section 5.2.) Haskell's `do` notation allows one to write instead

```
> allPairs :: Monad m => (m a, m b) -> m (a,b)
> allPairs m n = do { v <- m ; w <- n ; return (v,w) }
```

and this syntax works for an arbitrary monad, not just for lists.

2.5 Subclasses of monad

In order to model some aspects of tactic languages, we need more features than are provided by monads in general. For this purpose, we introduce some subclasses of monads, which encapsulate these special cases.

The subclass `MonadZero` of `Monad` has a distinguished constant:

```
> class Monad m => MonadZero m where
>   mzero :: m a
```

The canonical instance of this class is the `Maybe` type:

```
> instance MonadZero Maybe where
>   mzero = Nothing
```

but lists are also an instance (with the empty list as the zero). As the name suggests, this is a zero of the monadic composition:

- `mzero >>= k = mzero`

The subclass `MonadPlus` provides a binary operator, with which to combine monadic values:

```
> class MonadZero m => MonadPlus m where
>   mplus :: m a -> m a -> m a
```

The canonical instance is the type of (possibly-empty) lists:

```
> instance MonadPlus [ ] where
>   mplus = (++)
```

but the `Maybe` type is also an instance.

As the names suggest, `mplus` and `mzero` should form a monoid; moreover, `bind` should distribute backwards over `mplus`:

- `(m 'mplus' n) 'mplus' p = m 'mplus' (n 'mplus' p)`;
- `m 'mplus' mzero = m`;
- `mzero 'mplus' m = m`;
- `(m 'mplus' n) >>= k = (m >>= k) 'mplus' (n >>= k)`.

We have made `MonadPlus` a subclass of `MonadZero` for simplicity, but we might have chosen instead to make it a direct subclass of `Monad` and define a new subclass of both `MonadZero` and `MonadPlus`; then a type of non-empty lists would be a `MonadPlus` but not a `MonadZero`. (In fact, the Haskell standard prelude defines `MonadPlus` to have both `mzero` and `mplus`, and the option we have chosen here refines that class hierarchy.)

We introduce a third subclass `MonadCut`, with an operator `mcut` which, informally, prunes any non-determinism from a monadic value:

```
> class Monad m => MonadCut m where
>   mcut :: m a -> m a
```

Cutting is trivial on the `Maybe` and `State` monads, since they have no non-determinism anyway:

```
> instance MonadCut Maybe where
>   mcut = id

> instance MonadCut (State s) where
>   mcut = id
```

but on the list monad it prunes the list to length one:

```
> instance MonadCut [ ] where
>   mcut = take 1
```

The laws here seem rather more complicated, and perhaps there is scope for further exploration and refinement. Obvious laws are that failing computations are deterministic, so pruning has no effect:

- `mcut mzero = mzero`,

and that `mcut` yields deterministic computations, so pruning is idempotent:

- `mcut . mcut = mcut`.

A computation that yields at least one result will yield no more when pruned:

- `mcut (return v 'mplus' m) = return v`,

(and so, taking `m = mzero` in the above, `return v` is always deterministic). Non-determinism may be pruned within a computation if it is not needed later:

- `mcut (m 'mplus' n) = mcut (mcut m 'mplus' mcut n)`,
- `mcut (k >>= id) = mcut (k >>= mcut)`.

Other laws of `mcut` are used in [MGW96] to prove various properties of tactic programs.

3 Angel

Angel [MGW96] is a generic tactic language. Initially it was intended to support proofs in the goal-directed style, that is, providing a framework for the composition of primitive inference rules in the construction of backwards proofs, but it turns out to be more general than this. Term rewriting, for example, which in Cambridge LCF [Pau87] is directed by a separate set of operators from those which describe tactics, could also be described using Angel.

Because Angel is a small language, its semantic description is quite clean and easy to reason about. Nevertheless it is able to describe a large class of useful algorithms. The language is named Angel because the account in [MGW96] makes tactics angelically nondeterministic. That is, a tactic is a relation between goals, rather than a simple function. Thus we may arrange that a compound tactic will fail only if there is no possible path from input to output: an implementation must avoid dead-end paths, typically achieves this by a backtracking search. The angelic style removes the need for the tactic programmer to program backtracking explicitly.

Primitive inference rules (or rewrites) are referenced using atomic tactics of the form ‘**rule** *R*’. Other atomic tactics are **skip** (which always succeeds, leaving its goal unchanged) and **fail**, which always fails. Tactics may be sequentially composed (;), or placed in alternation (!). For efficiency reasons, it may be appropriate to limit the scope of the angelic choice, so ‘!*t*’ restricts the nondeterminism to the scope of *t*. This notation resembles the ‘cut’ of Prolog, but thanks to its scoping, the semantics of this ‘!’ are more compositional (similar to Isabelle’s **DETERM** [Pau94]).

For an example, consider tactics over a logical language of first-order predicates. Let the rule *alphaConv* perform alpha-conversion (renaming bound variables in a quantified expression), and *allIntro* perform some kind of forall-introduction step. This typically has a side-condition, and so may in some situations need to be preceded by alpha-conversion if failure is to be avoided. A tactic which does this, then, is:

$$t_1 \equiv (\mathbf{skip} \mid \mathbf{rule} \textit{alphaConv}) ; \mathbf{rule} \textit{allIntro} .$$

This tactic has the additional property that if post-composed with some other tactic, the possibility of alpha conversion before all-introduction remains open, whereas $!t_1$ fixes the choice so that it cannot be revisited. (In a system displaying referential transparency, this tactic will be deficient, since the alpha conversion will always choose the same new bound variable, based only on the predicate presented to the rule. We will address this issue later.)

The semantics given for Angel in [MGW96] is expressed using a simple theory of lists, based on [Bir86]. The lists are used to record alternative outcomes of tactic applications, thus offering backtracking/angelic nondeterminism. This is reflected in the name *Angel*.

A simple (in the sense that it imposes total deterministic interpretations on the primitive rules) translation of the tactic combinators to Haskell would be:

```

> skip g      = [g]
> fail g      = []
> rule r g    = [r g]
> alt t1 t2 g = t1 g ++ t2 g
> sequ t1 t2 g = concat (map t2 (t1 g))
> cut t g     = take 1 (t g)

```

so that the tactic t_1 above may be expressed as:

```

> sequ (alt skip (rule alphaConv)) (rule allIntro)

```

which may be expanded to

```

concat . (map (\ g -> [allIntro g])) . (\ g -> [g] ++ [alphaConv g])

```

and hence simplified to

```

(\ g -> [allIntro g, allIntro (alphaConv g)])

```

The cut version of the tactic — which does not ‘backtrack’ by lazily presenting alternatives — composes `take 1` with this expression.

4 Generalized Tactic Model

Several readers of [MGW96] commented on the similarity of the lists used there with various other algebraic structures, captured with the categorical notion of a *monad*. With this in mind, we may abstract away from the list-based presentation of tactic semantics, using an arbitrary monad instead. Different instantiations for the monad will lead to different tactic languages (with different properties, as we shall see).

4.1 Interpretation of a tactic in a monad

We assume basic types for primitive rules and for variables of the tactic language:

```

> type Rule = String

```

The terms of the tactic language are then expressed in the obvious way with the following datatype:

```

> data Tactic = Rule Rule
>             | Skip
>             | Fail
>             | Seq Tactic Tactic
>             | Alt Tactic Tactic
>             | Cut Tactic

```

The interpretation of terms in the tactic language using a particular monad is then captured with the function `int`, which we now discuss. For a particular monad `m` and object-level expression type `e`, this function basically turns a `Tactic` into a function of type `e -> m e`, which takes an expression and yields a computation with rewritten expressions as results. However, `int` also takes as parameter a function `r` of type `Rule -> e -> m e` giving the interpretation of primitive rules. Moreover, the monad `m` must be a member of all the monad subclasses described in Section 2.5, to allow interpretation of the failing, alternation and cut combinators. We therefore declare

```
> int :: (Monad m, MonadZero m, MonadPlus m, MonadCut m) =>
>       (Rule -> e -> m e) -> Tactic -> e -> m e
```

The definition of `int` is given in Figure 1; below, we explain each clause of the definition of `int` in turn.

```
> int :: (Monad m, MonadZero m, MonadPlus m, MonadCut m) =>
>       (Rule -> e -> m e) -> Tactic -> e -> m e
> int rule (Rule r) = \ e -> rule r e
> int rule Skip     = \ e -> return e
> int rule Fail     = \ e -> mzero
> int rule (Seq t u) = \ e -> (int rule t e >>= \ e' -> int rule u e')
> int rule (Alt t u) = \ e -> int rule t e 'mplus' int rule u e
> int rule (Cut t)   = \ e -> mcut (int rule t e)
```

Fig. 1. Interpretation of a tactic in a monad

- The interpretation of primitive rules is determined by the parameter `r`:


```
> int r (Rule rule) = \ e -> r rule e
```
- The tactic `Skip` corresponds to the unit of the monad, returning the object-level term unchanged and having no computational effect:


```
> int r Skip = \ e -> return e
```
- The tactic `Fail` yields no result:


```
> int r Fail = \ e -> mzero
```
- Sequential composition of tactics corresponds to binding in the monad — the argument is rewritten by the two tactics in turn:


```
> int r (Seq t u) = \ e ->
>   (int r t e >>= \ e' -> int r u e')
```

These complex ‘bind’ expressions are exactly what Haskell’s ‘do’-notation is designed to clarify: we could define equivalently

```
> int r (Seq t u) = \ e ->
>   do { e' <- int r t e ; e'' <- int r u e' ; return e'' }
```

A third way of looking at this is in terms of the pointfree definition of a monad as a triple (\mathcal{M}, η, μ) ; if we define $\mu m = m \gg= \text{id}$, which translates monadic bind into monadic multiplication, then we could define sequential composition of two tactics equivalently as Kleisli composition of their interpretations:

```
> int r (Seq t u) = mu . fmap (int r u) . int r t
```

- Alternation of tactics corresponds to the `mplus` method of the `MonadPlus` class:

```
> int r (Alt t u) = \ e -> int r t e 'mplus' int r u e
```

(here, 'f' in Haskell turns the two-argument function `f` into an infix binary operator).

- Similarly, the `Cut` combinator is interpreted using the `mcut` member of the `MonadCut` class:

```
> int r (Cut t) = \ e -> mcut (int r t e)
```

4.2 Some example interpretations

Using the list monad given above with the `int` function gives rise to the Angel semantics presented in [MGW96], as produced by the simple interpretation given in Section 3 above. Suppose that we define the tactic

```
> t1 :: Tactic
> t1 = (Skip 'Alt' Rule "alphaConv") 'Seq' (Rule "allIntro")
```

and assume an interpretation `rule` for primitive rules. Then the interpretation `int t1` of the tactic in the `List` monad is

```
> int t1 g = rule "allIntro" g ++
>           concat (map (rule "allIntro") (rule "alphaConv" g))
```

essentially as before. Such behaviour is also possible in *Isabelle* [Pau89], though several combinators are offered there.

Interpretation in the exception monad gives rise (essentially) to the tactic language of Edinburgh (and Cambridge) LCF. To be more precise, LCF allows side-effects in tactics (see below); the logical framework 2OBJ [GSHH92] implemented precisely this semantics. For clarity, we could define

```
> try (Just x) _ = Just x
> try Nothing y = y
```

to try a second computation only if a first fails, and

```
> lift f (Just x) = f x
> lift f Nothing = Nothing
```

to lift a monadic function with a plain argument to work on monadic arguments, then the interpretation of `t1` in the `Maybe` monad is

```
> t1 g = try (rule "allIntro" g)
>         (lift (rule "allIntro") (rule "alphaConv" g))
```

(In fact, `try` is just `mplus`, and `lift f x` is `x >>= f`.)

The semantics of systems such as LCF requires state, and the `State` monad given above implements this. Such a state is necessary for the practical implementation of rules such as alpha-conversion: by allowing a state component to record information about variable names in use, we may ensure the selection of an entirely fresh name when we need it. State alone forms a monad, but not all of the foregoing subclasses of monad. Therefore, a practical implementation such as LCF will use `State` together with `Maybe`. We have not explored this monad combination here, though it is one of the combinations mentioned in [KW93].

A more general combination (capturing more accurately, for example, the choice semantics possible in Isabelle) involves combining `State` with lists. There are several ways to accomplish such a combination, but not all are equally good. Straightforward composition of functors suggests either $S \Rightarrow (List\ B \times S)$, in which all outcomes of a non-deterministic computation share the same final state, or $List\ (S \Rightarrow (B \times S))$, which is not obviously a monad. The most promising approach seems to be to use $S \Rightarrow List\ (B \times S)$. Here, each alternative outcome gets its own copy of the state; this is the choice defined by King and Wadler.

In Haskell, this functor would be defined

```
> data LState s a = LSt (s -> [(a,s)])
> unLSt (LSt p) = p
```

Installation as an instance of `Functor` is simplest expressed as a list comprehension:

```
> instance Functor (LState s) where
>   fmap f (LSt p) = LSt (\ s -> [ (f a, s') | (a,s') <- p s ])
```

For a plain monad, we define

```
> instance Monad (LState s) where
>   return a = LSt (\ s -> [(a,s)])
>   LSt p >>= f = LSt (\ s -> [ (a',s'') | (a,s') <- p s,
>                                     (a',s'') <- unLSt (f a) s' ])
```

The instances of the three subclasses of `Monad` lift in a straightforward way from the list monad:

```
> instance MonadZero (LState s) where
>   mzero = LSt (\ s -> [])

> instance MonadPlus (LState s) where
>   LSt p 'mplus' LSt q = LSt (\ s -> p s ++ q s)

> instance MonadCut (LState s) where
>   mcut (LSt p) = LSt (\ s -> take 1 (p s))
```

Most proof tools in the LCF family have an interactive mode of operation, in which various goals and subgoals can be selected for consideration, or deferred until later. This is generally a very high level of interaction, and not part of the consideration of tactics. In Ergo 5 [MNU97a] however, those facilities are fully part of the tactic language — which in other respects implements state with lists, as above. In summary, the user is allowed to `defer` treatment of a goal at any time, whereupon it is placed in a pool of open goals, to be re-opened at any time. We believe that the Ergo 5 tactic semantics therefore corresponds to a combination of lists, state, and continuations. This assertion has yet to be tested against the Prolog code for Ergo 5.

Of course, a huge class of other possible monads exists, and it may be that others of these represent useful semantics for tactic languages, perhaps in combination with the features explored above.

5 Extensions

The previous section defined an interpretation of a simple tactic language in a variety of monads. We have omitted from that initial treatment two significant aspects of Angel, namely *recursive tactics* and *structural combinators*. The first of these does not interact significantly with the monadic interpretation; we explain the necessary adjustments in Section 5.1. The second requires some kind of *generic programming* for a proper treatment, but in Section 5.2 we outline a close approximation in Haskell.

5.1 Treatment of recursion

Angel also provides for recursively-defined tactics ($\mu X \bullet tac(X)$), where X is a variable and $tac(X)$ a tactic in which X may appear as if it were itself a tactic. (Of course, the ‘ μ ’ here has nothing to do with the multiplier of a monad.) The tactic ($\mu X \bullet tac(X)$) behaves like $tac(X)$, but with each occurrence of X behaving like ($\mu X \bullet tac(X)$). The introduction of recursion in the language necessary introduces also the possibility of non-termination of tactic programs, for which we define the atomic tactic **abort**. This is different from mere failure of a tactic to apply, and is typically not something one would ever write in a tactic program, but (following Morgan [Mor94]) we find it convenient to be able to reason about it.

We can extend our interpretation to cover recursion as follows. We define a type of variable names:

```
> type Var = String
```

and introduce three new constructors for terms in the tactic language:

```
> data Tactic = ...
>             | Mu Var Tactic
>             | Var Var
>             | Abort
```

The interpretation function `int` takes an extra argument, an environment `rho` of type `[(Var, e -> m e)]` recording the interpretation of the tactic variables in scope, leading to the type declaration

```
> int :: (Monad m, MonadZero m, MonadPlus m, MonadCut m) =>
>       (Rule -> e -> m e) -> [(Var, e -> m e)] ->
>       Tactic -> e -> m e
```

The three extra clauses of the definition are as follows:

- Variable bindings augment the environment with a new maplet, and are interpreted as ‘letrec’ bindings on account of the recursive `where` clause (that is, recursion in the tactic language corresponds to recursion in the implementation language):

```
> int r rho (Mu v t) = h where h = int r ((v,h):rho) t
```

- Variable use depends on the environment, and fails for unbound variables:

```
> int r rho (Var v) = \ e -> case lookup v rho of
>   Nothing -> mzero
>   Just h -> h e
```

(here, the function `lookup :: Eq a => a -> [(a,b)] -> Maybe b` from the standard prelude looks up an entry in an association list).

- **abort** really corresponds to an erroneous tactic program:

```
> int r rho Abort = error "Aborting tactic"
```

The other clauses remain unchanged, apart from having to pass around the argument `rho`.

5.2 Treatment of structural combinators

In [MGW96] the language Angel is extended to provide for *structural combinators*. When an operator \clubsuit appears as the top-level constructor in an object level term $(e_1 \clubsuit e_2)$, applying the tactic $t_1 \boxtimes t_2$ involves the application of t_1 to e_1 and t_2 to e_2 , leaving the \clubsuit constructor in place. (Clearly, this combinator is not validity-preserving for all object-level constructors; some monotonicity property is needed.) This mechanism allows Angel tactics to be applied to the leaves of a proof tree under construction: we may apply tactics selectively within the list of open nodes by lifting the tactics over the constructor for the open node list.

We can extend our interpretation of tactics within a monad to cover structural combinators, with a little effort. We redefine the datatype `Tactic` as follows:

```
> data Tactic f g = Rule Rule
>               | Skip
>               | Fail
>               | Seq (Tactic f g) (Tactic f g)
>               | Alt (Tactic f g) (Tactic f g)
>               | Cut (Tactic f g)
>               | Struct (g (Tactic f g))
```

The two type constructor parameters `f` and `g` are both required solely for the structural combinator case. Informally, `f` is the base functor of the recursive type of object-level terms, and `g` gives the members of this base functor; but this informal description is expanded and explained below when we come to discussing the interpretation of the tactic language in a particular monad.

The object-level terms can now no longer be arbitrary, because structural tactics must interact with their structure. We therefore represent these terms by a recursive datatype `Expr`:

```
> data Functor f => Expr f = Fix (f (Expr f))
> unFix (Fix x) = x
```

Here, the parameter `f` is the base functor of the type, and `Expr f` is (intended to be) isomorphic to `f (Expr f)`; the isomorphism is given in one direction by the type constructor `Fix`, and in the other by the function `unFix`. The same `f` is intended to be the first parameter of the tactic datatype constructor `Tactic`.

For example, consider a simple datatype of boolean expressions, constructed from literals using negation, conjunction and disjunction. We represent this by the following datatype:

```
> data BoolExprF b = Lit Char | Neg b | And b b | Or b b
> type BoolExpr = Expr BoolExprF
```

The term $p \wedge \neg q$ would be represented by the expression

```
Fix (And (Fix (Lit 'p')) (Fix (Neg (Fix (Lit 'q')))))
```

Angel would provide four structural combinators for this term type, one for each variant. (The base case of literals might be omitted.) We simplify the definition by providing a single structural combinator, covering all variants simultaneously. This single structural combinator needs to take five tactics as arguments: one tactic to be applied in case the term is a negation; two tactics, one for each argument, in case it is a conjunction; and two more, in case it is a disjunction. We therefore provide a datatype `BoolSubs`, for the possible subexpressions of a boolean expression:

```
> data BoolSubs a = BS a (a,a) (a,a)
```

The relationship between `BoolExprF` and `BoolSubs` is captured by an auxiliary type class, which we call `Subs`:

```
> class Subs f g where
>   distribute :: g a -> f b -> f (a,b)
```

The method `distribute` labels an expression of type `f b` with additional labels of type `a` drawn from a data structure of type `g a`. (If the `f` is single-shaped, then `g` and `f` will be equal and `distribute` will be a kind of zip [HB97]; but in general, `f` will be a sum with several variants, and the sums in `f` will be expanded to products in `g`.) The types `BoolExprF` and `BoolSubs` instantiate this class:

```

> instance Subs BoolExprF BoolSubs where
>   distribute (BS _ _ _) (Lit c) = Lit c
>   distribute (BS b _ _) (Neg e) = Neg (b,e)
>   distribute (BS _ (b1,b2) _) (And e1 e2) = And (b1,e1) (b2,e2)
>   distribute (BS _ _ (b1,b2)) (Or e1 e2) = Or (b1,e1) (b2,e2)

```

Such a `g` is intended to be the second argument to the type constructor `Tactic`. The `Struct` constructor takes a `g` of tactics, and appropriate ones are applied to the subterms of a term.

One more issue needs to be addressed. Once each subterm of a term has been rewritten, we end up with an outermost object-level term constructor with monadic subterms. Those monadic effects should be promoted outside the term constructor and collected into a single overall monadic effect. This promotion is captured by the following type class:

```

> class Promotable f where
>   promote :: Monad m => f (m a) -> m (f a)

```

and instantiated for boolean expressions as follows:

```

> instance Promotable BoolExprF where
>   promote (Lit c) = return (Lit c)
>   promote (Neg x) = do { e <- x ; return (Neg e) }
>   promote (And x y) = do { e1 <- x ; e2 <- y ; return (And e1 e2) }
>   promote (Or x y) = do { e1 <- x ; e2 <- y ; return (Or e1 e2) }

```

These ingredients are sufficient to extend the interpretation of a tactic within a monad to cover structural combinators too. We enforce instances of these two classes on our tactics:

```

> data (Promotable f, Subs f g) => Tactic f g = ...

```

We revise the type of `int` to

```

> int :: (Functor m, Monad m, MonadZero m, MonadPlus m, MonadCut m,
>        Functor f, Subs f g, Promotable f) =>
>        (Rule -> Expr f -> m (Expr f)) ->
>        Tactic f g -> Expr f -> m (Expr f)

```

and add the clause for structural combinators:

```

> int r (Struct subs) =
>   fmap Fix . promote . fmap (uncurry (int r)) . distribute subs . unFix

```

Let us unpack this rather dense expression. We are given an expression `e`, and we strip the constructor `Fix` from `e = Fix x`, yielding `x :: f (Expr f)`; distribute the structural rules `subs`, yielding a value of type `f (Tactic f g, Expr f)`; apply each tactic to the corresponding subexpression, yielding `f (m (Expr f))`; promote the monadic effects, yielding `m (f (Expr f))`; reapply the constructor, yielding `m (Expr f)`.

Note that the monadic aspects of terms constructed from monadic values can be collected in different ways; in particular, there is a choice as to whether

the effect of a lefthand subterm precedes or follows that of a righthand subterm. However, it seems reasonable to impose a left-to-right ordering on subterms, allowing instances of the `Promotable` class to be mechanically derived. Similarly, the subexpression type (like `BoolSubs`) and the instantiation of the `Subs` type could also be mechanically derived. We see this as an elegant application of Generic Haskell [Hin00], but have not yet had time to explore this connection.

Another paper [MNU97b] explores the algebraic treatment of *associative* structural combinators in the *Gumtree* interpretation of Angel. Associative matching potentially increases the complexity of tactic evaluation quite considerably, but through an algebraic treatment of tactic arity (essentially, a simple typing scheme for tactics), considerable optimisations can be made. We expect that this generalised treatment of structural combinators could be incorporated into our monadic treatment, and hope to explore that possibility in future.

6 Conclusions

One of the contributions of the paper on Angel was to show how a large class of tactic programs could be treated algebraically, giving for the first time a way to discuss the flow of control and essential use of *failure* in tactic languages of the LCF family.

In this paper, we have generalised that treatment, and shown how a semantics using monads can give rise to many of the algebraic laws previously considered as essential definitions of the tactic language. The foregoing discussion of extensions points to much scope for future work on structural combinators. Such a language may be seen as an economical way to describe proof procedures at a high level, wherein their efficiency and optimisation can be considered. The algebraic properties used in such considerations will determine in which monads — and so, upon which proof platforms — such proof procedures can be executed.

References

- [BdM97] Richard Bird and Oege de Moor. *Algebra of Programming*. Series in Computer Science. Prentice Hall International, Hemel Hempstead, 1997.
- [Bir86] Richard S. Bird. An introduction to the theory of lists. Technical Monograph PRG-56, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK, 1986.
- [dMG00] Oege de Moor and Jeremy Gibbons. Pointwise relational programming. In *LNCS 1816: Algebraic Methodology and Software Technology*, pages 371–390, May 2000.
- [GMW79] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer-Verlag, 1979.
- [GSHH92] Joseph Goguen, Andrew Stevens, Hendrik Hilberdink, and Keith Hobley. 2OBJ: A Metalogical Theorem Prover based on Equational Logic. *Philosophical Transactions of the Royal Society, Series A*, 339:69–86, 1992. Also in C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, Prentice-Hall, 1992.

- [HB97] Paul Hoogendijk and Roland Backhouse. When do datatypes commute? In Eugenio Moggi and Guiseppe Rosolini, editors, *LNCS 1290: Category Theory and Computer Science*, pages 242–260. Springer-Verlag, September 1997.
- [Hin00] Ralf Hinze. A new approach to generic functional programming. In *Principles of Programming Languages*. ACM, 2000.
- [HS85] C. A. R. Hoare and J. C. Shepherdson, editors. *Mathematical Logic and Programming Languages*. Prentice Hall, 1985.
- [JD93] Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report RR-1004, DCS, Yale, December 1993.
- [JHA⁺99] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98: A non-strict, purely functional language. www.haskell.org/onlinereport, February 1999.
- [KW93] David J. King and Philip Wadler. Combining monads. In J. Launchbury and P. M. Sansom, editors, *Functional Programming, Glasgow 1992*. Springer, 1993.
- [MGW96] A. P. Martin, P. H. B. Gardiner, and J. C. P. Woodcock. A tactic calculus. *Formal Aspects of Computing*, 8(4):479–489, 1996. An abridged version appears in the printed journal; the full version is available in the electronic supplement to Formal Aspects of Computing, 8E, pp244–285. http://link.springer.de/link/service/journals/00165/supp/list94_96.htm.
- [Mil84] R. Milner. The use of machines to assist in rigorous proof. *Philosophical Transactions of the Royal Society, London. Series A*, 312:411–422, 1984. Also in [HS85].
- [ML98] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, second edition, 1998.
- [MNU97a] A. Martin, R. Nickson, and M. Utting. A tactic language for Ergo. In Lindsay Groves and Steve Reeves, editors, *Formal Methods Pacific '97*, Springer Series in Discrete Mathematics and Theoretical Computer Science, Singapore, May 1997. Springer-Verlag. Also appears as TR97-16, Software Verification Research Centre, The University of Queensland, QLD 4072, Australia.
- [MNU97b] Andrew Martin, Ray Nickson, and Mark Utting. Improving Angel’s parallel operator: Gumtree’s approach. Technical Report 97-15, Software Verification Research Centre, The University of Queensland, QLD 4072, Australia, 1997.
- [Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings Fourth Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, Washington, D.C., 1989.
- [Mor94] Carroll Morgan. *Programming from Specifications*. Series in Computer Science. Prentice-Hall International, second edition, 1994.
- [Pau87] Lawrence C. Paulson. *Logic and Computation—Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.
- [Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989. Also University of Cambridge Computer Laboratory Technical Report No. 130.

- [Pau94] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture notes in Computer Science*. Springer Verlag, Berlin; New York, 1994. ‘With contributions by Tobias Nipkow’.
- [Sch84] David A Schmidt. A programming notation for tactical reasoning. In R. E. Shostak IV, editor, *7th International Conference on Automated Deduction*. Springer-Verlag, LNCS Volume 170, 1984.
- [Spi90] Mike Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42, 1990.
- [Wad89] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.
- [Wad92] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, USA, January 1992.

Acknowledgements

The first author is grateful to Mike Spivey and Paul Gardiner for suggesting the connection of the work on *Angel* with the monadic ideas.