Backtracking, Interleaving, and Terminating Monad Transformers

(Functional Pearl)

Oleg Kiselyov
FNMOC
oleg@pobox.com

Chung-chieh Shan
Harvard University
ccshan@post.harvard.edu

Daniel P. Friedman
Indiana University
dfried@indiana.edu

Amr Sabry
Indiana University
sabry@indiana.edu

Abstract

We design and implement a library for adding backtracking computations to any Haskell monad. Inspired by logic programming, our library provides, in addition to the operations required by the *MonadPlus* interface, constructs for fair disjunctions, fair conjunctions, conditionals, pruning, and an expressive top-level interface. Implementing these additional constructs is easy in models of backtracking based on streams, but not known to be possible in continuation-based models. We show that all these additional constructs can be *generically* and monadically realized using a single primitive *msplit*. We present two implementations of the library: one using success and failure continuations; and the other using control operators for manipulating delimited continuations.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.6 [*Programming Techniques*]: Logic Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Control primitives

General Terms Languages

Keywords continuations, control delimiters, Haskell, logic programming, Prolog, streams.

1. Introduction

One of the benefits of monadic programming is that it generalises over all computational side effects or notions of computation [16, 17], thus supporting custom evaluation modes like non-determinism and backtracking [29, 30]. Using monads to express non-determinism and backtracking is far from a theoretical curiosity. Haskell's *MonadPlus* type class, which defines a backtracking monad interface, has found many practical applications [20], ranging from those envisioned for McCarthy's *amb* operator [15] and its descendents [25], to transactions [28], pattern combinators [27], and failure handling [21].

In a functional pearl [11], Hinze describes backtracking monad transformers that support non-deterministic choice and a Prolog-like *cut* with delimited extent. Hinze aimed to systematically derive his monad transformers in two ways, yielding a term implementation and a (more efficient) context-passing implementation. The most basic backtracking operations, failure and non-deterministic choice, are indeed systematically derived from their specifications. But when it came to *cut*, creative insight was still needed. Furthermore, the resulting term implementation is no longer based on a free term algebra, and the corresponding context-passing implementation performs pattern-matching on the context. As Hinze notes [11], this context-passing implementation differs from a traditional continuation-passing-style (CPS) implementation that handles continuations abstractly. In other words, the implementation is not directly amenable to a direct-style implementation using control operators.

Most existing backtracking monad transformers, including the ones presented by Hinze, suffer from three deficiencies in practical use: unfairness, confounding negation with pruning, and a limited ability to collect and operate on the final answers of a non-deterministic computation. First, the straightforward depth-first search performed by most implementations of *MonadPlus* is not fair: a non-deterministic choice between two alternatives tries every solution from the first alternative before any solution from the second alternative. When the first alternative offers an infinite number of solutions, the second alternative is never tried, making the search *incomplete*. Indeed, as our examples in Section 3 show, fair backtracking helps more logic programs terminate. Naturally, the importance of fairness has been recognised before (*e.g.*, by Seres and Spivey [23, 26], who also present a simple term implementation based on streams). Our contribution in this regard is to implement fair disjunctions and conjunctions in monad *transformers* and using control operators and continuations.

The second deficiency in many existing backtracking monads is the adoption of Prolog's *cut*, which confounds negation with pruning. Theoretically speaking, each of negation and pruning independently makes logic programming languages more expressive [9, 18]. Pruning also allows an implementation to reclaim stor-

ICFP'05 September 26–28, 2005, Tallinn, Estonia.

age and thus run some logic programs in constant space [18]. But negation does not necessarily imply pruning. In fact, Naish [18] points out that Prolog's *cut* is best understood as a combination of two operators: a logical if-then-else (also known as soft-cut, or negation as failure) and don't-care non-determinism (also known as *once*). Thus we separate the implementation and expressive power of these two operators and eschew the overloaded *cut* in our library.

The third practical deficiency is the often-forgotten top-level interface: how to run and interact with a computation that may return an infinite number of answers? The most common solution is to provide a *stream* that can be consumed or processed at the top-level as desired. But in the case of monad transformers, this solution only works if the base monad is non-strict (such as Haskell's lazy list monad and *LazyST*). In the case where the base monad is strict, the evaluation may diverge by forcing the evaluation of the entire stream, even if we only desire one answer. A less common solution is to explicitly include in the top-level request the maximum number of answers to return. Such an interface is trivial to implement if the backtracking effect is internally realized using streams, but apparently impossible if the effect is internally realized using continuations or control operators. Indeed, no existing system that uses continuations seems to provide such an interface. We however show how to uniformly implement this interface in our model.

To summarise our contributions, we implement a backtracking monad transformer with fair disjunctions, fair conjunctions, soft-cut, *once*, and an expressive top-level interface. In addition to discussing the standard stream-based implementation, we show two technically-challenging implementations, one based on CPS and the other based on a "control channel":

1. The CPS implementation uses a success continuation alongside a failure continuation. It is efficient in two ways:
   (a) It does not pattern-match on these continuations but only invokes them.
   (b) It is the result of CPS-transforming a direct-style implementation that runs deterministic code "at full speed"—that is, without any interpretive overhead—insofar as the base monad to which the monad transformer is applied runs at full speed with mutable state and delimited control.
2. Underscoring this last point, the control-channel implementation uses fully-polymorphic multiprompt delimited continuations [7]. Thus our monad transformer factors through delimited control. This implementation can be extended with more sophisticated search strategies that handle left recursion without tabling, and that avoid pitfalls that make depth-first search incomplete and breadth-first search impractical. We omit such extensions here and refer the interested reader to the Kanren project [10].

We achieve fair disjunctions and conjunctions, negation, pruning, and an expressive top-level interface across these two implementations by requiring of them a single operation beyond the *MonadPlus* interface, called *msplit*. Roughly, *msplit* means to look ahead for one solution. It has the signature:

$$msplit :: (Monad\ m, LogicT\ t, MonadPlus\ (t\ m)) \Rightarrow t\ m\ a \to t\ m\ (Maybe\ (a, t\ m\ a))$$

where m is the underlying monad that provides arbitrary effects, and t is the monad transformer designed and implemented in this paper. Intuitively, *msplit* computes the first solution (if any) and suspends the rest of the computation. Although it seems that *msplit* simply de-constructs a list or stream, it is not so easy to implement when $t\ m\ a$ is not a stream (indeed, not even a recursive type), as is the case for our CPS and control-channel implementations. The *msplit* operation does however let us treat the transformed monad as a stream, even when it is not. In particular, we can observe not just the first solution from a backtracking computation but an arbitrary number of solutions, even using an implementation not based on streams.

This paper is a literate Haskell 98 program, except that we need the commonly implemented extension of rank-2 polymorphism [19] for the control operators of Section 5.2. All the code described in the paper is available at `http://pobox.com/~oleg/ftp/packages/LogicT.tar.gz` under the MIT License.

2. Basic Backtracking Computations: *MonadPlus*

The *MonadPlus* interface provides two primitives, *mzero* and *mplus*, for expressing backtracking computations. The command *mplus* introduces a choice junction, and *mzero* denotes failure:

```
class Monad m ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

The precise set of laws that a *MonadPlus* implementation should satisfy is not agreed upon [1], but there is reasonable agreement on the following laws [11]. (We discuss what kind of equivalence $\simeq$ means in Section 3.4.)

Definition 2.1 (Laws for *MonadPlus*).

$$\begin{array}{lcl} mplus\ a\ mzero & \simeq & a \\ mplus\ mzero\ a & \simeq & a \\ mplus\ a\ (mplus\ b\ c) & \simeq & mplus\ (mplus\ a\ b)\ c \\ mzero \gg\!= k & \simeq & mzero \\ (mplus\ a\ b) \gg\!= k & \simeq & mplus\ (a \gg\!= k)\ (b \gg\!= k) \end{array}$$

The intuition behind these laws is that *mplus* is a disjunction of goals and $\gg\!=$ is a conjunction of goals. The conjunction evaluates the goals from left-to-right and is not symmetric.

Using these operations, we may write some simple examples:

```
t1, t2, t3 :: MonadPlus m ⇒ m Int
t1 = mzero
t2 = return 10 `mplus` return 20 `mplus` return 30
t3 = msum (map return [10, 20, 30])
```

The first example represents a choice that leads to failure. The second and third examples are identical, using a library definition of *msum*: they both represent a computation with three choices, each succeeding with a different integer.

For simple examples like these, the built-in list monad is an adequate implementation of the *MonadPlus* interface. The empty list denotes failure; a singleton list denotes a deterministic computation; and a list with more than one element denotes multiple successful results returned by multiple choices. To run such examples, we can trivially convert the answers generated by the multiple choices into a stream of answers:

```
runList :: [a] → [a]
runList = id
```

Indeed, *runList* t_1 returns the empty list, and *runList* t_2 and *runList* t_3 both return the list $[10, 20, 30]$.

The list monad imposes an interpretive overhead on even deterministic computations, because it constructs and destructs singleton lists over and over again. Moreover, it takes quadratic time to enumerate all the solutions of a program like [11]:

```
( ... (return 1 `mplus` return 2) `mplus` ... return n)
```

Other, more efficient implementations of backtracking have been proposed using the two-continuation model [8] and delimited control operators [5]. We revisit the various implementations of backtracking after we enrich the interface of *MonadPlus* with additional

operators that are considered useful for realistic programming applications.

3. A More Expressive Interface

In this section, we extend the bare-bones *MonadPlus* interface with four combinators, for fair disjunctions, fair conjunctions, conditionals, and pruning. But first, let us generalise the *runList* function to monads other than the list monad.

3.1 Running Computations

To run commands in a backtracking monad, we use a function *runL*, which is discussed in detail in Section 6. For now it suffices to think of *runL* as having the following type:

$$runL :: Maybe\ Int \to L\ a \to [a]$$

where L is the backtracking monad in question. When the first argument to *runL* is *Nothing*, all answers are produced. But when the first argument to *runL* is *Just n*, at most *n* answers are produced.

3.2 Interleaving (Fair Disjunction)

Many realistic logic programs make a potentially infinite number of non-deterministic choices. For example, the computation:

$$odds :: MonadPlus\ m \Rightarrow m\ Int$$
$$odds = (return\ 1)\ `mplus`\ (odds \gg\!= \lambda a \to return\ (2 + a))$$

succeeds an infinite number of times. Running *runL* (*Just* 5) *odds* produces the list $[1, 3, 5, 7, 9]$.

Computations that succeed an infinite number of times cannot be combined naïvely with other computations. For example, $odds\ `mplus`\ t_3$ never considers t_3 and thus the execution of the program:[1]

$$runL\ (Just\ 1)\ (\mathbf{do}\ x \leftarrow odds\ `mplus`\ t_3 \quad \mathbf{if}\ even\ x\ \mathbf{then}\ return\ x\ \mathbf{else}\ mzero)$$

diverges without ever succeeding. In this case, however, the three answers 10, 20, and 30 that could be returned by t_3 if it were executed are all even. In other words, the entire computation could diverge due to the infinite number of successes with odd numbers generated by *odds*.

More abstractly, in addition to the laws in Definition 2.1, *mplus* satisfies an extra law:

$$m_1\ `mplus`\ m \quad \simeq \quad m_1$$

whenever m_1 is a computation that can backtrack arbitrarily many times. This law is undesirable as it compromises completeness. This undesirable property is a direct consequence of the associativity of *mplus*: it holds up to finite observations in *any* implementation of *MonadPlus* which satisfies the laws in Definition 2.1 (including the specific *List* monad).

It would thus be useful to have a new primitive *interleave* such that:

$$runL\ (Just\ 10)\ (odds\ `interleave`\ t_3)$$

would produce $[1, 10, 3, 20, 5, 30, 7, 9, 11, 13]$. This would allow:

$$runL\ (Just\ 1)\ (\mathbf{do}\ x \leftarrow odds\ `interleave`\ t_3 \quad \mathbf{if}\ even\ x\ \mathbf{then}\ return\ x\ \mathbf{else}\ mzero)$$

to succeed with the answer 10.

[1] For the code examples in this section, it is tempting to write ... $ **do** ..., but that would not work with our control-channel implementation of backtracking in Section 5.2, because the type of the computation passed to that *runL* must be polymorphic, which the "predicative" rank-2 polymorphism in Haskell [19] does not allow in an argument to $.

3.3 Fair Conjunction

The distributivity law from Definition 2.1 states that:

$$(mplus\ a\ b) \gg\!= k \quad \simeq \quad mplus\ (a \gg\!= k)\ (b \gg\!= k)$$

If $a \gg\!= k$ is a computation that can backtrack arbitrarily many times, then *mplus* never considers $b \gg\!= k$, which means that in this case the following two expressions become equivalent:

$$(mplus\ a\ b) \gg\!= k \quad \simeq \quad a \gg\!= k$$

Thus the unfairness of disjunction (*mplus*) causes the unfairness of conjunction ($\gg\!=$). For example, the program:

$$\mathbf{let}\ oddsPlus\ n = odds \gg\!= \lambda a \to return\ (a + n)$$
$$\mathbf{in}\ runL\ (Just\ 1)\ (\mathbf{do}\ x \leftarrow (return\ 0\ `mplus`\ return\ 1) \gg\!= oddsPlus \quad \mathbf{if}\ even\ x\ \mathbf{then}\ return\ x\ \mathbf{else}\ mzero)$$

diverges, even though there exists an infinite number of answers from $return\ 1 \gg\!= oddsPlus$. Therefore, in addition to a fair *mplus* we need a fair $\gg\!=$, which we denote with $\gg\!-$. Using such a combinator, the program:

$$\mathbf{let}\ oddsPlus\ n = odds \gg\!= \lambda a \to return\ (a + n)$$
$$\mathbf{in}\ runL\ (Just\ 1)\ (\mathbf{do}\ x \leftarrow (return\ 0\ `mplus`\ return\ 1) \gg\!- oddsPlus \quad \mathbf{if}\ even\ x\ \mathbf{then}\ return\ x\ \mathbf{else}\ mzero)$$

succeeds with the answer 2.

3.4 Laws of *interleave* and $\gg\!-$

By design, *interleave* and $\gg\!-$ are fair analogues of *mplus* and $\gg\!=$. In order to state the analogues for the laws in Definition 2.1, it is helpful to realize that every non-deterministic computation can be represented as either *mzero* or $return\ a\ `mplus`\ mr$ for some a and mr.

Definition 3.1 (Laws for Fair *MonadPlus*).

$$\begin{array}{lcl} interleave\ mzero\ m & \simeq & m \\ interleave\ (return\ a\ `mplus`\ m_1)\ m_2 & \simeq & \\ \quad return\ a\ `mplus`\ (interleave\ m_2\ m_1) & & \\ mzero \gg\!- k & \simeq & mzero \\ (mplus\ (return\ a)\ m) \gg\!- k & \simeq & \\ \quad interleave\ (k\ a)\ (m \gg\!- k) & & \end{array}$$

There are no explicit laws for computations of the form:

$$interleave\ m\ mzero$$

but we can reason about such computations as follows. Either m is *mzero* and hence the entire expression is equivalent to *mzero*, or m can be represented as $return\ a\ `mplus`\ mr$ and then:

$$\begin{array}{lcl} interleave\ m\ mzero & \simeq & \\ \quad interleave\ (return\ a\ `mplus`\ mr)\ mzero & \simeq & \\ \quad return\ a\ `mplus`\ (interleave\ mzero\ mr) & \simeq & \\ \quad return\ a\ `mplus`\ mr & \simeq & m \end{array}$$

The main use of *interleave* and $\gg\!-$ is in avoiding divergence when composing computations with a possibly infinite number of answers. In finitary cases, *interleave* and $\gg\!-$ are observationally equivalent to *mplus* and $\gg\!=$ if the notion of observation does not include the *order* of elements in the final list of answers. More precisely, Hinze [11] interprets an equivalence $a \simeq b$ between monadic computations a and b to mean that $runL\ Nothing\ a$ and $runL\ Nothing\ b$ produce identical streams of answers (where the order of the answers is significant). In this paper, we always interpret the equivalence the same way. This is important when we

later consider non-deterministic computations layered over arbitrary monadic computations. In such cases the order of the non-deterministic answers must indeed be assumed to be observable.

We should however mention in passing a different approach, which asserts that the order of answers should not be part of the observational semantics of non-deterministic computations. In that case, *runL Nothing a* and *runL Nothing b* need only return the same *multiset* of answers. Using this—weaker, less deterministic and more liberal notion of equivalence based on multisets—the laws of Definition 3.1 become an instance of the simpler laws of Definition 2.1.

3.5 Soft cut (Conditional)

In Haskell, one can use ordinary conditionals within a sequence of commands. While these constructs are quite useful, a *logical* conditional is still wanting. The conventional conditional constructs can easily express the situation when one computation depends on the success of another. For example, the non-deterministic computation *odds*, which produces an odd number, can be "restricted" to only succeed when the odd number is divisible by another number:

```
iota n   = msum (map return [1..n])

test_oc = runL (Just 10)
               (do n ← odds
                   guard (n > 1)
                   d ← iota (n − 1)
                   guard (d > 1 ∧ n `mod` d ≡ 0)
                   return n)
```

The result is $[9, 15, 15, 21, 21, 25, 27, 27, 33, 33]$. We use *guard* from the standard *Monad* library to filter out only those numbers generated by *odds* that are evenly divisible by some number *d* between 1 and *n* exclusive. (The presence of duplicates in the result will be discussed in the next section.)

The existing constructs are of no help however if we want to restrict the computation *odds* by filtering those odd numbers that are *not* divisible by any *d* in the given range, *i.e.*, to produce odd prime numbers. For this case, we need the common paradigm in logic programming of "negation as finite failure", which performs a logical computation when some other computation fails.

What is needed in this case is a special logical conditional operator that we call *ifte*. The operation *ifte t th el* should evaluate as follows. First the computation *t* is executed. If it succeeds with at least one result, the entire *ifte* computation is equivalent to $t \gg\!= th$. Otherwise, the entire computation becomes equivalent to *el*.

The construct *ifte* is equivalent to Prolog's *soft-cut* (*->) and Mercury's if-then-else construct. A similar construct has been proposed for a Haskell logic monad [3]. The behaviour of this construct is given by the following laws.

Definition 3.2 (Laws for *ifte*).

```
ifte (return a) th el              ≃  th a
ifte mzero th el                   ≃  el
ifte (return a `mplus` m) th el    ≃  th a `mplus` (m >>= th)
```

The first two equivalences formalise the basic intuition of the construct. The third equivalence is more interesting: as soon as the test command succeeds once, the *th* branch is immediately executed and the *el* branch can never be tried. Thus:

```
ifte (m1 `mplus` m2) th el   ≠
  (ifte m1 th el) `mplus` (ifte m2 th el)
```

In other words, the context *ifte* [ ] *th el* interacts in an unusual way with *mplus*. Because the *el* branch is only attempted when the test fails on the initial (rather than on a backtracking) try, *ifte* is particularly useful [4] for "explaining failure." For example, the *el* branch may contain a computation that records (e.g., prints) the fact and circumstances of failure of that particular test, and thus helps avoid uninformative, silent failures. Another common application of soft-cut, mentioned by Andrew Bromage [4], is committing to a heuristic (expressed as the test of *ifte*) if the heuristics applies. Section 7 illustrates with such an application.

Example 3.1. *With ifte we can now modify the example at the beginning of the section to generate the odd prime numbers:*

```
test_op = runL (Just 10)
               (do n ← odds
                   guard (n > 1)
                   ifte (do d ← iota (n − 1)
                            guard (d > 1 ∧ n `mod` d ≡ 0))
                        (const mzero)
                        (return n))
```

The result is $[3, 5, 7, 11, 13, 17, 19, 23, 29, 31]$.

3.6 Pruning (Once)

The operator *ifte* is in some sense a pruning primitive. Another important pruning primitive is *once*, which selects, generally non-deterministically, one solution out of possibly many. The operator *once* greatly improves efficiency as it can be used to avoid useless backtracking and therefore to dispose of data structures that hold information needed for backtracking (*e.g.*, choice points). The *once* primitive is also important for expressiveness, as it expresses "don't care non-determinism." For example, without it, non-deterministic polynomial-time Datalog queries are inexpressible [9].

Naish [18] suggests a simple example that motivates the use of *once*. The example is based on the following code, which shows how to sort a list by generating all permutations and testing them:

```
bogosort l          = do p ← permute l
                         if sorted p then return p else mzero

sorted (e1 : e2 : r) = e1 ⩽ e2 ∧ sorted (e2 : r)
sorted _             = True

permute [ ]          = return [ ]
permute (h : t)      = do {t' ← permute t; insert h t'}

insert e [ ]         = return [e]
insert e l@(h : t)   = return (e : l) `mplus`
                       do {t' ← insert e t; return (h : t')}
```

Despite being a bit contrived, this example is characteristic of logic programming: generate candidate solutions and then test them. The function *bogosort* can have more than one answer in case the list to sort has duplicates. For example:

```
runL Nothing (bogosort [5, 0, 3, 4, 0, 1])
```

produces two answers that differ in the order of the first two elements: $[[0, 0, 1, 3, 4, 5], [0, 0, 1, 3, 4, 5]]$. Clearly this order is not observable, and we only need any one of the answers, which we can express by changing the definition of *bogosort* to be:

```
bogosort' l = once (do p ← permute l
                       if sorted p then return p else mzero)
```

The change does not constrain the use of *bogosort* in a larger program which itself uses backtracking. It is just that *bogosort'* avoids backtracking during the sorting itself because it is useless and wasteful.

In more general situations, different solutions may not be equivalent, and yet we may be satisfied with any of them for the purposes of a particular application. For example, in model checking we are

usually satisfied with the first counterexample. As another example, our *test_op* in Example 3.1, which computes odd prime numbers, calculates all the factors of a composite number, which is wasteful for primality testing. If any factor is found, the number is not prime, and there is no need to look for more factorisations. In other words, *test_op* could be modified as follows:

```
test_op′ =
  runL (Just 10)
       (do n ← odds
           guard (n > 1)
           ifte (once (do d ← iota (n − 1)
                          guard (d > 1 ∧ n `mod` d ≡ 0)))
                (const mzero)
                (return n))
```

The use of *ifte* and *once* implements "negation as failure." As Andrew Bromage explains [4], this pattern can be abstracted into the following construct:

```
gnot  :: (Monad m, LogicT t, MonadPlus (t m)) ⇒
         t m a → t m ()
gnot m = ifte (once m) (const mzero) (return ())
```

Clearly, if *m* succeeds then *gnot m* fails, and if *m* fails then *gnot m* succeeds. Moreover, after the first time *gnot m* fails, there is no reason to backtrack into *m*; any more results it might produce will just be ignored.

4. Splitting Computations

Remarkably, the additional primitives in the more expressive interface presented in the previous section can all be implemented using one basic new abstraction *msplit*. We begin by formalising the extended interface as a Haskell type class that can be instantiated with one method: *msplit*. We give all the remaining operators default definitions.

4.1 The Monad *LogicM*

The class *LogicM* in Figure 1 formalises the interface discussed in the previous section. It includes the functions used there: *interleave*, ≫−, *ifte*, and *once*, with default implementations using the function *msplit*. Conspicuously absent from the *LogicM* class is a function *runL*. It turns out that it can also be easily expressed using *msplit*. We discuss that implementation for a more general case of a monad transformer *LogicT* in Section 6.

Intuitively *msplit* takes a computation and determines if that computation fails or succeeds at least once. Operationally, we think of *msplit* as running its input computation looking for the first successful choice, providing a sort of "lookahead" of size one. The behaviour of *msplit* can be formalised using the following two laws.

Definition 4.1 (Laws for *msplit*).

```
msplit mzero                  ≃  return Nothing
msplit (return a `mplus` m)   ≃  return (Just (a, m))
```

The first law formalises that a computation that fails cannot be split. The second law states that a computation that succeeds at least once can be split into the first result and the rest of the computation.

Using the default implementations in *LogicM* we can verify the axioms for our primitives assuming *msplit* satisfies its axioms. For example, we can verify:

```
    ifte (return a) th el
≃   do r ← msplit (return a)
       case r of
         Nothing → el
         Just (sg1, sg2) → (th sg1) `mplus` (sg2 >>= th)
```

```
class MonadPlus m ⇒ LogicM m where
  msplit            :: m a → m (Maybe (a, m a))

  interleave        :: m a → m a → m a
  interleave sg1 sg2 =
    do r ← msplit sg1
       case r of
         Nothing → sg2
         Just (sg11, sg12) →
           (return sg11) `mplus` (interleave sg2 sg12)

  (≫−)              :: m a → (a → m b) → m b
  sg ≫− g           =
    do r ← msplit sg
       case r of
         Nothing → mzero
         Just (sg1, sg2) → interleave (g sg1) (sg2 ≫− g)

  ifte              :: m a → (a → m b) → m b → m b
  ifte t th el      =
    do r ← msplit t
       case r of
         Nothing → el
         Just (sg1, sg2) → (th sg1) `mplus` (sg2 >>= th)

  once              :: m a → m a
  once m            =
    do r ← msplit m
       case r of
         Nothing → mzero
         Just (sg1, _) → return sg1
```

Figure 1. The class *LogicM*

```
≃   do r ← msplit (return a `mplus` mzero)
       case r of
         Nothing → el
         Just (sg1, sg2) → (th sg1) `mplus` (sg2 >>= th)
≃   do r ← return (Just (a, mzero))
       case r of
         Nothing → el
         Just (sg1, sg2) → (th sg1) `mplus` (sg2 >>= th)
≃   case Just (a, mzero) of
      Nothing → el
      Just (sg1, sg2) → (th sg1) `mplus` (sg2 >>= th)
≃   (th a) `mplus` (mzero >>= th)
≃   th a
```

4.2 Implementing *msplit* Using Lists

The main technical challenge addressed in the paper is in implementing *msplit* in monads that use continuations. The implementation of *msplit* in the case of the list monad is straightforward and provides some helpful intuition.

```
newtype SSG a = Stream [a]
unSSG (Stream str) = str

instance Monad SSG where
  return e          = Stream [e]
  (Stream es) >>= f = Stream (concat (map (unSSG ∘ f) es))

instance MonadPlus SSG where
  mzero                          = Stream [ ]
  (Stream es1) `mplus` (Stream es2) = Stream (es1 ++ es2)
```

```
class MonadTrans t ⇒ LogicT t where
  msplit      :: (Monad m, MonadPlus (t m)) ⇒
                 t m a → t m (Maybe (a, t m a))
  interleave  :: (Monad m, MonadPlus (t m)) ⇒
                 t m a → t m a → t m a
  (≫−)        :: (Monad m, MonadPlus (t m)) ⇒
                 t m a → (a → t m b) → t m b
  ifte        :: (Monad m, MonadPlus (t m)) ⇒
                 t m a → (a → t m b) → t m b → t m b
  once        :: (Monad m, MonadPlus (t m)) ⇒
                 t m a → t m a
```

Figure 2. The class *LogicT*

```
instance LogicM SSG where
  msplit (Stream [ ])     = return Nothing
  msplit (Stream (h : t)) = return (Just (h, Stream t))
```

The implementation is essentially the *List* monad that is already present in Haskell. As argued earlier, the reliance on the list monad has several drawbacks, many of which are discussed by Hinze [11]. In addition, the above implementation cannot be easily modified to become a monad transformer since *List* as a transformer can only be applied to commutative monads [12].

4.3 The Monad Transformer *LogicT*

Instead of working with the fixed monad *LogicM*, we would like to uniformly add *msplit* to other monads, thus augmenting arbitrary computations with our backtracking facilities.

In Haskell, this can be achieved by using *monad transformers*. A monad transformer *t* is defined using a class:

```
class MonadTrans t where
  lift :: Monad m ⇒ m a → t m a
```

Intuitively computations in a base monad *m* are lifted to computations in a transformed monad *t m*. The lifting satisfies the following laws:

$$\begin{array}{lcl} lift \circ return & \simeq & return \\ lift\ (m \gg\!= k) & \simeq & lift\ m \gg\!= lift \circ k \end{array}$$

The specification of *LogicM* can be turned into a monad transformer by simply copying the functions and giving them the required generalised types. Indeed the interface to the class *LogicT* in Figure 2 is essentially identical to the one of *LogicM*; for brevity we have not repeated the default implementations of the methods. The precise relationship between the two figures is that the class *LogicM* can be recovered by applying the monad transformer *LogicT* to the identity monad.

By inheriting from the library class *MonadTrans*, the class *LogicT* also includes the method *lift* that injects computations of the underlying monad of type *m a* into backtracking computations of type *t m a*. For example, *lift* (*putStrLn* `"text"`) ≫ *mzero* is a backtracking computation which performs a side-effect in the *IO* monad and then fails.

The laws of *msplit* postulated in Definition 4.1 should be generalised to handle the additional "lifted" effects from the underlying monad [14]. In the first law, instead of considering a computation *mzero*, we should consider more generally a computation *lift m* ≫ *mzero* which might perform computational effects in the underlying monad before failing. Similarly in the second law, instead of considering a computation *return a* ‘*mplus*‘ tm_1, we should consider more generally a computation *lift m* ‘*mplus*‘ tm_1 which returns the first result after performing some arbitrary effects in the underlying monad. These generalisations are minimal and only describe the morphisms that lift trivially. In some cases, a morphism *m* in the underlying monad may not be trivially lifted as *lift m* but rather as a new morphism *tm* that combines the backtracking effects and underlying effects in non-trivial ways. This is similar to what happens when lifting *callcc* through the state monad transformer for example [14].

We motivate the new form of the laws by considering the generalised left-hand side of the second law: *msplit* (*lift m* ‘*mplus*‘ tm_1). The computation tm_1 has type *t m a* whereas the result of *msplit* has the type *t m* (*Maybe* (*a*, *t m a*)), which makes relating these values inconvenient. Therefore, we define:

```
reflect :: (Monad m, LogicT t, MonadPlus (t m)) ⇒
           Maybe (a, t m a) → t m a
reflect r = case r of
              Nothing → mzero
              Just (a, tmr) → return a ‘mplus‘ tmr

rr tm   = msplit tm >>= reflect
```

Now, *rr tm* has the same type as *tm* and hence the two can be more directly related. Direct inspection of the code of *reflect* shows that it introduces no effects at the source monad level and it does not affect the values of the monadic computation—the latter fact is obvious from the type, which is polymorphic over any *a* and any source monad *m*. We can now state the generalised laws.

Definition 4.2 (Generalised Laws for *msplit*).

$$\begin{array}{lcl} rr\ (lift\ m \gg mzero) & \simeq & lift\ m \gg mzero \\ rr\ (lift\ m\ \text{`}mplus\text{`}\ tma) & \simeq & lift\ m\ \text{`}mplus\text{`}\ (rr\ tma) \end{array}$$

5. Implementations of *LogicT*

We now focus our attention on the main technical challenge of the paper: the implementation of the *LogicT* monad transformer. We provide two implementations that manipulate continuations, either explicitly or implicitly using control operators.

5.1 CPS Implementation

The CPS-based implementation introduces the type constructor *SFKT* for functions accepting success and failure continuations. The answer type is fully polymorphic:

```
newtype SFKT m a =
  SFKT (∀ ans. SK (m ans) a → FK (m ans) → m ans)
unSFKT (SFKT a) = a

type FK ans     = ans
type SK ans a   = a → FK ans → ans
```

The concrete type of monadic actions is *SFKT m a*, where *m* is the source monad to be transformed.

The following instance declarations specify that *SFKT m* is a *Monad* and *MonadPlus*, and that *SFKT* is a monad transformer. These instance declarations are quite straightforward, and match the implementation of the monad transformer (without *cut*) given by Hinze [11].

```
instance Monad m ⇒ Monad (SFKT m) where
  return e = SFKT (λsk fk → sk e fk)
  m >>= f =
    SFKT (λsk → unSFKT m (λa → unSFKT (f a) sk))

instance Monad m ⇒ MonadPlus (SFKT m) where
  mzero     = SFKT (λ_ fk → fk)
  m1 ‘mplus‘ m2 =
    SFKT (λsk fk → unSFKT m1 sk (unSFKT m2 sk fk))
```

```
instance MonadTrans SFKT where
  lift m = SFKT (λsk fk → m >>= (λa → sk a fk))
```

As Hinze explains, this "context-passing" implementation improves on the naïve term implementation by removing an interpretive layer. But in order to augment the above implementation with control over backtracking (*e.g.*, *cut*), Hinze changes the representation of contexts in order to pattern-match against them, restoring an interpretive layer. We show however that this is not necessary: we can maintain the abstract representation of continuations and support *msplit*. But indeed, contrary to the situation in Section 4.2 where we implement *msplit* using lists, the implementation of *msplit* for the two-continuation monad transformer *SFKT* is more challenging:

```
instance LogicT SFKT where
  msplit tma = lift (unSFKT tma ssk (return Nothing))
    where ssk a fk = return (Just (a, (lift fk >>= reflect)))
```

Intuitively, to split a computation *tma*, we supply it with two custom success and failure continuations. If the failure continuation is immediately invoked, we get *lift* (*return Nothing*) which is essentially another way of expressing *mzero* in the transformed monad. If we encounter several non-deterministic choices and the success continuation is invoked in one of the cases with an answer *a*, we return this answer *a* and a suspension that can be used to continue the exploration of the other choices. This is reminiscent of the list implementation but works even if *t m a* for arbitrary *m* is generally not a recursive data type.

Since the correctness of *msplit* is not so obvious, we outline a proof in the remainder of the section. The first law of *msplit*:

$$rr\ (lift\ m \gg mzero) \quad \simeq \quad lift\ m \gg mzero$$

follows from the monadic laws, the definition of *reflect* and the observation:

$$msplit\ (lift\ m \gg mzero) \quad \simeq \quad lift\ m \gg (return\ Nothing),$$

which can be derived from the code of the above instance declarations. To prove the second law of *msplit*:

$$rr\ (lift\ m\ `mplus`\ tma) \quad \simeq \quad lift\ m\ `mplus`\ (rr\ tma)$$

we observe that:

$$\begin{array}{l} lift\ m\ `mplus`\ tma \quad \simeq \\ \quad SFKT\ (\lambda sk\ fk \to (\lambda sk\ fk \to m \gg= (\lambda a \to sk\ a\ fk)) \\ \qquad\qquad sk \\ \qquad\qquad (unSFKT\ tma\ sk\ fk)) \quad \simeq \\ \quad SFKT\ (\lambda sk\ fk \to m \gg= (\lambda a \to sk\ a\ (unSFKT\ tma\ sk\ fk))) \end{array}$$

Thus we have:

$$\begin{array}{l} msplit\ (lift\ m\ `mplus`\ tma) \quad \simeq \\ \quad lift\ m \gg= (\lambda a \to return\ (Just\ (a, rr\ tma))) \end{array}$$

The desired result follows from the definition of *rr* and monadic laws in the *m a* and *t m a* monads.

Furthermore, if we define $g\ f\ v\ tm = unSFKT\ (rr\ tm)\ f\ v$ we can easily obtain, from the definition of *rr*, that:

$$\begin{array}{lcl} g\ f\ v\ mzero & \simeq & v \\ g\ f\ v\ (lift\ m\ `mplus`\ tm_1) & \simeq & m \gg= (\lambda a \to f\ a\ (g\ f\ v\ tm_1)) \end{array}$$

that is, *g* is essentially *fold*, which clarifies the meaning of the function *rr*.

5.2 Implementation Using Delimited Control

The implementation based on control operators uses an extension of the fully answer-type-polymorphic delimited continuation framework developed by Dybvig *et. al.* [7]. The framework provides two type constructors of interest: *CC* for computations that manipulate delimited continuations and *Prompt* for control delimiters. We first extend the implementation to make *CC* a monad transformer and, using this extension, define the following small library of control operators:

```
promptP :: Monad m ⇒
           (Prompt r a → CC r m a) → CC r m a

abortP  :: Monad m ⇒
           Prompt r a → CC r m a → CC r m b

shiftP  :: Monad m ⇒
           Prompt r b →
           ((CC r m a → CC r m b) → CC r m b) →
           CC r m a
```

The constructors *Prompt* and *CC* are parametrized by a type parameter *r* that refers to the *control region* associated with the computation. Intuitively a delimiter of type *Prompt r a* was created in a control region indexed by type *r* and its uses cannot escape from that control region. The type parameter *r* allows computations in the monad to be *encapsulated* using a construct of the following type:

```
runCC :: Monad m ⇒ (∀ r. CC r m a) → m a
```

The control operators we implement above have the following intuitive explanation. The operator *promptP* creates a new delimiter, pushes it on the stack, and invokes its argument with access to the newly-pushed delimiter. The execution of the argument can later abort up to this occurrence of the prompt using *abortP*, or capture the continuation up to this occurrence of the prompt using *shiftP*. For example, in the following expression:

```
runIdentity (runCC (promptP $ λp →
                      do x ← shiftP p (λk → k (k (return 2)))
                         return (x + 1)))
```

the evaluation proceeds by first creating a new prompt and pushing it on the stack. The evaluation of the **do**-expression then pushes the context **do** $\{x \leftarrow [\,]; return\ (x + 1)\}$ on the stack before evaluating the *shiftP* expression. This expression captures the continuation up to the prompt *p*, reifies it as a function, and applies it twice to the argument 2.

This instance of *LogicT* uses the *CC* library to define the type constructor *SR* as follows:

```
newtype SR r m a =
  SR (∀ ans. ReaderT (Prompt r (Tree r m ans)) (CC r m) a)
unSR (SR f) = f

data Tree r m a = HZero
                | HOne a
                | HChoice a (CC r m (Tree r m a))
```

At the core of the definition of the type *SR r m a* is a computation of type *CC r m a* that manipulates continuations using control operators instead of having an explicit success and failure continuation as in the previous section. The computation executes in the context of an environment implemented using *ReaderT*. The environment holds the most recently-pushed control delimiter, to which the computation should abort if it fails. Computations in the *ReaderT* monad transformer are executed using the library function *runReaderT*, and access the environment using the library function *ask*, which in our case has the type:

```
ReaderT (Prompt r (Tree r m ans)) (CC r m)
        (Prompt r (Tree r m ans)).
```

The *SR r m* monad is essentially the direct-style version of the two-continuation implementation in the previous section. Previously, non-determinism was realized with the help of success and

failure continuations. The success continuation receives not only the produced value but also a (failure) continuation to invoke if that value was "not good enough" (failure when processing that value). Here, the success continuation is represented by an implicit stack of pending computations with control delimiters marking the choice junctions. A non-deterministic choice is then represented by capturing the delimited continuation up to the closest delimiter and trying each of the branches using that continuation; failure is represented by aborting the current delimited continuation.

In the two-continuation model, there was no special way to represent deterministic computations, which produce exactly one value. In the current model, we find it useful to distinguish deterministic results from non-deterministic results by using the data type *Tree r m a*. A deterministic result is marked by tagging it with *HOne*. A failure is represented by *HZero* and a choice is represented by *HChoice a r*. The latter value describes both the result *a* of the performed computation and the not-yet-executed computation that represents the other part of the choice. It is possible to represent deterministic computations as *HChoice a* (*return HZero*) at the expense of obscuring the definitions and losing some performance.

The following definition shows that the type *SR r m* is an instance of *Monad*:

```
instance Monad m ⇒ Monad (SR r m) where
  return e       = SR (return e)
  (SR m) >>= f = SR (m >>= (unSR ∘ f))
```

The definition shows that deterministic computations are executed "normally"—that is, as if they were in the base monad *m*. And since *SR* is a **newtype**, tagging with *SR* and untagging with *unSR* do not take any run time.

We then show that *SR r m* is an instance of *MonadPlus*:

```
instance Monad m ⇒ MonadPlus (SR r m) where
  mzero =
    SR (ask >>= λpr → lift (abortP pr (return HZero)))
  m1 `mplus` m2 =
    SR (ask >>= λpr →
        lift $ shiftP pr $ λsk →
        do f1 ← sk (runReaderT (unSR m1) pr)
           let f2 = sk (runReaderT (unSR m2) pr)
           compose_trees f1 f2)

compose_trees :: Monad m ⇒
                 Tree r m a →
                 CC r m (Tree r m a) →
                 CC r m (Tree r m a)
compose_trees HZero r          = r
compose_trees (HOne a) r       = return $ HChoice a r
compose_trees (HChoice a r') r =
  return $ HChoice a $ r' >>= (λv → compose_trees v r)
```

A failing computation *mzero* simply aborts to the current delimiter with the result *HZero*. A non-deterministic choice is slightly more complicated: we capture the continuation *sk* up to the current delimiter. The first alternative m_1 is immediately executed in that continuation; the second alternative m_2 is suspended. The function *compose_trees* builds a new *Tree* combining the results obtained from the first branch with the suspension from the second branch.

Before discussing the implementation of *msplit* we discuss a necessary function used to implement *msplit*. This function *reify* represents the computational effect of backtracking in an algebraic data type as described in two recent papers [24, 13]:

```
reify   :: Monad m ⇒ SR r m a → CC r m (Tree r m a)
reify m = promptP (λpr → runReaderT (unSR m) pr >>=
                         (return ∘ HOne))
```

As the code shows, the function *reify m* creates a new prompt, sets it, executes the computation, and tags the result with *HOne*. If the computation *m* executes deterministically, we get the result *HOne a* where *a* is the resulting value. If the computation *m* fails, then the continuation (*return* ∘ *HOne*) is aborted and we instead get *HZero* as the resulting value. Finally, let us illustrate the case where the computation *m* is about to execute *mplus* m_1 m_2 using the following example:

```
reify (m0 >>= (λx → k1 x `mplus` k2 x) >>= k3)
```

where m_0 is deterministic. This example is equivalent to:

```
do HOne x ← reify m0
   f1val ← reify (k1 x >>= k3)
   let f2comp = reify (k2 x >>= k3)
   compose_trees f1val f2comp
```

Here *f1val* is the result (reified into the *Tree* data type) of the first choice of *mplus*, and *f2comp* is the *computation* that corresponds to the second choice. If *f1val* is *HZero*—that is, the first choice eventually fails (either in *k1* or in *k3*), then *compose_trees* will run the computation *f2comp* and the (reified) result of the latter shall be the result of the original computation. In other words, if the first choice fails, we execute the other one. If the first choice finishes deterministically, *i.e.*, if *f1val* is *HOne a*, then *compose_trees* represents the available choice by creating a result *HChoice a f2comp*, which suspends the computation *f2comp* to be later explored if needed as a possible alternative solution. Finally, if *f1val* is itself the choice *Choice a r′*—that is, the execution *k1 x >>= k3* has (unexplored) alternative branches (represented by *r′*), the function *compose_trees* essentially composes the unexplored choices *r′* with the unexplored choice *f2comp*. One may think of that "composition" as a rotation of a binary decision tree, or joining a branch *f2comp* to a binary node (a, r').

It helps to contrast our way of handling non-determinism with the regular Prolog approach, epitomized in the Warren Abstract Machine (WAM). The WAM includes a stack, which contains both environments and choice points [22]. The stack of the WAM is represented by the regular execution stack of the Haskell system. In the above example, *f2comp* (or more generally the computation *sk* (*runReaderT* (*unSR* m_2) *pr*) in the implementation of *mplus*) represents the choice point. The function *compose_trees* essentially handles the "top-most" choice point (the *CP* register of the WAM).

Our function *compose_trees* opens up more flexible policies of handling the choice points. For example, once *compose_trees* determines that *f1val* is *HZero* (that is, represents failure), it may not run *f2comp*. Rather, it may tag *f2comp* with a special tag like *Incomplete* and return that. When one *mplus* is nested in another, the outer instances of *compose_trees*, having received the *Incomplete f2comp*, may then decide, either to run that *f2comp*, or to try other choices, if any. Thus we can implement alternative evaluation policies that avoid divergence in situations like (*odds* ≫ *mzero*) `mplus` *m*, that is, when disjoining (on the left) a computation that diverges on backtracking.

Finally we implement *msplit* for the monad transformer *SR r*:

```
instance MonadTrans (SR r) where
  lift m = SR (lift (lift m))

instance LogicT (SR r) where
  msplit m = SR (lift (reify m >>= (return ∘ reflect_sr)))
    where reflect_sr HZero          = Nothing
          reflect_sr (HOne a)       = Just (a, mzero)
          reflect_sr (HChoice a r1) =
            Just (a, SR (lift r1) >>=
                     (return ∘ reflect_sr) >>=
                     reflect)
```

This implementation of *msplit* is more complicated than the CPS-based one because it must maintain the proper polymorphism of the answer type by not letting the type variable *ans* escape. Given the computation *m*, we first *reify* it. This will tell us if the computation fails, completes deterministically, or completes with one answer and the choice of an alternative solution. This is precisely the information that we need to know for *msplit*. The problem with the *msplit* implementation is that r_1 in *HChoice a* r_1 has the type *CC r m* (*Tree r m a*) (of the reified computation), whereas *msplit* must produce *SR r m a*. Thus we must be able to go from the reified computation *Tree r m a* "back" to the computation *m a*. The recursion in the implementation of *msplit* accomplishes that goal: of reflecting a value *HChoice a r* back into the computation *return a* ‘*mplus*‘ *r*. The recursion of *reflect_sr* is a consequence of the fact that *Tree r m a* is a recursive data type (although *SR r m a* is generally not, depending on *m*). Note that *msplit* invokes *reify*, which sets the *prompt*, overriding the prompt set by the top-level *reify*. This "dynamic scoping" inherent in delimited control [2] is essential for *msplit*.

In summary, we have implemented the *LogicT* interface to support "direct-style" programming with a rich combination of several computational effects: of the base monad *m*, the computational of the *CC* monad, and of the *Reader* monad. Although the present *SR r m a* implementation may have a large cost because of the layering of several effects and because of the inherent cost of the *CC* monad, it is still worth using, at least for prototyping. Often "direct-style" implementations are clearer and lend themselves to deeper insights. Even though the *SFKT* monad seems more efficient (and we would recommend using it in production, because all its costs are quantifiable and not large), *SR r m a* seems to be better suited for prototyping of various choice-point selection policies and other advanced implementations (suspensions, constraint-propagation, etc.) of logical programming systems.

6. Running Computations

We now implement *runL*, to run the backtracking monad and observe its results as a list of answers. The simplest solution is to define the function *solve* for a particular monad, *e.g.*, *SFKT*:

```
solve                :: (Monad m) ⇒ SFKT m a → m [a]
solve (SFKT m) =
  m (λa fk → fk >>= (return ∘ (a:))) (return [])
```

which is identical to the one provided by Hinze [11]. This function runs the backtracking computation of type *SFKT m a* and collects all the answers in a list (to be observed in the source *m* monad). One may think that this function is sufficient: to observe at most *n* answers, we need to examine the prefix of the resulting list of at most that size. The rest of the answers will not be produced. However, the latter is only true if the source monad *m* is non-strict. If however *m* is strict (*e.g.*, *IO*), it is clear from the definition of *solve* that *all* the answers of *SFKT m a* will be produced and collected into the list, even if we need only a few of them. This also means that applying *solve* to a computation *SFKT IO a* that has an infinite number of answers (such as *odds*) will diverge.

Thus we need a more general function *runL*, to which we can pass the maximum number of answers we wish to observe. That function will run the backtracking computation to the extent needed to observe that many answers, no more. Therefore, *runL* can be safely used with non-deterministic computations with an infinite number of answers over a strict monad.

Implementing *runL* for the *SFKT* monad however appears to be all but impossible. To run a computation, we need to pass it a success and failure continuation. The success continuation receives one answer and a computation *fk* to run to get more answers. We can easily disregard the failure computation after the first answer:

```
observe              :: Monad m ⇒ SFKT m a → m a
observe (SFKT m) =
  m (λa fk → return a) (fail "no answer")
```

Or we can run that *fk* computation after the first answer, as in *solve*, which gives us *all* answers. There does not seem to be a way to run *fk* only a certain number of times, as the interface of *SFKT* does not let us pass any counter from one invocation of the success continuation to the next.

Here again *msplit* helps. It turns out that we can implement *runL*—moreover, we can implement a more general operation *bagofN* and even *unfold*. Furthermore, we can implement *bagofN* in a way that does not depend on the implementation of the backtracking monad transformer. The operation *bagofN* is similar to Prolog's *bagof* iterator. The latter collects all answers of a given goal in a list. Our *bagofN* is more general, as it lets the user specify the maximum number of answers wanted. This more general *bagofN* is not expressible in Prolog using *bagof* or other primitives, without resorting to destructive operations on the Prolog database:

```
bagofN :: (Monad m, LogicT t, MonadPlus (t m)) ⇒
          Maybe Int → t m a → t m [a]
bagofN (Just n) _ | n ⩽ 0 = return []
bagofN n m                = msplit m >>= bagofN′
  where bagofN′ Nothing          = return []
        bagofN′ (Just (a, m′)) =
          bagofN (fmap pred n) m′ >>= (return ∘ (a:))
```

If the first argument to *bagofN* is *Just n*, it selects at most *n* answers. Again, the operation *msplit* let us treat the backtracking monad as if it were a stream, regardless of its actual implementation. The partially-applied *bagofN Nothing* is equivalent to the function *sols* of Hinze [11]. But there is no equivalent there of the more challenging and more expressive *bagofN* (*Just n*).

The result type of *bagofN* is *t m* [*a*]: we are still in the transformed monad. To get back to the source monad *m*, we need to "observe" [11] the produced list value. The observation function is necessarily specific to the backtracking implementation.[2] For the *SFKT* monad, it is given above. For the *SR* monad, it is as follows:

```
observe_sr :: Monad m ⇒ (∀ r. SR r m a) → m a
observe_sr m = runCC (reify m >>= pick1)
  where pick1 HZero          = fail "no answer"
        pick1 (HOne a)       = return a
        pick1 (HChoice a r) = return a
```

Here, we *reify* the computation *m* into *Tree r m a*, then pick the first answer from the *Tree*, disregarding any other choices.

With the help of *observe* we can now write the function *runL* that we have used to run our examples:

```
type L a = ∀ r. SR r Identity a
runL     :: Maybe Int → L a → [a]
runL n m = runIdentity (observe_sr (bagofN n m))
```

We also reveal the type of the backtracking monad *L* that we introduced in 3.1. For our examples, the type is the transformer *SR r* over the identity monad. The examples also run with the *SFKT* transformer.

As an example of using the backtracking transformer over a non-trivial (and strict!) monad, we modify Example 3.1 to print all the factors that are produced:

[2] Ideally, the function *observe* should be part of the *LogicT* class, but this is not possible because of the universally quantified type variable *r* in the last implementation.

```
test_opio = print =<< observe (bagofN (Just 10)
                  (do n ← odds
                      guard (n > 1)
                      ifte (do d ← iota (n − 1)
                               guard (d > 1 ∧ n `mod` d ≡ 0)
                               liftIO (print d))
                           (const mzero)
                           (return n)))
```

The source *IO* monad lets us print out the intermediate results. This approach is far more robust than using *Debug.Trace*, as the output of the latter is hard to predict. The comparison with Example 3.1 demonstrates the advantage of having a monad *transformer*: the bulk of the code of Example 3.1 remains the same. We merely add *liftIO* (*print d*) and the printing of the final result.

7. A Larger Example: Tic Tac Toe

Tic Tac Toe, Reversi and many other strategic boardgames are good examples of heuristic search. The Tic Tac Toe code, suggested by Andrew Bromage on the Haskell mailing list [4], illustrates many features of our monad transformer *LogicT* in conducting basic minimax search coupled with two heuristics. The present code is a generalisation of that by Andrew Bromage: It solves instances of the problem with boards of size $n \times n$ and where m consecutive marks are required for a win (such as Gomoku). We also added explicit limits on the depth and breadth of the search. Without the limits, the program is too slow for interactive play on boards larger than 3×3. The code accompanying the article includes the complete program.

We begin by declaring the basic types for representing the board and the marks:

```
data Mark = X | O deriving (Ord, Eq, Show)
type Loc = (Int, Int)
type Board = FiniteMap Loc Mark
data Game = Game{winner :: Maybe (Loc, Mark),
                 moves  :: [Loc],
                 board  :: Board}
```

The type *Loc* describes (*row*, *column*) coordinates of one board cell, as a pair of integers within $[0\,..\,n-1]$. A finite map *Board* maps the coordinates of marked locations to their marks. We use type *Mark* to identify players as well. The current state of the game is a record of the current board position, the list of available moves (*i.e.*, unmarked cells) and the indicator of the winner.

The function

```
new'game :: Game
```

initialises the board. The function

```
take'move :: Mark → Loc → Game → Game
take'move p loc g =
  Game{ moves  = delete loc (moves g),
        board  = board',
        winner = let (n, l) = max'cluster board' p loc
                 in if n ⩾ m then Just (l, p) else Nothing}
  where board' = addToFM (board g) loc p
```

accounts for a move (*i.e.*, the placement of a mark on a previously empty cell) and generates the new game state. The function *max'cluster* :: *Board* → *Mark* → *Loc* → (*Int*, *Loc*) computes the number of consecutive marks of the same sort around a given location, maximized over all possible directions.

Let us define the player procedure that takes the player's mark, the game state, and, non-deterministically, makes a move and returns the new game state together with an estimate of the game score for the player.

```
type PlayerProc t (m :: ∗ → ∗) =
  Mark → Game → t m (Int, Game)
```

The main game function can then be defined:

```
game :: (MonadPlus (t m), LogicT t,
         Monad m, MonadIO (t m)) ⇒
        (Mark, PlayerProc t m) →
        (Mark, PlayerProc t m) →
        t m ()
game player1 player2 = game' player1 player2 new'game
  where game' player@(p, proc) other'player g
          | Game{winner = Just k} ← g
            = liftIO (putStrLn ((show k) ++ " wins!"))
          | Game{moves = []} ← g
            = liftIO (putStrLn "Draw!")
          | otherwise
            = do (_, g') ← once (proc p g)
                 liftIO (putStrLn $ show'board (board g'))
                 game' other'player player g'
```

The expression *once* (*proc p g*) means that once the player has made the move, the move is committed and cannot be un-played.

Our playing strategy is the basic minimax search. We first check if we reached the terminal, goal state.

```
ai' :: (MonadPlus (t m), Monad m, LogicT t) ⇒
       PlayerProc t m
ai' p g = ai'lim m 6 p g
  where ai'lim dlim blim p g
          | Game{winner = Just _} ← g
            = return (estimate'state p g, g)
          | Game{moves = []} ← g
            = return (estimate'state p g, g)
          | otherwise
            = minmax ai'lim dlim blim p g
```

If not, we pick such a successor state that minimizes the score for our opponent assuming the opponent always makes its best move:

```
minmax :: (MonadPlus (t m), Monad m, LogicT t) ⇒
          (Int → Int → PlayerProc t m) →
          (Int → Int → PlayerProc t m)
minmax self dlim blim p g =
  do wbs ← bagofN (Just blim)
                  (do m ← choose (moves g)
                      let g' = take'move p m g
                      if dlim ⩽ 0
                        then return (estimate'state p g', g')
                        else do (w, _) ← self (dlim − 1) blim
                                              (other'player p) g'
                                return (−w, g'))
     return (maximumBy (λ(x, _) (y, _) → compare x y) wbs)
```

The number *dlim* limits the depth of the search and the number *blim* limits the number of moves considered at each step. The function *choose* non-deterministically chooses one move out of all available, and the function *estimate'state* :: *Mark* → *Game* → *Int* estimates the game score for the given player, as a signed integer in $[-score'win\,..\,score'win]$. The larger the integer, the better the position. We see the application of *bagofN* operation of *LogicT*.

Unfortunately, this code is too slow. Even for such a simple game on a 3×3 board, the search space is noticeably large. Andrew Bromage has pointed out two safe heuristics. They are based on the following function, which determines if there is a move that immediately leads to victory for the player *p*:

```
first'move'wins p g =
  do m ← choose (moves g)
     let g' = take'move p m g
     guard (maybe False (λ(_, p') → p' ≡ p) (winner g'))
     return (m, (score'win, g'))
```

We can change our play function as follows:

```
ai' :: (MonadPlus (t m), Monad m, LogicT t) ⇒
       PlayerProc t m
ai' p g = ai'lim m 6 p g
  where
  ai'lim dlim blim p g
    | Game{winner = Just _} ← g
      = return (estimate'state p g, g)
    | Game{moves = [ ]} ← g
      = return (estimate'state p g, g)
    | otherwise
      = ifte (once (first'move'wins p g))
             (return ∘ snd)
             (ifte (once (first'move'wins (other'player p) g))
                   (λ(m, _) → do let g' = take'move p m g
                                 (w, _) ← ai'lim dlim blim
                                            (other'player p) g'
                                 return (−w, g'))
                   (minmax ai'lim dlim blim p g))
```

If there is a winning move, we take it, without further ado. We are only interested in one such move, hence *once*. If we cannot immediately win but our opponent can on the next move, we block that move. The *ifte* forms signify the commitment to a heuristic once it applies. If, and only if, none applies, we do the minimax search. To let the computer play against itself, we run the following computation:

```
a12a1 :: IO () = observe $ game (X, ai') (O, ai')
```

The complete code also includes a function for a human player, so one can play against the computer. Although currently the choice and scoring functions are simplistic, and the search limits are low, the play is good enough to be entertaining.

8. Related Work

Seres and Spivey [23] explored fair conjunction, but their implementation relied exclusively on streams, instead of being purely monadic with operators ≫− and *interleave*. Our solution not only handles the stream representation but at least two other representations, Federhen's two-continuation model [8] and the "control channel." Wand and Vaillancourt [31] formally related the stream and two-continuation semantics of backtracking, but they did not consider more general monadic streams or splitting of the backtracking computation.

Hinze [11] described backtracking transformers that support non-deterministic choice, and limited-extent Prolog-like *cut*. His final, efficient context-passing implementation was explicitly not continuation-passing because it required pattern-matching on the context. In addition, he ignored the problem of managing termination, which we address with *interleave* and ≫−. Furthermore, we added the ability to select any given number of answers. This is of course easy using streams, but difficult in the two-continuation and "control-channel" solutions; we believe that we are the first to implement these monadically. One weakness of our approach as compared to Hinze's is that he shows how to *derive* the transformers, with the promise of mechanization. That promise is not completely fulfilled however when *cut* is involved. Our approach is akin to the one he contrasts with in his introduction: "Because it works."

CPS-based implementations of Prolog with *cut* were discussed by de Bruin and de Vink [6], who used three continuations, for success, failure, and *cut*. In this paper, we have shown a CPS-based system with negation and Prolog-like pruning that uses only two continuations.

9. Conclusion

We have introduced a backtracking monad transformer, which, in addition to the *MonadPlus* interface, provides fair conjunctions and disjunctions, logical conditional and pruning (don't-care nondeterminism), and selecting an arbitrary number of answers. We have described two implementations of the transformer, a CPS one with two continuations, and a direct-style one based on a Haskell library of delimited continuations [7]. All additional backtracking operations are implemented generically, in terms of one operation *msplit*.

Our *msplit* operation lets us treat the backtracking transformed monad as if it were a stream—even when the monad is not a stream and not even of a recursive type (which is the case for both our implementations). We can therefore observe not just the first solution from a backtracking computation but an arbitrary number of solutions.

In future research, we plan on using our direct-style implementation to implement sophisticated backtracking policies that can handle, for example, left recursion without tabling.

Acknowledgments

We would like to thank the anonymous reviewers for their corrections and suggestions. We also thank Andrew Bromage for helpful suggestions.

References

[1] MonadPlus. `http://www.haskell.org/hawiki/MonadPlus`, 2005.

[2] Ariola, Z. M., Herbelin, H., and Sabry, A. A type-theoretic foundation of continuations and prompts. In *ACM SIGPLAN International Conference on Functional Programming* (2004), ACM Press, New York, pp. 40–53.

[3] Bromage, A. Initial (term) algebra for a state monad. `http://www.haskell.org/pipermail/haskell-cafe/2005-January/008259.html`, Jan. 2005.

[4] Bromage, A. A MonadPlusT with fair operations and pruning. `http://www.haskell.org/pipermail/haskell/2005-June/016037.html`, June 2005.

[5] Danvy, O., and Filinski, A. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice* (1990), ACM Press, New York, pp. 151–160.

[6] de Bruin, A., and de Vink, E. P. Continuation semantics for PROLOG with cut. In *TAPSOFT, Vol.1* (1989), J. Díaz and F. Orejas, Eds., vol. 351 of *Lecture Notes in Computer Science*, Springer, pp. 178–192.

[7] Dybvig, R. K., Peyton Jones, S. L., and Sabry, A. A monadic framework for delimited continuations. Tech. Rep. TR615, Department of Computer Science, Indiana University, June 2005.

[8] Federhen, S. A mathematical semantics for PLANNER. Master's thesis, University of Maryland, 1980.

[9] Fosca Giannotti, D. P., and Zaniolo, C. Semantics and expressive power of nondeterministic constructs in deductive databases. *Journal of Computer and System Sciences 62*, 1 (2001), 15–42.

[10] Friedman, D. P., and Kiselyov, O. A declarative applicative logic programming system. `http://kanren.sourceforge.net/`, 2005.

[11] Hinze, R. Deriving backtracking monad transformers. In *ICFP '00: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming* (2000), ACM Press, pp. 186–197.

[12] Jones, M. P., and Duponcheel, L. Composing monads. Tech. Rep. YALEU/DCS/RR-1004, Department of Computer Science, Yale University, New Haven, 1993.

[13] Kiselyov, O. How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible. Tech. Rep. TR611, Department of Computer Science, Indiana University, 2005.

[14] Liang, S., Hudak, P., and Jones, M. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1995), ACM Press, pp. 333–343.

[15] McCarthy, J. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems* (1963), P. Braffort and D. Hirschberg, Eds., North-Holland, Amsterdam, pp. 33–70.

[16] Moggi, E. An abstract view of programming languages. Tech. Rep. ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1990.

[17] Moggi, E. Notions of computation and monads. *Information and Computation 93*, 1 (1991), 55–92.

[18] Naish, L. Pruning in logic programming. Tech. Rep. 95/16, Department of Computer Science, University of Melbourne, Melbourne, Australia, June 1995.

[19] Peyton Jones, S. L., and Shields, M. B. Practical type inference for arbitrary-rank types. `http://research.microsoft.com/~simonpj/papers/putting/`, Apr. 2004.

[20] Pierce, B. C., et al. What is MonadPlus good for? `http://www.haskell.org/pipermail/haskell-cafe/2005-February/009072.html`, Feb. 2005.

[21] Roundy, D. What is MonadPlus good for? `http://www.haskell.org/pipermail/haskell-cafe/2005-February/009081.html`, Feb. 2005.

[22] Roy, P. V. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming 19–20* (1994), 385–441.

[23] Seres, S., Spivey, J. M., and Hoare, C. A. R. Algebra of Logic Programming. In *ICLP* (1999), pp. 184–199.

[24] Shan, C. Shift to control. In *Proceedings of the 5th workshop on Scheme and Functional Programming* (2004), O. Shivers and O. Waddell, Eds., pp. 99–107. Technical report, Computer Science Department, Indiana University, 2004.

[25] Siskind, J. M., and McAllester, D. A. Nondeterministic Lisp as a substrate for constraint logic programming. In *AAAI-93: Proceedings of the 11th National Conference on Artificial Intelligence* (11–15 July 1993), AAAI Press, pp. 133–138.

[26] Spivey, J. M. Combinators for breadth-first search. *Journal of Functional Programming 10*, 4 (2000), 397–408.

[27] Tullsen, M. First class patterns. In *Practical Aspects of Declarative Languages, 2nd International Workshop* (2000), E. Pontelli and V. S. Costa, Eds., vol. 1753 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–15.

[28] Turk, R. What is MonadPlus good for? `http://www.haskell.org/pipermail/haskell-cafe/2005-February/009086.html`, Feb. 2005.

[29] Wadler, P. L. Comprehending monads. *Mathematical Structures in Computer Science 2*, 4 (1992), 461–493.

[30] Wadler, P. L. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1992), ACM Press, pp. 1–14.

[31] Wand, M., and Vaillancourt, D. Relating models of backtracking. In *ACM SIGPLAN International Conference on Functional Programming* (2004), pp. 54–65.