Backtracking, Interleaving, and Terminating Monad Transformers

(Oppositional Pearl)

Oleg Kiselyov
FNOSC
oleg@pobox.com

Chung-chiieh 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 constructions 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 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, confusing 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 Sprey [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-
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 inter-
face: 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 solu-
tion is to explicitly include in the top-level request the maximum
number of answers to return. Such an interface is trivial to imple-
ment 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 imple-
       mentation 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 implementa-
tion 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 with-
out 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, prun-
ing, and an expressive top-level interface across these two imple-
mentations 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:

\[ \text{msplit} :: (\text{Monad m, LogicT t, MonadPlus t m}) \Rightarrow t \text{ m a} \rightarrow t \text{ m (Maybe a, t m a)} \]

where \(m\) is the underlying monad that provides arbitrary effects,
and \(t\) is the monad transformer designed and implemented in this
paper. Intuitively, \(\text{msplit}\) computes the first solution (if any) and
suspects the rest of the computation. Although it seems that \(\text{msplit}\)
simply de-constructs a list or stream, it is not so easy to implement
when \(t \text{ m a}\) is not a stream (indeed, not even a recursive type), as
is the case for our CPS and control-channel implementations. The
\(\text{msplit}\) 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 polymor-
phism [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, \(\text{mzero}\) and
\(\text{mplus}\), for expressing backtracking computations. The command
\(\text{mplus}\) introduces a choice junction, and \(\text{mzero}\) denotes failure:

\[
\begin{align*}
\text{class Monad m \Rightarrow MonadPlus m where} \\
\text{mzero :: m a} \\
\text{mplus :: m a \rightarrow m a \rightarrow m a}
\end{align*}
\]

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 \(\equiv\)
means in Section 3.4.)

**Definition 2.1 (Laws for MonadPlus).**

\[
\begin{align*}
\text{mplus a mzero} & \equiv a \\
\text{mplus mzero a} & \equiv a \\
\text{mplus a (mplus b c)} & \equiv \text{mplus (mplus a b) c} \\
\text{mzero >>= k} & \equiv \text{mzero} \\
\text{mplus a (mplus b k)} & \equiv \text{mplus (mplus a b) k} \\
\end{align*}
\]

The intuition behind these laws is that \(\text{mplus}\) 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:

\[
\begin{align*}
t_1, t_2, t_3 :: \text{MonadPlus m} \Rightarrow m \text{ Int} \\
t_1 = \text{mzero} \\
t_2 = \text{return 10 \text{ 'mplus' return 20 \text{ 'mplus' return 30}} \\
t_3 = \text{msum (map return \{10, 20, 30\})}
\end{align*}
\]

The first example represents a choice that leads to failure. The
second and third examples are identical, using a library definition of
\(\text{msum}\): 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 com-
putation; and a list with more than one element denotes multiple
successful results returned by multiple choices. To run such exam-
pies, we can trivially convert the answers generated by the multiple
choices into a stream of answers:

\[
\text{runList :: [a] \rightarrow [a]} \\
\text{runList = id}
\]

Indeed, \(\text{runList t}_1\) returns the empty list, and \(\text{runList t}_2\) and
\(\text{runList t}_3\) both return the list \([10, 20, 30]\).

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

\[
( \ldots \text{return 1 'mplus' return 2 'mplus' \ldots \text{return n})
\]

Other, more efficient implementations of backtracking have been
proposed using the two-continuation model [8] and delimited con-
trol operators [5]. We revisit the various implementations of back-
tracking 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:

\[ \text{runL} :: \text{Maybe Int} \rightarrow L a \rightarrow [a] \]

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:

\[
\text{odds} :: \text{MonadPlus} m \Rightarrow m \text{Int} \\
\text{odds} = (\text{return} 1) \cdot m\text{plus} \cdot (\text{odds} \gg\gg \lambda a \rightarrow \text{return} (2 + a))
\]

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 naively with other computations. For example, \( \text{odds} \cdot \text{mplus} \cdot \text{t} \) never considers \( \text{t} \) and thus the execution of the program:

\[
\text{runL} (\text{Just} 1) (\text{do} \ x \leftarrow \text{odds} \cdot \text{mplus} \cdot \text{t}3 \newline \quad \text{if even} \ x \ \text{then return} \ x \ \text{else mzero})
\]

diverges without ever succeeding. In this case, however, the three answers 10, 20, and 30 that could be returned by \( \text{t}3 \) 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:

\[
m1 \cdot \text{mplus} \cdot m = m1
\]

whenever \( m1 \) 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:

\[
\text{runL} (\text{Just} 10) (\text{odds} \cdot \text{interleave} \cdot \text{t}3)
\]

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

\[
\text{runL} (\text{Just} 1) (\text{do} \ x \leftarrow \text{odds} \cdot \text{interleave} \cdot \text{t}3 \newline \quad \text{if even} \ x \ \text{then return} \ x \ \text{else mzero})
\]

to succeed with the answer 10.

3.3 Fair Conjunction

The distributivity law from Definition 2.1 states that:

\[(\text{mplus} \ a \ b) \gg\gg k \equiv \text{mplus} \ (a \gg\gg k) \ (b \gg\gg k)\]

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

\[(\text{mplus} \ a \ b) \gg\gg k \equiv a \gg\gg k\]

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

\[
\text{let oddPlus} \ n = \text{odds} \gg\gg \lambda a \rightarrow (a + n) \\
\text{in runL} (\text{Just} 1) \newline \quad (\text{do} \ x \leftarrow (\text{return} 0 \cdot \text{mplus} \cdot \text{return} 1) \gg\gg \text{oddPlus} \newline \quad \quad \text{if even} \ x \ \text{then return} \ x \ \text{else mzero})
\]

diverges, even though there exists an infinite number of answers from return 1 \( \gg\gg \) oddPlus. Therefore, in addition to a fair mplus we need a fair \( \gg\gg \), which we denote with \( \gg\gg \). Using such a combinator, the program:

\[
\text{let oddPlus} \ n = \text{odds} \gg\gg \lambda a \rightarrow (a + n) \\
\text{in runL} (\text{Just} 1) \newline \quad (\text{do} \ x \leftarrow (\text{return} 0 \cdot \text{mplus} \cdot \text{return} 1) \gg\gg \text{oddPlus} \newline \quad \quad \text{if even} \ x \ \text{then return} \ x \ \text{else mzero})
\]

succeeds with the answer 2.

3.4 Laws of interleave and \( \gg\gg \)

By design, interleave and \( \gg\gg \) are fair analogues of mplus and \( \gg\gg \). 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).

\[
\text{interleave} \ m \ \text{mzero} \equiv m \\
\text{interleave} \ (\text{return} \ a \cdot \text{mplus} \cdot m1) \ m2 \equiv \text{return} \ a \cdot \text{mplus} \cdot (\text{interleave} \ m1 \ m2) \\
\text{mzero} \gg\gg k \equiv \text{mzero} \\
(\text{mplus} \ (\text{return} \ a) \ m) \gg\gg k \equiv \text{interleave} \ (k \ a) \ (m \gg\gg k)
\]

There are no explicit laws for computations of the form:

\[
\text{interleave} \ m \ \text{mzero}
\]

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:

\[
\text{interleave} \ m \ \text{mzero} = \\
\text{interleave} \ (\text{return} \ a \cdot \text{mplus} \cdot \text{mr}) \ \text{mzero} = \\
\text{return} \ a \cdot \text{mplus} \cdot (\text{interleave} \ \text{mzero} \ \text{mr}) = \\
\text{return} \ a \cdot \text{mplus} \cdot \text{mr} = m
\]

The main use of interleave and \( \gg\gg \) is in avoiding divergence when composing computations with a possibly infinite number of answers. In finitary cases, interleave and \( \gg\gg \) are observationally equivalent to mplus and \( \gg\gg \) 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 \( a \equiv b \) 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 \texttt{mplus} 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 succeeds on the success of another. For example, the non-deterministic computation \texttt{odds}, which produces an odd number, can be “restricted” to only succeed when the odd number is divisible by another number:

\[
\begin{align*}
\text{ifte} & \quad mzero \quad \text{th el} \\
\text{ifte} & \quad \text{mzero} \quad \text{th el} \\
\text{ifte} & \quad (\text{mzero} \quad \text{mplus} \quad m) \quad \text{th el} \\
\text{ifte} & \quad (\text{mzero} \quad \text{th el}) \quad \text{mplus} \\
\end{align*}
\]

In other words, the context \texttt{ifte [ ] th el} interacts in an unusual way with \texttt{mplus}. Because the \texttt{el} branch is only attempted when the test fails on the initial (rather than on a backtracking) try, \texttt{ifte} is particularly useful [4] for “explaining failure.” For example, the \texttt{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 \texttt{ifte}) if the heuristics applies. Section 7 illustrates with such an application.

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

\[
\begin{align*}
\text{test\_op} & = \text{runL (Just 10)} \\
\text{do} & \quad n \leftarrow \text{odds} \\
\text{guard} & \quad (n > 1) \\
\text{ifte} & \quad (\text{do} \quad d \leftarrow \text{iota} (n - 1) \\
\text{guard} & \quad (d > 1 \land n \mod d \equiv 0) \\
\text{return} & \quad n)
\end{align*}
\]

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

### 3.6 Pruning (Once)

The operator \texttt{ifte} is in some sense a pruning primitive. Another important pruning primitive is \texttt{once}, which selects, generally non-deterministically, one solution out of possibly many. The operator \texttt{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 \texttt{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 \texttt{once}. The example is based on the following code, which shows how to sort a list by generating all permutations and testing them:

\[
\begin{align*}
\text{bogosort} \quad l & = \text{do} \quad p \leftarrow \text{permute} \quad l \\
\text{if} & \quad \text{sorted} \quad p \quad \text{then} \quad \text{return} \quad p \quad \text{else} \quad \text{mzero}
\end{align*}
\]

\[
\begin{align*}
\text{sorted} & \quad (e_1 : e_2 : r) = e_1 \leq e_2 \quad \text{and} \quad \text{sorted} \quad (e_2 : r) \\
\text{sorted} & \quad = \quad \text{True} \\
\text{permute} & \quad [] = \quad \text{return} \quad [] \\
\text{permute} & \quad (h : t) = \text{do} \quad t' \leftarrow \text{permute} \quad t; \text{insert} \quad h \quad t' \\
\text{insert} & \quad e \quad [] = \quad \text{return} \quad [e] \\
\text{insert} & \quad e \quad l@(h : t) = \text{return} \quad (e : l) \mod \text{mplus}' \\
\text{do} & \quad t' \leftarrow \text{insert} \quad e \quad t; \text{return} \quad (h : t')
\end{align*}
\]

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

\[
\text{runL Nothing (bogosort [5, 0, 3, 4, 0, 1])}
\]

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 \texttt{bogosort} to be:

\[
\text{bogosort'} \quad l = \text{once} \quad (\text{do} \quad p \leftarrow \text{permute} \quad l \\
\text{if} & \quad \text{sorted} \quad p \quad \text{then} \quad \text{return} \quad p \quad \text{else} \quad \text{mzero})
\]

The change does not constrain the use of \texttt{bogosort} in a larger program which itself uses backtracking. It is just that \texttt{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:

\[
\text{test\_op'} = 
\begin{array}{c}
\text{runL (Just (10))} \\
\text{(do \(n \leftarrow \text{odds}\) guard (\(n > 1\)) ife (\(\text{once (do \(d \leftarrow \text{iota (n - 1)}\) guard (d > 1 \land n \mod d \equiv 0\))\)\)}))}
\end{array}
\]

\[
\text{(const mzero)}
\end{array}
\]

\(\text{return } n\)

\(\text{return }\)

\(\text{msplit}\)

\(\text{msplit}\)

\(\text{do r }\leftarrow \text{msplit sg}_1\)

\[
\text{case r of}
\begin{array}{c}
\text{Nothing } \rightarrow \text{sg}_2
\end{array}
\]

\[
\text{Just (sg}_{11}, sg_{12}) \rightarrow
\]

\[
\text{mplus}' \text{ (interleave sg}_2, sg_{12})
\]

\[
\text{(>>=)} \quad \rightarrow \text{m b } m b
\]

\[
\text{sg} \gg g =
\]

\[
\text{do r }\leftarrow \text{msplit t}\)
\]

\[
\text{case r of}
\begin{array}{c}
\text{Nothing } \rightarrow \text{mzero}
\end{array}
\]

\[
\text{Just (sg}_{11}, sg_{2}) \rightarrow \text{interleave (g sg}_{1}) (sg_{2} \gg g)
\]

\[
\text{ife \quad \rightarrow \text{m b } m b}
\]

\[
\text{ife t } \text{el } =
\]

\[
\text{do r }\leftarrow \text{msplit m}\)
\]

\[
\text{case r of}
\begin{array}{c}
\text{Nothing } \rightarrow \text{mzero}
\end{array}
\]

\[
\text{Just (sg}_{11}, \_ ) \rightarrow \text{return sg}_1
\]

\[
\text{\textbf{4. Splitting Computations}}
\]

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, >>-, ife, 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).**

\[
\begin{array}{c}
\text{msplit mzero } \Rightarrow \text{return Nothing}
\end{array}
\]

\[
\begin{array}{c}
\text{msplit (return a \text{'mplus' } m) } \Rightarrow \text{return (Just (a, m))}
\end{array}
\]

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:

\[
\begin{array}{c}
\text{ife \(r \leftarrow \text{msplit (return a) th el }\)}
\end{array}
\]

\[
\begin{array}{c}
\text{case r of}
\end{array}
\]

\[
\begin{array}{c}
\text{Nothing } \rightarrow \text{el}
\end{array}
\]

\[
\begin{array}{c}
\text{Just (sg}_{11}, sg_{2}) \rightarrow \text{th sg}_{1'} \text{'mplus' } (sg_{2} \gg th)
\end{array}
\]

\[
\text{\textbf{4.2 Implementing msplit Using Lists}}
\]

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.

\[
\text{newtype SSG a }=\text{ Stream [a]}
\]

\[
\text{unSSG (Stream str) }=\text{ str}
\]

\[
\begin{array}{c}
\text{instance Monad SSG where}
\end{array}
\]

\[
\begin{array}{c}
\text{return } e = \text{ Stream } [e] \\
\text{(Stream es) } \gg f = \text{ Stream } (\text{concat } \text{map } (\text{unSSG } f) \text{ es})
\end{array}
\]
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
(r >>) :: (Monad m, MonadPlus (t m)) ⇒
  t m a → (a → t m b) → t m b
if e :: (Monad m, MonadPlus (t m)) ⇒
  t m a → (a → t m b) → t m b
once :: (Monad m, MonadPlus (t m)) ⇒
  t m a → t m a

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 Hinch [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:

lift o return ≅ return
lift (m >>= k) ≅ lift m >>= lift o k

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, we should consider more generally a computation lift m ‘mplus’ tm, 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). The computation tm 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, tm) → return a ‘mplus’ tm

Now, 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).

\[ \text{reflect} (\text{msplit } (\text{lift } m \Rightarrow \text{mzero})) \Rightarrow \text{lift } m \Rightarrow \text{mzero} \]
\[ \text{reflect} r = \text{case } r \text{ of} \]
\[ \text{Nothing} \Rightarrow \text{mzero} \]
\[ \text{Just} (a, \text{tm}) \Rightarrow \text{return } a \Rightarrow \text{tm} \Rightarrow \text{a plus } \text{tm} \]

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:

\[ \text{newtype } \text{SFKT } m a = \]
\[ \text{SFKT } (\text{\textit{f}} \text{\textit{ans}}, \text{\textit{k}} (\text{\textit{m}} \text{\textit{ans}}) a → \text{\textit{f}} \text{\textit{K}} (\text{\textit{m}} \text{\textit{ans}}) a) \]
\[ \text{SFKT } \text{\textit{a}} = a \]

\[ \text{\textit{type}} \text{\textit{FK}} \text{\textit{ans}} = \text{\textit{ans}} \]
\[ \text{\textit{type}} \text{\textit{SK}} \text{\textit{ans}} a = a → \text{\textit{FK}} \text{\textit{ans}} a \]

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 Hinch [11].

\[ \text{instance } \text{Monad } m ⇒ \text{Monad } (\text{SFKT } m) \text{ where} \]
\[ \text{return } f = \text{SFKT } (\text{\textit{f}} \text{\textit{K}} \text{\textit{f}} \text{\textit{k}} → \text{\textit{sk}} \text{\textit{f}}) \]
\[ \text{m} \Rightarrow f \Rightarrow \text{SFKT } (\text{\textit{f}} \text{\textit{K}} \text{\textit{f}} → \text{\textit{sk}} \text{\textit{f}}) \]

\[ \text{instance } \text{Monad } m ⇒ \text{MonadPlus } (\text{SFKT } m) \text{ where} \]
\[ \text{mzero} = \text{SFKT } (\text{\textit{f}} \text{\textit{K}} \text{\textit{f}} → \text{\textit{f}}) \]
\[ \text{m} \Rightarrow \text{m plus } \text{m} \text{\textit{m}} = \text{SFKT } (\text{\textit{m}} \text{\textit{K}} \text{\textit{m}} \text{\textit{f}} → \text{\textit{m}} \text{\textit{sk}} \text{\textit{m}} \text{\textit{f}}) \]

\[ \text{msplit} (\text{\textit{f}} \text{\textit{K}} \text{\textit{f}} \text{\textit{k}} → \text{\textit{sk}} \text{\textit{f}}) \]
\[ \text{msplit} (\text{\textit{f}} \text{\textit{K}} \text{\textit{f}} → \text{\textit{sk}} \text{\textit{f}}) \]
\[ \text{msplit} (\text{\textit{f}} \text{\textit{K}} \text{\textit{f}} → \text{\textit{sk}} \text{\textit{f}}) \]
\[ \text{msplit} (\text{\textit{f}} \text{\textit{K}} \text{\textit{f}} → \text{\textit{sk}} \text{\textit{f}}) \]
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:

\[ r r (\text{lift} \ m >>= \text{mzero}) \cong \text{lift} \ m >>= \text{mzero} \]

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

\[ \text{msplit} \ (\text{lift} \ m >>= \text{mzero}) \cong \text{lift} \ m >>= \text{return} \text{Nothing}, \]

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

\[ r r (\text{lift} \ m \text{msplus}' \text{tma}) \cong \text{lift} \ m \text{msplus}' \ (\text{rr} \ \text{tma}) \]

we observe that:

\[ \text{lift} \ m \text{msplus}' \text{tma} \cong SFKT (λsk fk → (λsk fk → m >>= (λa → sk a fk)) sk (unSFKT tma sk fk)) \cong SFKT (λsk fk → m >>= (λa → sk a (unSFKT tma sk fk))) \]

Thus we have:

\[ \text{msplit} \ (\text{lift} \ m \text{msplus}' \text{tma}) \cong \text{lift} \ m >>= (λa → \text{return} \ (\text{Just} \ (a, \text{rr} \ \text{tma}))) \]

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:

\[ g f v \text{mzero} \cong v \]

\[ g f v \ (\text{lift} \ m \text{msplus}' \text{tm}_1) \cong m >>= (λa → f a (g f v \text{tm}_1)) \]

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:

\[
promptP :: \text{Monad} m ⇒
\begin{align*}
\text{Prompt} \ r \ a & → \text{CC} \ r \ m \ a \\
\text{abortP} :: \text{Monad} m ⇒
\text{Prompt} \ r \ a & → \text{CC} \ r \ m \ a
\end{align*}
\]

\[
\text{shiftP} :: \text{Monad} m ⇒
\text{Prompt} \ r \ b & →
(\text{CC} \ r \ m \ a → \text{CC} \ r \ m \ b) → \text{CC} \ r \ m \ a
\]

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:

\[ \text{runCC} :: \text{Monad} m ⇒ (∀ r. \text{CC} \ r \ m \ a) → m a \]

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:

\begin{verbatim}
runIdentity (runCC (promptP $ λp →
  do x ← shiftP p (λk → k (λ (return 2)))
    return (x + 1)))
\end{verbatim}

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 ← []; 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:

\begin{verbatim}
newtype SR r m a =
  SR (∀ ans. ReaderT (Prompt r (Tree r m ans))) (CC r m a)
unsSR (SR f) = f
\end{verbatim}

\begin{verbatim}
data Tree r m a = HZero |
  | HOne a |
  | HChoice a (CC r m (Tree r m a))
\end{verbatim}

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:

\[ \text{ReaderT} \ (\text{Prompt} \ r \ (\text{Tree} \ r \ m \ a)) \ (\text{CC} \ r \ m) \]

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 \texttt{Tree r m a}. A deterministic result is marked by tagging it with \texttt{HOne}. A failure is represented by \texttt{HZero} and a choice is represented by \texttt{HChoice a r}. The latter value describes both the result \texttt{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 \texttt{HChoice a (return HZero)} at the expense of obscuring the definitions and losing some performance.

The following definition shows that the type \texttt{SR r m} is an instance of \texttt{Monad}:

\[
\text{instance Monad m} \Rightarrow \text{Monad (SR r m)} \text{ where}
\]

\[
\begin{align*}
\text{return } e & = \text{SR (return } e) \\
\text{(SR } m) & \Rightarrow = f = \text{SR (m } \Rightarrow (\text{unsSR } f))
\end{align*}
\]

The definition shows that deterministic computations are executed “normally”—that is, as if they were in the base monad \texttt{m}. And since \texttt{SR} is a \texttt{newtype}, tagging with \texttt{SR} and untagging with \texttt{unsSR} do not take any run time.

We then show that \texttt{SR r m} is an instance of \texttt{MonadPlus}:

\[
\text{instance Monad m} \Rightarrow \text{MonadPlus (SR r m) where}
\]

\[
\begin{align*}
\text{mzero} & = \text{SR (ask } \Rightarrow \rightarrow \text{ lift (abortSR pr } \text{pr (return HZero)))} \\
m_1 \cdot \text{mplus } m_2 & = \text{SR (ask } \Rightarrow \rightarrow \text{ lift } \$ \text{shiftPR } \text{pr } \$ \text{ask } \rightarrow \text{do } f_1 \leftarrow \text{sk } \text{runReaderT (unsSR } m_1) \text{pr} \\
& \quad \text{let } f_2 = \text{sk } \text{runReaderT (unsSR } m_2) \text{pr} \\
& \quad \text{compose_trees } f_1 f_2
\end{align*}
\]

\[
\text{compose_trees :: Monad m } \Rightarrow \text{Tree r m a } \rightarrow \text{CC r m (Tree r m a) } \rightarrow \text{CC r m (Tree r m a)}
\]

\[
\begin{align*}
\text{compose_trees } \text{HZero } r & = r \\
\text{compose_trees } \text{HOne } a \text{ r} & = \text{return } \$ \text{HChoice a } r \\
\text{compose_trees } \text{HChoice a } r' \text{ r} & = \text{return } \$ \text{HChoice a } r' \Rightarrow (\lambda v \rightarrow \text{compose_trees } v r)
\end{align*}
\]

A failing computation \texttt{mzero} simply aborts to the current delimiter with the result \texttt{HZero}. A non-deterministic choice is slightly more complicated: we capture the continuation \texttt{sk} up to the current delimiter. The first alternative \texttt{m1} is immediately executed in that continuation; the second alternative \texttt{m2} is suspended. The function \texttt{compose_trees} builds a new \texttt{Tree} combining the results obtained from the first branch with the suspension from the second branch.

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

\[
\begin{align*}
\text{reify} & = \text{Monad m } \Rightarrow \text{SR r m a } \rightarrow \text{CC r m (Tree r m a)}
\end{align*}
\]

\[
\text{reify } m = \text{promptP (\lambda pr } \rightarrow \text{runReaderT (unsSR } m) \text{pr } \Rightarrow (\text{return } \Rightarrow \text{HOne}))
\]

As the code shows, the function \texttt{reify} \texttt{m} creates a new prompt, sets it, executes the computation, and tags the result with \texttt{HOne}. If the computation \texttt{m} executes deterministically, we get the result \texttt{HOne a} where \texttt{a} is the resulting value. If the computation \texttt{m} fails, then the continuation \texttt{(return o HOne)} is aborted and we instead get \texttt{HZero} as the resulting value. Finally, let us illustrate the case where the computation \texttt{m} is about to execute \texttt{mplus} \texttt{m1} \texttt{m2} using the following example:

\[
\text{reify } (m_0 \Rightarrow (\lambda x \rightarrow k_1 x \cdot \text{mplus } k_2 x) \Rightarrow k_3)
\]

where \texttt{m0} is deterministic. This example is equivalent to:

\[
\text{do HOne } x \leftarrow \text{reify } m_0
\]

\[
\text{flval } \leftarrow \text{reify } (k_1 x \Rightarrow k_3)
\]

\[
\text{let } \text{f2comp } = \text{reify } (k_2 x \Rightarrow k_3)
\]

\[
\text{compose_trees } \text{flval } \text{f2comp}
\]

Here \texttt{flval} is the result (reified into the \texttt{Tree} data type) of the first choice of \texttt{mplus}, and \texttt{f2comp} is the computation that corresponds to the second choice. If \texttt{flval} is \texttt{HZero}—that is, the first choice eventually fails (either in \texttt{k1} or in \texttt{k3}), then \texttt{compose_trees} will run the computation \texttt{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, \texttt{i.e.,} if \texttt{flval} is \texttt{HOne a}, then \texttt{compose_trees} represents the available choice by creating a result \texttt{HChoice a f2comp}, which suspends the computation \texttt{f2comp} to be later explored if needed as a possible alternative solution. Finally, if \texttt{flval} is itself the choice \texttt{Choice a r’}—that is, the execution \texttt{k1 x \Rightarrow k3} has (unexplored) alternative branches (represented by \texttt{r’}), the function \texttt{compose_trees} essentially composes the unexplored choices \texttt{r’} with the unexplored choice \texttt{f2comp}. One may think of that “composition” as a rotation of a binary decision tree, or joining a branch \texttt{f2comp} to a binary node \texttt{(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, \texttt{f2comp} (or more generally the computation \texttt{sk (runReaderT (unsSR m2) pr}) in the implementation of \texttt{mplus}) represents the choice point. The function \texttt{compose_trees} essentially handles the “top-most” choice point (the CP register of the WAM).

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

Finally we implement \texttt{msplit} for the monad transformer \texttt{SR r}:

\[
\text{instance MonadTrans (SR r) where}
\]

\[
\begin{align*}
\text{lift } m & = \text{SR (lift (lift m))} \\
\text{instance LogicT (SR r) where}
\end{align*}
\]

\[
\begin{align*}
\text{msplit } m & = \text{SR (lift (reify m \Rightarrow (\text{return } \Rightarrow \text{reflect_sr}))} \\
\text{where} \text{reflect_sr HZero} & = \text{Nothing} \\
\text{reflect_sr (HOne a)} & = \text{Just (a, mzero)} \\
\text{reflect_sr (HChoice a r')} & = \text{Just (a, SR (lift r') \Rightarrow} \Rightarrow \text{return } \Rightarrow \text{reflect_sr})
\end{align*}
\]
This implementation of \textit{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 \textit{ans} escape. Given the computation \textit{m}, we first \textit{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 \textit{msplit}. The problem with the \textit{msplit} implementation is that \textit{r1} in \textit{HChoice} \textit{a} \textit{r1} has the type \textit{CC r m} (Tree \textit{r m a}) (of the reified computation), whereas \textit{msplit} must produce \textit{SR r m a}. Thus we must be able to go from the reified computation Tree \textit{r m a} “back” to the computation \textit{m a}. The recursion in the implementation of \textit{msplit} accomplishes that goal: of reflecting a value \textit{HChoice} \textit{a} \textit{r} back into the computation \textit{return a ’mplus’ r}’s The recursion of \textit{reflect_r} is a consequence of the fact that Tree \textit{r m a} is a recursive data type (although SR \textit{r m a} is generally not, depending on \textit{m}). Note that \textit{msplit} invokes \textit{reify}, which sets the \textit{prompt}, overriding the \textit{prompt} set by the top-level \textit{reify}. This “dynamic scoping” inherent in delimited control [2] is essential for \textit{msplit}.

In summary, we have implemented the \textit{LogicT} interface to support “direct-style” programming with a rich combination of several computational effects: of the base monad \textit{m}, the computational of the \textit{CC} monad, and of the \textit{Reader} monad. Although the present \textit{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 \textit{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 \textit{SFKT m} monad seems more efficient (and we would recommend using it in production, because all its costs are quantifiable and not large), \textit{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 logic programming systems.

6. Running Computations

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

\[
\text{solve} :: (Monad m) \Rightarrow \text{SFKT} m a \rightarrow m [a]
\]

\[
\text{solve} (\text{SFKT} m) = m (\lambda a \text{fk} \rightarrow \text{fk} \gg> \text{return } a (a)) (\text{return } [])
\]

which is identical to the one provided by Hinze [11]. This function runs the backtracking computation of type \textit{SFKT} \textit{m a} and collects all the answers in a list (to be observed in the source \textit{m} monad).

One may think that this function is sufficient: to observe at most \textit{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 \textit{m} is non-strict. If \textit{m} is strict (e.g., \textit{IO}), it is clear from the definition of \textit{solve} that all the answers of \textit{SFKT} \textit{m a} will be produced and collected into the list, even if we need only a few of them. This also means that applying \textit{solve} to a computation \textit{SFKT IO a} that has an infinite number of answers (such as \textit{odds}) will diverge.

Thus we need a more general function \textit{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, \textit{runL} can be safely used with non-deterministic computations with an infinite number of answers over a strict monad.

Implementing \textit{runL} for the \textit{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 \textit{fk} to run to get more answers. We can easily disregard the failure computation after the first answer:

\[
\text{observe} :: \text{Monad m} \Rightarrow \text{SFKT} m a \rightarrow m a
\]

\[
\text{observe} (\text{SFKT} m) = m (\lambda a \text{fk} \rightarrow \text{fk} \gg> \text{return } \text{fail } \text{"no answer"})
\]

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

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

\[
\text{bagofN} :: (\text{Monad m}, \text{LogicT t}, \text{MonadPlus} (t m)) \Rightarrow\text{Maybe } \text{Int} \rightarrow \text{m } \text{a} \rightarrow \text{m } [a]
\]

\[
\text{bagofN} \text{Just n} \_ \text{n } \leq 0 \Rightarrow \text{return } []
\]

\[
\text{where} \text{bagofN} ^{\prime} \text{Nothing} = \text{return } []
\]

\[
\text{bagofN} ^{\prime} \text{(Just } (a, m')) =
\]

\[
\text{bagofN} ^{\prime} \text{(fmap pred m') } > > (\text{return } a (a))
\]

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

The result type of \textit{bagofN} is \text{t m} \text{a}; we are still in the transformed monad. To get back to the source monad \textit{m}, we need to “observe” [11] the produced list value. The observation function is necessarily specific to the backtracking implementation.3 For the \textit{SFKT} monad, it is given above. For the \textit{SR} monad, it is as follows:

\[
\text{observe}_{SR} :: \text{Monad m} \Rightarrow (\forall t. \text{SR r m a} \rightarrow m a)
\]

\[
\text{observe}_{SR} m = \text{runCC} (\text{reify m} > > \text{pick1})
\]

\[
\text{where} \text{pick1 } \text{HZero} = \text{fail } \text{"no answer"}
\]

\[
\text{pick1 } (\text{HOne a}) = \text{return } a
\]

\[
\text{pick1 } (\text{HChoice } a r) = \text{return } a
\]

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

With the help of \textit{observe} we can now write the function \textit{runL} that we have used to run our examples:

\[
\text{type } L a = \forall r. \text{SR r Identity a}
\]

\[
\text{runL} :: \text{Maybe } \text{Int} \rightarrow L a \rightarrow [a]
\]

\[
\text{runL } n m = \text{runIdentity (observe}_{SR} (\text{bagofN n m})
\]

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

3 Ideally, the function \textit{observe} should be part of the \textit{LogicT} class, but this is not possible because of the universally quantified type variable \textit{r} in the last implementation.
The main game function can then be defined:

game :: (MonadPlus t m, LogicT t, Monad m, MonadIO t m) \Rightarrow (Mark, PlayerProc t m) \rightarrow (Mark, PlayerProc t m) \rightarrow t m

where game' player1 player2 new'game

\[ \text{game' player1@}(t, \text{proc}) \text{ other'player } g \]
\[ = \begin{cases} \text{Game}\{\text{winner } = \text{Just } k}\} \leftrightarrow g \\
                    = \text{liftIO} (\text{putStrLn} (\text{show } k) ++ \text{" wins!"]) \\
                    = \text{liftIO} (\text{putStrLn} \text{"Draw!"}) \\
                    = \text{do} (\_ < g) \rightarrow \text{once } (\text{proc } p \\ g) \\
                    \text{otherwise} \\
                    = \text{do} (\_ < g) \rightarrow \text{once } (\text{proc } p \\ g) \\
                    = \text{minmax } a'\text{lim } dlim \text{ blim } p \\ g \]

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

\[ \text{minmax } :: (\text{MonadPlus } t m, \text{Monad } m, \text{LogicT } t) \Rightarrow (\text{Int } \rightarrow \text{Int } \rightarrow \text{PlayerProc } t m) \rightarrow (\text{Int } \rightarrow \text{Int } \rightarrow \text{PlayerProc } t m) \]
\[ = \text{do } \text{wbs } \leftarrow \text{bagofN } (\text{Just } \text{blim}) \\
\]
\[ \text{do } m \leftarrow \text{choose } (\text{moves } g) \\
\]
\[ \text{let } g' = \text{take'move } p \text{ m } g \\
\]
\[ \text{if } \text{dlim } \leq 0 \\
\]
\[ \text{then return } (\text{estimate'state } p \text{ g'}, g') \\
\]
\[ \text{else do } (w, \_ ) \leftarrow \text{self } (\text{dlim } - 1) \text{ blim} \\
\]
\[ \text{(other'player } p \text{) g'} \\
\]
\[ \text{return } (\neg w, g') \]
\[ \text{return } (\text{maximumBy } (\lambda(x, \_ ) (y, \_ ) \rightarrow \text{compare } x y) \text{ wbs}) \]

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

Unfortunately, this code is too slow. Even for such a simple game on a 3 \times 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 \text{p}:
first move wins p g =
   do m ← choose (moves g)
   let g' = take move p m g
   guard (maybe False (≡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 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
         = ife (once (first move wins p g))
            (return o snd)
            (ife (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 ife 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

Serres 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 non-determinism), 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


