Purely Functional Lazy Non-deterministic Programming

Sebastian Fischer
Christian-Albrechts University, Germany
sebf@informatik.uni-kiel.de

Oleg Kiselyov
FNMOC, CA, USA
oleg@pobox.com

Chung-chieh Shan
Rutgers University, NJ, USA
ccshan@cs.rutgers.edu

Abstract

Functional logic programming and probabilistic programming have demonstrated the broad benefits of combining laziness (non-strict evaluation with sharing of the results) with non-determinism. Yet these benefits are seldom enjoyed in functional programming, because the existing features for non-strictness, sharing, and non-determinism in functional languages are tricky to combine.

We present a practical way to write purely functional lazy non-deterministic programs that are efficient and perspicuous. We achieve this goal by embedding the programs into existing languages (such as Haskell, SML, and OCaml) with high-quality implementations, by making choices lazily and representing data with non-deterministic components, by working with custom monadic data types and search strategies, and by providing equational laws for the programmer to reason about their code.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.6 [*Programming Techniques*]: Logic Programming; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Design, Languages

Keywords Monads, side effects, continuations, call-time choice

1. Introduction

Non-strict evaluation, sharing, and non-determinism are all valuable features in functional programming. Non-strict evaluation lets us express infinite data structures and their operations in a modular way (Hughes 1989). Sharing lets us represent graphs with cycles, such as circuits (surveyed by Acosta-Gómez 2007), and express memoization (Michie 1968), which underlies dynamic programming. Since Rabin and Scott's Turing-award paper (1959), non-determinism has been applied to model checking, testing (Claessen and Hughes 2000), probabilistic inference, and search.

These features are each available in mainstream functional languages. A call-by-value language can typically model non-strict evaluation with thunks and observe sharing using reference cells, physical identity comparison, or a generative feature such as Scheme's `gensym` or SML's exceptions. Non-determinism can be achieved using `amb` (McCarthy 1963), threads, or first-class continuations (Felleisen 1985; Haynes 1987). In a non-strict language like Haskell, non-determinism can be expressed using a list monad (Wadler 1985) or another `MonadPlus` instance, and sharing can be represented using a state monad (Acosta-Gómez 2007; §2.4.1).

These features are particularly useful together. For instance, sharing the results of non-strict evaluation—known as call-by-need or lazy evaluation—ensures that each expression is evaluated at most once. This combination is so useful that it is often built-in: as `delay` in Scheme, `lazy` in OCaml, and memoization in Haskell.

In fact, many programs need all three features. As we illustrate in §2, lazy functional logic programming (FLP) can be used to express search problems in the more intuitive *generate-and-test* style yet solve them using the more efficient *test-and-generate* strategy, which is to generate candidate solutions only to the extent demanded by the test predicate. This pattern applies to property-based test-case generation (Christiansen and Fischer 2008; Fischer and Kuchen 2007; Runciman et al. 2008) as well as probabilistic inference (Goodman et al. 2008; Koller et al. 1997).

Given the appeal of these applications, it is unfortunate that combining the three features naively leads to unexpected and undesired results, even crashes. For example, `lazy` in OCaml is not thread-safe (Nicollet et al. 2009), and its behavior is unspecified if the delayed computation raises an exception, let alone backtracks. Although sharing and non-determinism can be combined in Haskell by building a state monad that is a `MonadPlus` instance (Hinze 2000; Kiselyov et al. 2005), the usual monadic encoding of non-determinism in Haskell loses non-strictness (see §2.2). The triple combination has also been challenging for theoreticians and practitioners of FLP (López-Fraguas et al. 2007, 2008). After all, Algol has made us wary of combining non-strictness with any effect.

The FLP community has developed a sound combination of laziness and non-determinism, *call-time choice*, embodied in the Curry language. Roughly, call-time choice makes lazy non-deterministic programs predictable and comprehensible because their declarative meanings can be described in terms of (and is often same as) the meanings of eager non-deterministic programs.

1.1 Contributions

We embed lazy non-determinism with call-time choice into mainstream functional languages in a shallow way (Hudak 1996), rather than, say, building a Curry interpreter in Haskell (Tolmach and Antoy 2003). This new approach is especially practical because these languages already have mature implementations, because functional programmers are already knowledgeable about laziness, and because different search strategies can be specified as `MonadPlus` instances and plugged into our monad transformer. Furthermore, we provide equational laws that programmers can use to reason about their code, in contrast to previous accounts of call-time choice based on directed, non-deterministic rewriting.

The key novelty of our work is that non-strictness, sharing, and non-determinism have not been combined in such a general way before in purely functional programming. Non-strictness and non-determinism can be combined using data types with non-deterministic components, such that a top-level constructor can be

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.

computed without fixing its arguments. However, such an encoding defeats Haskell's built-in sharing mechanism, because a piece of non-deterministic data that is bound to a variable that occurs multiple times may evaluate to a different (deterministic) value at each occurrence. We retain sharing by annotating programs explicitly with a monadic combinator for sharing. We provide a generic library to define non-deterministic data structures that can be used in non-strict, non-deterministic computations with explicit sharing.

Our library is implemented as a monad transformer and can, hence, be combined with arbitrary monads for non-determinism. We are, thus, not restricted to the list monad (which implements depth-first search) but can also use monads that backtrack more efficiently or provide a complete search strategy. The library does not directly support logic variables—perhaps the most conspicuous feature of FLP—and the associated solution techniques of narrowing and residuation, but logic variables can be emulated using non-deterministic generators (Antoy and Hanus 2006) or managed using an underlying monad of equality and other constraints.

We present our concrete code in Haskell, but we have also implemented our approach in OCaml. Our monadic computations perform competitively against corresponding computations in Curry that use non-determinism, narrowing, and unification.

1.2 Structure of the paper

In §2 we describe non-strictness, sharing, and non-determinism and why they are useful together. We also show that their naive combination is problematic, to motivate the explicit sharing of non-deterministic computations. In §3 we clarify the intuitions of sharing and introduce equational laws to reason about lazy non-determinism. Section 4 develops an easy-to-understand implementation in several steps. Section 5 generalizes and speeds up the simple implementation. We review the related work in §6 and then conclude.

2. Non-strictness, sharing, and non-determinism

In this section, we describe non-strictness, sharing, and non-determinism and explain why combining them is useful and non-trivial.

2.1 Lazy evaluation

Lazy evaluation is illustrated by the following Haskell predicate, which checks whether a given list of numbers is sorted:

```
isSorted :: [Int] -> Bool
isSorted (x:y:zs) = (x <= y) && isSorted (y:zs)
isSorted _        = True
```

In a non-strict language such as Haskell, the arguments to a function are only evaluated as much as demanded by the definition of the function. The predicate `isSorted` only demands the complete input list if it is sorted. If the list is not sorted, then it is only demanded up to the first two elements that are out of order.

As a consequence, we can apply `isSorted` to infinite lists and it will yield `False` if the given list is unsorted. Consider the following function that produces an infinite list:

```
iterate :: (a -> a) -> a -> [a]
iterate next x = x : iterate next (next x)
```

The test `isSorted (iterate (‘div‘2) n)` yields the result `False` if `n>0`. It does not terminate if `n<=0` because an infinite list cannot be identified as being sorted without considering each of its elements. In this sense, `isSorted` is not total (Escardó 2007).

A *lazy* evaluation strategy is not only non-strict. Additionally, it evaluates each expression bound to a variable at most once—even if the variable occurs more than once. For example, the variable `x` occurs twice in the right-hand side of the definition of `iterate`. Although `iterate` never evaluates its argument `x`, the duplication of a computation bound to `x` does *not* cause it to be evaluated twice. In a lazy language, the value of the expression `factorial 100` would only be computed once when evaluating `iterate (‘div‘2) (factorial 100)`. This property—called *sharing*—makes lazy evaluation strictly more efficient than eager evaluation, at least on some problems (Bird et al. 1997).

2.2 Non-determinism

Programming non-deterministically can simplify the declarative formulation of an algorithm. For example, many languages are easier to describe using non-deterministic rather than deterministic automata. As logic programming languages such as Prolog and Curry have shown, the expressive power of non-determinism simplifies programs because different non-deterministic results can be viewed individually rather than as members of a (multi-)set of possible results (Antoy and Hanus 2002).

In Haskell, we can express non-deterministic computations using lists (Wadler 1985) or, more generally, monads that are instances of the type class `MonadPlus`. A monad `m` is a type constructor that provides two polymorphic operations:

```
return :: a -> m a
(>>=)  :: m a -> (a -> m b) -> m b
```

The operation `return` builds a deterministic computation that yields a value of type `a`, and the operation `>>=` (“bind”) chains computations together. Haskell's `do`-notation is syntactic sugar for long chains of `>>=`. For example, the expression

```
do x <- e1
   e2
```

desugars into `e1 >>= \x -> e2`. If a monad `m` is an instance of `MonadPlus`, then two additional operations are available:

```
mzero :: m a
mplus :: m a -> m a -> m a
```

Here, `mzero` is the primitive failing computation, and `mplus` chooses non-deterministically between two computations. For the list monad, `return` builds a singleton list, `mzero` is an empty list, and `mplus` is list concatenation.

As an example, the following monadic operation computes all permutations of a given list non-deterministically:

```
perm :: MonadPlus m => [a] -> m [a]
perm []     = return []
perm (x:xs) = do ys <- perm xs
                 zs <- insert x ys
                 return zs

insert :: MonadPlus m => a -> [a] -> m [a]
insert x xs = return (x:xs)
      ‘mplus‘ case xs of
                []     -> mzero
                (y:ys) -> do zs <- insert x ys
                             return (y:zs)
```

The operation `perm` permutes a list by recursively inserting elements at arbitrary positions. To insert an element `x` at an arbitrary position in `xs`, the operation `insert` *either* puts `x` in front of `xs` *or* recursively inserts `x` somewhere in the tail of `xs` if `xs` is not empty.

Non-determinism is especially useful when formulating search algorithms. Following the *generate-and-test* pattern, we can find solutions to a search problem by non-deterministically describing candidate solutions and using a separate predicate to filter results. It is much easier for a programmer to express generation and testing separately than to write an efficient search procedure by hand, because the generator can follow the structure of the data

(to achieve completeness easily) and the test can operate on fully determined candidate solutions (to achieve soundness easily).

We demonstrate this technique with a toy example, permutation sort, which motivates our combination of non-strictness, sharing, and non-determinism. Below is a simple declarative specification of sorting. In words, to sort a list is to compute a permutation of the list that is sorted. The convenience function `guard` is *semi-deterministic*: it yields `()` if its argument is `True` and fails otherwise.

```
sort :: MonadPlus m => [Int] -> m [Int]
sort xs = do ys <- perm xs
             guard (isSorted ys)
             return ys
```

Unfortunately, this program is grossly inefficient, because it iterates through every permutation of the list. It takes about a second to sort 10 elements, more than 10 seconds to sort 11 elements, and more than 3 minutes to sort 12 elements. The inefficiency is mostly because we do not use the non-strictness of the predicate `isSorted`. Although `isSorted` rejects a permutation as soon as it sees two elements out of order, `sort` generates a complete permutation before passing it to `isSorted`. Even if the first two elements of a permutation are already out of order, exponentially many permutations of the remaining elements are computed.

In short, the usual, naive monadic encoding of non-determinism in Haskell loses non-strictness.

2.3 Retaining non-strictness by sacrificing sharing

The problem with the naive monadic encoding of non-determinism is that the arguments to a constructor must be deterministic. If these arguments are themselves results of non-deterministic computations, these computations must be performed completely before we can apply the constructor to build a non-deterministic result.

To overcome this limitation, we can redefine all data structures such that their components may be non-deterministic. A data type for lists with non-deterministic components is as follows:

```
data List m a = Nil | Cons (m a) (m (List m a))
```

We define operations to construct such lists conveniently:

```
nil :: Monad m => m (List m a)
nil = return Nil

cons :: Monad m => m a -> m (List m a) -> m (List m a)
cons x y = return (Cons x y)
```

We redefine the non-strict `isSorted` to test non-deterministic lists:

```
isSorted :: MonadPlus m => m (List m a) -> m Bool
isSorted ml = ml >>= \l ->
  case l of
   Cons mx mxs -> mxs >>= \xs ->
    case xs of
     Cons my mys -> mx >>= \x -> my >>= \y ->
      if x <= y then isSorted (cons (return y) mys)
                else return False
     _ -> return True
   _ -> return True
```

By generating lists with non-deterministic arguments, we can define a lazier version of the permutation algorithm.

```
perm :: MonadPlus m => m (List m a) -> m (List m a)
perm ml = ml >>= \l ->
  case l of Nil         -> nil
            Cons mx mxs -> insert mx (perm mxs)
```

Note that we no longer evaluate (bind) the recursive call of `perm` in order to pass the result to the operation `insert`, because `insert` now takes a non-deterministic list as its second argument.

```
insert :: MonadPlus m
       => m a -> m (List m a) -> m (List m a)
insert mx mxs = cons mx mxs
        `mplus` do Cons my mys <- mxs
                   cons my (insert mx mys)
```

The operation `insert` *either* creates a new list with the non-deterministic `mx` in front of the non-deterministic `mxs` *or* it inserts `mx` somewhere in the tail of `mxs`. Here, the pattern match in the `do`-expression binding is non-exhaustive. If the computation `mxs` returns `Nil`, the pattern-match failure is a failing computation.

Now, we can define a permutation-sort algorithm that lazily checks whether generated permutations are sorted:

```
sort :: MonadPlus m
     => m (List m Int) -> m (List m Int)
sort xs = let ys = perm xs in
          do True <- isSorted ys
             ys
```

Unfortunately, this version of the algorithm does not sort. It yields every permutation of its input, not only the sorted permutations. This is because the shared variable `ys` in the new definition of `sort` is bound to the non-deterministic *computation* yielding a permutation of the input rather than to the result of this computation. Consequently, `isSorted` checks whether there is a sorted permutation and, if so, `sort` yields an arbitrary permutation.

In short, the presence of non-deterministic components in data structures conflicts with the intuition that shared variables such as `ys` denote *values*, fully determined if not yet fully computed. In order for `sort` to work, the shared non-deterministic computation `ys`, used twice in `sort`, must yield the same result each time.

2.4 Explicit sharing

Our new approach to non-determinism is lazy in that it preserves both non-strictness and sharing. We provide a combinator `share` for explicit sharing, which can be used to introduce variables for non-deterministic computations that represent *values* rather than computations. The combinator `share` has the signature[1]

```
share :: m a -> m (m a)
```

where `m` is an instance of `MonadPlus` that supports explicit sharing. (We describe the implementation of explicit sharing in §§4–5.) The function `sort` can then be redefined to actually sort:

```
sort xs = do ys <- share (perm xs)
             True <- isSorted ys
             ys
```

In this version of `sort`, the variable `ys` denotes the same permutation wherever it occurs but is nevertheless only computed as much as demanded by the predicate `isSorted`.

3. Programming with lazy non-determinism

In this section we formalize the `share` combinator and specify equational laws with which a programmer can reason about non-deterministic programs with `share` and predict their observations. Before the laws, we first present a series of small examples to clarify how to use `share` and what `share` does.

3.1 The intuition of sharing

We define two simple programs. The computation `coin` flips a coin and non-deterministically returns either `0` or `1`.

```
coin :: MonadPlus m => m Int
coin = return 0 `mplus` return 1
```

[1] In fact, the signature has additional class constraints; see §5.

The function duplicate evaluates a given computation a twice.

```
duplicate :: Monad m => m a -> m (a, a)
duplicate a = do u <- a
                 v <- a
                 return (u,v)
```

3.1.1 Sharing enforces call-time choice

We contrast three ways to bind x:

```
dup_coin_let   = let x = coin in duplicate x
dup_coin_bind  = do x <- coin
                    duplicate (return x)
dup_coin_share = do x <- share coin
                    duplicate x
```

The programs dup_coin_let and dup_coin_bind do not use share, so we can understand their results by treating m as any non-determinism monad, such as the set monad. The program dup_coin_let binds the variable x to the non-deterministic computation coin. The function duplicate executes x twice—performing two independent coin flips—so dup_coin_let yields four answers, namely (0,0), (0,1), (1,0), and (1,1). In contrast, dup_coin_bind binds x, of type Int, to share not the coin computation but its result. The function duplicate receives a *deterministic* computation return x, whose two evaluations yield the same result, so dup_coin_bind yields only (0,0) and (1,1).

The shared computation x in dup_coin_share behaves like return x in dup_coin_bind: both arguments to duplicate are deterministic computations, which yield the same results even when evaluated multiple times. As in §§2.3–2.4, we wish to share the results of computations, and we wish variables to denote values. In dup_coin_bind, x has the type Int and indeed represents an integer. In dup_coin_share, x has the type m Int, yet it represents one integer rather than a set of integers. Thus dup_coin_share yields the same two results as dup_coin_bind.

3.1.2 Sharing preserves non-strictness

Shared computations, like the lazy evaluation of pure Haskell expressions, take place only when their results are needed. In particular, if the program can finish without a result from a shared computation, then that computation never happens. The sorting example in §2 shows how non-strictness can improve performance dramatically. Here, we illustrate non-strictness with two shorter examples:

```
strict_bind = do x <- undefined :: m Int
                 duplicate (const (return 2)
                                  (return x))

lazy_share  = do x <- share (undefined :: m Int)
                 duplicate (const (return 2) x)
```

The evaluation of strict_bind diverges, whereas lazy_share yields (2,2). Of course, real programs do not contain undefined or other intentionally divergent computations. We use undefined above to stand for an expensive search whose results are unused.

Alternatively, undefined above may stand for an expensive search that in the end fails to find any solution. If the rest of the program does not need any result from the search, then the shared search is not executed at all. Thus, if we replace undefined with mzero in the examples above, strict_bind would fail, whereas lazy_share would yield (2,2) as before.

3.1.3 Sharing recurs on non-deterministic components

We turn to data types that contain non-deterministic computations, such as List m a introduced in §2.3. We define two functions for illustration: the function first takes the first element of a List; the function dupl builds a List with the same two elements.

```
first :: MonadPlus m => m (List m a) -> m a
first l = l >>= \(Cons x xs) -> x

dupl :: Monad m => m a -> m (List m a)
dupl x = cons x (cons x nil)
```

The function dupl is subtly different from duplicate: whereas duplicate runs a computation twice and returns a data structure with the results, dupl returns a data structure containing the same computation twice without running it.

The following two examples illustrate the benefit of data structures with non-deterministic components.

```
heads_bind  = do x <- cons coin undefined
                 dupl (first (return x))

heads_share = do x <- share (cons coin undefined)
                 dupl (first x)
```

Despite the presence of undefined, the evaluation of both examples terminates and yields defined results. Since only the head of the list x is needed, the undefined tail of the list is not evaluated.

The expression cons coin undefined above denotes a deterministic computation that returns a data structure containing a non-deterministic computation coin. The monadic bind in heads_bind shares this data structure, coin and all, but not the result of coin. The monad laws entail that heads_bind yields cons coin (cons coin nil). When we later execute the latent computations (to print the result, for example), the two copies of coin will run independently and yield four outcomes [0,0], [0,1], [1,0], [1,1], so heads_bind is like dup_coin_let above. Informally, monadic bind performs only shallow sharing, which is not enough for data with non-deterministic components.

Our share combinator performs *deep* sharing: all components of a shared data structure are shared as well.[2] For example, the variable x in heads_share stands for a fully determined list with no latent non-determinism. Thus, heads_share yields only two outcomes, [0,0] and [1,1].

3.1.4 Sharing applies to unbounded data structures

Our final example involves a list of non-deterministic, unbounded length, whose elements are each also non-deterministic. The set of possible lists is infinite, yet non-strictness lets us compute with it.

```
coins :: MonadPlus m => m (List m Int)
coins = nil `mplus` cons coin coins

dup_first_coin = do cs <- share coins
                    dupl (first cs)
```

The non-deterministic computation coins yields every finite list of zeroes and ones. Unlike the examples above using undefined, each possible list is fully defined and finite, but there are an infinite number of possible lists, and generating each list requires an unbounded number of choices. Even though, as discussed above, the shared variable cs represents the fully determined result of such an unbounded number of choices, computing dup_first_coin only makes the few choices demanded by dupl (first cs). In particular, first cs represents the first element and is demanded twice, each time giving the same result, but no other element is demanded. Thus, dup_first_coin produces two results, [0,0] and [1,1].

3.2 The laws of sharing

We now formalize the intuitions illustrated above in a set of equational laws that hold up to observation as detailed in §3.3. We show

[2] Applying share to a function does not cause any non-determinism in its body to be shared. This behavior matches the intuition that invoking a function creates a copy of its body by substitution.

here how to use the laws to reason about—in particular, predict the results of—non-deterministic computations with `share`, such as the examples above. In §4, we further use the laws to guide an implementation.

The laws of our monad are shown in Figure 1. We write ret for `return`, $\emptyset$ for `mzero`, $\oplus$ for `` `mplus` ``, and $\bot$ for `undefined`.

First of all, our monad satisfies the monad laws, (Lret), (Rret) and (Bassc). Our monad is also a `MonadPlus` instance, but the laws for `MonadPlus` are not agreed upon (MonadPlus 2008); we include two commonly accepted laws, (Ldistr) and (Lzero). We do not however require that $\oplus$ be associative or that $\emptyset$ be a left or right unit of $\oplus$, so that our monad can be a weighted non-determinism monad, for which $\oplus$ means averaging weights. Weighted non-determinism is useful for probabilistic inference.

Using the laws in Figure 1, we can reduce a computation expression in our monad to an expression like $(\emptyset \oplus \cdots) \oplus (\mathsf{ret}\, v_1 \oplus \cdots)$, a (potentially infinite) tree whose branches are $\oplus$ and whose leaves are $\bot$, $\emptyset$, or $\mathsf{ret}\, v$. To observe the computation, we apply a function run to convert it to another `MonadPlus` instance such as the set monad. Figure 2 gives the laws of run. The right-hand sides use primes (ret', $\gg\!='$, $\emptyset'$, $\oplus'$) to refer to operations of the target monad.

Using the (Lret) and (Ldistr) laws, we can compute the result of the example `dup_coin_bind` above, which does not use `share`.

$$\begin{aligned}&(\mathsf{ret}\,0 \oplus \mathsf{ret}\,1) \gg\!= \lambda x.\, \mathsf{ret}\, x \gg\!= \lambda u.\, \mathsf{ret}\, x \gg\!= \lambda v.\, \mathsf{ret}\,(u,v)\\ &\quad = (\mathsf{ret}\,0 \oplus \mathsf{ret}\,1) \gg\!= \lambda x.\, \mathsf{ret}\,(x,x)\\ &\quad = (\mathsf{ret}\,0 \gg\!= \lambda x.\, \mathsf{ret}\,(x,x)) \oplus (\mathsf{ret}\,1 \gg\!= \lambda x.\, \mathsf{ret}\,(x,x))\\ &\quad = \mathsf{ret}\,(0,0) \oplus \mathsf{ret}\,(1,1)\end{aligned}$$

To show how the laws enforce call-time choice, we derive the same result for `dup_coin_share`, which is

$$\mathsf{share}\,(\mathsf{ret}\,0 \oplus \mathsf{ret}\,1) \gg\!= \lambda x.\, x \gg\!= \lambda u.\, x \gg\!= \lambda v.\, \mathsf{ret}\,(u,v).$$

We first use the (Choice) law to reduce $\mathsf{share}\,(\mathsf{ret}\,0 \oplus \mathsf{ret}\,1)$ to $\mathsf{share}\,(\mathsf{ret}\,0) \oplus \mathsf{share}\,(\mathsf{ret}\,1)$. To proceed further, we reduce $\mathsf{share}\,(\mathsf{ret}\,0)$ to $\mathsf{ret}\,(\mathsf{ret}\,0)$ (and $\mathsf{share}\,(\mathsf{ret}\,1)$ to $\mathsf{ret}\,(\mathsf{ret}\,1)$) using the (HNF) law (HNF is short for "head normal form"). In the law, c stands for a constructor with n non-deterministic components. Since 0 has no non-deterministic components, $n = 0$ and we have

$$\begin{aligned}&\mathsf{ret}\,(\mathsf{ret}\,0) \gg\!= \lambda x.\, x \gg\!= \lambda u.\, x \gg\!= \lambda v.\, \mathsf{ret}\,(u,v)\\ &\quad = \mathsf{ret}\,0 \gg\!= \lambda u.\, \mathsf{ret}\,0 \gg\!= \lambda v.\, \mathsf{ret}\,(u,v) = \mathsf{ret}\,(0,0)\end{aligned}$$

The overall result is thus the same as that for `dup_coin_bind`.

The preservation of non-strictness is illustrated by `lazy_share`. After reducing $\mathsf{const}\,(\mathsf{ret}\,2)\,x$ there to $\mathsf{ret}\,2$, we obtain

$$\mathsf{share}\,\bot \gg\!= \lambda x.\, \mathsf{duplicate}\,(\mathsf{ret}\,2).$$

Because x is unused, the result can be computed without evaluating the shared expression. And it is, as assured by the (Bot) law, which reduces $\mathsf{share}\,\bot$ to $\mathsf{ret}\,\bot$ (not to $\bot$). The (Lret) law then reduces the expression to $\mathsf{duplicate}\,(\mathsf{ret}\,2)$, and the final result is $\mathsf{ret}\,(2,2)$. The next section discusses (Bot) further. The (Fail) law works similarly.

We turn to data structures with non-deterministic components. Using the monad laws, the `heads_bind` example easily reduces to $\mathsf{ret}\,(\mathsf{Cons}\,\mathsf{coin}\,(\mathsf{ret}\,(\mathsf{Cons}\,\mathsf{coin}\,(\mathsf{ret}\,\mathsf{Nil}))))$, in which the constructor Cons takes two non-deterministic computations as arguments. Whereas applying run to observe results without non-deterministic components is a trivial matter of replacing $\emptyset$ by $\emptyset'$, $\oplus$ by $\oplus'$, and ret by ret' (so trivial as to be glossed over above), observing the result of `heads_bind` requires using the (rRet) law in Figure 2 in a non-trivial way, with c being Cons and n being 2. The result is

$$\begin{aligned}&\mathsf{run}\,\mathsf{coin} \gg\!=' \lambda y_1.\\ &\mathsf{run}\,(\mathsf{ret}\,(\mathsf{Cons}\,\mathsf{coin}\,(\mathsf{ret}\,\mathsf{Nil}))) \gg\!=' \lambda y_2.\\ &\mathsf{ret}'\,(\mathsf{Cons}\,(\mathsf{ret}'\,y_1)\,(\mathsf{ret}'\,y_2)),\end{aligned}$$

$$\mathsf{ret}\,x \gg\!= k = kx \qquad \text{(Lret)}$$

$$a \gg\!= \mathsf{ret} = a \qquad \text{(Rret)}$$

$$(a \gg\!= k_1) \gg\!= k_2 = a \gg\!= \lambda x.\, k_1 x \gg\!= k2 \qquad \text{(Bassc)}$$

$$\emptyset \gg\!= k = \emptyset \qquad \text{(Lzero)}$$

$$(a \oplus b) \gg\!= k = (a \gg\!= k) \oplus (b \gg\!= k) \qquad \text{(Ldistr)}$$

$$\mathsf{share}\,(a \oplus b) = \mathsf{share}\,a \oplus \mathsf{share}\,b \qquad \text{(Choice)}$$

$$\mathsf{share}\,\emptyset = \mathsf{ret}\,\emptyset \qquad \text{(Fail)}$$

$$\mathsf{share}\,\bot = \mathsf{ret}\,\bot \qquad \text{(Bot)}$$

$$\begin{aligned}\mathsf{share}\,(\mathsf{ret}\,(c\,x_1 \ldots x_n)) = {}& \mathsf{share}\,x_1 \gg\!= \lambda y_1 \ldots .\\ & \mathsf{share}\,x_n \gg\!= \lambda y_n.\, \mathsf{ret}\,(\mathsf{ret}\,(c\,y_1 \ldots y_n))\end{aligned} \qquad \text{(HNF)}$$

where c is a constructor with n non-deterministic components

Figure 1. The laws of a monad with non-determinism and sharing

$$\mathsf{run}\,\emptyset = \emptyset' \qquad \text{(rZero)}$$

$$\mathsf{run}\,(a \oplus b) = (\mathsf{run}\,a) \oplus' (\mathsf{run}\,b) \qquad \text{(rPlus)}$$

$$\begin{aligned}\mathsf{run}\,(\mathsf{ret}\,(c\,x_1 \ldots x_n)) = {}& \mathsf{run}\,x_1 \gg\!=' \lambda y_1 \ldots .\\ & \mathsf{run}\,x_n \gg\!=' \lambda y_n.\, \mathsf{ret}'\,(c\,(\mathsf{ret}'\,y_1) \ldots (\mathsf{ret}'\,y_n))\end{aligned} \qquad \text{(rRet)}$$

Figure 2. The laws of observing a monad with non-determinism and sharing in another monad with non-determinism

which eventually yields four solutions due to two independent observations of coin. In general, (rRet) ensures that the final observation yields only fully determined values.

To predict the result of `heads_share`, we need to apply the (HNF) law in a non-trivial way, with c being Cons and n being 2:

$$\begin{aligned}&\mathsf{share}\,(\mathsf{ret}\,(\mathsf{Cons}\,\mathsf{coin}\,\bot))\\ &= \mathsf{share}\,\mathsf{coin} \gg\!= \lambda y_1.\, \mathsf{share}\,\bot \gg\!= \lambda y_2.\, \mathsf{ret}\,(\mathsf{ret}\,(\mathsf{Cons}\,y_1\,y_2))\\ &= \mathsf{share}\,\mathsf{coin} \gg\!= \lambda y_1.\, \mathsf{ret}\,(\mathsf{ret}\,(\mathsf{Cons}\,y_1\,\bot))\\ &= \mathsf{share}\,(\mathsf{ret}\,0 \oplus \mathsf{ret}\,1) \gg\!= \lambda y_1.\, \mathsf{ret}\,(\mathsf{ret}\,(\mathsf{Cons}\,y_1\,\bot))\\ &= (\mathsf{share}\,(\mathsf{ret}\,0) \oplus \mathsf{share}\,(\mathsf{ret}\,1)) \gg\!= \lambda y_1.\, \mathsf{ret}\,(\mathsf{ret}\,(\mathsf{Cons}\,y_1\,\bot))\\ &= (\mathsf{share}\,(\mathsf{ret}\,0) \gg\!= \lambda y_1.\, \mathsf{ret}\,(\mathsf{ret}\,(\mathsf{Cons}\,y_1\,\bot)))\\ &\oplus (\mathsf{share}\,(\mathsf{ret}\,1) \gg\!= \lambda y_1.\, \mathsf{ret}\,(\mathsf{ret}\,(\mathsf{Cons}\,y_1\,\bot)))\\ &= \mathsf{ret}\,(\mathsf{ret}\,(\mathsf{Cons}\,(\mathsf{ret}\,0)\,\bot)) \oplus \mathsf{ret}\,(\mathsf{ret}\,(\mathsf{Cons}\,(\mathsf{ret}\,1)\,\bot))\end{aligned}$$

This derivation shows that applying share to $\mathsf{ret}\,(\mathsf{Cons}\,\mathsf{coin}\,\bot)$ exposes and lifts the latent choice coin in the list to the top level. Therefore, sharing a list that contains a choice is equivalent to sharing a choice of a list, so `heads_share` yields only two outcomes.

The (Bot) law and our discussion of `lazy_share` above suggest a more general law

$$\mathsf{share}\,a \gg\!= \lambda \_.\, b = b, \qquad \text{(Ignore)}$$

which says that any unused shared computation, not just $\bot$, can simply be skipped. This law implies that $\oplus$ is *idempotent*: $b \oplus b = b$. The proof of the implication is that

$$\begin{aligned}b \oplus b &= (\mathsf{share}\,(\mathsf{ret}\,0) \gg\!= \lambda \_.\, b) \oplus (\mathsf{share}\,(\mathsf{ret}\,1) \gg\!= \lambda \_.\, b)\\ &= \mathsf{share}\,\mathsf{coin} \gg\!= \lambda \_.\, b = b.\end{aligned}$$

Idempotence is justified if we observe a non-deterministic computation as a *set* of outcomes, that is, if we care only *whether* a particular result is produced, not how many times or in what order. This perspective on non-deterministic programming is popular; for instance, it is customary in FLP (López-Fraguas et al. 2007, 2008).

The (Ignore) law enables a simpler analysis of our last example program `dup_first_coin`, which creates an infinite number of choices but demands only a few of them. Without (Ignore), we can only reduce the program using (Choice) and (HNF) to $\emptyset \oplus (a \oplus b)$, where $a = d_0 \oplus (a \oplus a)$, $b = d_1 \oplus (b \oplus b)$, and

$$d_i = \mathsf{ret}\,(\mathsf{Cons}\,(\mathsf{ret}\,i)\,(\mathsf{ret}\,(\mathsf{Cons}\,(\mathsf{ret}\,i)\,(\mathsf{ret}\,\mathsf{Nil})))).$$

Using (Ignore), we can arrive at a simpler result with no duplicate solutions, namely $\emptyset \oplus (d_0 \oplus d_1)$.

3.3 Intuitions behind our laws

Call-time choice makes shared non-determinism feel like familiar call-by-value evaluation in monadic style, except $\emptyset$ and $\bot$ are treated like values. Indeed, the laws (Fail) and (Bot) would be subsumed by (HNF) if $\bot$ and $\emptyset$ were $\mathsf{ret}\,c$ for some c. The intuition of treating divergence like a value to express laziness guides standard formalizations of FLP (González-Moreno et al. 1999; López-Fraguas et al. 2007, 2008) that inspired our laws.

Still, for an equational law, (Bot) is unusual in two ways. First, (Bot) is not constructive: its left-hand side matches a computation that diverges, which is in general not decidable. Therefore, it does not correspond to a clause in the implementation of share, as we detail in §§4–5 below. Second, the function share is computable and thus monotonic in the domain-theoretic sense, so (Bot) entails that $\mathsf{share}\,a \geq \mathsf{share}\,\bot = \mathsf{ret}\,\bot$ for all a. In particular, we have by (Choice) that $\mathsf{share}\,a \oplus \mathsf{share}\,b = \mathsf{share}\,(a \oplus b) \geq \mathsf{ret}\,\bot$. How can a non-deterministic value $\mathsf{share}\,a \oplus \mathsf{share}\,b$ possibly be as defined as the deterministic value $\mathsf{ret}\,\bot$?

The key is that our laws hold only *up to observation*. That is, they only say that replacing certain expressions by others does not affect the (non)termination of well-typed programs[3] when the monad type constructors are held abstract (Hinze 2000; Lin 2006). Observing a computation in our monad requires applying run then observing the result in the target monad. Each of the two steps may identify many computations. For example, the order of choices may be unobservable because the target monad is the set monad.[4] Also, we may be unable to disprove that $\mathsf{share}\,a \oplus \mathsf{share}\,b$ is as defined as $\mathsf{ret}\,\bot$ because $\mathsf{run}\,(\mathsf{ret}\,\bot)$ diverges.

A positive example of our laws holding up to observation lies in Constructor-Based Rewriting Logic (CRWL) (González-Moreno et al. 1999), a standard formalization of FLP. To every term e (which we assume is closed in this informal explanation), CRWL assigns a denotation $[\![e]\!]$, the set of *partial values* that e can *reduce* to. A partial value is built up using constructors such as Cons and Nil, but any part can be replaced by $\bot$ to form a lesser value. A denotation is a downward-closed set of partial values.

López-Fraguas et al.'s Theorem 1 (2008) is a fundamental property of call-time choice. It states that, for every context C and term e, the denotation $[\![C[e]]\!]$ equals the denotation $\bigcup_{t \in [\![e]\!]} [\![C[t]]\!]$, i.e., the union of denotations of $C[t]$ where t is drawn from the denotation of e. Even if e is non-deterministic, the denotation of a large term that contains e can be obtained by considering each partial value e can reduce to. Especially, if e is an argument to a function that duplicates its argument, this argument denotes the same value wherever it occurs. The monadic operation $\oplus$ for non-deterministic choice resembles the CRWL operation `?` defined as follows:

```
x ? y -> x     x ? y -> y
```

Using the theorem above, we conclude that $[\![C[a?b]]\!] = [\![C[a]?C[b]]\!]$, which inspired our (Choice) law.

[3] without selective strictness via `seq`

[4] The set monad can be implemented in Haskell just like the list monad, with the usual `Monad` and `MonadPlus` instances that do not depend on `Eq` or `Ord`, as long as computations can only be observed using the `null` predicate.

4. Implementing lazy non-determinism

We start to implement `share` in this section. We begin with a very specific version and generalize it step by step. Revisiting the equational laws for `share`, we show how memoization can be used to achieve the desired properties. First, we consider values without non-deterministic components, namely values of type `Int`. We then extend the approach to non-deterministic components, namely lists of numbers. An implementation for arbitrary user-defined non-deterministic types in terms of a transformer for arbitrary instances of `MonadPlus` is given in §5.

4.1 The tension between late demand and early choice

Lazy evaluation means to evaluate expressions at most once and not until they are demanded. The law (Ignore) from the previous section, or more specifically, the laws (Fail) and (Bot) from Figure 1 formalize late demand. In order to satisfy these laws, we could be tempted to implement `share` as follows:

```
share :: Monad m => m a -> m (m a)
share a = return a
```

and so `share undefined` is trivially `return undefined`, just as the law (Bot) requires; (Fail) is similarly satisfied. But (Choice) fails, because $\mathsf{ret}\,(a \oplus b)$ is not equal to $\mathsf{ret}\,a \oplus \mathsf{ret}\,b$. For example, if we take `dup_coin_share` from §3.1.1 and replace `share` with `return`, we obtain `dup_coin_let`—which, as explained there, shares only a non-deterministic computation, not its result as desired. Instead of re-making the choices in a shared monadic value each time it is demanded, we must make the choices only once and reuse them for duplicated occurrences.

We could be tempted to try a different implementation of `share` that ensures that choices are performed immediately:

```
share :: Monad m => m a -> m (m a)
share a = a >>= \x -> return (return x)
```

This implementation satisfies the (Choice) law, but it does not satisfy the (Fail) and (Bot) laws. The (Lzero) law of `MonadPlus` shows that this implementation renders `share mzero` equal to `mzero`, which is observationally different from the `return mzero` required by (Fail). This attempt ensures early choice using early demand, so we get eager sharing, rather than lazy sharing as desired.

4.2 Memoization

We can combine late demand and early choice using memoization. The idea is to delay the choice until it is demanded, and to remember the choice when it is made for the first time so as to not make it again if it is demanded again.

To demonstrate the idea, we define a very specific version of `share` that fixes the monad and the type of shared values. We use a state monad to remember shared monadic values. A state monad is an instance of the following type class, which defines operations to query and update a threaded state component.

```
class MonadState s m where
  get :: m s
  put :: s -> m ()
```

In our case, the threaded state is a list of *thunks* that can be either unevaluated or evaluated.

```
data Thunk a = Uneval (Memo a) | Eval a
```

Here, `Memo` is the name of our monad. It threads a list of `Thunk`s through non-deterministic computations represented as lists.

```
newtype Memo a = Memo {
  unMemo :: [Thunk Int] -> [(a, [Thunk Int])] }
```

The instance declarations for the type classes `Monad`, `MonadState`, and `MonadPlus` are as follows:

```
instance Monad Memo where
  return x = Memo (\ts -> [(x,ts)])
  m >>= f  =
    Memo (concatMap (\(x,ts) -> unMemo (f x) ts)
         . unMemo m)

instance MonadState [Thunk Int] Memo where
  get    = Memo (\ts -> [(ts,ts)])
  put ts = Memo (\_  -> [((),ts)])

instance MonadPlus Memo where
  mzero       = Memo (const [])
  a `mplus` b =
    Memo (\ts -> unMemo a ts ++ unMemo b ts)
```

It is crucial that the thunks are passed to both alternatives separately in the implementation of `mplus`. The list of thunks thus constitutes a *first-class store* (Morrisett 1993)—using mutable global state to store the thunks would not suffice because thunks are created and evaluated differently in different non-deterministic branches.

We can implement a very specific version of `share` that works for integers in the `Memo` monad.

```
share :: Memo Int -> Memo (Memo Int)
share a = memo a

memo a =
 do thunks <- get
    let index = length thunks
    put (thunks ++ [Uneval a])
    return
     (do thunks <- get
         case thunks!!index of
           Eval x   -> return x
           Uneval a ->
            do x <- a
               thunks <- get
               let (xs,_:ys) = splitAt index thunks
               put (xs ++ [Eval x] ++ ys)
               return x)
```

This implementation of `share` adds an unevaluated thunk to the current store and returns a monadic action that, when executed, queries the store and either returns the already evaluated result or evaluates the unevaluated thunk before updating the threaded state. The argument `a` given to `share` is not demanded until the inner action is performed. Hence, this implementation of `share` satisfies the (Fail) and (Bot) laws. Furthermore, the argument is only evaluated once, followed by an update of the state to remember the computed value. Hence, this implementation of `share` satisfies the (Choice) law (up to observation, as defined in §4.4). If the inner action is duplicated and evaluated more than once, then subsequent calls will yield the same result as the first call due to memoization.

4.3 Non-deterministic components

The version of `share` just developed memoizes only integers. However, we want to memoize data with non-deterministic components, such as permuted lists that are computed on demand. So instead of thunks that evaluate to numbers, we redefine the `Memo` monad to store thunks that evaluate to lists of numbers now.

```
newtype Memo a = Memo {
  unMemo :: [Thunk (List Memo Int)]
         -> [(a, [Thunk (List Memo Int)])] }
```

The instance declarations for `Monad` and `MonadPlus` stay the same. In the `MonadState` instance only the state type needs to be adapted. We also reuse the `memo` function, which has now a different type. We could try to define `share` simply as a renaming for `memo` again:

```
share :: Memo (List Memo Int)
      -> Memo (Memo (List Memo Int))
share a = memo a
```

However, with this definition lists are not shared deeply. This behavior corresponds to the expression `heads_bind` where the head and the tail of the demanded list are still executed whenever they are demanded and may hence yield different results when duplicated. This implementation does not satisfy the (HNF) law.

We can remedy this situation by recursively memoizing the head and the tail of a shared list:

```
share :: Memo (List Memo Int)
      -> Memo (Memo (List Memo Int))
share a = memo (do l <- a
                   case l of
                     Nil       -> nil
                     Cons x xs -> do y  <- share x
                                     ys <- share xs
                                     cons y ys)
```

This implementation of `share` memoizes data containing non-deterministic components as deeply as demanded by the computation. Each component is evaluated at most once and memoized individually in the list of stored thunks.[5]

4.4 Observing non-deterministic results

In order to observe the results of a computation that contains non-deterministic components, we need a function (such as `run` in Figure 2) that evaluates all the components and combines the resulting alternatives to compute a non-deterministic choice of deterministic results. For example, we can define a function `eval` that computes all results from a non-deterministic list of numbers.

```
eval :: List Memo Int -> Memo (List Memo Int)
eval Nil         = return Nil
eval (Cons x xs) =
 do y  <- x  >>= eval
    ys <- xs >>= eval
    return (Cons (return y) (return ys))
```

The lists returned by `eval` are fully determined. Using `eval`, we can define an operation `run` that computes the results of a non-deterministic computation:

```
run :: Memo (List Memo Int) -> [List Memo Int]
run m = map fst (unMemo (m >>= eval) [])
```

In order to guarantee that the *observed* results correspond to *predicted* results according to the laws in §3.2, we place two requirements on the monad used to observe the computation (`[]` above). (In contrast, the laws in §3.2 constrain the monad used to express the computation (`Memo` above).)

Idempotence of mplus The (Choice) law predicts that the computation $\mathsf{run}\,(\mathsf{share}\,\mathsf{coin} \gg\!= \lambda\_.\,\mathsf{ret}\,\mathsf{Nil})$ gives $\mathsf{ret}'\,\mathsf{Nil} \oplus' \mathsf{ret}'\,\mathsf{Nil}$. However, our implementation gives a single solution $\mathsf{ret}'\,\mathsf{Nil}$ (following the (Ignore) law, as it turns out). Hence, we require $\oplus'$ to be *idempotent*; that is, $m \oplus' m = m$.

This requirement is satisfied if we abstract from the multiplicity of results (considering `[]` as the set monad rather than the list

[5] This implementation of `share` does not actually type-check because `share x` in the body needs to invoke the previous version of `share`, for the type `Int`, rather than this version, for the type `List Memo Int`. The two versions can be made to coexist, each maintaining its own state, but we develop a polymorphic `share` combinator in §5 below, so the issue is moot.

monad), as is common practice in FLP, or if we treat $\oplus'$ as averaging the weights of results, as is useful for probabilistic inference.

Distributivity of bind over mplus According to the (Choice) law, the result of the computation

$$\begin{aligned}&\mathsf{run}\,(\mathsf{share}\,\mathsf{coin} \gg\!= \lambda c.\,\mathsf{coin} \gg\!= \lambda y.\,c \gg\!= \lambda x.\\ &\quad \mathsf{ret}\,(\mathsf{Cons}\,(\mathsf{ret}\,x)\,(\mathsf{ret}\,(\mathsf{Cons}\,(\mathsf{ret}\,y)\,(\mathsf{ret}\,\mathsf{Nil})))))\end{aligned}$$

is the following non-deterministic choice of lists (we write $\langle x,y\rangle$ to denote $\mathsf{ret}'\,(\mathsf{Cons}\,(\mathsf{ret}'\,x)\,(\mathsf{ret}'\,(\mathsf{Cons}\,(\mathsf{ret}'\,y)\,(\mathsf{ret}'\,\mathsf{Nil})))))$.

$$(\langle 0,0\rangle \oplus' \langle 0,1\rangle) \oplus' (\langle 1,0\rangle \oplus' \langle 1,1\rangle)$$

However, our implementation yields

$$(\langle 0,0\rangle \oplus' \langle 1,0\rangle) \oplus' (\langle 0,1\rangle \oplus' \langle 1,1\rangle).$$

In order to equate these two trees, we require the following distributive law between $\gg\!='$ and $\oplus'$.

$$a \gg\!=' \lambda x.\,(f\,x \oplus' g\,x) = (a \gg\!=' f) \oplus' (a \gg\!=' g)$$

If the observation monad satisfies this law, then the two expressions above are equal (we write coin' to denote $\mathsf{ret}'\,0 \oplus' \mathsf{ret}'\,1$):

$$\begin{aligned}&(\langle 0,0\rangle \oplus' \langle 0,1\rangle) \oplus' (\langle 1,0\rangle \oplus' \langle 1,1\rangle)\\ &\quad = (\mathsf{coin}' \gg\!=' \lambda y.\,\langle 0,y\rangle) \oplus' (\mathsf{coin}' \gg\!=' \lambda y.\,\langle 1,y\rangle)\\ &\quad = \mathsf{coin}' \gg\!=' \lambda y.\,(\langle 0,y\rangle \oplus' \langle 1,y\rangle)\\ &\quad = (\langle 0,0\rangle \oplus' \langle 1,0\rangle) \oplus' (\langle 0,1\rangle \oplus' \langle 1,1\rangle).\end{aligned}$$

Hence, the intuition behind distributivity is that the observation monad does not care about the order in which choices are made. This intuition captures the essence of implementing call-time choice: we can perform choices on demand and the results are as if we performed them eagerly.

In general, it is fine to use our approach with an observation monad that does not match our requirements, as long as we are willing to abstract from the mismatch. For example, the list monad satisfies neither idempotence nor distributivity, yet our equational laws are useful in combination with the list monad if we abstract from the order and multiplicities of results. We also do not require that $\oplus'$ be associative or that $\emptyset'$ be a left or right unit of $\oplus'$.

5. Generalized, efficient implementation

In this section, we generalize the implementation ideas described in the previous section such that

1. arbitrary user-defined types with non-deterministic components can be passed as arguments to the combinator `share`, and
2. arbitrary instances of `MonadPlus` can be used as the underlying search strategy.

We achieve the first goal by introducing a type class with the interface to process non-deterministic data. We achieve the second goal by defining a monad transformer `Lazy` that adds sharing to any instance of `MonadPlus`. After describing a straightforward implementation of this monad transformer, we show how to implement it differently in order to improve performance significantly.

Both of these generalizations are motivated by practical applications in non-deterministic programming.

1. The ability to work with user-defined types makes it easier to compose deterministic and non-deterministic code and to draw on the sophisticated type and module systems of existing functional languages.
2. The ability to plug in different underlying monads makes it possible to express techniques such as breadth-first search (Spivey 2000), heuristics, constraint solving (Nordin and Tolmach 2001), and weighted results.

For example, we have applied our approach to express and sample from probability distributions as OCaml programs in direct style (Filinski 1999). With less development effort than state-of-the-art systems, we achieved comparable concision and performance (Kiselyov and Shan 2009).

The implementation of our monad transformer is available as a Hackage package at: `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/explicit-sharing-0.1`

5.1 Non-deterministic data

We have seen in the previous section that in order to share nested, non-deterministic data deeply, we need to traverse it and apply the combinator `share` recursively to every non-deterministic component. We have implemented deep sharing for the type of non-deterministic lists, but want to generalize this implementation to support arbitrary user-defined types with non-deterministic components. It turns out that the following interface to non-deterministic data is sufficient:

```
class MonadPlus m => Nondet m a where
  mapNondet
    :: (forall b . Nondet m b => m b -> m (m b))
    -> a -> m a
```

A non-deterministic type `a` with non-deterministic components wrapped in the monad `m` can be made an instance of `Nondet m` by implementing the function `mapNondet`, which applies a monadic transformation to each non-deterministic component. The type of `mapNondet` is a rank-2 type: the first argument is a polymorphic function that can be applied to non-deterministic data of any type.

We can make the type `List m Int`, of non-deterministic number lists, an instance of `Nondet` as follows.

```
instance MonadPlus m => Nondet m Int where
  mapNondet _ c           = return c

instance Nondet m a => Nondet m (List m a) where
  mapNondet _ Nil         = return Nil
  mapNondet f (Cons x xs) = do y  <- f x
                               ys <- f xs
                               return (Cons y ys)
```

The implementation mechanically applies the given transformation to the non-deterministic arguments of each constructor. In fact the implementation is so regular that we believe it can be easily automated using generic programming (Lämmel and Peyton Jones 2003) or Template Haskell. We plan to investigate this possibility.

An example for the use of `mapNondet` is the following operation, which computes the fully determined values from a non-deterministic value.

```
eval :: Nondet m a => a -> m a
eval = mapNondet (\a -> a>>=eval>>=return.return)
```

This operation generalizes the specific version for lists given in §4.4. In order to determine a value, we determine values for the arguments and combine the results. The bind operation of the monad nicely takes care of the combination.

Our original motivation for abstracting over the interface of non-deterministic data was to define the operation `share` with a more general type. In order to generalize the type of `share` to allow not only different types of shared values but also different monad type constructors, we define another type class.

```
class MonadPlus m => Sharing m where
  share :: Nondet m a => m a -> m (m a)
```

Non-determinism monads that support the operation `share` are instances of this class. We next define an instance of `Sharing` with the implementation of `share` for arbitrary non-deterministic types.

5.2 State monad transformer

The implementation of memoization in §4 uses a state monad to thread a list of thunks through non-deterministic computations. The straightforward generalization is to use a state monad transformer to thread thunks through computations in arbitrary monads. A state monad transformer adds the operations defined by the type class `MonadState` to an arbitrary base monad.

The type for `Thunk`s generalizes easily to an arbitrary monad:

```
data Thunk m a = Uneval (m a) | Eval a
```

Instead of using a list of thunks, we use a `ThunkStore` with the following interface. Note that the operations `lookupThunk` and `insertThunk` deal with thunks of arbitrary type.

```
emptyThunks :: ThunkStore
getFreshKey :: MonadState ThunkStore m => m Int
lookupThunk :: MonadState ThunkStore m
            => Int -> m (Thunk m a)
insertThunk :: MonadState ThunkStore m
            => Int -> Thunk m a -> m ()
```

There are different options to implement this interface. We have implemented thunk stores using the generic programming features provided by the `Data.Typeable` and `Data.Dynamic` modules but omit corresponding class contexts for the sake of clarity.

Lazy monadic computations can now be performed in a monad that threads a `ThunkStore`. We obtain such a monad by applying the `StateT` monad transformer to an arbitrary instance of `MonadPlus`.

```
type Lazy m = StateT ThunkStore m
```

For any instance `m` of `MonadPlus`, the type constructor `Lazy m` is an instance of `Monad`, `MonadPlus`, and `MonadState ThunkStore`. We only need to define the instance of `Sharing` ourselves, which implements the operation `share`.

```
instance MonadPlus m => Sharing (Lazy m) where
  share a = memo (a >>= mapNondet share)
```

The implementation of `share` uses the operation `memo` to memoize the argument and the operation `mapNondet` to apply `share` recursively to the non-deterministic components of the given value. The function `memo` resembles the specific version given in §4.2 but has a more general type.

```
memo :: MonadState ThunkStore m => m a -> m (m a)
memo a =
 do key <- getFreshKey
    insertThunk key (Uneval a)
    return (do thunk <- lookupThunk key
               case thunk of
                 Eval x   -> return x
                 Uneval b ->
                  do x <- b
                     insertThunk key (Eval x)
                     return x)
```

The only difference in this implementation of `memo` from before is that it uses more efficient thunk stores instead of lists of thunks.

In order to observe a lazy non-deterministic computation, we use the functions `eval` to compute fully determined values and `evalStateT` to execute actions in the transformed state monad.

```
run :: Nondet (Lazy m) a => Lazy m a -> m a
run a = evalStateT (a >>= eval) emptyThunks
```

This function is the generalization of the `run` function to arbitrary data types with non-deterministic components that are expressed in an arbitrary instance of `MonadPlus`.

This completes an implementation of our monad transformer for lazy non-determinism, with all of the functionality motivated in §§2–3.

5.3 Optimizing performance

We have applied some optimizations that improve the performance of our implementation significantly. We use the permutation sort in §2 for a rough measure of performance. The implementation just presented exhausts the search space for sorting a list of length 20 in about 5 minutes.[6] The optimizations described below reduce the run time to 7.5 seconds. All implementations run permutation sort in constant space (5 MB or less) and the final implementation executes permutation sort on a list of length 20 roughly three times faster than the fastest available compiler for Curry, the Münster Curry Compiler (MCC).

As detailed below, we achieve this competitive performance by

1. reducing the amount of pattern matching in invocations of the monadic bind operation,
2. reducing the number of store operations when storing shared results, and
3. manually inlining and optimizing library code.

5.3.1 Less pattern matching

The `Monad` instance for the `StateT` monad transformer performs pattern matching in every call to `>>=` in order to thread the store through the computation. This is wasteful especially during computations that do not access the store because they do not perform explicit sharing. We can avoid this pattern matching by using a different instance of `MonadState`.

We define the continuation monad transformer `ContT`:[7]

```
newtype ContT m a = C {
  unC :: forall w . (a -> m w) -> m w }

runContT :: Monad m => ContT m a -> m a
runContT m = unC m return
```

We can make `ContT m` an instance of the type class `Monad` without using operations from the underlying monad `m`:

```
instance Monad (ContT m) where
  return x = C (\c -> c x)
  m >>= k  = C (\c -> unC m (\x -> unC (k x) c))
```

An instance for `MonadPlus` can be easily defined using the corresponding operations of the underlying monad. The interesting exercise is to define an instance of `MonadState` using `ContT`. When using continuations, a reader monad—a monad where actions are functions that take an environment as input but do not yield one as output—can be used to pass state. More specifically, we need the following operations of reader monads:

```
ask   :: MonadReader s m => m s
local :: MonadReader s m => (s -> s) -> m a -> m a
```

The function `ask` queries the current environment, and the function `local` executes a monadic action in a modified environment. In combination with `ContT`, the function `local` is enough to implement state updates:

```
instance Monad m
      => MonadState s (ContT (ReaderT s m)) where
  get   = C (\c -> ask >>= c)
  put s = C (\c -> local (const s) (c ()))
```

[6] We performed our experiments on an Apple MacBook with a 2.2 GHz Intel Core 2 Duo processor using GHC with optimizations (`-O2`).

[7] This implementation differs from the definition shipped with GHC in that the result type `w` for continuations is higher-rank polymorphic.

With these definitions, we can define our monad transformer `Lazy`:

```
type Lazy m = ContT (ReaderT ThunkStore m)
```

We can reuse from §5.2 the definition of the `Sharing` instance and of the `memo` function used to define `share`.

After this optimization, searching all sorted permutations of a list of length 20 takes about 2 minutes rather than 5.

5.3.2 Fewer state manipulations

The function `memo` just defined performs two state updates for each shared value that is demanded: one to insert the unevaluated shared computation and one to insert the evaluated result. We can save half of these manipulations by inserting only evaluated head-normal forms and using lexical scope to access unevaluated computations. We use a different interface to stores now, again abstracting away the details of how to implement this interface in a type-safe manner.

```
emptyStore  :: Store
getFreshKey :: MonadState Store m => m Int
lookupHNF   :: MonadState Store m
            => Int -> m (Maybe a)
insertHNF   :: MonadState Store m
            => Int -> a -> m ()
```

Based on this interface, we can define a variant of `memo` that only stores evaluated head normal forms.

```
memo :: MonadState Store m => m a -> m (m a)
memo a =
 do key <- getFreshKey
    return (do hnf <- lookupHNF key
               case hnf of
                 Just x  -> return x
                 Nothing -> do x <- a
                               insertHNF key x
                               return x)
```

Instead of retrieving a thunk from the store on demand if it is not yet evaluated, we can use the action `a` directly because it is in scope. As a consequence, `a` cannot be garbage collected as long as the computation returned by `share` is reachable, which is a possible memory leak. We did not experience memory problems during our experiments, however. After this optimization, searching all sorted permutations of a list of length 20 takes 1.5 minutes rather than 2.

5.3.3 Mechanical simplifications

The final optimization is to

1. expand the types in `ContT (ReaderT State m)`,
2. inline all definitions of monadic operations,
3. simplify them according to monad laws, and
4. provide a specialized version of `memo` that is not overloaded.

This optimization, like the previous ones, affects only our library code and not its clients; for instance, we did not inline any definitions into our benchmark code. Afterwards, searching all sorted permutations of a list of length 20 takes 7.5 seconds rather than 1.5 minutes. This is the most impressive speedup during this sequence of optimizations, even though it is completely mechanical and should ideally be performed by the compiler.

Surprisingly, our still high-level and very modular implementation (it works with arbitrary monads for non-determinism and arbitrary types for nested, non-deterministic data) outperforms the fastest available Curry compiler. A Curry program for permutation sort, equivalent to the program we used for our benchmarks, runs for 25 seconds when compiled with MCC and `-O2` optimizations.

We have also compared our performance on *deterministic* monadic computations against corresponding non-monadic programs in Haskell and Curry. Our benchmark is to call the naive reverse function on long lists, which involves a lot of deterministic pattern-matching. In this benchmark, the monadic code is roughly 20% faster than the corresponding Curry code in MCC. The overhead compared to a non-monadic Haskell program is about the same order of magnitude.

Our library does not directly support narrowing and unification of logic variables but can emulate it by means of lazy non-determinism. We have measured the overhead of such emulation using a functional logic implementation of the `last` function:

```
last l | l =:= xs ++ [x] = x where x,xs free
```

This Curry function uses narrowing to bind `xs` to the spine of the init of `l` and unification to bind `x` and the elements of `xs` to the elements of `l`. We can translate it to Haskell by replacing `x` and `xs` with non-deterministic generators and implementing the unification operator `=:=` as equality check. When applying `last` to a list of determined values, the monadic Haskell code is about six times faster than the Curry version in MCC. The advantage of unification shows up when `last` is applied to a list of logic variables: in Curry, `=:=` can unify two logic variables deterministically, while an equality check on non-deterministic generators is non-deterministic and leads to search-space explosion. More efficient unification could be implemented using an underlying monad of equality constraints.

All programs used for benchmarking are available under: `http://github.com/sebfisch/explicit-sharing/tree/0.1.1`

6. Related work

In this section we compare our work to foundational and practical work in various communities. We refer to other approaches to implementing monads for logic programming. We also point out similarities to the problems solved for hardware description languages.

6.1 Functional logic programming

The interaction of non-strict and non-deterministic evaluation has been studied in the FLP community, leading to different semantic frameworks and implementations. They all establish *call-time choice*, which ensures that computed results correspond to strict evaluation. An alternative interpretation of call-time choice is that variables denote *values* rather than (possibly non-deterministic) computations. As call-time choice has turned out to be the most intuitive model for lazy non-determinism, we also adopt it.

Unlike approaches discussed below, however, we do not define a new programming language but implement our approach in Haskell. In fact, functional logic programs in Curry or Toy can be compiled to Haskell programs that use our library.

Semantic frameworks There are different approaches to formalizing the semantics of FLP. CRWL (González-Moreno et al. 1999) is a proof calculus with a denotational flavor that allows to reason about functional logic programs using inference rules and to prove program equivalence. Let Rewriting (López-Fraguas et al. 2007, 2008) defines rewrite rules that are shown to be equivalent to CRWL. It is more operational than CRWL but does not define a constructive strategy to evaluate programs. Deterministic procedures to run functional logic programs are described by Albert et al. (2005) in the form of operational big-step and small-step semantics.

We define equational laws for monadic, lazy, non-deterministic computations that resemble let rewriting in that they do not fix an evaluation strategy. However, we provide an efficient implementation of our equational specification that can be executed using an arbitrary `MonadPlus` instance. Hence, our approach is a step towards closing the gap between let rewriting and the operational semantics, as it can be seen as a monadic let calculus that can be executed but does not fix a search strategy.

Implementations There are different compilers for FLP languages that are partly based on the semantic frameworks discussed above. Moreover, the operational semantics by Albert et al. (2005) has been implemented as Haskell interpreters by Tolmach and Antoy (2003) and Tolmach et al. (2004). We do not define a compiler that translates an FLP language; nor do we define an interpreter in Haskell. We rather define a monadic language for lazy FLP *within* Haskell. Instead of defining data types for every language construct as the interpreters do, we only need to define new types for data with non-deterministic components. Instead of using an untyped representation for non-deterministic data, our approach is typed.

This tight integration with Haskell lets us be much more efficient than is possible using an interpreter. The KiCS compiler from Curry to Haskell (Braßel and Huch 2009) also aims to exploit the fact that many functional logic programs contain large deterministic parts. Unlike our approach, KiCS does not use monads to implement sharing but generates unique identifiers using impure features that prevent compiler optimizations on the generated Haskell code.

Naylor et al. (2007) implement a library for functional logic programming in Haskell which handles logic variables explicitly and can hence implement a more efficient version of unification. It does not support data types with non-deterministic components or user-defined search strategies. The authors discuss the conflict between laziness and non-determinism in §5.4 without resolving it.

Monad transformers Hinze (2000) derived monad transformers for backtracking from equational specifications. Spivey (2000) and Kiselyov et al. (2005) improved the search strategy in monadic computations to avoid the deficiencies of depth-first search. However, we are the first to introduce *laziness* in non-deterministic computations modeled using monads in Haskell. Any instance of `MonadPlus`—including those developed in these works—can be used in combination with our approach.

6.2 Call-by-need calculi

The combination of sharing and non-strictness—known as call-by-need or lazy evaluation—has been extensively investigated theoretically. The first "natural" semantics for call-by-need evaluation (Launchbury 1993; Seaman 1993) both rely on heaps, which store either evaluated or unevaluated bindings of variables. Later, Ariola et al. (1995), Ariola and Felleisen (1997), and Maraist et al. (1998) proposed call-by-need calculi to avoid the explicit heap: Maraist et al.'s sequence of let-bindings and Ariola and Felleisen's binding context play the role of a heap but use bindings present in the original program instead of creating fresh heap references.

The calculi developed equational theories to reason about call-by-need programs. However, the laws presented in these calculi are quite different from ours (Figure 1). Although Ariola et al. add constructors as an extension of their calculus, constructed values cannot have non-value components. To construct data lazily, one must explicitly let-bind computations of all components, however deeply nested. They do not have an analogue of our (HNF) law. Maraist et al. briefly discuss an extension for constructed values with non-value components; their law V^K corresponds to our law (HNF). Our laws (Choice), (Fail) and (Bot) are not reflected in any call-by-need calculus. Ariola et al. mention our law (Ignore) as a potential addition (adopted by Maraist et al. later). Unlike these call-by-need calculi, we do not need a special syntactic category of answers, since we introduce the notion of observation (Figure 2).

Since our implementations are based on store passing, they closely correspond to Launchbury's natural semantics (1993). Evaluating `memo a` returns a computation that behaves like a variable reference in Launchbury's semantics. Evaluating the variable reference for the first time evaluates the associated term and updates the store by binding the variable to the resulting value. The main difference of our evaluator is non-determinism. We cannot get by with a single global heap—we need first-class stores (Morrisett 1993), one for each branch of the non-deterministic computation.

Garcia et al. (2009) recently reduced call-by-need (again, without non-determinism) to stylized uses of delimited continuations. In particular, they simulate call-by-need in a call-by-value calculus with delimited control. We have similarly (Kiselyov and Shan 2009) embedded lazy probabilistic programs (Koller et al. 1997) in OCaml, a call-by-value language with delimited control. Like Garcia et al., we use first-class control delimiters to represent shared variables on the heap. A useful (Goodman et al. 2008) and straightforward generalization is to memoize probabilistic functions.

6.3 Hardware description languages

The problem of explicit sharing has also been addressed in the context of hardware description languages (Acosta-Gómez 2007; Bjesse et al. 1998). In order to model a circuit as an algebraic data type in a purely functional language, one needs to be able to identify shared nodes. The survey by Acosta-Gómez (2007; §2.4.1) discusses four different solutions to this problem:

Explicit labels clutter the code with identifiers that are—apart from expressing sharing—unrelated to the design of a circuit. Moreover, the programmer is responsible for passing unique labels in order to correctly model the nodes of a circuit.

State monads can be used to automate the creation of unique labels. However, writing monadic code is considered such a major paradigm shift in the context of circuit description that, for example, Lava (Bjesse et al. 1998) resorts to the next solution.

Observable sharing is the preferred solution because it maintains the usual recursive structure of the circuit description, but it requires impure features that often make it extremely complicated to reason about or debug programs (de Vries 2009).

Source transformations can also label the nodes of a circuit automatically. For example, Template Haskell can be used to add unique labels at compile time to unlabeled circuit descriptions.

Observable sharing is very similar to the approach used currently in KiCS (Braßel and Huch 2009). The problem of impure features—especially their hindering compiler optimizations—seems much more severe in FLP than in hardware description. As non-deterministic computations are usually expressed monadically in Haskell anyway, there is no paradigm shift necessary to use our monadic approach to sharing. It integrates smoothly by introducing a new operation to share the results of monadic computations.

7. Conclusions

We have presented an equational specification and an efficient implementation of non-strictness, sharing, and non-determinism embedded in a pure functional language.

Our specification (Figure 1) formalizes *call-time choice*, a combination of these three features that has been developed in the FLP community. This combination is intuitive and predictable because the results of computations resemble results of corresponding eager computations and shared variables represent fully determined values as opposed to possibly non-deterministic computations. Our equational laws for lazy non-determinism can be used to reason about the meaning of non-deterministic programs on a high level. They differ from previous formalizations of call-time choice, which use proof calculi, rewriting, or operational semantics. We describe intuitively the correspondence of López-Fraguas et al.'s formalizations (2007, 2008) with our laws as well as why our implementation satisfies them. A more formal treatment is left as future work.

Our implementation is novel in working with custom monadic data types and search strategies; in expressing the sharing of non-deterministic choices explicitly; and in implementing the sharing using first-class stores of typed data.

Our high-level monadic interface was crucial in order to optimize our implementation as described in §5.3. Initial comparisons of monadic computations with corresponding computations in Curry that use non-determinism, narrowing, and unification are very promising. We outperform the currently fastest Curry compiler (MCC) on the highly non-deterministic permutation sort algorithm. In our deterministic benchmark we incur acceptable overhead compared to pure Haskell. Simulated narrowing turned out competitive while simulated unification can lead to search space explosion. Our results suggest that our work can be used as a simple, high-level, and efficient implementation target for FLP languages.

Acknowledgments

We thank Greg Morrisett for pointing us to first-class stores, and Bernd Braßel and Michael Hanus for examining drafts of this paper.

References

Acosta-Gómez, Alfonso. 2007. Hardware synthesis in ForSyDe. Master's thesis, Dept. of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, Sweden.

Albert, Elvira, Michael Hanus, Frank Huch, Javier Oliver, and German Vidal. 2005. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation* 40(1):795–829.

Antoy, Sergio, and Michael Hanus. 2002. Functional logic design patterns. In *FLOPS*, 67–87.

———. 2006. Overlapping rules and logic variables in functional logic programs. In *ICLP*, 87–101.

Ariola, Zena M., and Matthias Felleisen. 1997. The call-by-need lambda calculus. *JFP* 7(3):265–301.

Ariola, Zena M., Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. 1995. The call-by-need lambda calculus. In *POPL*, 233–246.

Bird, Richard, Geraint Jones, and Oege de Moor. 1997. More haste, less speed: Lazy versus eager evaluation. *JFP* 7(5):541–547.

Bjesse, Per, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware design in Haskell. In *ICFP*, 174–184.

Braßel, Bernd, and Frank Huch. 2009. The Kiel Curry System KiCS. In *WLP 2007*, 195–205.

Christiansen, Jan, and Sebastian Fischer. 2008. EasyCheck – test data for free. In *FLOPS*, 322–336.

Claessen, Koen, and John Hughes. 2000. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP*, 268–279.

Escardó, Martín H. 2007. Infinite sets that admit fast exhaustive search. In *LICS*, 443–452.

Felleisen, Matthias. 1985. Transliterating Prolog into Scheme. Tech. Rep. 182, Computer Science Dept., Indiana University.

Filinski, Andrzej. 1999. Representing layered monads. In *POPL*, 175–188.

Fischer, Sebastian, and Herbert Kuchen. 2007. Systematic generation of glass-box test cases for functional logic programs. In *PPDP*, 63–74.

Garcia, Ronald, Andrew Lumsdaine, and Amr Sabry. 2009. Lazy evaluation and delimited control. In *POPL*, 153–164.

González-Moreno, Juan Carlos, Maria Teresa Hortalá-González, Francisco Javier López-Fraguas, and Mario Rodríguez-Artalejo. 1999. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming* 40(1):47–87.

Goodman, Noah, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A language for generative models. In *UAI*, 220–229.

Haynes, Christopher T. 1987. Logic continuations. *Journal of Logic Programming* 4(2):157–176.

Hinze, Ralf. 2000. Deriving backtracking monad transformers (functional pearl). In *ICFP*, 186–197.

Hudak, Paul. 1996. Building domain-specific embedded languages. *ACM Computing Surveys* 28(4es):196.

Hughes, John. 1989. Why functional programming matters. *The Computer Journal* 32(2):98–107.

Kiselyov, Oleg, and Chung-chieh Shan. 2009. Embedded probabilistic programming. In *Working conf. on domain specific lang.*

Kiselyov, Oleg, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers (functional pearl). In *ICFP*, 192–203.

Koller, Daphne, David McAllester, and Avi Pfeffer. 1997. Effective Bayesian inference for stochastic programs. In *AAAI*, 740–747.

Lämmel, Ralf, and Simon L. Peyton Jones. 2003. Scrap your boilerplate: A practical design pattern for generic programming. In *TLDI*, 26–37.

Launchbury, John. 1993. A natural semantics for lazy evaluation. In *POPL*, 144–154.

Lin, Chuan-kai. 2006. Programming monads operationally with Unimo. In *ICFP*, 274–285.

López-Fraguas, Francisco Javier, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. 2007. A simple rewrite notion for call-time choice semantics. In *PPDP*, 197–208.

———. 2008. Rewriting and call-time choice: The HO case. In *FLOPS*, 147–162.

Maraist, John, Martin Odersky, and Philip Wadler. 1998. The call-by-need lambda calculus. *JFP* 8(3):275–317.

McCarthy, John. 1963. A basis for a mathematical theory of computation. In *Computer programming and formal systems*, 33–70. North-Holland.

Michie, Donald. 1968. "Memo" functions and machine learning. *Nature* 218:19–22.

MonadPlus. 2008. MonadPlus. `http://www.haskell.org/haskellwiki/MonadPlus`.

Morrisett, J. Gregory. 1993. Refining first-class stores. In *Proceedings of the workshop on state in programming languages*.

Naylor, Matthew, Emil Axelsson, and Colin Runciman. 2007. A functional-logic library for Wired. In *Haskell workshop*, 37–48.

Nicollet, Victor, et al. 2009. Lazy and threads. `http://caml.inria.fr/pub/ml-archives/caml-list/2009/02/9fc4e4a897ce7a356674660c8cfa5ac0.fr.html`.

Nordin, Thomas, and Andrew Tolmach. 2001. Modular lazy search for constraint satisfaction problems. *JFP* 11(5):557–587.

Rabin, Michael O., and Dana Scott. 1959. Finite automata and their decision problems. *IBM Journal of Research and Development* 3:114–125.

Runciman, Colin, Matthew Naylor, and Fredrik Lindblad. 2008. SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In *Haskell symposium*, 37–48.

Seaman, Jill M. 1993. An operational semantics of lazy evaluation for analysis. Ph.D. thesis, Pennsylvania State University.

Spivey, J. Michael. 2000. Combinators for breadth-first search. *JFP* 10(4):397–408.

Tolmach, Andrew, and Sergio Antoy. 2003. A monadic semantics for core Curry. In *WFLP*, 33–46. Valencia, Spain.

Tolmach, Andrew, Sergio Antoy, and Marius Nita. 2004. Implementing functional logic languages using multiple threads and stores. In *ICFP*, 90–102.

de Vries, Edsko. 2009. Just how unsafe is unsafe. `http://www.haskell.org/pipermail/haskell-cafe/2009-February/055201.html`.

Wadler, Philip L. 1985. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *FPCA*, 113–128.