JFP **21** (4 & 5): 413–465, 2011. © Cambridge University Press 2011
doi:10.1017/S0956796811000189 First published online 16 August 2011

Purely functional lazy nondeterministic programming

SEBASTIAN FISCHER, OLEG KISELYOV and CHUNG-CHIEH SHAN
(*e-mail:* oleg@okmij.org)

Abstract

Functional logic programming and probabilistic programming have demonstrated the broad benefits of combining laziness (nonstrict evaluation with sharing of the results) with nondeterminism. Yet these benefits are seldom enjoyed in functional programming because the existing features for nonstrictness, sharing, and nondeterminism in functional languages are tricky to combine. We present a practical way to write purely functional lazy nondeterministic 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 nondeterministic components, by working with custom monadic data types and search strategies, and by providing equational laws for the programmer to reason about their code.

1 Introduction

Nonstrict evaluation, sharing, and nondeterminism are all valuable features in functional programming. Nonstrict 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 & Scott's Turing-award paper (1959), nondeterminism has been applied to model checking, testing (Claessen & Hughes, 2000), probabilistic inference, and search.

These features are each available in mainstream functional languages. A call-by-value language can typically model nonstrict 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. Nondeterminism can be achieved using `amb` (McCarthy, 1963), threads, or first-class continuations (Felleisen, 1985; Haynes, 1987). In a nonstrict language like Haskell, nondeterminism 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, Section 2.4.1).

These features are particularly useful together. For instance, sharing the results of nonstrict 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 Section 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 (Fischer & Kuchen, 2007; Christiansen & Fischer, 2008; Runciman *et al.*, 2008) as well as probabilistic inference (Koller *et al.*, 1997; Goodman *et al.*, 2008).

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 nondeterminism 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 nondeterminism in Haskell loses nonstrictness (see Section 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 nonstrictness with any effect.

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

1.1 Contributions

We embed lazy nondeterminism with call-time choice into mainstream functional languages in a shallow way (Hudak, 1996), rather than, say, building a Curry interpreter in Haskell (Tolmach & 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 using overloading via type classes. 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, nondeterministic rewriting.

The key novelty of our work is that nonstrictness, sharing, and nondeterminism have not been combined in such a general way before in purely functional programming. Nonstrictness and nondeterminism can be combined using data types with nondeterministic components such that a top-level constructor can be computed without fixing its arguments. However, such an encoding defeats Haskell's built-in sharing mechanism because a piece of nondeterministic 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 nondeterministic data structures that can be used in nonstrict, nondeterministic computations with explicit sharing annotations. Furthermore, we provide a Template Haskell library

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

to automatically derive the definitions of such nondeterministic data structures and their basic operations from conventional data type declarations.

Our 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 nondeterministic generators (Antoy & Hanus, 2006).

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 nondeterminism, narrowing, and unification. The complete code, along with many tests, is available on Hackage as package `explicit-sharing-0.9`.

1.2 Structure of the paper

We strove to make the paper accessible not only to Haskell programmers interested in nondeterminism, but also to Prolog and, in general, logic programmers interested in Haskell. Therefore, the paper begins by reviewing how to represent nondeterminism in a pure functional language and how to reason about such purely functional nondeterministic programs. In Section 2, we describe nonstrictness, sharing, and nondeterminism and why they are useful together. We also show that their naive combination is problematic, to motivate the explicit sharing of nondeterministic computations. In Section 3, we clarify the intuitions of sharing and introduce equational laws to reason about lazy nondeterminism. Section 4 develops an easy-to-understand implementation in several steps. Subtle issues of observing the results of nondeterministic computations are discussed in Section 5. Section 6 generalizes and speeds up the simple implementation and describes the automatic derivation of nondeterministic data types with corresponding type-class instances. In Section 7, we explain sharing across nondeterminism and how to avoid repeated sharing. We review related work in Section 8 and then conclude. The Appendix gives a complete example of using our library, including the definition of an interactive eval-print-like loop.

2 Nonstrictness, sharing, and nondeterminism

In this section, we describe nonstrictness, sharing, and nondeterminism and explain why combining them is useful and nontrivial. For completeness, we first describe representing nondeterminism in a pure functional language and illustrate equational reasoning about nondeterministic programs.

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 []       = True
```

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

```
isSorted [_]      = True
isSorted (x:y:ys) = (x <= y) && isSorted (y:ys)
```

In a nonstrict language, such as Haskell, the arguments to a function are evaluated only as far as needed (or, demanded) by the body of the function to compute the result. 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).

In a nonstrict language, variables can be bound to not yet evaluated expressions rather than only fully evaluated values like in a call-by-value language. This allows to evaluate arguments of functions only when they are needed. If a variable occurs several times in the body of a function, the bound expression is duplicated and may have to be evaluated several times. For example, the function `iterate` duplicates the variable `x`, which occurs twice in its body. Nonstrict languages can be distinguished by how they handle duplicated variables. In a language with a *call-by-name* evaluation strategy, duplicated unevaluated expressions are evaluated independently each time a duplicated occurrence is demanded. For example, the call `iterate (‘div‘2) (factorial 100)` yields a list that starts with the elements `factorial 100` and `factorial 100 ‘div‘ 2`. When passing this list to `isSorted` in a language with a call-by-name strategy, the expression `factorial 100` is evaluated twice. A *call-by-need* or *lazy* evaluation strategy prevents such duplicated evaluation, ensuring that each expression bound to a variable is evaluated at most once—even if the variable occurs more than once. In a lazy language, `iterate (‘div‘2) (factorial 100)` builds a list in which all occurrences of `factorial 100` are *shared* but none is evaluated. If we pass such a list to `isSorted`, the value of `factorial 100` is computed only once. This property—called *sharing*—makes lazy evaluation strictly more efficient than nonstrict evaluation without sharing. The controlled use of destructive updates represented by sharing also allows efficiency gains over purely functional eager programs, as demonstrated by Bird *et al.* (1997).

2.2 Nondeterminism

Programming nondeterministically can simplify the declarative formulation of an algorithm. As logic programming languages such as Prolog and Curry have shown, the expressive power of nondeterminism simplifies programs because different

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

nondeterministic results can be viewed individually rather than as members of a set of possible results (Antoy & Hanus, 2002).

In this section, we discuss nondeterminism and how it can be represented in the pure functional language Haskell. We add example programs in Curry to document the difference between a language with built-in nondeterminism and nondeterminism modeled in a deterministic language.

In a nondeterministic language like Curry, expressions evaluate to any of possibly many values. For example, the Curry declaration

```
coin = 0
coin = 1
```

defines a (zero-argument) function `coin` that yields `0` or `1` when evaluated.

Nondeterministic expressions can be combined using built-in or user-defined functions. Such combinations are themselves nondeterministic. For example, in Curry, the expression `1+coin` yields `1` or `2` nondeterministically and `coin+coin` yields `0`, `1`, or `2`.

To model nondeterministic computations in a pure functional language, the implicit computational effect of nondeterminism must be made explicit. Haskell adopts a style of programming popularized by Wadler (1992) in which effectful computations are expressed in a domain-specific monadic sublanguage embedded into a pure functional language. Nondeterministic computations are expressed using monads that are instances of the type class `MonadPlus`.

We now briefly review using monads to model nondeterminism. We recall that a monad `m` is a type constructor that provides two polymorphic operations:

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

Following Moggi (1989), we call monadic values *computations*. 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`, which builds a bigger computation out of `e1` and `e2`. That bigger computation performs (or executes) `e1`, binds the resulting value to the variable `x`, and then executes `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 nondeterministically between two computations.

One of the implementations of `MonadPlus` is a list monad (Wadler, 1985): `return` builds a singleton list, `mzero` is an empty list, and `mplus` is list concatenation. We

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

encourage the reader however *not* to think of `MonadPlus m => m a` values as lists. One should think of them as expressions in a small language of nondeterminism, whose expression forms are `return`, `(>>=)`, `mzero`, and `mplus`; that language is embedded in Haskell. We should reason about nondeterministic expressions using the equational laws explained below rather than in terms of any particular implementation.

The Curry definition of `coin` shown above can be translated to Haskell using `return` and `mplus`:

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

We have translated the two Curry definition clauses into arguments of `mplus` in a single Haskell clause and wrapped the integers with calls to `return`. The type of `coin` is `m Int` for an arbitrary `MonadPlus` instance `m`; that is, `coin` is a nondeterministic computation that *yields*, or results in, an integer. In Curry, the type of `coin` is just `Int` and nondeterminism is implicit.

The following Curry program demonstrates how to reason about nondeterministic computations.

```
coin' | x+y > 0 = x
 where x = coin
       y = coin
```

This function binds two variables `x` and `y` to the results of calls to `coin` and yields the result of `x` if at least one of the variables is bound to `1`. To reason about possible results of this function, we check all possible combinations of bindings for `x` and `y` and observe that both `0` and `1` are possible results of `coin'`.

To translate `coin'` into Haskell, we use `do`-notation instead of a `where` clause to bind the variables. The convenience function `guard` is *semideterministic*, meaning that it yields at most one result: it yields `()` if its argument is `True` and fails otherwise.

```
coin' :: MonadPlus m => m Int
coin' = do x <- coin
           y <- coin
           guard (x+y > 0)
           return x

guard :: MonadPlus m => Bool -> m ()
guard b = if b then return () else mzero
```

To reason about the Haskell version of the `coin'` function, we can pick a specific instance of `MonadPlus`, inline the definitions of the monadic operations, and compute the results by evaluating the resulting expression using Haskell's reduction rules. A more high-level approach, which is independent of a specific instance of `MonadPlus`, is to reason via equational laws. `MonadPlus` instances need to satisfy the monad laws as well as additional laws for the interaction of `mzero` and `mplus` with the `>>=` combinator. Figure 1 shows the laws, writing ret for `return`, $\emptyset$ for `mzero` and

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

$$\mathsf{ret}\ x \gg\!= k = k\ x \qquad \text{(Lret)}$$
$$a \gg\!= \mathsf{ret} = a \qquad \text{(Rret)}$$
$$(a \gg\!= k_1) \gg\!= k_2 = a \gg\!= \lambda x.\, k_1 x \gg\!= k_2 \qquad \text{(Bassc)}$$
$$\emptyset \gg\!= k = \emptyset \qquad \text{(Lzero)}$$
$$(a \oplus b) \gg\!= k = (a \gg\!= k) \oplus (b \gg\!= k) \qquad \text{(Ldistr)}$$

Fig. 1. The laws of a monad with nondeterminism.

$\oplus$ for '`mplus`'. The additional laws, involving `mzero` and `mplus`, are not agreed upon (MonadPlus 2008). Figure 1 shows our choice: the inclusion of (Lzero) and (Ldistr) and omitting monoid laws for $\emptyset$ and $\oplus$. Because of the (Ldistr) law, we do *not* regard `Maybe` as a monad for nondeterminism (despite Haskell's `MonadPlus` instance for `Maybe`), since in the `Maybe` monad, $(\mathsf{ret}\ \mathsf{False} \oplus \mathsf{ret}\ \mathsf{True}) \gg\!= \mathsf{guard}$ equals $\emptyset$ instead of $\emptyset \oplus \mathsf{ret}\,()$.

To use these laws to reason about the monadic version of `coin'`, we desugar the `do`-notation and apply the laws of Figure 1:

$$\begin{aligned}
&\mathsf{coin} \gg\!= \lambda x.\, \mathsf{coin} \gg\!= \lambda y.\, \mathsf{guard}\,(x + y > 0) \gg\!= \lambda_.\, \mathsf{ret}\ x \\
&\quad = (\mathsf{ret}\,0 \oplus \mathsf{ret}\,1) \gg\!= \lambda x.\, \mathsf{coin} \gg\!= \lambda y.\, \mathsf{guard}\,(x + y > 0) \gg\!= \lambda_.\, \mathsf{ret}\ x \quad \text{(Def. coin)} \\
&\quad = (\mathsf{ret}\,0 \gg\!= \lambda x.\, \mathsf{coin} \gg\!= \lambda y.\, \mathsf{guard}\,(x + y > 0) \gg\!= \lambda_.\, \mathsf{ret}\ x) \\
&\qquad \oplus (\mathsf{ret}\,1 \gg\!= \lambda x.\, \mathsf{coin} \gg\!= \lambda y.\, \mathsf{guard}\,(x + y > 0) \gg\!= \lambda_.\, \mathsf{ret}\ x) \qquad \text{(Ldistr)} \\
&\quad = (\mathsf{coin} \gg\!= \lambda y.\, \mathsf{guard}\,(0 + y > 0) \gg\!= \lambda_.\, \mathsf{ret}\,0) \\
&\qquad \oplus (\mathsf{coin} \gg\!= \lambda y.\, \mathsf{guard}\,(1 + y > 0) \gg\!= \lambda_.\, \mathsf{ret}\,1) \qquad \text{(Lret)} \\
&\quad = ((\mathsf{guard}\,(0 + 0 > 0) \gg\!= \lambda_.\, \mathsf{ret}\,0) \oplus (\mathsf{guard}\,(0 + 1 > 0) \gg\!= \lambda_.\, \mathsf{ret}\,0)) \\
&\qquad \oplus ((\mathsf{guard}\,(1 + 0 > 0) \gg\!= \lambda_.\, \mathsf{ret}\,1) \oplus (\mathsf{guard}\,(1 + 1 > 0) \gg\!= \lambda_.\, \mathsf{ret}\,1)) \\
&\qquad\qquad \text{(Def. coin, Ldistr, Lret)} \\
&\quad = ((\emptyset \gg\!= \lambda_.\, \mathsf{ret}\,0) \oplus (\mathsf{ret}\,() \gg\!= \lambda_.\, \mathsf{ret}\,0)) \\
&\qquad \oplus ((\mathsf{ret}\,() \gg\!= \lambda_.\, \mathsf{ret}\,1) \oplus (\mathsf{ret}\,() \gg\!= \lambda_.\, \mathsf{ret}\,1)) \qquad \text{(Def. guard)} \\
&\quad = (\emptyset \oplus \mathsf{ret}\,0) \oplus (\mathsf{ret}\,1 \oplus \mathsf{ret}\,1) \qquad \text{(Lzero, Lret)}
\end{aligned}$$

We call equational reasoning steps that involve the monad laws (such as those in Figure 1) *execution* of a nondeterministic computation. We tacitly use other equational laws such as α, β, η-conversions, arithmetic, and pair construction and projections.

As a more complex example for a nondeterministic program, we consider a function that yields a permutation of a given list nondeterministically. First, we show the Curry version, where nondeterminism is implicit.

```
perm :: [a] -> [a]
perm []     = []
perm (x:xs) = insert x (perm xs)

insert :: a -> [a] -> [a]
insert x xs     = x : xs
insert x (y:ys) = y : insert x ys
```

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

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. In Curry, any matching clause of a defined operation is applied, not necessarily the first matching clause as in Haskell where the first clause of `insert` would catch all applications.

The Haskell version of `perm` takes a list as argument and returns a nondeterministic computation that yields a permutation of this list.

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

The function `insert` shown below takes a list as second argument. Since `perm xs` is a list-yielding computation rather than a list, we cannot pass it to `insert` directly; we have to build a bigger computation that first performs the computation `perm xs`, binds the result to `ys`, and then performs the computation `insert x ys`. The function `insert` receives as its second argument the result of the `perm xs` computation rather than the computation itself. Our translation therefore corresponds to a call-by-value strategy rather than the call-by-need strategy of Curry—a difference that will become important below.

```
insert :: MonadPlus m => a -> [a] -> m [a]
insert x xs = insert_head `mplus` insert_tail xs
 where insert_head        = return (x:xs)
       insert_tail []     = mzero
       insert_tail (y:ys) = do zs <- insert x ys
                               return (y:zs)
```

To translate `insert` into monadic style, we merge the different clauses of Curry's `insert` into a single clause using `mplus`, with a helper function `insert_tail` to pattern-match on `xs`. We explicitly yield `mzero` if the argument to `insert_tail` is empty, which corresponds to pattern match failure in Curry.

2.3 Sharing of nondeterminism

Nondeterminism is especially useful when formulating search algorithms. Following the *generate-and-test* pattern, we can find solutions to a search problem by nondeterministically 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).

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

We demonstrate this technique with a toy example, permutation sort,[1] which motivates the combination of nonstrictness, sharing, and nondeterminism. Below is a simple declarative specification of sorting in Curry. In words, to sort a list is to compute one of its sorted permutations.

```
sort :: [Int] -> [Int]
sort xs | isSorted ys = ys
 where ys = perm xs
```

This definition of `sort` makes crucial use of call-time choice: On one hand, the result `ys` of `perm` is evaluated only to the extent demanded by `isSorted`. (The function `isSorted` in Curry looks like the one in Haskell, Section 2.1.) In this sense, `ys` can be seen as a computation that yields a permutation nondeterministically and is executed during the evaluation of `isSorted`. On the other hand, the second occurrence of `ys` on the right-hand side of `sort` denotes *the same* permutation as was passed to `isSorted`. In this sense, `ys` can be seen as a single, but not yet evaluated, permutation.

A naive Haskell encoding of `sort` uses `do`-notation to bind the result of `perm`, and the `guard` function to select a sorted permutation.

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

Unfortunately, this Haskell program is grossly inefficient compared to the Curry program above. Using the list monad, it takes about a second to sort 10 elements, more than 10 s to sort 11 elements, and more than 3 min to sort 12 elements. The reason for the inefficiency is that, as explained above, the employed monadic encoding corresponds to call-by-value, rather than call-by-need, evaluation. The function `isSorted` takes a list of integers. To obtain such a list, `sort` must perform the computation `perm xs` first. However, `isSorted` can reject a permutation as soon as two elements are out of order, without knowing the rest of the permutation. The naive Haskell encoding of Curry's `perm` offers no way to compute the permuted list partly, to the needed extent. As a result, many nondeterministic choices are performed needlessly, leading to the observed inefficiency.

In short, the usual, naive monadic encoding of nondeterminism in Haskell loses nonstrictness. In the next two sections, we develop an alternative monadic encoding of nondeterminism that models the nonstrictness of nondeterministic Curry executions more closely.

[1] Permutation sort is a standard example in the FLP community to demonstrate call-time choice because it is a small program that combines nondeterminism, nonstrictness, and sharing in a nontrivial way. It is frequently used to test the performance of Curry systems.

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

2.4 Retaining nonstrictness by sacrificing sharing

The naive monadic encoding of nondeterminism loses nonstrictness because the arguments to a constructor cannot be monadic. For example, the second argument of the constructor for ordinary lists

```
(:) :: a -> [a] -> [a]
```

has the type `[a]` rather than `m [a]`. We cannot use this constructor to build lists with a nondeterministic spine, that is, a list whose tail is a nondeterministic computation (a monadic value).

To overcome this limitation, we redefine data structures so that their components may be nondeterministic. A data type for lists with nondeterministic 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)
```

Unlike `(:)`, `cons` and `Cons` take monadic values as arguments. We redefine the nonstrict `isSorted` to test nondeterministic lists:

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

We define a lazier version of the monadic permutation algorithm that generates lists with nondeterministic components.

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

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

Unlike the previous version in Section 2.2, this version of `perm` takes as argument a computation that yields a list with monadic components. In the following, we call such computations nondeterministic lists. A crucial difference from the previous version of `perm` is that we no longer execute the recursive call of `perm` in order to pass the result to the operation `insert`, because `insert` now takes a nondeterministic 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` insert_tail mxs
 where insert_tail mxs = do Cons my mys <- mxs
                            cons my (insert mx mys)
```

The operation `insert` *either* creates a new list with the nondeterministic `mx` in front of the nondeterministic `mxs` *or* inserts `mx` somewhere in the tail of `mxs`. Unlike the previous version of `insert`, this version creates a nondeterministic list with a nondeterministic spine. Also, note that the pattern match in the `do`-expression binding is nonexhaustive. If the computation `mxs` yields `Nil`, the pattern-match failure is a failing computation.

Our second attempt at permutation sort in Haskell now checks lazily 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 an arbitrary permutation of its input, not necessarily a sorted one. This is because the shared variable `ys` in the new definition of `sort` is bound to a monadic value `m (List m Int)`, which is a nondeterministic *computation* yielding a permutation of the input. In the previous version of `sort`, we bound `ys` (using `do`-notation) to a (deterministic) result of a nondeterministic permutation computation. The new `sort` makes sure that there is a sorted permutation but then yields an arbitrary permutation. In other words, this monadic encoding of nondeterminism corresponds to nondeterministic call-by-name, rather than call-by-need, evaluation.[2]

In short, the presence of nondeterministic components in data structures complicates nondeterministic programming, for example, expressing Curry algorithms in Haskell. In Curry, variables denote *values*, fully determined if not yet fully computed. In the Curry version of `sort`, both occurrences of the variable `ys` denote the same result of the permutation function. A faithful encoding of lazy nondeterminism must preserve this intuition. In order for the monadic `sort` to work, the shared nondeterministic computation `ys`, used twice in `sort`, must yield the same result each time.

[2] In contrast to a deterministic language, a call-by-name evaluation strategy in a nondeterministic language can lead to more results than a call-by-need strategy because nondeterminism in duplicated expressions is executed independently.

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

2.5 Explicit sharing

Our new approach to nondeterminism is lazy in that it preserves both nonstrictness and sharing. We provide a combinator `share`, which can be used explicitly to introduce a variable that stands for fully determined values. One may regard the application `share m` as an explicit sharing annotation on the expression `m`. The combinator `share` has the signature[3]

```
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 Sections 4–6.) 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 nondeterminism

In this section, we formalize the `share` combinator and specify equational laws with which a programmer can reason about nondeterministic 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 we designed `share` to do.

3.1 The intuition of sharing

We demonstrate the intuition of sharing using the function `duplicate`, which executes 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
```

[3] In fact, the signature has additional class constraints (see Section 6).

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

Table 1. *Three ways to bind the result of a nondeterministic computation*

computation	number of results	`coin` is executed...
`dup_coin_let`	4	... twice inside `duplicate`
`dup_coin_bind`	2	... once before `duplicate`
`dup_coin_share`	2	... once inside `duplicate`

The programs `dup_coin_let` and `dup_coin_bind` do not use `share`, so we can reason about them as we did for `coin'` in Section 2.2: using the laws of `MonadPlus` in Figure 1. The differences among all three programs are summarized in Table 1.

The program `dup_coin_let` binds the variable `x` to the nondeterministic computation `coin`. Substituting `coin` for `x` and then for the argument of `duplicate` gives us

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

which, by the (Ldistr) and (Lret) laws, is equal to

$$(\mathsf{ret}\,(0,0) \oplus \mathsf{ret}\,(0,1)) \oplus (\mathsf{ret}\,(1,0) \oplus \mathsf{ret}\,(1,1)),$$

with four so-called primitive choices, each a pair of numbers. The function `duplicate` executes its argument `x` twice—performing two independent coin flips.

In contrast, `dup_coin_bind` binds `x` to the result of the `coin` computation rather than to `coin` itself. Whereas in `dup_coin_let`, the variable `x` is of the type `m Int`, in `dup_coin_bind` it is of type `Int`. Applying the equational laws to `dup_coin_bind` gives a different expression,

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

which, by the (Lret) law, is equal to

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

We no longer substitute `coin` in the body of `duplicate`; rather, we substitute `return x`, which is a *deterministic* computation, whose two executions yield the same result. Hence, the expression `dup_coin_bind` equals $\mathsf{ret}\,(0,0) \oplus \mathsf{ret}\,(1,1)$.

On one hand, `dup_coin_share` resembles `dup_coin_let` in that the variable `x` is of the type `m Int`, so `x` is bound to a computation. Unlike in `dup_coin_let`, however, `x` is bound to a *shared* `coin` computation, which behaves quite like `return x` in `dup_coin_bind`. As in Sections 2.4 and 2.5, we wish to share the results of computations, and we wish shared computations to denote determined 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` is equal to `dup_coin_bind`.

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

3.1.2 Sharing preserves nonstrictness

The sorting example in Section 2 shows how nonstrictness can improve performance significantly. Here, we illustrate nonstrictness with two shorter examples that demonstrate that, unlike monadic bind, `share` is lazy although, as seen above, it has a similar effect on the number of results as monadic bind:

```
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)
```

Whereas `strict_bind` is equal to `undefined` (that is, diverges), `lazy_share` is equal to $\mathsf{ret}(2,2)$. 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, by the law (Lzero) `strict_bind` would become equal to `mzero` (that is, fail), but `lazy_share` would still be equal to $\mathsf{ret}\langle 2,2\rangle$ after this change.

3.1.3 Intuitions of observation

So far, we have been content to show that `dup_coin_bind` equals $\mathsf{ret}(0,0) \oplus \mathsf{ret}(1,1)$, equating an expression to a tree of primitive choices. Our library provides a (partial) function results that can be used to collect the results of a nondeterministic computation with sharing in a set, from which the results can be extracted and printed, for example. This function lets us *observe* the nondeterministic computations with sharing. (We will see in Section 3.4 that results is an instance of the general observation function collect, which is not limited to sets.)

Our implementation restricts observable computations in two ways:

1. The computation can only use the `MonadPlus` operations and `share`. No other operations can enter sharing computations.
2. The results must not contain any reference to the sharing monad. Rather, they must have the form of *answers*, that is, trees of primitive choices.

In Section 5, we discuss how we use the type system to enforce these restrictions.

A consequence of the second restriction is that we can only observe fully determined results without nondeterministic components. We cannot observe a call to the `share` function, which yields a computation, or to our `sort` function, which yields a nondeterministic list (see Section 2.4), because the results of these functions are not admissible as answers. In order to pass the result of the `sort` function to results, we need to convert it into a list without nondeterministic components, for example by using the following function.

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

```
convertList :: Monad m => List m a -> m [a]
convertList Nil           = return []
convertList (Cons mx mxs) = do x <- mx
                               xs <- mxs >>= convertList
                               return (x:xs)
```

It is the responsibility of users of the library to apply such a conversion function prior to observing the result of a sharing computation. To relieve users from writing conversions, we provide below an overloaded conversion function that works for arbitrary types with nondeterministic components (Section 6.2) and a way to automatically derive instances for specific types (Section 6.3).

Applying `convertList` forces the execution of any nondeterminism hidden in the components of a nondeterministic list. For example, `cons coin (cons coin nil)` is a deterministic computation producing a list with two nondeterministic components, each of which is an independent coin flip. We cannot observe such a list directly, but we can observe `cons coin (cons coin nil) >>= convertList`, as a set {`[0,0]`,`[0,1]`,`[1,0]`,`[1,1]`}.

The Appendix shows a complete example of how to use the library by defining nondeterministic data-types, monadic operations on them, and calling results to enumerate converted results. It also shows how to define a print loop that resembles the eval-print loops of interactive Curry systems.

3.1.4 Sharing recurs on nondeterministic components

We now explain how `share` acts on values with nondeterministic components, as in the type `List m a` introduced in Section 2.4. 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 = do Cons x xs <- l
             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` executes a computation twice and returns a data structure with the results, `dupl` does not execute the given computation but returns a data structure with two copies of the computation.

The following two examples illustrate the benefit of data structures with nondeterministic components.

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

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

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

Despite the presence of `undefined`, neither expression is equal to `undefined`; both yield defined results. Since only the head of the list `x` is demanded, the undefined tail of the list is not executed.

The expression `cons coin undefined` above denotes a deterministic computation that returns a data structure containing a nondeterministic 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` is equal to `cons coin (cons coin nil)`, containing two copies of `coin`. Observing `heads_bind >>= convertList` executes these latent computations and gives the set `{[0,0],[0,1],[1,0],[1,1]}`. Therefore, `heads_bind` is like `dup_coin_let` above. Informally, monadic bind performs only shallow sharing, which is not enough for data with nondeterministic components.

Our `share` combinator performs *deep* sharing: all components of a shared data structure are shared as well.[4] For example, the variable `x` in `heads_share` stands for a fully determined list with no latent nondeterminism. Observing `heads_share >>= convertList` thus gives only the set `{[0,0],[1,1]}`.

3.1.5 Sharing applies to unbounded data structures

Our final example involves a list of nondeterministic, unbounded length, all of whose elements are also nondeterministic. The set of possible lists is infinite, yet nonstrictness 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 nondeterministic 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 every list requires an unbounded number of choices. Even though, as discussed above, the shared variable `cs` represents a 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, applying `results` to observe `dup_first_coin >>= convertList` gives the set `{[0,0],[1,1]}`. As we will see, `dup_first_coin` equals `dupl mzero `mplus` heads_share`.

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

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

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

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

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

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

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

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

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

$$\mathsf{share}\ a \gg\!= \lambda x.\, b = b \qquad \text{where } x \text{ is not free in } b \qquad \text{(Ignore)}$$

$$\mathsf{share}\ (\mathsf{share}\ e \gg\!= k) = \mathsf{share}\ e \gg\!= \mathsf{share} \circ k \qquad \text{(Flat)}$$

$$\mathsf{share}\ a \gg\!= \lambda x.\, k\ x\ (\mathsf{share}\ x) = \mathsf{share}\ a \gg\!= \lambda x.\, k\ x\ (\mathsf{ret}\ x) \qquad \text{(Repeat)}$$

$$\mathsf{share}\ a \gg\!= \lambda x.\, (f\ x \oplus g\ x) = (\mathsf{share}\ a \gg\!= f) \oplus (\mathsf{share}\ a \gg\!= g) \qquad \text{(Rdistr)}$$

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

where c is a constructor with n nondeterministic components

Fig. 2. The laws of a monad with nondeterminism and sharing.

3.2 The laws of sharing

As demonstrated in Section 2.2, the laws for nondeterminism monads (Figure 1) let us reason about nondeterministic computations and in particular predict their results. This section introduces additional laws for the interaction of monadic operations with `share`, formalizing the intuitions developed above. The new laws are shown in Figure 2, where we also repeat the laws for nondeterminism monads to ease reference. In this section, we show how to use the laws to reason about nondeterministic computations with `share`, such as the examples in Section 3.1. In Section 4, we further use the laws to guide an implementation.

In Section 3.1, we used the laws (Lret) and (Ldistr) of Figure 1 to deduce that `dup_coin_bind` equals $\mathsf{ret}\,(0,0) \oplus \mathsf{ret}\,(1,1)$. To show how the laws of Figure 2 entail 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, which entails that $\mathsf{share}\ (\mathsf{ret}\ 0 \oplus \mathsf{ret}\ 1)$ equals $\mathsf{share}\ (\mathsf{ret}\ 0) \oplus \mathsf{share}\ (\mathsf{ret}\ 1)$. The (HNF) law then further equates $\mathsf{share}\ (\mathsf{ret}\ 0)$ to $\mathsf{ret}\ (\mathsf{ret}\ 0)$ and $\mathsf{share}\ (\mathsf{ret}\ 1)$ to $\mathsf{ret}\ (\mathsf{ret}\ 1)$. In the (HNF) law (HNF is short for "head normal form"), c stands for a constructor with n nondeterministic components. Since the constructor 0 has no nondeterministic components, by setting n to 0, we have

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

and similarly for the $\mathsf{share}\,(\mathsf{ret}\,1)$ case. We thus obtain that `dup_coin_share` equals `dup_coin_bind`.

Generally, we cannot simplify the computation $e \gg\!= \lambda u.\, e \gg\!= \lambda v.\, \mathsf{ret}\ (u, v)$ to $e \gg\!= \lambda u.\, \mathsf{ret}\ u \gg\!= \lambda v.\, \mathsf{ret}\ (u, v)$ and hence to $e \gg\!= \lambda u.\, \mathsf{ret}\ (u, u)$. For example, if e is

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

`coin`, as in `dup_coin_let`, then this transformation is invalid. However, it is valid to replace e with $\mathsf{ret}\,u$ if we know e is deterministic, such as $\mathsf{ret}\,x$ for some x. Intuitively, a computation created by `share` behaves like a deterministic computation. Accordingly, the (Choice) law allows to identify shared nondeterministic computations with their results.

The preservation of nonstrictness is illustrated by `lazy_share` in Section 3.1.2. After simplifying $\mathsf{const}\,(\mathsf{ret}\,2)\,x$ there to $\mathsf{ret}\,2$, we obtain

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

Because x is unused, we apply the (Ignore) law, giving us $\mathsf{duplicate}\,(\mathsf{ret}\,2)$, and the final result is $\mathsf{ret}\,(2,2)$.

A more interesting example of preserving nonstrictness is the following seemingly "left-recursive" code.

$$\mathsf{zeros} \stackrel{\text{def}}{=} \mathsf{share}\,\mathsf{zeros} \gg\!= \lambda x.\,\mathsf{ret}\,0 \oplus x$$

This code is not unusual: if we replace share with ret, we get a computation that yields infinitely many choices, each zero. The computation zeros is the same. More precisely, let us prove that $\mathsf{zeros} = \mathsf{ret}\,0 \oplus \mathsf{zeros}$. To start, we use the law (Rdistr), followed by (Ignore) for one of the resulting choices.

$$\begin{aligned}
\mathsf{zeros} &= \mathsf{share}\,\mathsf{zeros} \gg\!= \lambda x.\,\mathsf{ret}\,0 \oplus x && \text{(Definition of zeros)}\\
&= (\mathsf{share}\,\mathsf{zeros} \gg\!= \lambda x.\,\mathsf{ret}\,0) \oplus (\mathsf{share}\,\mathsf{zeros} \gg\!= \lambda x.\,x) && \text{(Rdistr)}\\
&= \mathsf{ret}\,0 \oplus (\mathsf{share}\,\mathsf{zeros} \gg\!= \mathsf{id}) && \text{(Ignore)}
\end{aligned}$$

(We cannot apply (Ignore) to zeros immediately because the result of $\mathsf{share}\,\mathsf{zeros}$ is used on the right-hand side of bind—but only in some choices.) We examine $\mathsf{share}\,\mathsf{zeros} \gg\!= \mathsf{id}$:

$$\begin{aligned}
&\mathsf{share}\,\mathsf{zeros} \gg\!= \mathsf{id}\\
&\quad= \mathsf{share}(\mathsf{share}\,\mathsf{zeros} \gg\!= \lambda x.\,\mathsf{ret}\,0 \oplus x) \gg\!= \mathsf{id} && \text{(Definition of zeros)}\\
&\quad= \mathsf{share}\,\mathsf{zeros} \gg\!= \lambda x.\,\mathsf{share}(\mathsf{ret}\,0 \oplus x) \gg\!= \mathsf{id} && \text{(Flat)}\\
&\quad= \mathsf{share}\,\mathsf{zeros} \gg\!= \lambda x.\,(\mathsf{share}(\mathsf{ret}\,0) \gg\!= \mathsf{id}) \oplus (\mathsf{share}\,x \gg\!= \mathsf{id}) && \text{(Choice)}\\
&\quad= \mathsf{share}\,\mathsf{zeros} \gg\!= \lambda x.\,(\mathsf{ret}(\mathsf{ret}\,0) \gg\!= \mathsf{id}) \oplus (\mathsf{share}\,x \gg\!= \mathsf{id}) && \text{(HNF, } n = 0)\\
&\quad= \mathsf{share}\,\mathsf{zeros} \gg\!= \lambda x.\,\mathsf{ret}\,0 \oplus (\mathsf{share}\,x \gg\!= \mathsf{id}) && \text{(Lret)}\\
&\quad= (\mathsf{share}\,\mathsf{zeros} \gg\!= \lambda x.\,\mathsf{ret}\,0) \oplus (\mathsf{share}\,\mathsf{zeros} \gg\!= (\lambda x.\,\mathsf{share}\,x \gg\!= \mathsf{id})) && \text{(Rdistr)}\\
&\quad= \mathsf{ret}\,0 \oplus (\mathsf{share}\,\mathsf{zeros} \gg\!= (\lambda x.\,\mathsf{share}\,x \gg\!= \mathsf{id})) && \text{(Ignore)}\\
&\quad= \mathsf{ret}\,0 \oplus ((\mathsf{share}\,\mathsf{zeros} \gg\!= \lambda x.\,\mathsf{share}\,x) \gg\!= \mathsf{id}) && \text{(Bassc)}\\
&\quad= \mathsf{ret}\,0 \oplus ((\mathsf{share}\,\mathsf{zeros} \gg\!= \lambda x.\,\mathsf{ret}\,x) \gg\!= \mathsf{id}) && \text{(Repeat, } k = \lambda y.\,\mathsf{id})\\
&\quad= \mathsf{ret}\,0 \oplus (\mathsf{share}\,\mathsf{zeros} \gg\!= \mathsf{id}) && \text{(Rret)}
\end{aligned}$$

We just showed that $\mathsf{zeros} = \mathsf{ret}\,0 \oplus (\mathsf{share}\,\mathsf{zeros} \gg\!= \mathsf{id}) = \mathsf{share}\,\mathsf{zeros} \gg\!= \mathsf{id}$, so $\mathsf{zeros} = \mathsf{ret}\,0 \oplus \mathsf{zeros}$ as claimed. That is, we could have defined zeros without using

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

share in the first place. This is not a surprise because the computation returned by share zeros in the definition of zeros is only used once. This proof relies on the laws (Flat) and (Repeat). Without (Flat), we could not get rid of the pattern share(share zeros $\gg= \ldots$). Similarly, without (Repeat), we would be stuck at the second-but-last line of the derivation.

We turn to values with nondeterministic components. We have seen that `heads_bind` (which does not use `share`) equals `cons coin (cons coin nil)`, which yields a nondeterministic list with two independent occurrences of `coin`. That is why the computation `heads_bind >>= convertList` yields four different results. To predict the result of `heads_share`, we need to apply the (HNF) law in a nontrivial way, with c being Cons and n being 2:

$$\begin{aligned}
&\mathsf{share}\,(\mathsf{cons\ coin\ undefined}) \gg= \lambda x.\,\mathsf{dupl}(\mathsf{first}\ x)\\
&= \mathsf{share\ coin} \gg= \lambda y_1.\,\mathsf{share\ undefined} \gg= \lambda y_2.\,\mathsf{ret}\,(\mathsf{cons}\ y_1\ y_2) \gg= \lambda x.\,\mathsf{dupl}(\mathsf{first}\ x)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{HNF } n=2,\ \text{Bassc})\\
&= \mathsf{share\ coin} \gg= \lambda y_1.\,\mathsf{share\ undefined} \gg= \lambda y_2.\,\mathsf{dupl}\ y_1 \qquad (\text{Lret, Def. first})\\
&= \mathsf{share\ coin} \gg= \lambda y_1.\,\mathsf{dupl}\ y_1 \qquad (\text{Ignore})\\
&= (\mathsf{share}\,(\mathsf{ret}\,0) \gg= \lambda y_1.\,\mathsf{dupl}\ y_1) \oplus (\mathsf{share}\,(\mathsf{ret}\,1) \gg= \lambda y_1.\,\mathsf{dupl}\ y_1)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{Def. coin, Choice})\\
&= \mathsf{dupl}(\mathsf{ret}\,0) \oplus \mathsf{dupl}(\mathsf{ret}\,1) \qquad (\text{HNF } n=0,\ \text{Lret})
\end{aligned}$$

(Again we use the (Ignore) law to discard an unused shared computation, namely, the list tail ignored by first.) This derivation shows that applying share to ret (Cons coin undefined) 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. That is why the computation `heads_share >>= convertList` equals an answer with only two primitive choices.

The (Ignore) law also comes in handy when analyzing the expression `dup_first_coin`, which creates an infinite number of choices but demands only a few of them. Using (Choice), (HNF), and (Ignore), we conclude that `dup_first_coin` equals

$$\mathsf{dupl}\ \emptyset \ \oplus\ (\mathsf{dupl}\,(\mathsf{ret}\,0)\ \oplus\ \mathsf{dupl}\,(\mathsf{ret}\,1))$$

Therefore, observing `dup_first_coin >>= convertList` gives {`[0,0]`,`[1,1]`}.

3.3 Intuitions behind our laws

In this section, we motivate the less-conventional laws of Figure 2 and relate the equational laws with observations of lazy nondeterministic programs with explicit sharing.

Early choice. The motivation for our (Choice) law comes from 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

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

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 (so it always contains the least partial value $\bot$).

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. Thus, even if e is nondeterministic, the denotation of a large term that contains e can be obtained by separately considering each partial value that 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 nondeterministic choice resembles the CRWL operation `?`, which yields one of its arguments nondeterministically and is 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.

The intuition behind the (Choice) law is that choices are made *as if* they are executed eagerly. However, actual eager execution is not an option as it does not comply to the intuition of late demand.

Late demand. The laws (Rdistr) and (Ignore) show a different perspective on sharing in nonstrict programs: we can proceed in a derivation (e.g., make a choice) without executing a shared computation, if that computation is not demanded. While the law (Ignore) applies if the value of the shared computation is never used, (Rdistr) lets us postpone the shared computation to after a subsequent choice that does not need it. The two laws are present in one form or another in call-by-need calculi, which permit evaluation in the body of a let expression if the let-bound variable is not needed right away. In particular, in the HOLet calculus of López-Fraguas *et al.* (2008), our (Ignore) law is called (Elim) and a more general version of our (Rdistr) is called (Contx). A law similar to (Rdistr) was used in a parsing library by Claessen (2004) to arrange for multiple parsers to process input in parallel so that nondeterminism does not cause a space leak.

Consequences of our laws. Our laws are term equalities on nondeterministic computations with sharing. Although it is convenient to reason using the laws, they equate more terms than it might first appear.

First, the (Rdistr) law implies that some choices can be reordered, as demonstrated by the following derivations. Starting from the same expression, we can proceed in two different ways—with or without using the (Rdistr) law. We can proceed without using it:[5]

$$\mathsf{share}\,(\mathsf{ret}\,0 \oplus \mathsf{ret}\,1) \gg\!= \lambda a.\,(\mathsf{ret}\,0 \oplus \mathsf{ret}\,10) \gg\!= \lambda y.\,a \gg\!= \lambda x.\,\mathsf{ret}(x+y)$$
$$= (\mathsf{share}\,(\mathsf{ret}\,0) \oplus \mathsf{share}\,(\mathsf{ret}\,1)) \gg\!= \lambda a.\,(\mathsf{ret}\,0 \oplus \mathsf{ret}\,10) \gg\!= \lambda y.\,a \gg\!= \lambda x.\,\mathsf{ret}(x+y)$$

[5] Some steps are omitted for brevity.

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

$$= ((\mathsf{ret}\,0 \oplus \mathsf{ret}\,10) \gg\!= \lambda y.\, \mathsf{ret}(0 + y)) \oplus ((\mathsf{ret}\,0 \oplus \mathsf{ret}\,10) \gg\!= \lambda y.\, \mathsf{ret}(1 + y))$$
$$= (\mathsf{ret}\,0 \oplus \mathsf{ret}\,10) \oplus (\mathsf{ret}\,1 \oplus \mathsf{ret}\,11)$$

We can also use the (Rdistr) law in the second step:

$$\begin{aligned}
&\mathsf{share}\,(\mathsf{ret}\,0 \oplus \mathsf{ret}\,1) \gg\!= \lambda a.\, (\mathsf{ret}\,0 \oplus \mathsf{ret}\,10) \gg\!= \lambda y.\, a \gg\!= \lambda x.\, \mathsf{ret}(x + y)\\
&\quad = \mathsf{share}\,(\mathsf{ret}\,0 \oplus \mathsf{ret}\,1) \gg\!= \lambda a.\, (a \gg\!= \lambda x.\, \mathsf{ret}(x + 0)) \oplus (a \gg\!= \lambda x.\, \mathsf{ret}(x + 10))\\
&\quad = (\mathsf{share}\,(\mathsf{ret}\,0 \oplus \mathsf{ret}\,1) \gg\!= \lambda x.\, \mathsf{ret}(x + 0))\\
&\qquad \oplus (\mathsf{share}\,(\mathsf{ret}\,0 \oplus \mathsf{ret}\,1) \gg\!= \lambda x.\, \mathsf{ret}(x + 10))\\
&\quad = (\mathsf{ret}\,0 \oplus \mathsf{ret}\,1) \oplus (\mathsf{ret}\,10 \oplus \mathsf{ret}\,11)
\end{aligned}$$

Our laws thus equate $(\mathsf{ret}0 \oplus \mathsf{ret}10) \oplus (\mathsf{ret}1 \oplus \mathsf{ret}11)$ and $(\mathsf{ret}0 \oplus \mathsf{ret}1) \oplus (\mathsf{ret}10 \oplus \mathsf{ret}11)$. Consequently, if it happens that $\emptyset$ is the left and the right unit of $\oplus$, (Rdistr) and (Choice) imply that $\oplus$ is commutative.

Second, the (Choice) and (Ignore) laws imply that $\oplus$ is idempotent:

$$c = \mathsf{share}(a \oplus b) \gg\!= \lambda\_.\, c = (\mathsf{share}\, a \gg\!= \lambda\_.\, c) \oplus (\mathsf{share}\, b \gg\!= \lambda\_.\, c) = c \oplus c$$

Our laws thus equate $c \oplus c$ and c.

3.4 Observational equivalence

Recall from Section 3.1.3 that eventually we have to observe the results of a nondeterministic computation with sharing, so that we can print them out, for example. We introduced the (partial) observation function results to "run" a computation with sharing and compute the results as a set.

The function results uses the more general function collect for observing nondeterministic computations with sharing.[6]

```
results a = collect (liftM Data.Set.singleton a)
```

The type of computed results must be an instance of the type class `Nondet` that defines a method failure for failing computations and `(?)` for nondeterministic choice.

```
class Nondet n where
  failure :: n
  (?)     :: n -> n -> n
```

Valid definitions of the `(?)` operation are idempotent and allow choice reordering:

$$a \,?\, a = a \qquad \text{(Idempotence)}$$
$$(a \,?\, b) \,?\, (c \,?\, d) = (a \,?\, c) \,?\, (b \,?\, d) \qquad \text{(Choice Reordering)}$$

These laws are derived from the remarks in Section 3.3. Choice reordering reflects right distributivity of sharing computations. While the (Rdistr) law of Figure 2 allows to permute a single choice with subsequent choices, choice reordering allows to permute a single choice with any other choice. If failure and `(?)` form a monoid,

[6] We describe the types of results and collect in Section 5.

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

$$\mathsf{collect}\,\emptyset = \mathsf{failure} \tag{rZero}$$
$$\mathsf{collect}\,(a \oplus b) = \mathsf{collect}\,a\ ?\ \mathsf{collect}\,b \tag{rPlus}$$
$$\mathsf{collect}\,(\mathsf{ret}\,v) = v \tag{rRet}$$

Fig. 3. The laws of observing a computation with nondeterminism and sharing. The laws, in particular, (rRet), apply only if both of their sides are well typed. See Section 5 for the typing of collect.

then choice reordering is equivalent to commutativity. We define several instances of `Nondet` (including one for `Data.Set.Set a`) for observing nondeterministic computations with sharing in Section 5.1.

Recall that we can only observe trees of primitive choices, or answers. Answers are built using the monadic operations $\emptyset$, $\oplus$, and ret. The partial function collect acts homomorphically on $\emptyset$, $\oplus$, and ret, as depicted in Figure 3. Thus, to predict the observation of a computation with sharing, we have to show it equivalent to an answer and then apply the laws of Figure 3. (There are expressions that cannot be observed; see Section 5 for an example.)

Let e_1 and e_2 be two nondeterministic computations with sharing that are equal according to the laws of Figure 2. Suppose that both $\mathsf{collect}\,e_1$ and $\mathsf{collect}\,e_2$ are well typed, so the type of each of the two applications, if defined, is an instance of `Nondet`. If $\mathsf{collect}\,e_1$ is defined (because e_1 is an answer, for instance), then we conjecture that $\mathsf{collect}\,e_2$ is defined too and $\mathsf{collect}\,e_1 = \mathsf{collect}\,e_2$. In short, we conjecture that collect maps sharing computations that are equal according to Figure 2 to equal results in an arbitrary (valid) instance of `Nondet`. Colloquially speaking, we need $\mathsf{collect}\,e$ to reduce e. Of course, this step depends on the implementation of collect, which we consider next.

4 Implementing lazy nondeterminism

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 nondeterministic components, namely, values of type `Int`. We then extend the approach to values with nondeterministic components, namely, lists of numbers. Finally, in Section 6, we give an implementation for arbitrary user-defined nondeterministic types and for arbitrary instances of `Nondet`.

4.1 The tension between late demand and early choice

Lazy execution means to execute shared computations at most once and not until they are demanded. The laws (Fail) and (Ignore) from Figure 2 formalize such 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
```

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

and so `share mzero` is trivially `return mzero`, just as the law (Fail) requires; (Ignore) is clearly satisfied too. 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 Section 3.1.1 and replace `share` with `return`, we obtain `dup_coin_let`—which, as explained there, shares only a nondeterministic computation, not its result as desired. Instead of remaking 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 does not satisfy the (Fail) and (Ignore) laws. The (Lzero) law of `MonadPlus` shows that this implementation renders `share mzero` equal to `mzero`, not the `return mzero` required by (Fail). This attempt to ensure early choice using early demand leads to 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 observation type 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*, initially empty. Each thunk 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 `Thunks` through nondeterministic computations represented as lists.[7]

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

This version of the `Memo` monad threads thunks of type `Thunk Int` and can therefore only share integers. The instance declarations for the type classes `Monad`, `MonadState`, and `MonadPlus` are as follows:

[7] In this preliminary implementation, we use lists to represent sets of results.

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

```
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 nondeterministic 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 :: MonadState [Thunk a] m => m a -> m (m 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 evaluation of the thunk `a` (in the line `x <- a` of the last `do` form) may change the state; therefore, we need to query the state `thunks` again in the next line before updating the state.

This implementation of `share` satisfies the (Fail) and (Ignore) laws because the argument `a` given to `share` is not demanded until the inner action is performed. Thanks to the properties (Idempotence) and (Choice Reordering) in Section 3.4, which are true of lists viewed as sets, we conjecture that the (Choice) law is satisfied as well. We argue informally for this conjecture as follows. Consider the computation

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

$\mathsf{share}\,(a \oplus b) \gg\!= \lambda x.\, c$, which our implementation executes by passing an expanded store to c.

1. If executing c does not access the newly stored thunk, then the result of the computation $\mathsf{share}\,(a \oplus b) \gg\!= \lambda x.\, c$ does not depend on what thunk is stored, so it is same as the result of $\mathsf{share}\,a \gg\!= \lambda x.\, c$ and of $\mathsf{share}\,b \gg\!= \lambda x.\, c$ and, by (Idempotence), of

$$(\mathsf{share}\,a \gg\!= \lambda x.\, c) \oplus (\mathsf{share}\,b \gg\!= \lambda x.\, c)$$

   as desired.
2. If executing c accesses the newly stored thunk $a \oplus b$ right away, then we might as well nondeterministically choose whether to store the thunk a or the thunk b. In other words, the result of the computation $\mathsf{share}\,(a \oplus b) \gg\!= \lambda x.\, c$ is same as the result of

$$(\mathsf{share}\,a \gg\!= \lambda x.\, c) \oplus (\mathsf{share}\,b \gg\!= \lambda x.\, c)$$

   as desired.
3. Finally, suppose executing c makes a nondeterministic choice (between c_1 and c_2, say) without immediately accessing the newly stored thunk. Then, the result of the computation $\mathsf{share}\,(a \oplus b) \gg\!= \lambda x.\, c$ does not depend on whether the thunk is stored before or after c makes its choice, so it is same as the result of

$$(\mathsf{share}\,(a \oplus b) \gg\!= \lambda x.\, c_1) \oplus (\mathsf{share}\,(a \oplus b) \gg\!= \lambda x.\, c_2).$$

   Because (Choice Reordering) assures that the computations

$$((\mathsf{share}\,a \oplus \mathsf{share}\,b) \gg\!= \lambda x.\, c_1) \oplus ((\mathsf{share}\,a \oplus \mathsf{share}\,b) \gg\!= \lambda x.\, c_2)$$

   and

$$(\mathsf{share}\,a \oplus \mathsf{share}\,b) \gg\!= \lambda x.\, c_1 \oplus c_2$$

   have the same result, we have reduced satisfying (Choice) for c to satisfying (Choice) for c_1 and c_2.

We further conjecture that this argument generalizes from observing lists viewed as sets to observing other valid instances of `Nondet`, and from sharing integers to sharing data with nondeterministic components. The latter generalization is the topic of the next section.

4.3 Nondeterministic components

The version of `share` just developed memoizes only integers. However, we want to memoize data with nondeterministic 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.

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

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

The instance declarations for `Monad` and `MonadPlus` stay the same. In the `MonadState` instance, the state type needs to be changed to `[Thunk (List Memo Int)]`. 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. For example, in the expression

```
share (cons (return 0 `mplus` return 1) nil)
```

the `cons` expression producing the `Cons` value will be memoized, but the components of the `Cons` value will not. 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 nondeterministic components as deeply as demanded by the computation. Each component is executed at most once and memoized individually in the list of stored thunks.[8]

5 Observing nondeterministic computations

In order to observe the results of a computation that yields a value with nondeterministic components, we need

1. a function (such as `convertList`) that executes all nondeterministic components and combines the resulting alternatives to compute a nondeterministic choice of deterministic results, and
2. a function (such as results) that computes the results as a set.

[8] 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 Section 6 below, so the issue is moot.

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

We can define an operation `runMemo` that computes the results of a nondeterministic computation as a list:

```
runMemo :: Memo a -> [a]
runMemo m = map fst (unMemo m [])
```

However, the type of this definition is too permissive. For example, we are free to call `runMemo (share nil)`, which results in a value of type `[Memo (List Memo Int)]`. But the computations inside the resulting list cannot be sensibly executed again using `runMemo` because they access a reference into a store that has already been dismissed by the first call to `runMemo`.

The situation is very similar to returning references from ST-monad computations:

```
smuggle :: ST s (ST s Int)
smuggle = do x <- newSTRef 42
             return (readSTRef x)
```

We can use `smuggle` in ST-monad computations; however, observing `smuggle` using `runST` would attempt to "smuggle" out the ST reference `x` past `runST`. If the attempt succeeded, we would get a value that contains a reference to an already disposed heap. To prevent such attempts, the ST monad is parameterized by a polymorphic type variable `s` (Launchbury & Peyton Jones, 1994). The higher-rank type of `runST` ensures that the result of `runST` does not contain references to the heap used for running the computation.

Using this idea, we assign a higher-rank type to our observation function to restrict the kind of computations that can be observed, as described in Section 3.1.3. An attempt to execute `results (share nil)` leads to a type error in our implementation described in Section 6.

In Section 6.1, we provide a type class `Sharing`, which extends nondeterminism monads with the `share` operation. We let the function `results` have the following type:

```
results :: Ord a => (forall s . Sharing s => s a) -> Data.Set.Set a
results a = collect (liftM Data.Set.singleton a)
```

As with the function `runST` for the `ST` monad, this type ensures that

1. sharing computations can only use the operations of `MonadPlus` and `share` and
2. the result of type `a` cannot contain any reference to the sharing monad `s`.

We cannot even observe `nil` because it is of type `s (List s a)` for some monad `s` and, hence, `results nil` would be of type `Set (List s a)`. The type variable `s`, however, is not allowed to escape. Although the value `nil` does not actually contain any reference to `s`, we need to apply `convertList` in order to observe `nil`. This is perfectly reasonable because from the type `List s a` it is not apparent whether the corresponding list contains nondeterministic components (i.e., is nonempty) or not (i.e., is empty).

We also cannot observe computations that yield functions that involve shared computations. For example, the computation `share coin >>= \x -> return`

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

`(\y -> x)` has the type `Sharing s => s (a -> s Int)`. Applying results to this expression leads to a type error because the type variable `s` escapes.

5.1 Observation types

The function collect used in the definition of results has the following type:

```
collect :: Nondet n => (forall s . Sharing s => s n) -> n
```

For the definition of results to type check, we must make `Data.Set.Set a` an instance of `Nondet` for instances `a` of `Ord`. We implement failure as empty set and `(?)` as set union:

```
instance Ord a => Nondet (Data.Set.Set a) where
  failure = Data.Set.empty
  (?)     = Data.Set.union
```

This instance is valid if the `Ord` instance for `a` actually defines an ordering relation (which we assume) because then set union is idempotent and allows reordering of choices (as it is associative and commutative).

Users may be tempted to use laws of the observation type to reason about sharing computations and indeed, our implementation of sharing in Section 6 inherits the monoid laws for $\oplus$ and $\emptyset$ from the underlying set type.

In combination with the laws of Figure 2, such additional laws for the sharing monad may lead to unexpected identities. For example, if $\emptyset$ is a unit of $\oplus$, then we can derive

$$\begin{aligned}
\mathsf{ret}\,(\mathsf{ret}\,0) &= \mathsf{share}\,(\mathsf{ret}\,0) && \text{(HNF)}\\
&= \mathsf{share}\,(\mathsf{ret}\,0 \oplus \emptyset) && \text{(Unit)}\\
&= \mathsf{share}\,(\mathsf{ret}\,0) \oplus \mathsf{share}\,\emptyset && \text{(Choice)}\\
&= \mathsf{ret}\,(\mathsf{ret}\,0) \oplus \mathsf{ret}\,\emptyset. && \text{(HNF, Fail)}
\end{aligned}$$

The expressions in this equation chain all have the type `Sharing s => s (s Int)`, so the higher-rank type of results prevents us from projecting the left- and right-hand sides into the type `Data.Set.Set (Data.Set.Set Int)`. In other words, we cannot distinguish the left- and right-hand sides by applying the results function. The function

```
noEmpty :: Data.Set.Set (Data.Set.Set a) -> Bool
noEmpty s = Data.Set.null (Data.Set.filter Data.Set.null s)
```

yields `True` when applied to $\{\{0\}\}$ but `False` when applied to $\{\{0\},\emptyset\}$. However, the type of results prohibits the observation of this inconsistency. If we apply the monadic `join` function[9] to prevent the type variable `s` for the sharing monad from escaping then the inconsistency disappears.

[9] `join x = x >>= id`

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

The set type is not the only observation type that satisfies idempotence and choice reordering. For example, we can define a function that checks whether a nondeterministic computation with sharing yields a result by making `Bool` an instance of `Nondet`.

```
instance Nondet Bool where
  failure = False
  (?)     = (||)
```

Based on this instance, we can define a function `hasResult` that checks whether the given computation yields a result.

```
hasResult :: (forall s . Sharing s => s a) -> Bool
hasResult a = collect (liftM (const True) a)
```

This function is similar to `not . Data.Set.null . results` but does not compute an intermediate set of results and sometimes terminates even if the given computation can yield an unbounded number of results.

Both instances defined so far form a commutative monoid because `(?)` is associative and commutative and `failure` is its unit. As an example for an instance that does not form a monoid consider the type of probability distributions defined as

```
type Dist a = Data.Map.Map a Rational
```

We can define a `Nondet` instance for `Dist a` by averaging weights in the definition of `(?)`.

```
instance Ord a => Nondet (Dist a) where
  failure = Data.Map.empty
  d1 ? d2 = Data.Map.map (/2) (Data.Map.unionWith (+) d1 d2)
```

This definition of `(?)` is not associative and failure is no unit. But `(?)` is idempotent and satisfies choice reordering such that this instance is valid. Like in the previous instances, `(?)` is also commutative because weights are averaged evenly. A biased version of `(?)` (that picks an argument with a probability different from $\frac{1}{2}$) would not be commutative but still allow choice reordering.

The following function can be used to observe the probability distribution of a nondeterministic computation with sharing.

```
resultDist :: Ord a => (forall s . Sharing s => s a) -> Dist a
resultDist a = collect (liftM (\x -> Data.Map.singleton x 1) a)
```

The probability distribution computes for each result the probability that it would be returned by a random strategy that picks one alternative of `mplus` with the probability $\frac{1}{2}$. The following session using an interactive Haskell environment demonstrates that `mzero` and `mplus` do not satisfy the monoid laws.

```
ghci> resultDist (return 1)
fromList [(1,1 % 1)]
ghci> resultDist (mzero `mplus` return 1)
```

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

```
fromList [(1,1 % 2)]
ghci> resultDist (return 1 `mplus` (return 2 `mplus` return 3))
fromList [(1,1 % 2),(2,1 % 4),(3,1 % 4)]
ghci> resultDist ((return 1 `mplus` return 2) `mplus` return 3)
fromList [(1,1 % 4),(2,1 % 4),(3,1 % 2)]
```

A computation that may fail yields results with total probability less than one, and the individual probabilities for computations that yield one of many different results depend on how the calls to `mplus` are nested.

We can nest `mplus` to construct computations that yield a result with a probability $m/2^n \leqslant 1$ for nonnegative integers m and n, but it is more convenient to provide an additional operation that lets users set probabilities freely. Although we can use the `Dist` type to observe sharing computations, the type of collect restricts us from using such additional operations to construct sharing computations with freely set probabilities.

To clarify the interaction of the `share` combinator with probabilistic inference, we compare observations of `dup_coin_let`, `dup_coin_bind`, and `dup_coin_share` (see Section 3.1.1) with the `Set` type with corresponding observations with the `Dist` type. The observations with the `Set` type reflect the results derived in Sections 3.1.1 and 3.2.

```
ghci> results dup_coin_let
fromList [(0,0),(0,1),(1,0),(1,1)]
ghci> results dup_coin_bind
fromList [(0,0),(1,1)]
ghci> results dup_coin_share
fromList [(0,0),(1,1)]
```

When observing these computations with the `Dist` type, the results are the same but have associated probabilities.

```
ghci> resultDist dup_coin_let
fromList [((0,0),1 % 4),((0,1),1 % 4),((1,0),1 % 4),((1,1),1 % 4)]
ghci> resultDist dup_coin_bind
fromList [((0,0),1 % 2),((1,1),1 % 2)]
ghci> resultDist dup_coin_share
fromList [((0,0),1 % 2),((1,1),1 % 2)]
```

Like with the `Set` type, the results of `dup_coin_bind` and `dup_coin_share` are exactly the same. The shared `coin` computation in `dup_coin_share` causes only one nondeterministic choice although it is executed twice inside `duplicate`.

If a nondeterministic computation with sharing can yield an unbounded number of results, it is useful to compute them incrementally, while ignoring the order and multiplicities of computed results. For this purpose, we provide a function `resultList` that returns a list of results in the `IO` monad.

```
resultList :: (forall s . Sharing s => s a) -> IO [a]
```

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

We use the `IO` type to account for the unspecified order and multiplicities of computed results (which is similar to the role `IO` plays in imprecise exceptions (Peyton Jones *et al*., 1999)). As the result list is computed lazily, this function can be used to implement an interactive eval-print loop for querying results (see the Appendix).

6 Generalized, efficient implementation

In this section, we generalize the implementation ideas described in Section 4 such that

1. arbitrary user-defined types with nondeterministic components can be passed as arguments to the combinator `share`,
2. user-defined types with nondeterministic components can be converted into corresponding types without nondeterministic components, and
3. arbitrary instances of `Nondet` can be used to observe nondeterministic computations with sharing.

We achieve the first and second goal by introducing type classes with the interface to share and convert nondeterministic data. We achieve the third goal by defining a monad `Lazy n` that turns any instance `n` of `Nondet` into a monad for nondeterminism with sharing.

All of these generalizations are motivated by practical applications in nondeterministic programming.

1. The ability to work with user-defined types makes it possible to draw on the sophisticated type and module systems of existing functional languages.
2. The ability to convert between types with nondeterministic components and corresponding types without makes it easier to compose deterministic and nondeterministic code.
3. The ability to plug in different underlying search types makes it possible to express techniques such as weighted results as demonstrated in Section 5.1.

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 & Shan, 2009).

Our library is available as package `explicit-sharing-0.9` on Hackage.

6.1 Sharing nondeterministic data

We have seen in the previous section that, in order to share nested, nondeterministic data deeply, we need to traverse it and apply the combinator `share` recursively to every nondeterministic component. We have implemented deep sharing for nondeterministic lists, but want to generalize this implementation to support arbitrary user-defined types with nondeterministic components. We define the following type class that allows arbitrary user-defined types to be passed to `share`:

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

```
class MonadPlus m => Shareable m a where
  shareArgs :: MonadPlus n
            => (forall b . Shareable m b => m b -> n (m b))
            -> a -> n a
```

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

For example, we can make `List m Int`, the type of nondeterministic lists of numbers, an instance of `Shareable m` as follows.

```
instance MonadPlus m => Shareable m Int where
  shareArgs _ c           = return c

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

The implementation mechanically applies the given transformation to the nondeterministic arguments of each constructor.

Based on the `Shareable` type class, we can 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 :: Shareable m a => m a -> m (m a)
```

Nondeterminism monads that support the operation `share` are instances of this class. In Section 6.4, we define an instance of `Sharing`.

6.2 Converting nondeterministic data

We have seen in Section 3.1.3 that data structures with nondeterministic components are not observable. To observe them, a programmer must convert them to a (generally different) data structure without nondeterministic components, forcing the latent nondeterminism hidden in the monadic components. We introduced such a conversion function `convertList`, which turns lists with nondeterministic components into ordinary Haskell lists. It is the responsibility of the programmer to define such conversion functions for each data type with nondeterministic components.

A conversion function traverses a value of a data type and converts each encountered component in turn. To clarify the pattern of structural induction and to ease writing conversion functions, we introduce the type class `Convertible`.

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

```
class MonadPlus m => Convertible m a b where
  convert :: a -> m b
```

The type class has no functional dependency, because one source data type `a` may be converted to several data types. The desired target type `b` is determined from the context or from a user-provided type annotation. The conversion may force latent nondeterminism and hence has to be performed in a nondeterminism monad `m`.

Converting deterministic data is trivial:

```
instance MonadPlus m => Convertible m Int Int where
  convert c           = return c
```

We provide another `Convertible` instance to convert nondeterministic lists into ordinary Haskell lists.

```
instance Convertible m a b => Convertible m (List m a) [b] where
  convert Nil         = return []
  convert (Cons x xs) = do y  <- x  >>= convert
                           ys <- xs >>= convert
                           return (y:ys)
```

Finally, we provide a second `Convertible` instance for lists, to convert in the other direction.

```
instance Convertible m a b => Convertible m [a] (List m b) where
  convert []     = nil
  convert (x:xs) = cons (convert x) (convert xs)
```

This instance lets us convert ordinary Haskell lists, for example of type `[Int]`, into nondeterministic lists, of type `List m Int`, to be used in lazy nondeterministic computations.

6.3 Deriving nondeterministic data types

The previous two sections showed that, for each data type with nondeterministic components, the user has to provide instances of the type classes `Shareable` and `Convertible`. These instances are quite regular, following the pattern of structural induction. Defining them by hand is tedious and error-prone. We have written a tool that uses Template Haskell to synthesize a monadic data type declaration and corresponding instances automatically. For example, given the definition

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

running our tool like

```
$(derive monadic ''List)
```

produces the data type declaration

```
data MList m a = MNil | MCons (m a) (m (MList m a))
```

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

which is the same as the declaration given in Section 2.4 but using different type and constructor names. Additionally, our tool produces the following functions.

```
mNil       :: Monad m => m (MList m a)
mCons      :: Monad m => m a -> m (MList m a) -> m (MList m a)
matchMList :: Monad m => m (MList m a)
                      -> m b -> (m a -> m (MList m a) -> m b) -> m b
```

`mNil` and `mCons` are defined like `nil` and `cons` in Section 2.4. `matchMList` can be used for matching nondeterministic lists against patterns and hides the application of monadic bind to execute latent nondeterminism:

```
matchMList ml n c = do l <- ml
                       case l of
                         MNil       -> n
                         MCons x xs -> c x xs
```

In addition to the data type declaration with con- and de-structor functions, our tool also generates the following instances of the `Shareable` and `Convertible` type classes:

```
instance Shareable m a => Shareable m (MList m a)
instance Convertible m a b => Convertible m (List a) (MList m b)
instance Convertible m a b => Convertible m (MList m a) (List b)
```

The generated implementations of these instances are the same as the implementations in Sections 6.1 and 6.2, modulo renaming constructors.

Our derivation tool can also be used to generate data types where only some components are nondeterministic. To demonstrate how, we define a data type for strict lists:

```
data ListS a = NilS | ConsS !a (ListS a)
```

The first component of the cons cell is defined to be strict. When building such a list, GHC will arrange to force the computations for the list elements; the spine of the list will still be computed lazily. Our tool takes the strictness annotation as a determinism annotation, generating the following declaration:

```
data MListS m a = MNilS | MConsS !a (m (MListS m a))
```

The spine of the list is produced lazily and possibly nondeterministically; the elements of the list are deterministic and evaluated eagerly like in `ListS`.

The types of the auxiliary con- and de-structors generated for strict lists resemble those generated for the lazy list type, but the definitions account for the deterministic list elements. For example, the generated definition of the `mConsS` function uses monadic bind to ensure that the first argument of `MConsS` is deterministic:

```
mConsS :: Monad m => m a -> m (MListS m a) -> m (MListS m a)
mConsS mx mxs = do x <- mx
                   return (MConsS x mxs)
```

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

The second argument `mxs` is passed to `MConsS` without using bind, hence maintaining latent nondeterminism.

A consequence of our use of strictness annotations to signify determinism is that our tool cannot construct a data type with deterministic but lazy components. For example, to define a type of infinite streams with nondeterministic elements, users should provide the type definition and a corresponding constructor function manually.

Strictness annotations also influence the type-class instances. The `Shareable` instance for strict nondeterministic lists is generated as follows:

```
instance Shareable m a => Shareable m (MListS m a) where
  shareArgs _ MNilS         = return MNilS
  shareArgs f (MConsS x xs) = do y  <- shareArgs f x
                                 ys <- f xs
                                 return (MConsS y ys)
```

Instead of calling `f x` in order to share `x` recursively, we compute `shareArgs f x`. Note that, unlike in the `Shareable` instance for `MList`, the type of the variable `y` is `a`, not `m a`. For converting from the nonmonadic to the monadic type, we use the generated constructor functions, which take care of deterministic components:

```
instance Convertible m a b =>
         Convertible m (ListS a) (MListS m b) where
  convert NilS         = mNilS
  convert (ConsS x xs) = mConsS (convert x) (convert xs)
```

When converting in the other direction, we use the bind operation on possibly nondeterministic components and convert deterministic components directly:

```
instance Convertible m a b =>
         Convertible m (MListS m a) (List b) where
  convert MNilS         = return NilS
  convert (MConsS x xs) = do y  <- convert x
                             ys <- xs >>= convert
                             return (ConsS y ys)
```

With this ability to automatically derive "monadic versions" of data types, Haskell challenges dedicated FLP languages. With additional support for translating nonmonadic function declarations into monadic ones, lazy nondeterministic programs could be written in Haskell as elegantly as in a language like Curry. We believe that Template Haskell can also be used to convert function declarations, for example, to convert the nonmonadic definition of `isSorted` given in Section 2.1 into the monadic definition given in Section 2.4.

Our converter relies on the Derive tool[10] and is available as a module in our Hackage package.

[10] Available as package `derive` on Hackage.

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

6.4 Using an arbitrary state monad

The implementation of memoization in Section 4 uses a state monad to thread a list of thunks through nondeterministic computations. The straightforward generalization is to rely on the `MonadPlus` and `MonadState` type classes without fixing a concrete instance.

The type for `Thunks` 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, which clearly resembles a first-class store (Morrisett, 1993). Note that the operations `lookupThunk` and `insertThunk` deal with thunks of arbitrary type.

```
type Key    -- abstract
emptyThunks :: ThunkStore
getFreshKey :: MonadState ThunkStore m => m Key
lookupThunk :: MonadState ThunkStore m => Key -> m (Thunk m a)
insertThunk :: MonadState ThunkStore m => Key -> 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 modules `Data.Typeable` and `Data.Dynamic` but omit corresponding class contexts for the sake of clarity.

Lazy monadic computations can be performed in a monad that threads a `ThunkStore`. In Section 6.5, we define a type constructor `Lazy` that transforms any instance `n` of `Nondet` into an instance `Lazy n` of `MonadPlus` and `MonadState ThunkStore`. We can define the instance of `Sharing`, which implements the operation `share`, in terms of this interface.

```
instance Nondet n => Sharing (Lazy n) where
  share a = memo (a >>= shareArgs share)
```

The implementation of `share` uses the operation `memo` to memoize the argument and the operation `shareArgs` to apply `share` recursively to the nondeterministic components of the given value. The function `memo` resembles the specific version given in Section 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.

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

We could be tempted to use the predefined type `State ThunkStore a` to represent nondeterministic computations with sharing. Unfortunately, this type does not allow to use `failure` and `(?)` for implementing `mzero` and `mplus` because they would require `a` to be an instance of `Nondet`. Instead, we use a well-known technique to implement state monads based on continuations. This is not only more efficient than using the predefined `State` type but also solves the problem of using the `Nondet` operations to implement a `MonadPlus` instance.

6.5 *Efficient implementation*

We now give an efficient implementation of the monad `Lazy n` for nondeterministic computations with sharing. We use the permutation sort in Section 2 for a rough measure of performance. We develop our implementation in several steps. 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 as fast as the fastest available compiler for Curry, the Münster Curry Compiler (MCC).[11]

As detailed below, we achieve this competitive performance by

1. using a continuation-based state monad to thread the store of thunks,
2. reducing the number of store operations when storing shared results, and
3. manually inlining and optimizing library code.

6.5.1 *Continuation-based state monad*

The `Monad` instance for the predefined `State` monad 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`:

```
newtype ContT n m a = C { unC :: (a -> m n) -> m n }

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

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

```
instance Monad (ContT n 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 defined using the corresponding operations of the `Nondet` class if the result type `m n` of continuations supports them:

[11] We performed our experiments on a Lenovo ThinkPad with a 2.13 GHz Intel Core i7 processor using GHC 7.0.3 with optimizations (`-O2`).

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

```
instance Nondet (m n) => MonadPlus (ContT n m) where
  mzero     = C (\c -> failure)
  mplus a b = C (\c -> unC a c ? unC b c)
```

This instance relies on `m n` to be an instance of `Nondet` but the result type `a` of monadic computations is unrestricted.

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. The type constructor `Reader s` is a reader monad for any type `s`, for example, `s = ThunkStore`.

```
newtype Reader s a = Reader { runReader :: s -> a }
```

In combination with `ContT`, the function `local` is enough to implement state updates:

```
instance MonadState s (ContT n (Reader s)) where
  get   = C (\c -> ask >>= c)
  put s = C (\c -> local (const s) (c ()))
```

`Reader s n` inherits `Nondet` operations from an underlying `Nondet` instance `n`:

```
instance Nondet n => Nondet (Reader s n) where
  failure = Reader (\s -> failure)
  a ? b   = Reader (\s -> runReader a s ? runReader b s)
```

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

```
type Lazy n = ContT n (Reader ThunkStore)
```

We can reuse from Section 6.4 the definition of the `Sharing` instance and of the `memo` function used to define `share`.

The function `collect`, used to observe nondeterministic computations with sharing using the underlying `Nondet` instance, can be defined as follows.

```
collect :: Nondet n => (forall s . Sharing s => s n) -> n
collect a = runReader (runContT a) emptyThunks
```

As discussed in Section 5, we use a higher-rank type to restrict observable computations.

6.5.2 *Fewer state manipulations*

The function `memo` defined in Section 6.4 performs two state updates for each shared value that is demanded: one to insert the unevaluated shared computation and one

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

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 slightly 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 Key
lookupHNF   :: MonadState Store m => Key -> m (Maybe a)
insertHNF   :: MonadState Store m => Key -> a -> m ()
```

The result of `lookupHNF` is undefined if the given key is not present in the store, and the result of `insertHNF` is undefined if it is. Both functions now operate on values rather than thunks and the `HNF` suffix (for head normal form) is meant to signify that.

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)
```

This definition of `memo` does not store the unevaluated argument `a` directly after generating a fresh key. Therefore, it is crucial that the store interface allows to generate keys independently of manipulating the stored values.

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. The original implementation had a possible memory leak too: `a` is first placed into the threaded store and—if its result is not required—remains in the store throughout the computation. Fixing this leak would seem to require support from the garbage collector. We did not experience memory problems during our experiments, however.

6.5.3 Mechanical simplifications

The final optimization is to

1. expand the types in `ContT n (Reader State)`,
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. We

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

believe our final optimizations should really be performed by the compiler, but did not find a way to achieve this.

Surprisingly, our still high-level and very modular implementation (it works with arbitrary `Nondet` instances and arbitrary types for nested, nondeterministic data) is as efficient as the fastest available Curry compiler. Like the program used for our benchmarks, an equivalent Curry program for permutation sort runs for about 15 s when compiled with MCC and `-O2` optimizations.

We have also compared our performance on *deterministic* monadic computations against corresponding nonmonadic programs in Haskell and Curry. Our benchmark is to call the naive reverse function on long lists (10,000 elements), which involves a lot of deterministic pattern-matching. In this benchmark, the monadic code is roughly five times slower than the corresponding Curry code in MCC, which is as fast as a deterministic version in Haskell.

Our library does not directly support narrowing and unification of logic variables but can emulate it by means of lazy nondeterminism. 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 nondeterministic generators and implementing the unification operator `=:=` as equality check. When applying `last` to a list of determined values, the monadic Haskell code is about twice as fast as 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 nondeterministic generators is nondeterministic and leads to search-space explosion.

All programs used for benchmarking are available online.[12]

7 Characteristics of explicit sharing

In this section, we discuss subtleties such as sharing across nondeterminism (Section 7.1) and repeated sharing of already explicitly shared computations (Section 7.2). The two issues do not affect the correctness of programs that use explicit sharing, but they may significantly affect the performance of some classes of these programs. We describe causes of and workarounds for these performance problems.

7.1 Sharing across nondeterminism

Our implementation of nondeterminism with sharing does not provide what Braßel & Huch (2007) call sharing *across* nondeterminism. Although shared results are reused if accessed more than once in one branch of the computation, shared computations are executed independently in different nondeterministic branches

[12] `http://github.com/sebfisch/explicit-sharing/tree/0.9`

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

of the computation. This is not a problem with respect to correctness because different nondeterministic branches are independent. It is, however, a performance penalty, when shared computations are deterministic but recomputed in independent nondeterministic branches of the computation.

We demonstrate this penalty using the function `shareND` that shares the computation `count n` in `n` nondeterministic branches:

```
shareND :: Sharing m => Int -> m Int
shareND n = do x <- share (count n)
               foldr mplus mzero (replicate n x)
```

The function `count` is a placeholder for an expensive monadic computation—its run time is linear in the given argument.

```
count :: Monad m => Int -> m Int
count 0 = return 0
count n = do m <- count (n-1)
             return (m+n)
```

When observing the result of `shareND n` as a set, the computation `count n` is executed `n` times—once in each nondeterministic branch—which leads to quadratic run time in total.

However, the computation `count n` is deterministic and, hence, it is not necessary to share its result explicitly using the `share` combinator. Therefore, we can define a variant `shareND'` of `shareND` that does not use `share` and can be executed directly using an arbitrary instance of `MonadPlus`:

```
shareND' :: MonadPlus m => Int -> m Int
shareND' n = foldr mplus mzero (replicate n (count n))
```

We can now execute `shareND' n` in the list monad without first using a monad with explicit sharing and then observing the results. The result of the computation `count n` is then shared by Haskell's built-in sharing because `count n` is a data term of type `[Int]`. As a consequence, it is only computed once and not recomputed in every nondeterministic branch. Thus, the total run time of executing `shareND' n` in the list monad is linear, not quadratic, in `n`.

The reason that the list monad supports sharing across nondeterminism but our monad with explicit sharing does not is that monadic computations in the list monad are represented as data, namely lists, but monadic computations in our monad with explicit sharing are represented as functions. When executing `count n` in the list monad, Haskell's built-in sharing has the effect that the result of `count n`, a singleton list, is only computed once and shared in the list generated by `replicate`. In our monad with explicit sharing, the result of `count n` is a function. This function is shared in the list generated by `replicate` and applied repeatedly when observing the results.

Although sharing across nondeterminism reduces the run time of certain lazy nondeterministic programs, it also increases the memory usage of others. Using Haskell's built-in sharing to memoize nondeterministic computations—not only

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

their results—can lead to prohibitive memory requirements of programs that share computations that require expensive search. The Curry compiler KiCS implements sharing across nondeterminism and Braßel & Huch (2007) discuss its benefit using more practical examples than we present here. On the other hand, KiCS suffers from the described memory problems: when executing our running example of sorting by testing if a shared permutation is sorted, the program generated by KiCS uses an exponential amount of memory. Our implementation uses constant memory because only the result of the expensive search is shared, not the corresponding search space. It is unclear which execution model is preferable or how the advantages of both can be combined in a single, purely functional, implementation.

7.2 Repeated sharing

Our library does not prevent users from writing code that shares nondeterministic computations repeatedly. Consider, for example, the following code, where the result of a computation `a` is shared three times:

```
do b <- share a
   c <- share b
   d <- share c
   d
```

Such code wastes space in the threaded store: by the first application of `share` a reference, say r_b, is reserved and `b` is a computation that, when performed, stores its result using the reference r_b or retrieves an already stored value using r_b. Similarly, `c` is a computation that, when performed, stores or retrieves its result using another reference, say r_c. In the end, when `d` is performed, the store contains three entries, stored under r_b, r_c, and r_d, which all point to the result of `a`.

The `share` function cannot inspect its argument to detect whether it is already shared because of the (Ignore) law: if the result of a call to `share` is not used later, the call to `share` must not have any effect—especially, it must not diverge if the given argument diverges. By inspecting the argument given to `share`, we would destroy the nonstrictness property that we expect from `share`.

Although it seems useless to write programs, such as the above, that share the same computation repeatedly, it is not always easy to avoid, as we now demonstrate. Repeated sharing may lead to prohibitive memory requirements because a single entry is copied an exponential number of times. The following example first demonstrates such exponential store blowup and then shows how to avoid it using a custom `Shareable` instance.

We define a data type for binary trees and automatically derive its monadic version.

```
data Bin = Tip | Bin Bin Bin
$(derive monadic ''Bin)
```

Our benchmark is to create a complete monadic binary tree and compute its size afterward. Here is the `size` function:

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

```
size :: Monad m => m (MBin m) -> m Int
size t = matchMBin t (return 1)
                     (\l r -> do m <- size l
                                 n <- size r
                                 return (m+n+1))
```

It uses the derived function `matchMBin` to pattern match on the given tree `t`, recursively computes the sizes of the left and right subtrees of branch nodes, and increments their sum to return the total size. We apply `size` to a complete binary tree generated as follows:

```
complete :: Sharing m => Int -> m (MBin m)
complete 0     = mTip
complete (n+1) = do t <- share (complete n)
                    mBin t t
```

The argument of `complete` determines the height of the generated tree. If it is zero, `complete` generates a leaf using `mTip`. If the argument is positive then the result of the recursive call is shared and used as left and right subtree of a new branch node. Since our tree is monadic, both arguments of the constructed branch node `mBin` are the monadic computation `t` that yields an `MBin m` value. Because of explicit sharing, this computation is executed only once and its result is reused.

Remember that explicit sharing is performed deeply, i.e., monadic components of an argument of `share` are shared recursively. The result of `complete` is a branch node with already shared components. When calling `share` on the result of a recursive call to `complete`, the already shared components are shared again. Especially, the number of calls to `share` on the single shared leaf of the created tree is exponential in the height of the generated tree and, hence, the leaf is inserted into the threaded store exponentially often.

We have instrumented our library code to output the number of used references after observing a computation that computes the size of a complete binary tree with the `run` function. As expected, this number is exponential in the height of the generated tree and we also noticed that the program uses a lot of memory due to the large threaded store.

The situation can be improved significantly by using a custom monadic data type for monadic binary trees instead of the automatically derived one. We also write a corresponding custom `Shareable` instance that avoids sharing components of an already shared value. Our new `MBin` is similar to the one that would be derived automatically but uses an additional flag in branch nodes to tell whether this node has already been shared:

```
data MBin m = MTip | MBin Bool (m (MBin m)) (m (MBin m))
```

When creating a branch node using `mBin`, the flag is set to `False`.

```
mBin :: Monad m => m (MBin m) -> m (MBin m) -> m (MBin m)
mBin l r = return (MBin False l r)
```

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

We omit the definition of `mTip` because leaves do not have nondeterministic components and do not need an extra flag. We also omit the `matchMBin` function and the `Convertible` instances, which ignore the additional argument of type `Bool`.

The crucial change is in the `Shareable` instance of the modified `MBin` type:

```
instance MonadPlus m => Shareable m (MBin m) where
  shareArgs _ MTip = return MTip
  shareArgs f (MBin isShared l r)
    | isShared  = return (MBin True l r)
    | otherwise = do x <- f l; y <- f r
                     return (MBin True x y)
```

The flag is set to `True` in the results of the `shareArgs` function and if it is already set on an argument of `shareArgs` then the argument is just returned unchanged instead of passing the nondeterministic components to the function `f`. Note that the components of a value with a set flag are always results of the `share` function and that processing them again would only blow up the store without leading to additional sharing. This intuition is formalized by the (Repeat) law presented in Figure 2.

When executing our benchmark using the modified `MBin` type and custom `Shareable` instance, we observe that the number of used references in the store is now linear, not exponential, in the height of the generated tree. As a consequence, the performance of the program improves significantly although the run time is, of course, still exponential because the size of the generated tree is.

The shown technique of marking shared values avoids that components of already shared nondeterministic data structures are shared repeatedly. It can be implemented without changing our library code and only requires custom declarations for the nondeterministic data types with corresponding instances. This technique does not, however, avoid repeated sharing like in the example at the beginning of this subsection: although nondeterministic components of `a` would not be shared repeatedly, the top-level constructor of `a` would still be added to the store three times. Avoiding repeated sharing also of top-level constructors is difficult, if not impossible, without sacrificing nonstrictness of the `share` combinator.

8 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.

8.1 Functional logic programming

The interaction of nonstrict and nondeterministic 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

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

to strict evaluation. An alternative interpretation of call-time choice is that variables denote *values* rather than (possibly nondeterministic) computations. As call-time choice has turned out to be the most intuitive model for lazy nondeterminism, 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, nondeterministic 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 `Nondet` instance. Hence, our approach is a step toward 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. Unlike what is common practice in FLP formalizations, our monadic approach distinguishes finite failure, represented as `mzero`, from diverging computations, and preserves the tree structure of answers.

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 & 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 nondeterministic components. Instead of using an untyped representation for nondeterministic 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 & 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 FLP in Haskell that handles logic variables explicitly and can hence implement a more efficient version of unification. It does not support data types with nondeterministic components or user-defined search

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

strategies. The authors discuss the conflict between laziness and nondeterminism in Section 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 nondeterministic computations modeled using monads in Haskell.

8.2 Call-by-need calculi

The combination of sharing and nonstrictness—known as call-by-need or lazy evaluation—has been extensively investigated theoretically. The first two "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 & 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 & 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 2). Although Ariola *et al.* add constructors as an extension of their calculus, constructed values cannot have nonvalue components. To construct data lazily, one must explicitly let-bind computations of all components, no matter how deeply nested. They do not have an analogue of our (HNF) law. Maraist *et al.* briefly discuss an extension for constructed values with nonvalue components; their law V^K corresponds to our law (HNF). Our laws (Choice) and (Fail) 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 3). Ariola *et al.* obtain equational laws as the congruence closure of (context-compatible) reductions.

Our `share` combinator resembles the *pruning* operation in Kitchin *et al.*'s Orc language (2009) for orchestrating concurrent processes. In particular, Launchbury & Elliott's embedding of Orc into Haskell (2010) provides pruning in the form of an `eagerly` combinator, which even has the same type signature as our `share`. Launchbury & Elliott propose some equational laws for `eagerly`, as we do for `share`. We do not purport to have come up with the definitive set of laws, and neither do they.

First-class stores. Since our implementations are based on store passing, they closely correspond to Launchbury's natural semantics (1993). Executing `memo a` returns a computation that behaves like a variable reference in Launchbury's semantics. Executing the variable reference for the first time evaluates the associated result and updates the store by binding the variable to this result. The main difference of

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

our evaluator is nondeterminism. We cannot get by with a single global heap—we need first-class stores (Morrisett, 1993), one for each branch of the nondeterministic computation.

Garcia *et al.* (2009) recently reduced call-by-need (again, without nondeterminism) 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 & 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.

The correspondence of our `ThunkStore` (Section 6.4) to first-class stores is not perfect. Once the result of a memoized computation is recorded in our store, it can never be removed (unless the whole store is disposed). That creates a possible memory leak (although we did not encounter problems in our experiments). We cannot remove items from our store, unless we can prove that the corresponding memoized computations are no longer used. Doing so would require cooperation with garbage collection.

8.3 Hardware description languages

The problem of explicit sharing has also been addressed in the context of hardware description languages (Bjesse *et al.*, 1998; Acosta-Gómez, 2007). 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, Section 2.4.1) discusses four different solutions to this problem:

Explicit labels. 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. 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. 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. 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 & Huch, 2009). The problem of impure features—especially their hindering compiler optimizations—seems much more severe in FLP than in hardware description. As nondeterministic computations are usually expressed monadically in Haskell anyway,

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

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.

9 Conclusions

We have presented an equational specification and an efficient implementation of nonstrictness, sharing, and nondeterminism embedded in a pure functional language.

Our specification (Figure 2) 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 nondeterministic computations. Our equational laws for lazy nondeterminism can be used to reason about the meaning of nondeterministic 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 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 and search types, in expressing the sharing of nondeterministic 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 Section 6.5. Initial comparisons of monadic computations with corresponding computations in Curry that use nondeterminism, narrowing, and unification are very promising. We achieve the performance of the currently fastest Curry compiler (MCC) on the highly nondeterministic 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.

Appendix. Complete example of using our library

In this Appendix, we discuss the pragmatics of using our implementation by showing the complete implementation of lazy monadic permutation sort in one place.

Figure A1 contains the definition of `sort` and the required auxiliary functions. Figure A2 contains the implementation of the monadic list data type along with auxiliary functions and necessary type-class instances. The definitions in Figure A2 can be imported from the module `Data.Monadic.List` but we include them here to guide the definition of other monadic data types and corresponding instances. Users who are satisfied with the default definitions of monadic data types, constructor functions, and type-class instances can derive them as described in Section 6.3.

Based on definitions in Figure A1, we can use the `convert` function (Section 6.2) to create computations that can be observed as follows.

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

```
import Control.Monad.Sharing

sort :: Sharing m => m (List m Int) -> m (List m Int)
sort xs = do ys <- share (perm xs)
             True <- isSorted ys
             ys

perm :: MonadPlus m => m (List m a) -> m (List m a)
perm xs = do l <- xs; case l of
                        Nil       -> nil
                        Cons y ys -> insert y (perm ys)

insert :: MonadPlus m => m a -> m (List m a) -> m (List m a)
insert x xs = cons x xs `mplus` do Cons y ys <- xs
                                   cons y (insert x ys)

isSorted :: Monad m => m (List m Int) -> m Bool
isSorted ml =
  do l <- ml
     case l of
       Nil         -> return True
       Cons mx mxs ->
         do xs <- mxs
            case xs of
              Nil         -> return True
              Cons my mys ->
                do x <- mx; y <- my
                   if x <= y
                     then isSorted (cons (return y) mys)
                     else return False
```

Fig. A1. The lazy monadic permutation sort function.

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

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 xs = return (Cons x xs)

instance Shareable m a => Shareable m (List m a) where
  shareArgs Nil         = return Nil
  shareArgs (Cons x xs) = do y <- share x
                             ys <- share xs
                             return (Cons y ys)

instance Convertible m a b => Convertible m [a] (List m b) where
  convert []     = nil
  convert (x:xs) = cons (convert x) (convert xs)

instance Convertible m a b => Convertible m (List m a) [b] where
  convert Nil         = return []
  convert (Cons x xs) = do y <- x >>= convert
                           ys <- xs >>= convert
                           return (y:ys)
```

Fig. A2. Implementation of lists with monadic components.

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

```
printResults :: Show a => [a] -> IO ()
printResults []     = putStrLn "No results"
printResults (x:xs) = printNext x xs

printNext :: Show a => a -> [a] -> IO ()
printNext x xs = print x >> printMore xs

printMore :: Show a => [a] -> IO ()
printMore []     = return ()
printMore (x:xs) =
  do putStr "more? [(Y)es, (n)o, (a)ll]: "
     s <- getLine
     if null s then printNext x xs
       else case toLower (head s) of
              'y' -> printNext x xs
              'n' -> return ()
              'a' -> mapM_ print (x:xs)
              _   -> printMore (x:xs)
```

Fig. A3. Printing a list of results interactively.

```
permComp :: forall s . Sharing s => s [Int]
permComp = perm (convert [1..(3::Int)] :: s (List s Int)) >>= convert

permSortComp :: Sharing s => s [Int]
permSortComp = sort (convert [10,9..(1::Int)]) >>= convert
```

`permComp` is a computation that computes a permutation of the list `[1,2,3]` nondeterministically.[13] `permSortComp` sorts the list `[10,9..1]` using the lazy nondeterministic permutation sort function.

As both computations are polymorphic over the used sharing monad, we can pass them to the function `results` to produce a set of results. Below, we use the observation function `resultList` (Section 5.1) which returns, in the `IO` monad, a list of (possibly duplicated, unordered) results that can be processed interactively using `printResults` (Figure A3). A session in the interactive Haskell Environment GHCi that uses this function may look as follows.

```
ghci> resultList permComp >>= printResults
[1,2,3]
more? [(Y)es, (n)o, (a)ll]: yes
[1,3,2]
more? [(Y)es, (n)o, (a)ll]: all
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
ghci> resultList permSortComp >>= printResults
```

[13] This definition of `permComp` uses the language extension `ScopedTypeVariables` to restrict the result type of the inner call of the `convert` function.

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press

```
[1,2,3,4,5,6,7,8,9,10]
ghci>
```

Acknowledgments

We thank Greg Morrisett for pointing us to first-class stores, Bernd Braßel and Michael Hanus for examining drafts of this paper, and our reviewers for insightful suggestions that helped improve this work.

References

Acosta-Gómez, A. (2007) *Hardware Synthesis in ForSyDe*. Master's thesis, Stockholm, Sweden: Department of Microelectronics and Information Technology, Royal Institute of Technology.

Albert, E., Hanus, M., Huch, F., Oliver, J. & Vidal, G. (2005) Operational semantics for declarative multi-paradigm languages. *J. Symb. Comput.* **40**(1), 795–829.

Antoy, S. & Hanus, M. (2002) Functional logic design patterns. In *Proceedings of Symposium on Functional and Logic Programming (FLOPS)*, pp. 67–87.

Antoy, S. & Hanus, M. (2006) Overlapping rules and logic variables in functional logic programs. In *Proceedings of International Conference on Logic Programming*, pp. 87–101.

Ariola, Z. M. & Felleisen, M. (1997) The call-by-need lambda calculus. *J. Funct. Program.* **7**(3), 265–301.

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M. & Wadler, P. (1995) The call-by-need lambda calculus. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pp. 233–246.

Bird, R., Jones, G. & de Moor, O. (1997) More haste, less speed: Lazy versus eager evaluation. *J. Funct. Program.* **7**(5), 541–547.

Bjesse, P., Claessen, K., Sheeran, M. & Singh, S. (1998) Lava: Hardware design in Haskell. In *Proceedings of International Conference on Functional Programming (ICFP)*, pp. 174–184.

Braßel, B. & Huch, F. (2007) On a tighter integration of functional and logic programming. In *Proceedings of Asian Symposium on Programming Languages and Systems (APLAS)*, Shao, Z. (ed), LNCS, vol. 4807, Berlin: Springer, pp. 122–138.

Braßel, B. & Huch, F. (2009) The Kiel Curry System KiCS. In *Proceedings of Workshop on Logic Programming (WLP)*, 195–205.

Christiansen, J. & Fischer, S. (2008) EasyCheck—Test data for free. In *Proceedings of Symposium on Functional and Logic Programming (FLOPS)*, pp. 322–336.

Claessen, K. (2004) Parallel parsing processes. *J. Funct. Program.* **14**(6), 741–757.

Claessen, K. & Hughes, J. (2000). QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of International Conference on Functional Programming (ICFP)*, pp. 268–279.

Escardó, M. H. (2007) Infinite sets that admit fast exhaustive search. In *Proceedings of IEEE Symposium on Logic in Computer Science (LICS)*, pp. 443–452.

Felleisen, M. (1985) *Transliterating Prolog into Scheme*. Technical Report 182. Computer Science Department, Indiana University.

Filinski, A. (1999) Representing layered monads. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pp. 175–188.

Fischer, S. & Kuchen, H. (2007) Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of Principles and Practice of Declarative Programming (PPDP)*, pp. 63–74.

Garcia, R., Lumsdaine, A. & Sabry, A. (2009) Lazy evaluation & delimited control. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pp. 153–164.

González-Moreno, J. C., Hortalá-González, M. T., López-Fraguas, F. J. & Rodríguez-Artalejo, M. (1999) An approach to declarative programming based on a rewriting logic. *J. Log. Program.* **40**(1), 47–87.

Goodman, N., Mansinghka, V. K., Roy, D., Bonawitz, K. & Tenenbaum, J. B. (2008). Church: A language for generative models. In *Proceedings of Conference on Uncertainty in Artificial Intelligence*, pp. 220–229.

Haynes, C. T. (1987) Logic continuations. *J. Log. Prog.* **4**(2), 157–176.

Hinze, R. (2000) Deriving backtracking monad transformers (functional pearl). In *Proceedings of International Conference on Functional Programming (ICFP)*, pp. 186–197.

Hudak, P. (1996) Building domain-specific embedded languages. *ACM Comput. Surv.* **28**(4es): 196.

Hughes, J. (1989) Why functional programming matters. *Comput. J.* **32**(2), 98–107.

Kiselyov, O. & Shan, C. (2009) Embedded probabilistic programming. In *Proceedings of the Working Conference on Domain-Specific Languages*, pp. 360–384.

Kiselyov, O., Shan, C., Friedman, D. P. & Sabry, A. (2005) Backtracking, interleaving, and terminating monad transformers (functional pearl). In *Proceedings of International Conference on Functional Programming (ICFP)*, pp. 192–203.

Kitchin, D., Quark, A., Cook, W. R. & Misra, J. (2009) The Orc programming language. In *Proceedings of FMOODS 2009 and FORTE 2009 (Formal Techniques for Distributed Systems)*, Lee, D., Lopes, A. & Poetzsch-Heffter, A. (eds), LNCS 5522. Berlin: Springer, pp. 1–25.

Koller, D., McAllester, D. & Pfeffer, A. (1997) Effective Bayesian inference for stochastic programs. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pp. 740–747.

Launchbury, J. (1993) A natural semantics for lazy evaluation. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pp. 144–154.

Launchbury, J. & Elliott, T. (2010) Concurrent orchestration in Haskell. In *Haskell Symposium*, Gibbons, J. (ed). ACM Press, pp. 79–90.

Launchbury, J. & Peyton Jones, S. L. (1994) Lazy functional state threads. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, pp. 24–35.

López-Fraguas, F. J., Rodríguez-Hortalá, J. & Sánchez-Hernández, J. (2007) A simple rewrite notion for call-time choice semantics. In *Proceedings of Principles and Practice of Declarative Programming (PPDP)*, pp. 197–208.

López-Fraguas, F. J., Rodríguez-Hortalá, J. & Sánchez-Hernández, J. (2008) Rewriting and call-time choice: The HO case. In *Proceedings of Symposium on Functional and Logic Programming (FLOPS)*, pp. 147–162.

Maraist, J., Odersky, M. & Wadler, P. (1998) The call-by-need lambda calculus. *J. Funct. Program.* **8**(3), 275–317.

McCarthy, J. (1963) A basis for a mathematical theory of computation. In *Proceedings of Computer Programming and Formal Systems*. North-Holland, pp. 33–70.

Michie, D. (1968) "Memo" functions and machine learning. *Nature* **218**, 19–22.

Moggi, E. (1989) Computational lambda-calculus and monads. In *Proceedings of Logic in Computer Science (LICS)*. Piscataway, NJ, USA: IEEE Press, pp. 14–23.

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

Morrisett, J. G. (1993) Refining first-class stores. In *Proceedings of the Workshop on State in Programming Languages*, pp. 73–87.

Naylor, M., Axelsson, E. & Runciman, C. (2007) A functional-logic library for Wired. In *Proceedings of Haskell Workshop*, pp. 37–48.

Nicollet, V., Minsky, Y. & Leroy, X. (2009) Lazy and threads. `http://caml.inria.fr/pub/ml-archives/caml-list/2009/02/9fc4e4a897ce7a356674660c8cfa5ac0.fr.html`.

Peyton Jones, S. L., Reid, A., Hoare, T., Marlow, S. & Henderson, F. (1999) A semantics for imprecise exceptions. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, pp. 25–36.

Rabin, M. O. & Scott, D. (1959) Finite automata and their decision problems. *IBM J. Res. Dev.* **3**, 114–125.

Runciman, C., Naylor, M. & Lindblad, F. (2008) SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In *Proceedings of Haskell Symposium*, pp. 37–48.

Seaman, J. M. (1993) *An Operational Semantics of Lazy Evaluation for Analysis*. PhD. thesis. Pennsylvania State University.

Spivey, J. M. (2000) Combinators for breadth-first search. *J. Funct. Program.* **10**(4), 397–408.

Tolmach, A. & Antoy, S. (2003) A monadic semantics for core Curry. In *Proceedings of Workshop on Functional and Logic Programming (WFLP)*, Valencia, Spain, pp. 33–46.

Tolmach, A., Antoy, S. & Nita, M. (2004) Implementing functional logic languages using multiple threads and stores. In *Proceedings of International Conference on Functional Programming (ICFP)*, pp. 90–102.

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

Wadler, P. 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 *Proceedings of Functional Programming Languages and Computer Architecture*, pp. 113–128.

Wadler, P. L. (1992) The essence of functional programming. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*. ACM Press, pp. 1–14.

https://doi.org/10.1017/S0956796811000189 Published online by Cambridge University Press