All Sorts of Permutations (Functional Pearl)

Jan Christiansen
Flensburg University of Applied Sciences, Germany
jan.christiansen@hs-flensburg.de

Nikita Danilenko
University of Kiel, Germany
nda@informatik.uni-kiel.de

Sandra Dylus
University of Kiel, Germany
sad@informatik.uni-kiel.de

Abstract

The combination of non-determinism and sorting is mostly associated with permutation sort, a sorting algorithm that is not very useful for sorting and has an awful running time.

In this paper we look at the combination of non-determinism and sorting in a different light: given a sorting function, we apply it to a non-deterministic predicate to gain a function that enumerates permutations of the input list. We get to the bottom of necessary properties of the sorting algorithms and predicates in play as well as discuss variations of the modelled non-determinism.

On top of that, we formulate and prove a theorem stating that no matter which sorting function we use, the corresponding permutation function enumerates all permutations of the input list. We use free theorems, which are derived from the type of a function alone, to prove the statement.

Categories and Subject Descriptors D.1.4 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs

General Terms Languages, Algorithms

Keywords Haskell, monads, non-determinism, permutation, sorting, free theorems

1. Introduction

In a functional language, non-deterministic functions can be expressed by using lists to model multiple non-deterministic results. In order to explicitly distinguish list values in the common sense and list values that are used to model non-determinism, we introduce the following type synonym, which is simply a renaming of the list data type. That is, in the following we use the type ND for all lists that represent non-deterministic choices.

$$\mathbf{type}\ ND\ \alpha = [\alpha]$$

We can naturally extend well-known functions to support non-determinism by simply applying functions to all non-deterministic results and combine all the non-deterministic results of these applications. In the list model we can use the function $concatMap$ – whose type is $(\alpha \to ND\ \beta) \to ND\ \alpha \to ND\ \beta$ in a non-deterministic setting – to apply a non-deterministic function to all choices of a non-deterministic value and combine the results via concatenation.

[Copyright notice will appear here once 'preprint' option is removed.]

Let us consider the following Haskell function[1]

$$filterND :: (\alpha \to ND\ Bool) \to [\alpha] \to ND\ [\alpha],$$

which is a non-deterministic extension of the well-known higher-order function $filter$. It is folklore knowledge that some non-deterministic extensions of predicate-based higher-order functions can be used to derive new non-deterministic functions by using a predicate that yields $True$ and $False$. For example, when we apply $filterND$ to the non-deterministic predicate

$$coinPredND :: \alpha \to ND\ Bool$$
$$coinPredND\ _ = [True, False]$$

we get a function that non-deterministically enumerates all sublists of a given list.

Intuitively, when we apply $filterND$ to $coinPredND$ and a list xs, the resulting function non-deterministically chooses to keep or remove it from the result list for every element of xs. This decision is made for every element in the argument list independently, hence, we get all sublists of the argument list.

Similarly, this "trick" can be used to implement a function enumerating all permutations by sorting with a non-deterministic binary predicate. That is, for some non-deterministic version of a sorting algorithm

$$sortND :: (\alpha \to \alpha \to ND\ Bool) \to [\alpha] \to ND\ [\alpha],$$

when applied to the non-deterministic comparison function

$$coinCmpND :: \alpha \to \alpha \to ND\ Bool$$
$$coinCmpND\ _\ _ = [True, False]$$

the resulting function enumerates all permutations of its argument.

Although improbable, to the best of our knowledge this connection has been first noted in the context of functional logic programming on the mailing list of Curry in a thread by Fischer and Christiansen (2009). A functional logic programming language can be considered as a functional language with built-in non-determinism similar to the non-determinism provided by the above construction.

Another thread on the Haskell mailing list by Pollard et al. (2009) discusses these connections in more detail and in the context of the functional programming language Haskell. These two mailing list threads raise the following interesting questions.

- Does the comparison function have to be consistent? That is, does the comparison function have to make the same decisions if one element is less or equal to another one when it is invoked multiple times within a specific non-deterministic branch.
- Does the comparison function have to be transitive? The correctness of most sorting algorithms relies on the fact that the comparison function is transitive.

[1] The predefined Haskell function is called $filterM$ and actually has a more general type, namely $Monad\ \mu \Rightarrow (\alpha \to \mu\ Bool) \to [\alpha] \to \mu\ [\alpha]$, but for the sake of accessibility we consider the more specific type here.

- Does the enumeration version of a sorting algorithm generate results multiple times or results that are not actually permutations of the input list?
- Do we have to use a set-based rather than a multiset-based (like with lists) approach for non-determinism in order to enumerate every permutation exactly once?

Finally, the thread by Pollard et al. (2009) contains some informal reasoning why every sorting algorithm should indeed be able to generate all permutations of a list.

> "Every sorting algorithm $[Int] \rightarrow [Int]$ that actually sorts can describe every possible permutation (if there is a permutation that cannot be realised by the sorting algorithm then there is an input list that cannot be sorted). Hence, if this sorting algorithm is $sort\ p$ for some predicate p then there are possible decisions of p to produce every possible permutation. If p makes every decision non-deterministically then certainly the specific decisions necessary for any specific permutation are also made."
>
> Sebastian Fischer

If you keep on reading, you can expect to read about the following.

- We answer all the questions raised by the two mailing list threads by a case study of common sorting algorithms.
- We formally prove the quote above, namely, that the non-deterministic extension of any sorting algorithm enumerates every permutation if it is applied to the predicate $coinCmpND$. Instead of considering a list version, we use a monadic extension of these functions to prove this statement.
- It might not be surprising, that the non-deterministic extension of every sorting algorithm is able to enumerate all permutations. However, we will see that there are a couple of sorting functions that enumerate every permutation exactly once, even if they are applied to a predicate as simple as $coinCmp$.
- In contrast there are sorting functions that enumerate elements that are not even permutations of the argument, when they are applied to $coinCmp$. As assumed there even is a sorting algorithm that relies on a non-deterministic predicate that respects transitivity in order to enumerate all permutations exactly once. However, it is none of the algorithms that first come to mind.

We can generalise the list-based non-determinism presented above to a more general, monadic approach. Let us consider a slightly different implementation of the non-deterministic function $coinCmpND$.

$$coinCmpND :: \alpha \rightarrow \alpha \rightarrow ND\ Bool$$
$$coinCmpND\ _\ _ = singleton\ True +\!\!+ singleton\ False$$

Instead of constructing the list explicitly, we use a function $singleton$, which yields a singleton list, and the list concatenation $+\!\!+$. In order to generalise the function $coinCmpND$ we use the type class $MonadPlus$.

Here and in the following we introduce potentially advanced concepts like $MonadPlus$ in info boxes like the following; readers familiar with a concept can skip the according box.

Type class $MonadPlus$

An instance of the type class $MonadPlus$ is a type constructor μ that provides the following operations.

$$mzero :: \mu\ \alpha$$
$$(\oplus) :: \mu\ \alpha \rightarrow \mu\ \alpha \rightarrow \mu\ \alpha$$

List instance of $MonadPlus$

The list data type is an instance of $MonadPlus$.

$$\textbf{instance}\ MonadPlus\ [\,]\ \textbf{where}$$
$$mzero = [\,]$$
$$xs \oplus ys = xs +\!\!+ ys$$

Every type constructor – like lists – that is an instance of the type class $MonadPlus$ has to be an instance of the type class $Monad$ as well. In the following we introduce the type class $Monad$.

Type class $Monad$

An instance of the type class $Monad$ is a type constructor μ that provides the following operations.

$$return :: \alpha \rightarrow \mu\ \alpha$$
$$(\gg\!=) :: \mu\ \alpha \rightarrow (\alpha \rightarrow \mu\ \beta) \rightarrow \mu\ \beta$$

List instance of $Monad$

The list data type is an instance of $Monad$.

$$\textbf{instance}\ Monad\ [\,]\ \textbf{where}$$
$$return\ x = singleton\ x$$
$$xs \gg\!= f = concatMap\ f\ xs$$

As the list data type is an instance of the type classes $Monad$ and $MonadPlus$, we can generalise $coinCmpND$ as follows. We replace the functions $singleton$ and $+\!\!+$ by their monadic counterparts $return$ and $\oplus$, respectively. In the end we obtain the following generalised definition of a non-deterministic comparison function $coinCmp$.

$$coinCmp :: MonadPlus\ \mu \Rightarrow \alpha \rightarrow \alpha \rightarrow \mu\ Bool$$
$$coinCmp\ _\ _ = return\ True \oplus return\ False$$

Readers that are not so familiar with the type classes $Monad$ and $MonadPlus$ can always think of the type μ as the type ND, of $return$ as $singleton$ and of $\oplus$ as $+\!\!+$ in the following code examples. If the reader prefers a more abstract view, the functions $mzero$, $return$, and $mplus$ can be considered as the basic building blocks for non-determinism. Here, $mzero$ represents a failure, that is, no result, $return$ lifts a single value into a non-deterministic context, that is, $return$ represents a single result, and $\oplus$ is a non-deterministic choice between two non-deterministic values.

The function $coinCmp$ explicitly introduces non-determinism – as it chooses between the values $True$ and $False$. Therefore, $coinCmp$ uses the type class $MonadPlus$. In contrast neither the non-deterministic filter nor the non-deterministic sorting function introduces any kind of non-determinism. All non-determinism is provided by the potentially non-deterministic predicate. As these functions do not introduce non-determinism, we can use the less strong type class $Monad$ instead of $MonadPlus$ for their types.

$$filterM :: Monad\ \mu \Rightarrow (\alpha \rightarrow \mu\ Bool) \rightarrow [\alpha] \rightarrow \mu\ [\alpha]$$
$$sortM\ \ :: Monad\ \mu \Rightarrow (\alpha \rightarrow \alpha \rightarrow \mu\ Bool) \rightarrow [\alpha] \rightarrow \mu\ [\alpha]$$

In the following, we call a function that is polymorphic in a monadic type constructor a monadic function and use a subscript type in order to denote a concrete instance of such a monadic function. For example, if fM is a monadic function, we denote the concrete instance by fM_{κ}, where κ is the concrete monad. That is, $filterM_{[\,]}$ corresponds to $filterND$. Moreover, in order to keep the code short, we use the following type synonym for non-deterministic comparison functions.

$$\textbf{type}\ Cmp\ \alpha\ \mu = \alpha \rightarrow \alpha \rightarrow \mu\ Bool$$

We will use the terminology of the *less than or equal* relation on integers even when we talk about an arbitrary comparison function to keep things simple. That is, when we say that *value A is smaller than value B*, we are referring to the *less than* relation that is provided by the context. Finally, our reasoning will not take general recursion into account. Instead our proofs will be "morally correct" in the sense of Danielsson et al. (2006). That is, although we only consider total functions and finite data structures, the statements still hold with these restrictions in a language like Haskell.

2. Insertion Sort

The first sorting algorithm we consider is insertion sort. We begin with a simple warm-up exercise and implement a standard pure version. At first, we implement a function that inserts an element into a list. The element is inserted in front of the first element in the list that is greater than or equal to the element to insert.

```
insert :: (α → α → Bool) → α → [α] → [α]
insert _ x []              = [x]
insert p x yys@(y : ys) =
  if p x y then x : yys else y : insert p x ys
```

By means of $insert$ we can define a function to sort a list as follows.

```
insertSort :: (α → α → Bool) → [α] → [α]
insertSort _ []       = []
insertSort p (x : xs) = insert p x (insertSort p xs)
```

As an example of the application, it hopefully comes as no surprise that $insertSort\ (\leqslant)\ xs$ sorts the elements of the list xs in ascending order with respect to $\leqslant$.

In order to apply this sorting function to a non-deterministic predicate, we have to lift it to a monadic context. More specifically we have to transform a function of type

$$(\alpha \to \alpha \to \quad Bool) \to [\alpha] \to \quad [\alpha]$$

into a function of type

$$Monad\ \mu \Rightarrow (\alpha \to \alpha \to \mu\ Bool) \to [\alpha] \to \mu\ [\alpha]$$

This monadic extension has to satisfy only one simple requirement, namely, when we use no effect, that is, we instantiate the monad with the identity monad, the resulting function has to behave like the original function. The identity monad is a data type with a single constructor containing a value. This data type is an instance of the type class $Monad$. We use the name Id for the type and the constructor and $runId$ for a function that extracts the value.

Id instance of $Monad$

The data type Id is an instance of $Monad$.

```
instance Monad Id where
  return x = Id x
  Id x >>= f = Id (f x)
```

More formally, for every sorting function

$$sort\ ::\ (\alpha \to \alpha \to \quad Bool) \to [\alpha] \to \quad [\alpha]$$

the monadic extension

$$sortM :: Monad\ \mu \Rightarrow (\alpha \to \alpha \to \mu\ Bool) \to [\alpha] \to \mu\ [\alpha]$$

has to satisfy

$$runId \circ sortM_{Id}\ (\lambda x\ y \to Id\ (p\ x\ y)) \equiv sort\ p.$$

for all $p :: \tau \to \tau \to Bool$.

Here and in the following we use the letters α and μ for type variables and τ and κ for concrete type instances.

The monadic liftings of $insert$ and $insertSort$ are defined as follows[2].

```
insertM :: Monad μ ⇒ Cmp α μ → α → [α] → μ [α]
insertM _ x []              = return [x]
insertM p x yys@(y : ys) =
  p x y >>= λb →
    if b then return (x : yys)
         else fmap (y:) (insertM p x ys)
insertSortM :: Monad μ ⇒ Cmp α μ → [α] → μ [α]
insertSortM _ []       = return []
insertSortM p (x : xs) =
  insertSortM p xs >>= λys → insertM p x ys
```

The operation $\gg=$ of the $Monad$ type class plays the role of a sequencing operator between two (or more) expressions. In the context of monads, Haskell adopted the so-called **do**-notation that smooths the handling of these sequencing operators in function definitions. A **do**-block contains monadic expressions and $\leftarrow$-symbols, where each $\gg=$ operator is implicitly added between two monadic expressions that are separated by new lines.

We can rewrite the above function definitions and obtain the following code that explicitly uses **do**-notation. As a side benefit, the definition looks very similar to the original implementation and can thus be read naturally.

```
insertM :: Monad μ ⇒ Cmp α μ → α → [α] → μ [α]
insertM _ x []              = return [x]
insertM p x yys@(y : ys) = do
  b ← p x y
  if b then return (x : yys)
       else fmap (y:) (insertM p x ys)
insertSortM :: Monad μ ⇒ Cmp α μ → [α] → μ [α]
insertSortM _ []       = return []
insertSortM p (x : xs) = do
  ys ← insertSortM p xs
  insertM p x ys
```

Next, we want to test if the requirement of the monadic lifting holds for $insertSortM$ and it is still capable of sorting. For simplicity, we only consider the comparison of values of type Int to demonstrate the sorting capability, but we could use comparison functions for other types as well.

```
cmp_Id :: Cmp Int Id
cmp_Id x y = return (x ⩽ y)
```

Now we are ready to apply the monadic version of $insertSort$ to this predicate. Et voilà, we get the original sorting capability.

```
sort1 :: [Int]
sort1 = runId (insertSortM cmp_Id (reverse [1 .. 5]))
  -- [1, 2, 3, 4, 5]
```

It is good to know that the monadic extension of a sorting function can still sort the input list, but this is not the exciting application of this function. The monadic extension becomes far more interesting when we use $coinCmp$ as predicate: by means of the non-deterministic function $coinCmp$ we can define a function that enumerates all permutations.

```
perms1 :: [[Int]]
perms1 = insertSortM coinCmp [1 .. 3]
  -- [[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]
```

And, indeed, we get a function that enumerates exactly all permutations when we use the list monad.

[2] The function $fmap\ f\ m$ is defined as $m \gg= return \circ f$.

Monad Laws

$$(m \gg\!= f) \gg\!= g \equiv m \gg\!= \lambda x \to (f\ x \gg\!= g)$$
(Bind associativity)

$$return\ x \gg\!= f \equiv f\ x \quad \text{(Bind left identity)}$$

$$m \gg\!= return \equiv m \quad \text{(Bind right identity)}$$

MonadPlus Laws

$$mzero \oplus m \equiv m \quad \text{(MPlus left zero)}$$

$$m \oplus mzero \equiv m \quad \text{(MPlus right zero)}$$

$$m \oplus n \gg\!= f \equiv (m \gg\!= f) \oplus (n \gg\!= f)$$
(Bind-MPlus left distributive)

Miscellaneous

$$return\ x \oplus return\ x \equiv return\ x \quad \text{(MPlus idempotent)}$$

$$m \gg\!= \lambda x \to n \oplus o \equiv (m \gg\!= \lambda x \to n) \oplus (m \gg\!= \lambda x \to o)$$
(Bind-MPlus right distributive)

Figure 1. Various monadic rules.

What is happening here exactly? In order to "demystify" the behaviour of the non-deterministic predicate, we inline the predicate into the monadic function *insertM*. We use the suffix *NDM* for the non-deterministic versions that result from inlining the non-deterministic predicate. If we consider the empty list as second argument of *insertM*, we get the right hand side

$$return\ [x].$$

For the case of a non-empty list $y : ys$ we get the right hand side

$$return\ (x : y : ys) \oplus fmap\ (y{:})\ (insertCoin\ x\ ys)$$

for the application $insertM\ \ coinCmp\ \ x\ \ (y : ys)$. We do not present this transformation here as it is straightforward: it uses the definitions of *coinCmp* and *insertM* as well as a couple of monadic laws. Here and in the following we make use of a variety of laws related to instances of *Monad* and *MonadPlus*, these laws are presented in Figure 1. Inlining the predicate *coinCmp* into *insertM* yields the following definition.

```
insertNDM :: MonadPlus μ ⇒ α → [α] → μ [α]
insertNDM x []             = return [x]
insertNDM x yys@(y : ys) =
  return (x : yys) ⊕ fmap (y:) (insertNDM x ys)
```

The function *insertNDM*, which is derived after inlining *insertM* used with a non-deterministic predicate, is exactly the definition of a non-deterministic insertion that is used to define a permutation function, for example in the context of the functional logic programming language Curry (Hanus 1994). According to Sedgewick (1977) this permutation algorithm was developed independently by Trotter (1962) and Johnson (1963). The implementation of the *permutations* function in Haskell is also based on this approach but has been improved with respect to non-strictness via a mailing list discussion by van Laarhoven et al. (2007). Exploring these improvements would probably allow for a separate paper.

3. Selection Sort

In this section we consider the permutation algorithm that can be derived from selection sort. After the warm-up in the previous section, we feel comfortable enough not to implement the original selection sort implementation first, but directly provide its monadic

```
                   +-[1,2,3]
        +- 1 <= 2 -+
        |          |         +-[2,1,3]
        |          +- 1 <= 3 -+
        |                    +-[2,3,1]
+- 2 <= 3 -+
        |          +-[1,3,2]
        +- 1 <= 3 -+
                   |         +-[3,1,2]
                   +- 1 <= 2 -+
                             +-[3,2,1]
```

Figure 2. The decision tree of *insertSort*.

extension. Selection sort is based on the idea of finding the minimum of a list and putting this minimum in front of the result list. The argument list without the minimum is sorted recursively.

```
minM :: Monad μ ⇒ Cmp α μ → α → α → μ α
minM p x y = do
  b ← p x y
  return (if b then x else y)
minimumM :: Monad μ ⇒ Cmp α μ → [α] → μ α
minimumM _ [x]      = return x
minimumM p (x : xs) = do
  y ← minimumM p xs
  minM p x y
selectSortM :: (Eq α, Monad μ)
            ⇒ Cmp α μ → [α] → μ [α]
selectSortM _ [] = return []
selectSortM p xs = do
  x ← minimumM p xs
  fmap (x:) (selectSortM p (delete x xs))
```

Again, when considering the identity instance of this function, we get the original sorting function and if we use the non-deterministic predicate instead, we get a permutation enumeration.

```
perms2 :: [[Int]]
perms2 = selectSortM coinCmp [1 .. 3]
  -- [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1]
  -- , [1, 2, 3], [1, 3, 2], [3, 1, 2], [3, 2, 1]]
```

However, in contrast to insertion sort, the resulting function enumerates some permutations multiple times. More precisely, we get

$$2^{\frac{n(n-1)}{2}}$$

permutations, where n is the length of the argument list. Note that this function grows much faster than the number of permutations $n!$; for example, for $n = 10$ we have $n! = 3\ 628\ 800$ permutations, as if this is not bad enough, a modified version of *perms2* with the list $[1 .. 10]$ yields

$$2^{\frac{10*9}{2}} = 2^{45} = 35\ 184\ 372\ 088\ 832$$

results.

In order to understand the difference between insertion sort and selection sort, we have a look at the decision trees of these algorithms. Decision trees reflect all possible control flows for a sorting function depending on the results of the comparison function. Figure 2 shows the decision tree of insertion sort whereas Figure 3 shows the tree for selection sort[3]. The numbers in these

[3] The ASCII art diagrams are generated by running the monadic sorting function in the free monad and pretty printing the underlying *MonadPlus* structure.

decision trees can be considered as the position of the element with respect to the input list – note, in contrast to an index we are starting at position 1. That is, the term `2 <= 3` denotes the comparison of the element at position 2 with the element at position 3. The upper successor reflects the result *True* and the lower successor *False*.

We can observe that the tree for selection sort contains the same comparisons multiple times. For example, initially we compare 2 and 3 and after we have compared 1 with 2, we again compare 2 and 3. Obviously, the path that ends in the upper result underlined with `^^^` will never be taken by the original sorting algorithm. There is no pure comparison function that yields *True* for a comparison first and *False* later. The same argument applies to the second underlined result. In the context of a non-deterministic predicate, we call this behaviour *consistent*: a non-deterministic predicate behaves consistently, if it yields the same Boolean value for every application to the same pair of values.

As a next step, we define a modification of our non-deterministic predicate that is consistent. We benefit from our monadic implementation as we can simply add a state transformer to record the choices we make. By checking whether we have made a specific choice before, we get a consistent non-deterministic predicate.

We will not introduce the implementation of a state transformer here but refer the interested reader to its introduction by Liang et al. (1995). Intuitively, by adding a state transformer to an instance of *MonadPlus*, for every non-deterministic branch we add a separate state. We use this state to remember the choices we have made before within one non-deterministic branch.

For convenience we use a simple list to record choices, but we could as well use a more efficient data structure like a search or radix tree.

$$\textbf{type}\ \mathit{Choices}\ \alpha = [((\alpha,\alpha),\mathit{Bool})]$$

$$\mathit{noChoices} :: \mathit{Choices}\ \alpha$$
$$\mathit{noChoices} = [\,]$$

$$\mathit{addChoice} :: \alpha \to \alpha \to \mathit{Bool} \to \mathit{Choices}\ \alpha \to \mathit{Choices}\ \alpha$$
$$\mathit{addChoice}\ x\ y\ b\ cs = ((x,y),b) : cs$$

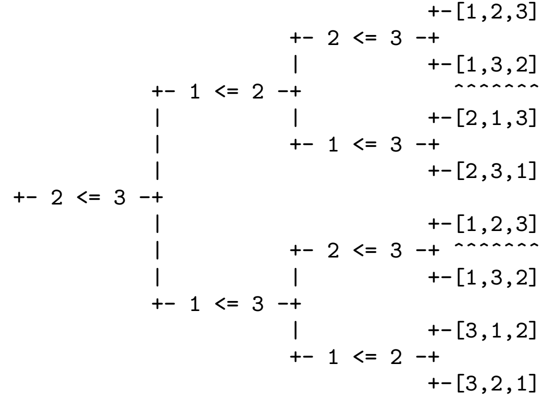
By means of *addChoice* we define a function that remembers the choice of a non-deterministic predicate. The following function records the choice of the provided non-deterministic predicate and additionally takes a function that transforms the list of choices after adding a new choice. This function will be the identity in the case of consistency and will be of greater interest later.

```
store :: (Eq α, MonadPlus μ)
      ⇒ (Choices α → Choices α)
      → Cmp α μ → Cmp α (StateT (Choices α) μ)
store update p x y = do
  b ← lift (p x y)
  modify (update ∘ addChoice x y b)
  return b
```

The final missing piece is a function that looks up whether we have made a choice before. If not, we use the provided state-based predicate to make the choice and store it. Otherwise, thus, if we have made the choice before, we simply yield this choice.

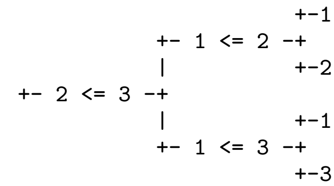
```
check :: (Eq α, MonadPlus μ)
      ⇒ Cmp α (StateT (Choices α) μ)
      → Cmp α (StateT (Choices α) μ)
check p x y = do
  s ← get
  maybe (p x y) return (lookup (x, y) s)
```

By means of these helper functions we define a non-deterministic choice that is consistent.



```
                                    +-[1,2,3]
                    +- 2 <= 3 -+
                    |           +-[1,3,2]
        +- 1 <= 2 -+             ^^^^^^^
        |           |           +-[2,1,3]
        |           +- 1 <= 3 -+
        |                       +-[2,3,1]
+- 2 <= 3 -+
        |                       +-[1,2,3]
        |           +- 2 <= 3 -+ ^^^^^^^
        |           |           +-[1,3,2]
        +- 1 <= 3 -+
                    |           +-[3,1,2]
                    +- 1 <= 2 -+
                                +-[3,2,1]
```

Figure 3. The decision tree of *selectSort*.



```
                    +-1
          +- 1 <= 2 -+
          |          +-2
+- 2 <= 3 -+
          |          +-1
          +- 1 <= 3 -+
                     +-3
```

Figure 4. The decision tree of *minimumM*.

```
consistentCoin :: (Eq α, MonadPlus μ)
               ⇒ Cmp α (StateT (Choices α) μ)
consistentCoin = check (store id coinCmp)
```

By making use of *consistentCoin* we indeed get a permutation enumeration function from *selectSortM* that enumerates exactly all permutations.

```
perms2Cons :: [[Int]]
perms2Cons =
  evalStateT (selectSortM consistentCoin [1 .. 3])
             noChoices
  -- [[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]
```

While this implementation does not enumerate permutations multiple times any more, we would like to derive an implementation that can do without the additional *State*. Instead of choosing a predicate that makes the right choices, we can as well use a set-based monad instead of a multiset-based monad to prevent duplicates. The following implementation uses a *Set* monad to prevent the enumeration of permutations multiple times.

```
perms2Set :: [[Int]]
perms2Set = Set.toList (selectSortM coinCmp [1 .. 3])
  -- [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

However, we do not want to remove the permutations after we have already enumerated them, because this would be quite inefficient. So, let us take a closer look at the algorithm. Figure 4 shows the decision tree of *minimumM* when we enumerate the permutations of $[1,2,3]$.

As we can see, *minimumM* enumerates 1 twice – even if we use the consistent comparison function *consistentCoin*. This repetition in *minimumM* causes the additional branches in Figure 3.

When we use a set-based monad, *minimumM* indeed generates 1 only once and the resulting function enumerates every permutation exactly once. However, we would like to avoid the additional comparisons that are performed by a set-based monad to

check whether an element was enumerated before. Therefore, in the following we will derive a purely non-deterministic implementation $minimumNDM$ in two steps.

We first inline the comparison $coinCmp$ into $minimumM$ and get the following implementation by using the definitions of $minM$, $coinCmp$, **if then else** and applying the monad law (Bind-MPlus left distributive).

$$
\begin{array}{l}
minimumNDM :: MonadPlus\ m \Rightarrow [\alpha] \rightarrow m\ \alpha \\
minimumNDM\ [x] \quad\quad = return\ x \\
minimumNDM\ (x : xs) = \textbf{do} \\
\quad y \leftarrow minimumNDM\ xs \\
\quad return\ x \oplus return\ y
\end{array}
$$

This definition still yields duplicate elements when we consider the list instance whereas it does not yield duplicates when we use the set monad.

This behaviour can be explained by the following observation. The set monad satisfies the laws (MPlus idempotent) and (Bind-MPlus right distributive) whereas the list monad does not. For example, (Bind-MPlus right distributive) does not hold as the following example shows.

$$
\begin{array}{l}
[1,2] \ggg\!= \lambda x \rightarrow return\ x \oplus return\ x \\
\equiv \\
[1,1,2,2] \\
\not\equiv \\
[1,2,1,2] \\
\equiv \\
([1,2] \ggg\!= \lambda x \rightarrow return\ x) \oplus ([1,2] \ggg\!= \lambda x \rightarrow return\ x)
\end{array}
$$

In a monad that satisfies (Bind-MPlus right distributive) we can change the implementation of $minimumNDM$ as follows.

$$
\begin{array}{l}
minimumNDM\ (x : xs) \ggg\!= \\
\quad \lambda y \rightarrow return\ x \oplus return\ y \\
\equiv \quad \{\ \text{(Bind-MPlus right distributive)}\ \} \\
(minimumNDM\ xs \ggg\!= \lambda y \rightarrow return\ x) \oplus \\
\quad (minimumNDM\ xs \ggg\!= \lambda y \rightarrow return\ y) \\
\equiv \quad \{\ \text{(Bind right identity)}\ \} \\
(minimumNDM\ xs \ggg\!= \lambda y \rightarrow return\ x) \oplus \\
\quad minimumNDM\ xs
\end{array}
$$

We end up with a definition where we apply $\ggg\!=$ to $return$ where the argument of $return$ is a constant value that does not depend on the first argument of $\ggg\!=$. In a monad that satisfies (MPlus idempotent) we can derive the following equality that can be used to simplify this expression.

Lemma 1. *For all* $xs :: [\tau]$ *and all* $c :: \tau$ *we have*

$$minimumNDM\ xs \ggg\!= \lambda x \rightarrow return\ c \equiv return\ c.$$

We can prove this statement by structural induction over xs by using the monad laws (Bind left identity), (Bind associativity), (Bind-MPlus left distributive), and (MPlus idempotent).

Thus, as a second step, with Lemma 1 at hand we can derive the following implementation.

$$
\begin{array}{l}
minimumSet :: MonadPlus\ \mu \Rightarrow [\alpha] \rightarrow \mu\ \alpha \\
minimumSet\ [x] \quad\quad = return\ x \\
minimumSet\ (x : xs) = return\ x \oplus minimumSet\ xs
\end{array}
$$

Note that $minimumSet$ is only equivalent to $minimumNDM$ if we consider an instance of $MonadPlus$ that satisfies the laws (MPlus idempotent) and (Bind-MPlus left distributive). In monads where (MPlus idempotent) or (Bind-MPlus right distributive) do not hold, like the list monad for example, $minimumSet$ might yield results in a different order and duplicates. For example, we have

$$minimumSet\ [1,2,3] \equiv [1,2,3]$$

and

$$minimumNDM_{[\,]}\ [1,2,3] \equiv [1,2,1,3].$$

Coincidentally, as before, we have defined a function that is used in the context of functional logic programming languages. The function $minimumSet$ enumerates the elements of a list non-deterministically and its definition resembles a function that is called $elemOf$ by Antoy and Hanus (2011). By means of $minimumSet$ we can define a selection sort based permutation enumeration.

$$
\begin{array}{l}
selectSortND :: (Eq\ \alpha, MonadPlus\ \mu) \Rightarrow [\alpha] \rightarrow \mu\ [\alpha] \\
selectSortND\ [\,]\ = return\ [\,] \\
selectSortND\ xs = \textbf{do} \\
\quad x \leftarrow minimumSet\ xs \\
\quad fmap\ (x:)\ (selectSortND\ (delete\ x\ xs))
\end{array}
$$

As we can see from the result, in comparison to $perms2Cons$, the order of enumeration has changed, but, on the positive side, the implementation does not require an additional state.

$$
\begin{array}{l}
perms2ND :: [[Int]] \\
perms2ND = selectSortND\ [1\,..\,3] \\
\quad \text{-- } [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
\end{array}
$$

Instead of choosing a random element from the list and removing it from the original input, we can as well combine $minimumM$ and $delete$ into a single function to prevent unnecessary traversals of the input list. For example, Gibbons and Hinze (2011) present an implementation of this fused function. We define a similar function $selectM$ as follows.

$$
\begin{array}{l}
selectM :: Monad\ \mu \Rightarrow Cmp\ \alpha\ \mu \rightarrow [\alpha] \rightarrow \mu\ (\alpha, [\alpha]) \\
selectM\ _\ [x] \quad\quad = return\ (x, [\,]) \\
selectM\ p\ (x : xs) = \textbf{do} \\
\quad (y, ys) \leftarrow selectM\ p\ xs \\
\quad b \leftarrow p\ x\ y \\
\quad return\ (\textbf{if}\ b\ \textbf{then}\ (x, xs)\ \textbf{else}\ (y, x : ys))
\end{array}
$$

Because a total function is preferable over a partial function, we could add a rule for the empty list like Gibbons and Hinze (2011) do. However, we would like to avoid using a $MonadPlus$ context instead of the $Monad$ context where possible. Because of the $Monad$ context, when applying $selectM$ to a non-deterministic predicate, we know that all non-determinism is introduced by the predicate only. We use this fact in order to prove that a monadic sorting function actually enumerates all permutations in Section 7.

4. Bubble Sort

We define an implementation of a bubble sort algorithm that bubbles the minimum element to the front of a list[4]. Of course, we could also bubble the maximum element to the end, but bubbling an element to the front allows for a more efficient selection of the minimum element and the rest of the list.

$$
\begin{array}{l}
bubbleM :: Monad\ \mu \Rightarrow Cmp\ \alpha\ \mu \rightarrow [\alpha] \rightarrow \mu\ [\alpha] \\
bubbleM\ _\ [x] \quad\quad = return\ [x] \\
bubbleM\ p\ (x : xs) = \textbf{do} \\
\quad y : ys \leftarrow bubbleM\ p\ xs \\
\quad b \leftarrow p\ x\ y \\
\quad return\ (\textbf{if}\ b\ \textbf{then}\ x : y : ys\ \textbf{else}\ y : x : ys)
\end{array}
$$

By means of this implementation we can define bubble sort.

[4] There is some dispute about the name, for example, the variation that bubbles the maximum element to the end is sometimes called *sinking sort*.

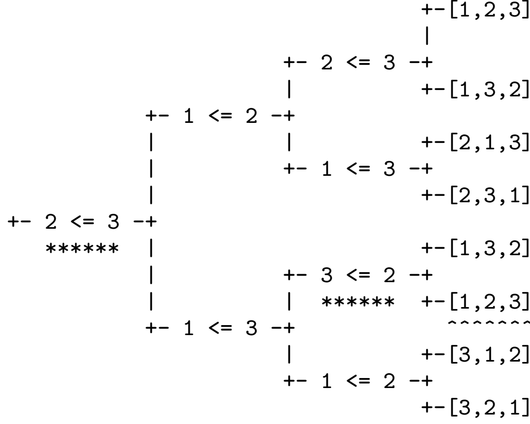


Figure 5. The decision tree of *bubbleSort*.

```
bubbleSortM :: Monad μ ⇒ Cmp α μ → [α] → μ [α]
bubbleSortM _ [] = return []
bubbleSortM p xs = do
  y : ys ← bubbleM p xs
  fmap (y:) (bubbleSortM p ys)
```

When we enumerate permutations with this algorithm, we get the same number of results as for selection sort. However, even if we use the consistent non-deterministic predicate, we get permutations multiple times.

When we take a look at the decision tree of *bubbleSort* in Figure 5, we see why the function enumerates permutations multiple times even if the predicate is consistent. The path that yields $[1, 2, 3]$ (underlined with `^^^` in Figure 5) compares the elements 2 and 3 twice (underlined with `***` in Figure 5). A sorting algorithm will never take this path because of the totality of the provided predicate. For a total predicate p we have $p\ x\ y$ or $p\ y\ x$ for all x and y. Therefore, if we have decided that `2 <= 3` does not hold, we later have to decide that `3 <= 2` does hold. Note, in the case that `2 <= 3` holds, totality does not imply that `3 <= 2` does not hold; hence, we cannot add any decision in this case.

We can get rid of the duplicates shown in the decision tree easily by closing the list of choices according to totality every time we add a new choice. Therefore, we define the following function to calculate the total closure of a list of choices.

```
totalClosure :: Eq α ⇒ Choices α → Choices α
totalClosure xs = nub (xs ++ concatMap add xs)
  where
    add ((x, y), b) = if b then [] else [((y, x), True)]
```

We pass this closure to *store* in order to calculate the total closure of the choices every time we add a new choice. The following state-based predicate is consistent and respects totality.

```
totalCoin :: (Eq α, MonadPlus μ)
          ⇒ Cmp α (StateT (Choices α) μ)
totalCoin = check (store totalClosure coinCmp)
```

Using this enriched predicate, the non-deterministic bubble sort yields the correct number of permutations.

```
perms3Total :: [[Int]]
perms3Total =
  evalStateT (bubbleSortM totalCoin [1 . . 3]) noChoices
  -- [[1, 2, 3], [1, 3, 2], [3, 1, 2], [2, 1, 3], [2, 3, 1], [3, 2, 1]]
```

Let us try to derive a non-deterministic version *bubbleSortND* that does not need a state-based predicate. When we apply the function *bubbleM* to *coinCmp*, the resulting function non-deterministically swaps pairs of consecutive elements in the argument list. That is, it yields $2^{(n-1)}$ non-deterministic results, because there are $n-1$ positions for a list of length n to swap two elements. For example, we get the following result for a list with three elements.

```
swaps :: [[Int]]
swaps = bubbleM coinCmp [1, 2, 3]
  -- [[1, 2, 3], [2, 1, 3], [1, 3, 2], [3, 1, 2]]
```

As the implementation of *bubbleSortM* splits the results of *bubbleM* into head and tail, we get multiple splits with the same head. The tails of the splits with the same head are not equal, but they contain the same elements. Because we recursively generate all permutations of theses tails and we get the same set of permutations if the lists contain the same elements, *bubbleSortM coinCmp* generates duplicates.

We can change the implementation of *bubbleM* to prevent this behaviour. In order to improve the implementation we consider the case that x, the first element of the input list, is smaller than y, the element we have bubbled to the front of the remaining list. In this case, instead of using the list $y : ys$ we can as well use the list xs because the result of *bubbleM* is supposed to contain the same list elements as its argument. We end up with the following implementation of *bubbleM*, where we have replaced the expression $x : y : ys$ in the **then** branch of the **if**-expression by $x : xs$.

```
bubbleM :: Monad μ ⇒ Cmp α μ → [α] → μ [α]
bubbleM _ [x]      = return [x]
bubbleM p (x : xs) = do
  y : ys ← bubbleM p xs
  b ← p x y
  return (if b then x : xs else y : x : ys)
```

When we use this implementation of *bubbleM*, the resulting *bubbleSortM coinCmp* yields exactly all permutations. However, this transformation is kind of cheating as we have removed the key property that makes *bubbleSortM* an implementation of bubble sort. In fact, we have simply transformed our bubble sort implementation into an implementation of selection sort. The function *bubbleM* as seen above is equivalent to *selectM* – that is a combination of *minimum* and *delete* – as defined in Section 3. The only difference is that *selectM* yields a pair of an element and a list while *bubbleM* simply yields a non-empty list where the first element of the list is the first component of the pair.

As an additional optimisation, the number of comparisons performed by bubble sort can be improved by adding an additional Boolean value. The value states whether *bubbleM* has performed any swap at all. If we have not performed a swap, the list is already sorted and we can stop. This implementation has a linear running time for pre-sorted lists instead of the quadratic running time of the naive implementation. If we use this improved implementation for enumerating permutations, some of the comparisons that cause inconsistent choices are not performed any more. Yet, while this optimisation reduces the number of inconsistent choices, it does not completely eliminate them. Therefore, we still need a non-deterministic predicate that is consistent and respects totality in order to enumerate exactly all permutations.

5. Quicksort

Let us consider the following monadic implementation of quicksort that is based on a monadic version of *filter* and has the reputation of being very declarative and compact.

```
quickSortM :: Monad μ ⇒ Cmp α μ → [α] → μ [α]
quickSortM _ []     = return []
quickSortM p (x : xs) = do
  ys ← filterM (λy → p y x) xs
  zs ← filterM (λy → fmap not (p y x)) xs
  ys' ← quickSortM p ys
  zs' ← quickSortM p zs
  return (ys' ++ [x] ++ zs')
```

When we use quicksort for enumerating permutations, the result is rather disappointing in contrast to the previous enumerations. The resulting list contains elements that are not even permutations of the original list, as the following example demonstrates.

```
perms4 :: [[Int]]
perms4 = quickSortM coinCmp [1, 2]
  -- [[2, 1], [2, 1, 2], [1], [1, 2]]
```

Besides the permutations $[1, 2]$ and $[2, 1]$ there are also the lists $[1]$ and $[2, 1, 2]$, which obviously are no permutations of the list $[1, 2]$.

One might think that these non-permutations are enumerated because the non-deterministic predicate does not respect transitivity, but the behaviour is caused by mere inconsistency. If we use the predicate that is consistent, the permutation function enumerates exactly all permutations.

```
perms4Cons :: [[Int]]
perms4Cons =
  evalStateT (quickSortM consistentCoin [1 .. 3])
             noChoices
  -- [[3, 2, 1], [2, 3, 1], [2, 1, 3], [3, 1, 2], [1, 3, 2], [1, 2, 3]]
```

How can this behaviour be explained? Let us inspect the implementation of *filterM* in detail. The predicate $\lambda y \rightarrow coinCmp\ y\ x$, which is used by the application of *quickSortM coinCmp*, is equivalent to *coinPred* for all x. The application

$$filterM\ coinPred\ xs$$

yields all subsequences of the list xs. For example, consider the following application.

```
subsequences :: [[Int]]
subsequences =
  filterM coinPred [1 .. 3]
  -- [[1, 2, 3], [1, 2], [1, 3], [1], [2, 3], [2], [3], []]
```

The second application of *filterM* in *quickSortM* also yields all subsequences of the input list, but in reversed order.

```
subsequencesRev :: [[Int]]
subsequencesRev =
  filterM (λy → fmap not (coinPred y)) [1 .. 3]
  -- [[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]
```

Now, let us consider the following part of the implementation of *quickSortM*.

```
filterTwiceNDM :: MonadPlus μ
    ⇒ (α → μ Bool) → [α] → μ ([α], [α])
filterTwiceNDM p xs = do
  subxs1 ← filterM p xs
  subxs2 ← filterM (λy → fmap not (p y)) xs
  return (subxs1, subxs2)
```

When we apply this function to *coinPred*, the resulting function enumerates the cross product of all subsequences of the input list.

```
splits :: [([Int], [Int])]
splits = filterTwiceNDM (coinCmp 0) [1, 2]
  -- [([1, 2], []), ([1, 2], [2]), ([1, 2], [1]), ([1, 2], [1, 2]), ...
```

In contrast when we use the consistent comparison function, we get only correct splits of the input list.

```
splitsState :: [([Int], [Int])]
splitsState =
  evalStateT (filterTwiceNDM (consistentCoin 0) [1, 2])
             noChoices
  -- [([1, 2], []), ([1], [2]), ([2], [1]), ([], [1, 2])]
```

With a consistent non-deterministic predicate, the second application of *filterM* will make the same choices as the first application of *filterM* for every non-deterministic branch. For example, if the first application removes all elements from the list, the second application will keep all elements of the list.

Instead of implementing *quickSortM* by means of *filterM*, we can as well implement it by means of *partitionM*. In this case the resulting monadic sorting function enumerates exactly all permutations even if the predicate is not consistent. As opposed to the implementation with two applications of filter, this implementation cannot make conflicting choices because the predicate is only used once per list element.

The non-deterministic version of *partition* splits a list non-deterministically into two sublists. Compared with the non-deterministic version of *filter*, *partition* yields all subsequences and for each subsequence it additionally yields the rest of the input list that is not part of the subsequence.

```
splits2 :: [([Int], [Int])]
splits2 = partitionM coinPred [1, 2]
  -- [([1, 2], []), ([1], [2]), ([2], [1]), ([], [1, 2])]
```

This example nicely illustrates the connection between the number of non-deterministic results in the list monad and the number of applications of the predicate in the deterministic version. More precisely, the quicksort implementation that is based on two applications of *filter* applies the predicate twice as often as the implementation that is based on partition. As the additional comparisons of the filter-based implementation are not necessary, these comparisons cause duplicate non-deterministic branches in the non-deterministic context.

6. Other Sorting Algorithms

In this section we will take a look at some additional sorting algorithms that we do not have considered in the previous sections. As mentioned in the introduction, there is actually a sorting algorithm, whose corresponding permutation algorithm only enumerates exactly all permutations, if the non-deterministic predicate respects transitivity. Yet, it is not one of the sorting algorithms that might come to mind. Even permutation sort – sometimes also referred to as stupid sort – does not rely on transitivity. Permutation sort uses a generate and test approach to sort a list. It enumerates all permutations of a list and filters the one permutation that is sorted. The predicate that checks whether a list is sorted naturally exploits transitivity as it typically only checks consecutive elements. If it did not employ transitivity, it would have to compare every element in the list with every other element. Nevertheless, we have found one algorithm that explicitly tests elements although it would not have to because of transitivity, namely patience sort by Mallows (1963).

Patience sort is based on the corresponding card game and consists of two phases. In the first phase the list to be sorted is divided into several piles, where each pile is already sorted. In order to get a list of sorted piles, each element of the input list is recursively added to the oldest (with respect to its creation) pile, whose top element is larger (with respect to the provided comparison function) than the considered element. If no such pile exists, a new pile with that element is created. The second phase

applies an n-way merge to the piles. Since all piles are already sorted, it suffices to consider only the current head elements of all piles and pick the minimum with respect to the comparison function. The minimum of all head elements will be the smallest element of all the remaining elements.

As we have already seen plenty of sorting algorithms, we do not show the implementation of a monadic version of patience sort here, but reference such an implementation as $patienceSortM$. As the following application shows, $patienceSortM$ even generates duplicates when we use the non-deterministic comparison function that respects totality.

```
perms5Cons :: [[Int]]
perms5Cons =
  evalStateT (patienceSortM totalCoin [1 .. 3])
             noChoices
  -- [[1, 2, 3], [2, 3, 1], [2, 1, 3], [1, 3, 2], [2, 1, 3], [3, 1, 2]
  -- , [3, 2, 1]]
```

In particular, the list $[2, 1, 3]$ is enumerated twice. While this might seem to be a minor problem, the number of duplicates grows quite fast, for example the same application yields 195 213 results for the list $[1 \mathinner{..} 8]$, while there are only 40 320 permutations of this list. So, why does $patienceSortM$ enumerate the list $[2, 1, 3]$ twice?

In order to understand the behaviour we illustrate the process of sorting the list $[5, 7, 2]$ with a deterministic version of patience sort. First, we will create a singleton pile with 5. In order to insert 7 into the list of piles we check $7 \leqslant 5$. As it does not hold, we add a new pile with 7 to the list of piles and end up with the list of piles $[[5], [7]]$. Next, we compare 2 with the head of the oldest pile namely 5. As $2 \leqslant 5$ holds, we insert 2 into the pile $[5]$ ending up with the list of piles $[[2, 5], [7]]$. In order to merge this list of piles we will first determine the minimum element of all heads of all piles. In particular we will check whether $2 \leqslant 7$ holds. However, as we know that $7 \leqslant 5$ does not hold, by totality $5 \leqslant 7$ must hold. Furthermore, we additionally know that $2 \leqslant 5$ holds, because we have already checked it when creating the piles. By transitivity we get $2 \leqslant 5 \leqslant 7$. That is, by transitivity (and totality) we would not have to check $2 \leqslant 7$. Because patience sort performs this comparison anyhow, in the non-deterministic setting the predicate might decide that $2 \leqslant 7$ does not hold although this decision conflicts with other decisions made before.

We can enumerate exactly all permutations using patience sort if we use a non-deterministic predicate that respects transitivity. If $transitiveClosure$ is a function that calculates the transitive closure of a list of choices, we can define a non-deterministic choice that respects transitivity as follows.

```
transitiveCoin :: (Eq α, MonadPlus μ)
               ⇒ Cmp α (StateT (Choices α) μ)
transitiveCoin = check (store cl coinCmp)
  where
    cl = totalClosure ∘ transitiveClosure ∘ totalClosure
```

Besides the implementations we have considered so far, we can derive permutation enumerations from all kinds of sorting functions. For example, when we apply the monadic version of an implementation of merge sort to the predicate $coinCmp$ the resulting function enumerates exactly all permutations. When we implement a permutation enumeration "by hand" we would probably not come up with an implementation like this. A similar argument applies to enumerating permutations by using a sorting function that is based on a binary search tree.

One final sorting algorithm we would like to mention here is heap sort. Heap sort can be considered as an improved version of selection sort. Instead of looking up the minimum element of a list on every pass, we initially create a heap structure from the list and use this structure to get and remove the minimum element efficiently. While the worst case complexity of selection sort is $\mathcal{O}(n^2)$, the worst case complexity of heap sort is $\mathcal{O}(n \log n)$. This reduction in the number of comparisons also improves the derived permutation enumeration. While the permutation enumeration that is based on selection sort enumerates permutations multiple times, the permutation enumeration that is based on heap sort enumerates exactly all permutations of a list. However, note that the worst case complexity of a sorting function does not determine the number of non-deterministic results of the corresponding permutation enumeration. For example, while the worst case complexity of insertion sort is n^2 the corresponding permutation enumeration enumerates exactly all permutations while selection sort has the same worst case complexity and enumerates permutations multiple times. Similarly, the two implementations of quicksort are in the same complexity class for the best, average, and worst case and still behave differently.

These examples illustrate the beauty of deriving permutation enumerations from sorting functions because we get as many permutation enumerations as there are sorting functions and their implementations can profit from improvements of the sorting functions. Interestingly, we can also derive a sorting function from every permutation enumeration by using the permutation sort approach. Yet, this way we end up with quite inefficient sorting functions.

7. Proving Fischer's Intuition

In this section we will make use of free theorems as presented by Wadler (1989). Free theorems are a means to prove statements about a function by only considering its type. The statements rely on the fact that a function cannot invent a value of a polymorphic type. Therefore, this style of proof especially allows for quite strong statements if the function type at hand is very general. A well-known trick to apply free theorems to a concrete problem is to make a function "more polymorphic" by introducing a higher-order argument that abstracts the non-polymorphic part. In our case, the higher-order argument is a predicate that abstracts the concrete type of the elements of the list we are sorting.

$$sortM :: Monad\ \mu \Rightarrow (\alpha \to \alpha \to \mu\ Bool) \to \mu\ [\alpha] \to \mu\ [\alpha]$$

Besides abstracting the type of the list elements, this function type also abstracts the non-deterministic context. More precisely, the type of $sortM$ does not even mention the $MonadPlus$ context. This context is only provided by the argument that we pass to $sortM$. In other words, the function $sortM$ itself cannot introduce any non-determinism.

In order to prove statements about function types that involve type constructor classes like monads, we make use of an extension of free theorems to cover type constructor classes as presented by Voigtländer (2009). We have to provide a relational interpretation of the type constructor to prove a statement about a monadic function. This relational interpretation is a function that takes a relation and yields a relation.

As we would like to prove that the non-deterministic variant of a sorting function actually enumerates all permutations of a given list, we have to express an *is element of* relation. The natural choice would be to use the function

$$elem :: Eq\ \alpha \Rightarrow \alpha \to [\alpha] \to Bool$$

that checks whether an element is found in a given list. In order to apply a free theorem we have to interpret the type of a function by a relation. That is, we would like to define a relation that expresses an *is element of* relation.

$$Elem := \{(x, xs) \mid elem\ x\ xs \equiv True\}$$

In the case of the list monad, instead of using $elem$, we can as well express an *is element of* relation by means of $(+\!\!+)$ and equality as follows.

$$Elem := \{(x, xs) \mid \exists ys, zs :: [\alpha] \;.\; ys +\!\!+ singleton\ x +\!\!+ zs \equiv xs\}$$

In contrast to the relation that is based on $elem$, we can generalise this relation to arbitrary $MonadPlus$ instances as follows.

$$Elem := \{(x, m) \mid \exists n, o :: \kappa\ \alpha \;.\; n \oplus_\kappa return_\kappa\ x \oplus_\kappa o \equiv m\}$$

In the case of the list monad this simply means that the element x is an element of the list m.

Before we start with the actual proof, we observe a connection between the *is element of* relation and the corresponding monadic bind operator. The following lemma states that if y is an element of m and x is an element of $g\ \ y$, then x is also an element of $m \ggg\!=_\kappa\ g$.

Lemma 2. *For all types τ, all MonadPlus instances κ, all $y :: \tau$, all $m :: \kappa\ \tau$ and all $g :: \tau \to \kappa\ \tau$ we have that if*

$$(y, m) \in Elem \quad \textit{and} \quad (x, g\ y) \in Elem \qquad (*)$$

then

$$(x, m \ggg\!=_\kappa\ g) \in Elem.$$

We do not prove this statement here, it is a straightforward application of (Bind-MPlus left distributive), (Bind left identity), and the preconditions $(*)$.

In order to prove statements about a function involving a type constructor class, we have to define a so-called $Monad$-action. A $Monad$-action M is a function that maps a relation over the "result type" of a monad to a relation that relates two concrete instances of a $Monad$. Furthermore, a $Monad$-action has to be compatible with $return$ and $\ggg\!=$ in the sense that if we consider values that are related, the images of $return$ and $\ggg\!=$ have to be related as well. We outline the precise statements shortly. If you are interested in more details, Voigtländer (2009) presents an introduction to $Monad$-actions as well as a couple of applications.

As a first step we define the following relational action that relates elements of the identity monad to elements of some specific instance κ of $MonadPlus$ by means of the relation[5] $Elem$.

$$E\ \mathcal{R} := \{\ (Id\ x, m) \mid \exists y \;.\; (x, y) \in \mathcal{R} \land (y, m) \in Elem\ \}$$

In order to use this relational action in a free theorem about monadic functions, we have to show that E is a $Monad$-action. Note that, because of the relation $Elem$, the type constructor κ has to be an instance of $MonadPlus$. Yet, we are only considering $Monad$-actions and not $MonadPlus$-actions in the following because the function we are considering, $sortM$, only has a $Monad$ context.

Lemma 3. *E is a $Monad$-action.*

Proving Lemma 3 requires showing two statements about the relation E. First, we need to show that for all relations $\mathcal{R}$ we have

$$(return_{Id}, return_\kappa) \in \mathcal{R} \to E\ \mathcal{R}\ .$$

The following info box introduces the idea of the operator $\to$ on relations that is used in this statement.

Function lifting of relations $\mathcal{R} \to \mathcal{S}$

For relations $\mathcal{R} : \alpha \leftrightarrow \beta$ and $\mathcal{S} : \gamma \leftrightarrow \delta$ the relation $\mathcal{R} \to \mathcal{S}$ relates functions of the type $\alpha \to \gamma$ to functions of the type $\beta \to \delta$, and we have

$$(f, g) \in \mathcal{R} \to \mathcal{S} \iff \forall\,(x, y) \in \mathcal{R} \;.\; (f\ x, g\ y) \in \mathcal{S}\ .$$

[5] Using the relational operations of multiplication ; and inverse $^{-1}$, $E\ \mathcal{R}$ can be written $E\ \mathcal{R} = Id^{-1}; \mathcal{R}; Elem$.

The statement about $return$ is easily shown using the definition of $Elem$ and the laws (MPlus left zero) and (MPlus right zero). As a second step, we need to verify that for all relations $\mathcal{R}$ and $\mathcal{S}$

$$((\ggg\!=_{Id}), (\ggg\!=_\kappa)) \in E\ \mathcal{R} \to (\mathcal{R} \to E\ \mathcal{S}) \to E\ \mathcal{S}$$

holds. The main ingredients in proving this statement are Lemma 2, as well as careful considerations of the relational arrow and the relation E. Clearly, the second statement is slightly more technical than the first one, since it involves more conditions and conclusions.

Now, we can use this $Monad$-action to prove a statement about $sortM$. In order to relate the results of two concrete instances of $sortM$ we have to relate the two arguments of these functions, that is, two comparison functions. More precisely, we will show that for an arbitrary $MonadPlus$ instance κ the results of the functions cmp_{Id} and $coinCmp_\kappa$ are related by the relation $E\ I$. The term I denotes the identity relation that relates elements of a specific type with itself. We have to use the identity relation, because the non-monadic content type of the result type of the two comparison functions is $Bool$. In the context of free theorems, non-polymorphic types like $Bool$ have to be interpreted by the identity relation.

As cmp_{Id} only takes arguments of type Int, the following statement only considers relations whose first component uses values of type Int.

Lemma 4. *For all $\mathcal{R} : Int \leftrightarrow \tau$ we have*

$$(cmp_{Id}, coinCmp_\kappa) \in \mathcal{R} \to \mathcal{R} \to E\ I.$$

Proof. Let $\mathcal{R} : Int \leftrightarrow \tau$ be a relation, $(x_1, y_1) \in \mathcal{R}$, and $(x_2, y_2) \in \mathcal{R}$. We want to show that

$$(Id\ (x_1 \leqslant x_2), coinCmp_\kappa\ y_1\ y_2) \in E\ I$$

and know that

$$cmp_{Id}\ x_1\ x_1 = Id\ (x_1 \leqslant x_2)$$

and

$$coinCmp_\kappa\ y_1\ y_2 = return_\kappa\ True \oplus_\kappa return_\kappa\ False.$$

We can distinguish two cases:

(1) $x_1 \leqslant x_2 \equiv True$

$return_\kappa\ True \oplus_\kappa return_\kappa\ False$
$\equiv$ { (MPlus left zero) }
$mzero_\kappa \oplus_\kappa return_\kappa\ True \oplus_\kappa return_\kappa\ False$
$\equiv$ { $x_1 \leqslant x_2 \equiv True$ }
$mzero_\kappa \oplus_\kappa return_\kappa\ (x_1 \leqslant x_2) \oplus_\kappa return_\kappa\ False$

(2) $x_1 \leqslant x_2 \equiv False$

$return_\kappa\ True \oplus_\kappa return_\kappa\ False$
$\equiv$ { (MPlus right zero) }
$return_\kappa\ True \oplus_\kappa return_\kappa\ False \oplus_\kappa mzero_\kappa$
$\equiv$ { $x_1 \leqslant x_2 \equiv False$ }
$return_\kappa\ True \oplus_\kappa return_\kappa\ (x_1 \leqslant x_2) \oplus_\kappa mzero_\kappa$

We get $(x_1 \leqslant x_2, coinCmp_\kappa\ y_1\ y_2) \in Elem$ by definition of $Elem$. In addition, we have $(x_1 \leqslant x_2, x_1 \leqslant x_2) \in I$. With these facts at hand, the definition of E yields

$$(cmp_{Id}\ x_1\ x_2, coinCmp_\kappa\ y_1\ y_2) \in E\ I. \qquad \square$$

Now we are ready to relate the results of two instances of $sortM$. In order to make the following statements more readable, we use the abbreviations

$$sort = sortM_{Id}\ cmp_{Id}$$

and

$$permute = sortM_\kappa\ coinCmp_\kappa.$$

We will only consider the relational action $E\ \mathcal{R}$ in the special case that $\mathcal{R}$ is a function. Note that in this case $(Id\ x, m) \in E\ f$ for some function f implies $(f\ x, m) \in Elem$.

Before we start with the actual proof, we provide a road map. Let τ be a type, $xs, ys :: [\tau]$, such that ys is a permutation of xs. In the following theorem we will show that $permute\ xs$ enumerates ys, in other words, $permute$ generates every permutation of a list.

Our approach is different from the intuition of Fischer as stated in Section 1. In order to prove the statement via the intuition we would have to provide an ordering that is able to generate ys from xs via "sorting" for every list xs and permutation ys. Instead of showing that the sorting function is able to generate every permutation, we will show that the sorting function is able to undo every permutation of a sorted list. In this case we can always use the same ordering independent of the permutation ys.

In order to illustrate the following proof we will consider a less general statement, which constitutes the underlying idea and is easier to grasp. That is, we will resort to lists of indices instead of using xs and ys. Working with indices is much simpler as we can use the well-known order on integers to sort them and do not have to bother about duplicate elements. That is, instead of showing that $permute\ xs$ generates ys, we will show that $permute\ is$ generates pis where $is = [0\,..\,length\ xs - 1]$ and pis is a permutation of is.

The key argument in the proof of this statement is the relation of two instances of $sortM$, namely $sort$ and $permute$. We relate these functions by using the free theorem for the function $sortM$. The free theorem for $sortM$ states that using related predicates yields related functions.

In a free theorem a type variable in a function type is interpreted by a relation. Because the function $sortM$ uses the polymorphic type variable μ twice – once in the result type of the functional argument and once in the result type of the function – we are able to connect the result of the functional argument and the result of the application of $sortM$. More precisely, we will show that two concrete functional arguments are related by the relational action E. By the free theorem for $sortM$, we get that the results of the application of $sortM$ to these functional arguments are related by E as well.

First, by Lemma 4 we have

$$(cmp_{Id}, coinCmp_{\kappa}) \in I \to I \to E\ I.$$

Second, by the free theorem for $sortM$, we get that the applications of $sortM$ to cmp_{Id} and $coinCmp_{\kappa}$ are related by $E\ I$ as well, thus, we get

$$(sort, permute) \in [I] \to E\ I.$$

List lifting of relations $[\mathcal{R}]$

For a function f we have

$$(x, y) \in [f] \iff y = map\ f\ x\ .$$

The identity relation I is the function id, thus

$$(x, y) \in [I] \iff y = map\ id\ x = x\ .$$

That is, for equal arguments the results of $sort$ and $permute$ are related by $E\ I$. This implies

$$(sort\ pis, permute\ pis) \in E\ I.$$

Third, because $is = [0\,..\,length\ xs - 1]$ and pis is a permutation of is, we have $sort\ pis \equiv Id\ is$ and, therefore,

$$(Id\ is, permute\ pis) \in E\ I.$$

Finally, by definition of E we have $(is, permute\ pis) \in Elem$. That is, $permute$ of the permutation of a list of indices actually yields the sorted list of indices.

This statement about sorting lists of indices can be generalised to arbitrary lists. However, as this generalisation requires additional setup, we apply a different approach that works for arbitrary lists as well. Instead of using the identity relation, we use a more general relation that connects a list with a list of indices. In order to prove this generalised statement we will need the following property that is often used when the behaviour of a function is characterised by its behaviour on a list of indices.

Lemma 5. *For all types τ and all lists $xs :: [\tau]$ we have*

$$map\ (xs!!)\ [0\,..\,length\ xs - 1] \equiv xs.$$

Now we are ready to tackle the main proof about sorting with a non-deterministic predicate.

Theorem 1. *For all types τ, all lists $xs :: [\tau]$, and all permutations $ys :: [\tau]$ of xs we have*

$$(ys, permute\ xs) \in Elem.$$

We prove this statement by proceeding as illustrated above but replacing the relation I by the relation $[(ys!!)]$ as shown in Appendix A.

Now we know that the monadic instance of any sorting function actually enumerates all permutations of a list. In other words, we have formally proven the intuition of Fischer as shown in Section 1.

8. Final Remarks

There is a lot more to the idea of applying a monadic function to a non-deterministic predicate than we would have thought in the beginning. While working on this idea, we discovered a lot of questions related to the enumeration of permutations that we did not follow in the end. We would like to address some of these directions here in order to present some potential topics for future work.

We did only consider the non-deterministic predicate $coinCmp$, while there are other non-deterministic predicates that can be used to affect the order of enumeration. For example, the following function lifts a predicate cmp to a non-deterministic context.

$$\begin{aligned}&liftCmp :: MonadPlus\ \mu\\ &\qquad \Rightarrow (\alpha \to \alpha \to Bool) \to Cmp\ \alpha\ \mu\\ &liftCmp\ p\ x\ y = return\ (p\ x\ y) \oplus return\ (not\ (p\ x\ y))\end{aligned}$$

When we use this function to lift a comparison function and pass it to a monadic version of merge sort, we get a special kind of permutation function: it enumerates permutations in lexicographical order.

Like listing the results in lexicographical order there are lots of properties related to permutations that have been investigated since the idea of enumerating permutations has been presented. For example, enumerating derangements, that is, enumerating all permutations where an element does not appear at its original position or enumerating all permutations of a sublist of a given list.

Yet, not only these permutation related topics emerged during this work; there are also topics that are related to the specific Haskell implementation of the algorithms. For example, we did not investigate the time complexity or the memory consumption of the presented permutation enumerations. While we need $n * n!$ steps to enumerate all permutations, we do not know whether the presented functions are even worse. Another interesting topic would be other monadic instances of the permutation enumerations besides the list or the set instance. For example, by using a random value generator monad we might be able to generate permutations efficiently by using sampling instead of simply enumerating them.

Finally, the two examples that were the motivation for conducting this research, namely non-deterministic filtering and sorting, share a common structure. More precisely, we can process both

cases in one if we abstract a type constructor φ and consider a function of type

$$(\varphi\ \alpha \to Bool) \to [\alpha] \to [\alpha].$$

In order to model filtering we use Id for φ whereas we use the following data type for sorting.

$$\textbf{data}\ Two\ \alpha = Two\ \alpha\ \alpha$$

Furthermore, we would like to consider other type constructors like a data type $Three$ and check how these functions are connected.

Acknowledgments

We would like to thank the anonymous reviewers who have helped to improve this paper through their criticism and suggestions. Moreover, we especially thank Koen Claessen for asking the right questions and investing a considerable amount of time for guiding us to improve the final version of this paper.

References

S. Antoy and M. Hanus. New Functional Logic Design Patterns. WFLP'11, pages 19–34. Springer LNCS 6816, 2011.

Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and Loose Reasoning is Morally Correct. POPL'06, pages 206–217. ACM, 2006.

Sebastian Fischer and Jan Christiansen. Curry mailing list, July 2009. URL `http://www.informatik.uni-kiel.de/~mh/curry/listarchive/0781.html`.

Jeremy Gibbons and Ralf Hinze. Just do it: Simple Monadic Equational Reasoning. ICFP'11, pages 2–14. ACM, 2011.

Michael Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, pages 583–628, 1994.

Selmer M Johnson. Generation of Permutations by Adjacent Transposition. *Mathematics of computation*, pages 282–285, 1963.

Sheng Liang, Paul Hudak, and Mark Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM, 1995.

Colin L Mallows. Patience Sorting. *SIAM review*, 1963.

George Pollard, Sebastian Fischer, Ganesh Sittampalam, and Ryan Ingram. Haskell mailing list, July 2009. URL `https://mail.haskell.org/pipermail/haskell-cafe/2009-July/064339.html`.

Robert Sedgewick. Permutation Generation Methods. *ACM Computing Surveys (CSUR)*, pages 137–164, 1977.

H. F. Trotter. Algorithm 115: Perm. *Commun. ACM*, pages 434–435, 1962.

Twan van Laarhoven et al. Haskell mailing list, December 2007. URL `https://mail.haskell.org/pipermail/haskell-cafe/2009-July/064339.html`.

Janis Voigtländer. Free Theorems Involving Type Constructor Classes: Functional Pearl. ICFP'09, pages 173–184. ACM, 2009.

Philip Wadler. Theorems for Free! FPCA'89, pages 347–359. ACM, 1989.

A. Proof of Theorem 1

Proof. Let τ be a type, $xs :: [\tau]$ a list, $ys :: [\tau]$ a permutation of xs, and $n = length\ xs - 1$.

Before we start, we have to formalise the notion of a permutation: ys being a permutation of xs is equivalent to the following.

- There exists a function $\sigma :: Int \to Int$, which is bijective when we restrict domain and codomain to $\{\, 0, \ldots, n \,\}$.
- For all $i \in \{\, 0, \ldots, n \,\}$ we have $ys\ !!\ i = xs\ !!\ \sigma\ i$.

Because we want to show that we are able to undo every permutation, we move σ to the left-hand side of the equation by using its inverse. That is, for all $i \in \{\, 0, \ldots, n \,\}$ we have

$$ys\ !!\ \sigma^{-1}\ i \equiv xs\ !!\ i.$$

Let $is = [0\, ..\, n]$ and $pis = map\ \sigma^{-1}\ is$. We are interested in the list pis because we know that it will be sorted by $sort$, hence, it holds $sort\ pis \equiv is$.

Additionally, we can relate pis and xs as follows.

$$
\begin{array}{l}
map\ (ys!!)\ pis \\
\equiv \quad \{\ \text{definition of } pis\ \} \\
map\ (ys!!)\ (map\ \sigma^{-1}\ is) \\
\equiv \quad \{\ \text{property of } map\ \} \\
map\ (\lambda i \to ys\ !!\ \sigma^{-1}\ i)\ pis \\
\equiv \quad \{\ \text{definition of } \sigma^{-1}\ \} \\
map\ (\lambda i \to xs\ !!\ i)\ is \\
\equiv \quad \{\ \text{Lemma 5}\ \} \\
xs
\end{array}
$$

Instead of using the identity relation in the proof sketch above, we can use the relation $(ys!!)$, that is, a relation that relates indices to the elements in ys. More precisely, we use the relation $[(ys!!)]$ where

$$(vs, ws) \in [(ys!!)]$$

implies

$$ws = map\ (ys!!)\ vs.$$

With the knowledge that $(ys!!) \in Rel(Int, \tau)$ holds, Lemma 4 yields

$$(cmp_{Id}, coinCmp_{\kappa}) \in (ys!!) \to (ys!!) \to E\ I$$

and by the free theorem of $sortM$, we get

$$(sort, permute) \in [(ys!!)] \to E\ [(ys!!)],$$

since we have

$$permute = sortM_{\kappa}\ coinCmp_{\kappa}$$

and

$$sort = sortM_{Id}\ cmp_{Id}.$$

Because $(pis, map\ (ys!!)\ pis) \in [(ys!!)]$ we get

$$(sort\ pis, permute\ (map\ (ys!!)\ pis)) \in E\ [(ys!!)].$$

On the other hand we have

$$
\begin{array}{l}
(sort\ pis, permute\ (map\ (ys!!)\ pis)) \\
\equiv \quad \{\ sort = sortM_{Id}\ cmp_{Id} \text{ is a sorting function }\} \\
(Id\ is, permute\ (map\ (ys!!)\ pis)) \\
\equiv \quad \{\ \text{equation above}\ \} \\
(Id\ is, permute\ xs)
\end{array}
$$

and thus

$$(Id\ is, permute\ xs) \in E\ [(ys!!)].$$

Now the definition of $E\ [(ys!!)]$ states there exists y, such that

$$(is, y) \in [(ys!!)]$$

and

$$(y, permute\ xs) \in Elem.$$

With this statement we get $y \equiv map\ (ys!!)\ is \equiv ys$, where the second equation is due to Lemma 5, which finally proves

$$(ys, permute\ xs) \in Elem.$$

That is, the permutation ys is an element of the resulting lists of lists $permute\ xs$. □