Reinventing Haskell Backtracking

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

Abstract: Almost ten years ago, Ralf Hinze has written a functional pearl on how to derive backtracking functionality for the purely functional programming language Haskell. In these notes, we show how to arrive at the efficient, two-continuation based backtracking monad derived by Hinze starting from an intuitive inefficient implementation that we subsequently refine using well known program transformations.

It turns out that the technique can be used to build monads for non-determinism from modular, independent parts which gives rise to various new implementations. Specifically, we show how the presented approach can be applied to obtain new implementations of breadth-first search and iterative deepening depth-first search.

1 Introduction

A conceptual divide tears apart two declarative paradigms: functional and logic programming. Combining them has a long tradition. Dedicated languages for multi-paradigm declarative programming show that the conceptual divide is not as big as one might expect [Han07], logic languages have incorporated support for directed, deterministic relations [SHC95], and lightweight support for logic features like backtracking has been implemented in purely functional languages. In an influential pearl, Ralf Hinze has shown how to derive lightweight backtracking for the functional programming language Haskell from an equational specification [Hin00].

Logic programming functionality can be incorporated into Haskell by expressing non-determinism explicitly as a computational effect modeled in the—today ubiquitous—framework of monads [Wad95]. In Haskell, a monad is a parametrised type m that supports the following operations.

$$\begin{aligned}&\mathit{return} :: a \to m\ a\\&(\gg\!=)\ \ :: m\ a \to (a \to m\ b) \to m\ b\end{aligned}$$

The operation return wraps an arbitrary value of type a as value of type $m\ \ a$ and $\gg\!=$ (pronounced 'bind') is used to apply monadic functions to monadically wrapped values. Haskell provides *syntactic sugar* for using these operations. For example, the expression $a \gg\!= \lambda x \to \mathit{return}\ (x + 1)$ can be written more conveniently using *do-notation*.

$$\begin{aligned}&\mathbf{do}\ x \leftarrow a\\&\quad\ \mathit{return}\ (x + 1)\end{aligned}$$

Monads provide a common interface for a variety of computational effects. In these notes we focus on non-determinism. Non-deterministic computations can be expressed monadically using two additional monadic combinators for failure and choice.

$$\begin{array}{l} \mathit{mzero} :: m\ a \\ \mathit{mplus} :: m\ a \rightarrow m\ a \rightarrow m\ a \end{array}$$

The shown monadic combinators can be interpreted in the context of non-determinism.

- mzero represents a failing computation, i.e., one without results;
- $\mathit{return}\ x$ represents a computation with the single result x;
- $\mathit{mplus}\ a\ b$ represents a computation that yields either a result of the computation a or one of the computation b; and
- $a \ggg\!= f$ applies the non-deterministic operation f to any result of a and yields any result of such an application.

We can use these combinators to define a function that yields an arbitrary element of a given list.

$$\begin{array}{l} \mathit{anyof} :: \mathit{MonadPlus}\ m \Rightarrow [a] \rightarrow m\ a \\ \mathit{anyof}\ [\,] \qquad\quad = \mathit{mzero} \\ \mathit{anyof}\ (x : \mathit{xs}) = \mathit{anyof}\ \mathit{xs}\ \text{`}\mathit{mplus}\text{`}\ \mathit{return}\ x \end{array}$$

The type signature specifies that the result of anyof can be expressed using a parametrised type m that is an instance of the type class $\mathit{MonadPlus}$, i.e., that is a monad for non-determinism that supports the operations that have just been introduced. The first rule of anyof uses the failing computation mzero to indicate that no result can be returned if the given list is empty. If it is nonempty, the second rule either returns a result of the recursive call to the tail of the list or the first element.

In Section 2, we introduce different implementations of parametrised types m that can be used to compute results of the non-deterministic operation anyof. Starting with an intuitive but inefficient implementation, we subsequently refine it using standard techniques. Specifically, we use difference lists to improve the asymptotic complexity of list concatenation in Section 2.1 and transform computations to continuation-passing style—which provides an implementation of monadic bind for free—in Section 2.2. In Section 2.3 we show that we arrive at the efficient implementation of backtracking previously derived by Hinze when combining these well-known techniques. We show in Section 3 how to use the developed ideas to find novel implementations of breadth-first search (Section 3.1) and iterative deepening depth-first search (Section 3.2). We compare different search strategies experimentally in Section 4 and finally point to related work briefly and conclude in Section 5.

2 Monadic Backtracking

The most intuitive implementation of the $\mathit{MonadPlus}$ type class uses *lists of successes* to represent results of non-deterministic computations. The four monadic operations are implemented on lists as follows. The failing computation is represented as empty list.

$$\begin{array}{l} \mathit{mzero} :: [\,a\,] \\ \mathit{mzero} = [\,] \end{array}$$

A deterministic computation with result x is represented as list with a single element x.

$$\begin{array}{l} \mathit{return} :: a \to [\,a\,] \\ \mathit{return}\ x = [\,x\,] \end{array}$$

To choose from the results of two non-deterministic computations, the results of both are concatenated using the append function $+\!\!+$.

$$\begin{array}{l} \mathit{mplus} :: [\,a\,] \to [\,a\,] \to [\,a\,] \\ \mathit{mplus}\ \mathit{xs}\ \mathit{ys} = \mathit{xs} +\!\!+\ \mathit{ys} \end{array}$$

Non-deterministic operations can be applied to any result of a given computation, e.g., by using a *list comprehension*.

$$\begin{array}{l} (\gg\!=) :: [\,a\,] \to (a \to [\,b\,]) \to [\,b\,] \\ \mathit{xs} \gg\!= f = [\,y \mid x \leftarrow \mathit{xs}, y \leftarrow f\ x\,] \end{array}$$

As Haskell lists implement the interface of the $\mathit{MonadPlus}$ type class, we can use lists to compute results of the non-deterministic operation anyof. For example, we can apply it to a list of numbers in order to get another list of the numbers that are contained in the list.

$$\begin{array}{l} >\ \mathit{anyof}\ [1\,..\,10] :: [\,\mathit{Int}\,] \\ [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] \end{array}$$

We provide an explicit type annotation in the call above which specifies that we want to use the list monad to compute the results. The resulting list does indeed contain each number of the given list but in reverse order. As different results of non-deterministic computations are independent their order is irrelevant, at least from a declarative point of view. We should expect different search strategies to enumerate results of non-deterministic computations in different orders.

If we compute results of anyof for long lists, we recognise that the list monad scales badly on this example. This is because of the specific implementation of anyof that uses a recursive call in the left argument of mplus. Actually, if we use the list monad then the implementation of anyof is the naive reverse function and, thus, has quadratic run time. We could change the implementation of anyof to avoid left recursion by swapping the arguments of mplus. However, we refrain from doing so and rather strive for a monad that can handle it gracefully.

2.1 Difference lists

The reason why the list monad scales so badly in case of left associative use of mplus is that the function $+\!\!+$ for list concatenation used for implementing mplus has linear run time in the length of its first argument. The standard technique to avoid this complexity is to use so called difference lists. A difference list is a function which takes a list as argument and yields a possibly longer list that ends with the given list. We can define the type of difference lists using Haskell's record syntax as follows.

$$\mathbf{newtype}\ \mathit{DiffList}\ a = \mathit{DiffList}\ \{\,(\not\!+\!\!\!+) :: [\,a\,] \to [\,a\,]\,\}$$

This declaration automatically generates a selector function

$$(\not\!+\!\!\!+) :: \mathit{DiffList}\ a \to [\,a\,] \to [\,a\,]$$

that can be used to append an ordinary list to a difference list. As an interface to difference lists, we need a function to construct the empty difference list;

$$\begin{aligned}
&\mathit{empty} :: \mathit{DiffList}\ a\\
&\mathit{empty} = \mathit{DiffList}\ \{\,(\not\!+\!\!\!+) = \mathit{id}\,\}
\end{aligned}$$

a function to construct a difference list with a single element;

$$\begin{aligned}
&\mathit{singleton} :: a \to \mathit{DiffList}\ a\\
&\mathit{singleton}\ x = \mathit{DiffList}\ \{\,(\not\!+\!\!\!+) = (x{:})\,\}
\end{aligned}$$

and a function to concatenate two difference lists.

$$\begin{aligned}
&(+\!\!\!+\!\!\!+) :: \mathit{DiffList}\ a \to \mathit{DiffList}\ a \to \mathit{DiffList}\ a\\
&a +\!\!\!+\!\!\!+ b = \mathit{DiffList}\ \{\,(\not\!+\!\!\!+) = (a\not\!+\!\!\!+) \circ (b\not\!+\!\!\!+)\,\}
\end{aligned}$$

The function $+\!\!\!+\!\!\!+$ is implemented via the function composition operator $\circ$ and has, thus, constant run time, which is the critical advantage compared to ordinary lists. The disadvantage of this representation is that we cannot perform pattern matching on difference lists without converting them back to ordinary lists. Such conversion can be performed by using $\not\!+\!\!\!+$ to stick the empty list at the end of a difference list.

$$\begin{aligned}
&\mathit{toList} :: \mathit{DiffList}\ a \to [\,a\,]\\
&\mathit{toList}\ a = a \not\!+\!\!\!+ [\,]
\end{aligned}$$

The three functions empty, $\mathit{singleton}$, and $+\!\!\!+\!\!\!+$ correspond exactly to the monadic combinators mzero, return, and mplus respectively. If we inline their definitions in the definition of anyof we obtain the following definition of $\mathit{reverse}$.

$$\begin{aligned}
&\mathit{reverse} :: [\,a\,] \to [\,a\,] \to [\,a\,]\\
&\mathit{reverse}\ [\,] \qquad\ \ = \mathit{id}\\
&\mathit{reverse}\ (x : \mathit{xs}) = \mathit{reverse}\ \mathit{xs} \circ (x{:})
\end{aligned}$$

This is the well-known linear-time implementation of the reverse function which uses an accumulating parameter to avoid repeated list concatenation. Unfortunately, we cannot instantiate the type parameter m of anyof with the type $\mathit{DiffList}$ which is no instance of $\mathit{MonadPlus}$. In order to make the type of difference lists a monad for non-determinism, we would need to implement the bind operator. Unfortunately, this is only possible by converting back and forth between difference and ordinary lists[1] which is unsatisfactory. We insist on a more elegant solution.

2.2 Continuation-passing style

To achieve a more elegant solution we need another well-known technique, viz. continuation-passing style. A function in continuation-passing style does not yield its result to the caller but is called with an additional function—a so called continuation—that expects the computed result as argument. For example, we could define integer addition in continuation-passing style as follows.

$$\begin{array}{l} \mathit{plusCPS} :: \mathit{Int} \to \mathit{Int} \to (\mathit{Int} \to a) \to a \\ \mathit{plusCPS}\ m\ n\ c = c\ (m + n) \end{array}$$

In general, if the result type of an ordinary function is a then the result type of the same function in continuation-passing style is $(a \to b) \to b$. The result type of the continuation is polymorphic. For example, we can pass print as a continuation to $\mathit{plusCPS}$ to print the computed result on the standard output.

$$\begin{array}{l} > \mathit{plusCPS}\ 17\ 4\ \mathit{print} \\ 21 \end{array}$$

2.2.1 CPS computations

We want to combine continuation-passing style with different effects modeled by a parametrised type that represents computations. For this purpose, it turns out beneficial to restrict the result type of continuations to use some parametrised type c. The type of so restricted computations in continuation-passing style is defined as follows.

$$\mathbf{newtype}\ \mathit{CPS}\ c\ a = \mathit{CPS}\ \{(\ggg\!\!-) :: \forall b.(a \to c\ b) \to c\ b\}$$

The CPS type uses so called rank-2 polymorphism to introduce the type variable b used in the result type of the continuation. We use Haskell's record syntax again to get the following selector function.

$$(\ggg\!\!-) :: \mathit{CPS}\ c\ a \to (a \to c\ b) \to c\ b$$

A value of type $\mathit{CPS}\ \ c\ \ a$ can be converted into a value of type $c\ \ a$ by passing it a continuation of type $a\ \to\ c\ \ a$ using $\ggg\!\!-$. We define a type class $\mathit{Computation}$ for

[1] $xs \ggg\!= f = \mathit{DiffList}\ \{(\not\!\!+\!\!+) = ([y \mid x \leftarrow \mathit{toList}\ xs, y \leftarrow \mathit{toList}\ (f\ x)]+\!\!+)\}$

parametrised types that can represent computations and support an operation *yield* that resembles the monadic operation *return*.

```
class Computation c where
  yield :: a → c a
```

We can now pass the operation *yield* as continuation using ≫– to run *CPS* values.

```
runCPS :: Computation c ⇒ CPS c a → c a
runCPS a = a ≫– yield
```

2.2.2 CPS monads for non-determinism

The gist of these notes is that *CPS c is a monad* for *any* parametrised type *c*. We get implementations of monadic operations *for free*.

```
instance Monad (CPS c) where
  return x = CPS {(≫–) = λc → c x}
  a ≫= f   = CPS {(≫–) = λc → a ≫– λx → f x ≫– c}
```

The last definition looks very clever. Fortunately, we do not need to invent it ourselves. It is the standard definition of monadic bind for continuation monads.

Monads for non-determinism need to support the additional operations *mzero* and *mplus*. We define another type class for parametrised types, this time to model computations that support failure and choice.

```
class Nondet n where
  failure :: n a
  choice :: n a → n a → n a
```

This type class is similar to the *MonadPlus* type class; *failure* resembles *mzero* and *choice* resembles *mplus*. However, the class *Nondet* does not require the parametrised type *n* to implement monadic bind, which the *MonadPlus* type class does. As we get monadic bind for free from the *CPS* type, we don't need to require it for types that represent non-deterministic computations.

CPS c is not only a monad for any *c*. If *n* is an instance of *Nondet* then *CPS n* is an instance of *MonadPlus*.

```
instance Nondet n ⇒ MonadPlus (CPS n) where
  mzero      = CPS {(≫–) = λ_ → failure}
  mplus a b = CPS {(≫–) = λc → choice (a ≫– c) (b ≫– c)}
```

In order to implement the operations for failure and choice we can simply dispatch to the corresponding operations of the *Nondet* class.

2.3 Efficient backtracking

Now we combine difference lists and continuation-passing style. We use the type $\mathit{DiffList}$ for difference lists and wrap it inside CPS to get an efficient implementation of the $\mathit{MonadPlus}$ type class. Note that we do not need to implement $\gg\!=$ on difference lists in order to obtain a monad on top of $\mathit{DiffList}$. We only need to implement the functions $\mathit{failure}$, yield, and choice that correspond to the monadic operations mzero, return, and mplus respectively. In order to be able to unwrap the $\mathit{DiffList}$ type from CPS, we need to provide an instance of $\mathit{Computation}$ for $\mathit{DiffList}$ and to make $\mathit{CPS}\ \mathit{DiffList}$ an instance of $\mathit{MonadPlus}$, we need to provide an instance of Nondet. Both instance declarations reuse operations for difference lists defined in Section 2.1.

```
instance Computation DiffList where
  yield   = singleton
instance Nondet DiffList where
  failure = empty
  choice  = (++)
```

We can now define efficient backtracking for non-deterministic computations.

```
backtrack :: CPS DiffList a → [a]
backtrack = toList ∘ runCPS
```

If we inline the **newtype** declarations $\mathit{DiffList}$ and CPS, we can see that the type $\mathit{CPS}\ \mathit{DiffList}\ a$ is the same as the following type.

$$\mathit{CPS}\ \mathit{DiffList}\ a \approx \forall b.(a \to [b] \to [b]) \to [b] \to [b]$$

This type is the well-known type used for two-continuation-based depth-first search. The first argument of type $a \to [b] \to [b]$ is called *success continuation* and the second argument of type $[b]$ is the so called $\mathit{failure\ continuation}$. If we inline the monadic operations, we can see that they resemble the operations derived by Hinze. The operation mzero yields the failure continuation.

```
mzero succ fail = fail
```

The return function passes the given argument to the success continuation and also passes the failure continuation for backtracking.

```
return x succ fail = succ x fail
```

The operation mplus passes the success continuation to both computations given as arguments and uses the results of the second computation as failure continuation of the first computation.

```
mplus a b succ fail = a succ (b succ fail)
```

The bind operation builds a success continuation that passes the result of the first computation to the given function.

$$(a \ggg\!= f)\ \mathit{succ}\ \mathit{fail} = a\ (\lambda x\ \mathit{fail} \rightarrow f\ x\ \mathit{succ}\ \mathit{fail})\ \mathit{fail}$$

These definitions have been devised from scratch earlier. We have obtained them by combining difference lists with continuation-passing style.

Using $\mathit{backtrack}$ to enumerate the results of calling anyof produces the same order of results as using the list monad.

```
> backtrack (anyof [1 .. 10])
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

However, the resulting list is computed more efficiently. The function $\mathit{backtrack} \circ \mathit{anyof}$ is a linear-time implementation of the reverse function. We can inline the monadic operations into the definition of anyof to verify this observation.

```
reverse' :: [a] → (a → [b] → [b]) → [b] → [b]
reverse' []       succ fail = fail
reverse' (x : xs) succ fail = reverse' xs succ (succ x fail)
```

If we specialise this definition for $\mathit{succ} = (:)$ then we obtain again the implementation of the reverse function that uses an accumulating parameter to achieve linear run time. The advantage of $\mathit{CPS}\ \mathit{DiffList}$ over $\mathit{DiffList}$ is that it has a natural implementation of monadic bind and can, hence, be used to execute monadic computations.

3 Different Search Strategies

In Section 2 we have seen how to reinvent an existing implementation of monadic backtracking. In this section we develop implementations of breadth-first search and iterative deepening depth-first search that we have not been aware of previously. Both implementations shown here are available in a Haskell package on Hackage [Fis09].

3.1 Breadth-first search

Backtracking can be trapped if the search space is infinite. If we apply anyof to an infinite list then the function $\mathit{backtrack}$ diverges without producing a result. Breadth-first search enumerates the search space in level order which results in a fair enumeration of all results.

```
newtype Levels n a = Levels {levels :: [n a]}
```

If the parameter n is an instance of Nondet we can merge the levels of a search space.

```
runLevels :: Nondet n ⇒ Levels n a → n a
runLevels = foldr choice failure ∘ levels
```

We could later use lists to represent individual levels but we use difference lists to benefit from more efficient concatenation. Thus, we define breadth-first search as follows.

$$\begin{aligned}&\mathit{levelSearch} :: \mathit{CPS}\ (\mathit{Levels}\ \mathit{DiffList})\ a \to [\,a\,]\\&\mathit{levelSearch} = \mathit{toList} \circ \mathit{runLevels} \circ \mathit{runCPS}\end{aligned}$$

We only need to provide instances of the type classes *Computation* and *Nondet* such that *levelSearch* can be applied to non-deterministic monadic computations. The definition of *yield* creates a single level that contains the given argument wrapped in the type n.

$$\begin{aligned}&\mathbf{instance}\ \mathit{Computation}\ n \Rightarrow \mathit{Computation}\ (\mathit{Levels}\ n)\ \mathbf{where}\\&\quad \mathit{yield}\ x = \mathit{Levels}\ \{\mathit{levels} = [\,\mathit{yield}\ x\,]\}\end{aligned}$$

The function *failure* is implemented as an empty list of levels and *choice* creates a new empty level (using the *failure* operation of the underlying parametrised type n) in front of the merged levels of the given computations.

$$\begin{aligned}&\mathbf{instance}\ \mathit{Nondet}\ n \Rightarrow \mathit{Nondet}\ (\mathit{Levels}\ n)\ \mathbf{where}\\&\quad \mathit{failure} \quad\;\;\; = \mathit{Levels}\ \{\mathit{levels} = [\,]\}\\&\quad \mathit{choice}\ a\ b = \mathit{Levels}\ \{\mathit{levels} = \mathit{failure} : \mathit{merge}\ (\mathit{levels}\ a)\ (\mathit{levels}\ b)\}\end{aligned}$$

The use of *failure* in the implementation of *choice* is crucial to achieve breadth-first search because it delays the results at deeper levels which are combined using *merge*.

$$\begin{aligned}&\mathit{merge} :: \mathit{Nondet}\ n \Rightarrow [\,n\ a\,] \to [\,n\ a\,] \to [\,n\ a\,]\\&\mathit{merge}\ [\,] \qquad\quad \mathit{ys} \qquad = \mathit{ys}\\&\mathit{merge}\ \mathit{xs} \qquad [\,] \qquad\; = \mathit{xs}\\&\mathit{merge}\ (x : \mathit{xs})\ (y : \mathit{ys}) = \mathit{choice}\ x\ y : \mathit{merge}\ \mathit{xs}\ \mathit{ys}\end{aligned}$$

We might feel inclined to generalise the type *Levels* to use an arbitrary parametrised type instead of lists to represent the collection of levels. Such a type would need to provide a *zip* operation to implement *merge* which we could require using another type class, e.g., *Zipable*. We refrain from such a generalisation in favour of a simpler description.

3.2 Iterative deepening depth-first search

Breadth-first search has an advantage when compared with depth-first search—it is fair. However, there is also a disadvantage. It needs a huge amount of memory to store complete levels of the search space. We can trade memory requirements for run time by using depth-first search to enumerate all results of the search space that are reachable within a certain depth limit and incrementally repeat the search with increasing depth limits.

We can define a type for depth-bounded search as a function that takes a depth limit and yields results that can be found within the given limit.

$$\mathbf{newtype}\ \mathit{Bounded}\ n\ a = \mathit{Bounded}\ \{(!) :: \mathit{Int} \to n\ a\}$$

The type parameter n is later required to be an instance of *Computation* and *Nondet* and holds the results of depth-bounded search. We use ordinary lists but omit corresponding instances for the list type.

We can define an instance of *Nondet* for *Bounded* n as follows. The implementation of *failure* uses the *failure* operation of the underlying type n.

```
instance Nondet n ⇒ Nondet (Bounded n) where
  failure    = Bounded {(!) = λ_ → failure}
  choice a b = Bounded {(!) = λd → if d ≡ 0 then failure
                                   else choice (a ! (d − 1)) (b ! (d − 1))}
```

The *choice* operation fails if the depth limit is exhausted. Otherwise, it calls the underlying *choice* operation on the given arguments with a decreased depth limit, reflecting that a choice descends one level in the search space.

Iteratively increasing the depth limit of a depth-bounded computation yields a list of levels.

```
levelIter :: (Computation n, Nondet n) ⇒
             Int → CPS (Bounded n) a → Levels n a
levelIter step a = Levels {levels = [(a ≫− yieldB) ! d | d ← [0, step ..]]}
  where
    yieldB x = Bounded {(!) = λd → if d < step then yield x else failure}
```

Between different searches the depth limit is incremented by *step*. If *step* equals one then the returned levels are really the levels of the search space. If it is greater then multiple levels of the search space are collected in a single level of the result.

Instead of *runCPS* (which is defined as (≫−*yield*)) we use a custom function *yieldB* and pass it as continuation to the given computation. This allows us to yield only those results where the remaining depth limit is small enough, i.e., that have not been enumerated in a previous search. We merge the different levels of iterative deepening search using an arbitrary instance of *Nondet*—using lists results in iterative deepening depth-first search.

```
iterDepth :: (Computation n, Nondet n) ⇒ Int → CPS (Bounded n) a → n a
iterDepth step = foldr choice failure ∘ levels ∘ levelIter step
```

This implementation of iterative deepening depth-first search is novel because it does not require the depth limit to be returned by depth-bound computations. If we wanted to implement ≫= directly on the type *Bounded* n we would need an updated depth limit as result of executing the first argument (see Spivey's implementation [Spi06]). We don't need to thread the depth limit explicitly when using the bind operation of the *CPS* type.

Both *levelSearch* and *iterDepth* can enumerate arbitrary infinite search spaces lazily.

```
> take 10 (levelSearch (anyof [1 ..])) ≡ take 10 (iterDepth 1 (anyof [1 ..]))
True
```

In fact, when using lists for the results, *iterDepth* 1 always returns them in the same order as *levelSearch* because it enumerates one level after the other from left to right.

4 Variations on a Theme

Using the types developed in the previous sections we can build numerous variants of the presented search strategies. In this section we compare experimentally three different versions of depth-first search and two versions of both breadth-first and iterative deepening depth-first search. All presented implementations can be built from the types developed in the previous sections: the parametrised types $[\,]$, $CPS\ [\,]$, and $CPS\ DiffList$ are all instances of $MonadPlus$ that implement depth-first search. The types $CPS\ (Levels\ [\,])$ and $CPS\ (Levels\ DiffList)$ implement breadth-first search and using $CPS\ (Bounded\ [\,])$ or $CPS\ (Bounded\ DiffList)$ results in iterative deepening depth-first search.

What if we keep following this pattern further? We can also build the types $CPS\ (x\ (y\ z))$ with $x, y \in \{Levels, Bounded\}$ and $z \in \{[\,], DiffList\}$. We can stack arbitrarily many layers of $Levels$ and $Bounded$ between CPS and $[\,]$ or $DiffList$. If we define instances $Computation\ (CPS\ c)$ and $Nondet\ c \Rightarrow Nondet\ (CPS\ c)$ similar to the $Monad$ and $MonadPlus$ instances for CPS then we can also include multiple layers of CPS between $Levels$ and $Bounded$. The inclined reader may investigate these types and the performance properties of the resulting strategies. We include some of them in our comparison.

4.1 Pythagorean triples

We measure run time and memory requirements of the different non-determinism monads using the $anyof$ function and a slightly more complex action that returns Pythagorean triples non-deterministically. A Pythagorean triple is a strictly increasing sequence of three positive numbers a, b, and c such that $a^2 + b^2 = c^2$.

```
pytriple :: MonadPlus m ⇒ m (Int, Int, Int)
pytriple = do a ← anyof [1 ..]; b ← anyof [a + 1 ..]; c ← anyof [b + 1 ..]
              guard (a ∗ a + b ∗ b ≡ c ∗ c)
              return (a, b, c)
```

The predefined function $guard :: MonadPlus\ m \Rightarrow Bool \rightarrow m\ ()$ fails if its argument is $False$ and we use it to filter Pythagorean triples from arbitrary strictly increasing sequences of three positive numbers.

That is a concise declarative specification of Pythagorean triples but can we execute it efficiently? It turns out that (unbounded) depth-first search is trapped in infinite branches of the search space and diverges without returning a result. We need a complete search strategy like breadth-first search or iterative deepening search to execute $pytriple$. In order to be able to compare those strategies with unbounded depth-first search, we use a variant $pytriple\_leq :: MonadPlus\ m \Rightarrow Int \rightarrow m\ (Int, Int, Int)$ that computes Pythagorean triples where all components are less or equal a given number. For this task the search space is finite and can also be explored using incomplete strategies.

	anyof	*pytriple*	*pytriple_leq*
$[\,]$	179s / 9MB	— / —	44s / 2MB
$CPS\ [\,]$	196s / 11MB	— / —	4s / 2MB
$CPS\ DiffList$	0s / 6MB	— / —	10s / 2MB
$CPS\ (Levels\ [\,])$	0s / 1MB	21s / 966MB	12s / 966MB
$CPS\ (Levels\ DiffList)$	0s / 1MB	23s / 966MB	13s / 966MB
$CPS\ (Bounded\ [\,])$	223s / 17MB	38s / 2MB	16s / 2MB
$CPS\ (Bounded\ DiffList)$	7s / 14MB	54s / 2MB	25s / 2MB
$CPS\ (Bounded\ (CPS\ [\,]))$	200s / 19MB	47s / 2MB	20s / 2MB
$CPS\ (Levels\ (Levels\ DiffList))$	0s / 1MB	1206s / 2041MB	24s / 1929MB

Table 1: Performance of different search strategies

4.2 Experimental results

The run time and memory requirements of the different strategies are depicted in Table 1.

anyof executes the call *anyof* $[1\,..\,50000]$ and enumerates the results w.r.t. to the strategies depicted in the leftmost column of the table.

pytriple enumerates 500 Pythagorean triples without an upper bound for their components. This benchmark can only be executed using complete strategies, there are no results for unbounded depth-first search.

pytriple_leq enumerates all 386 Pythagorean triples with an upper bound of 500 using all search strategies.

All benchmarks were executed on an Apple MacBook 2.2 GHz Intel Core 2 Duo with 4 GB RAM using a single core. We have used the Glasgow Haskell Compiler (GHC, version 6.10.3) with optimisations (-O -fno-full-laziness) to compile the code. When executing breadth-first search, we have provided an initial heap of 1 GB (+RTS -H1G). We have increased the depth-limit of iterative deepening search by 100 between different searches.

The *anyof* benchmark demonstrates the quadratic complexity of depth-first search strategies based on list concatenation. The corresponding search space is degenerated as it is a narrow but deep tree. Hence, there is noticeable overhead when performing iterative deepening search. The search space for enumerating Pythagorean triples is more realistic. With and without an upper limit, breadth-first search is faster than iterative deepening depth-first search but uses significantly more memory. Using difference lists instead of ordinary lists does not improve the performance of breadth-first search in our benchmarks. We have observed the memory requirements of iterative deepening depth-first search to be constant only when we disabled *let floating* by turning off the *full-laziness optimisation* of GHC. This optimisation increases sharing in a way that defeats the purpose of iteratively exploring the search space by recomputing it on purpose. Iterative deepening depth-first search incurs noticeable overhead compared to ordinary depth-first search which, however, can only be applied if the search space is finite.

Finally, we have tested two esoteric strategies, viz., $\mathit{CPS}\ (\mathit{Bounded}\ (\mathit{CPS}\ []))$ and $\mathit{CPS}\ (\mathit{Levels}\ (\mathit{Levels}\ \mathit{DiffList}))$. The former demonstrates that even wrapping the list type under multiple layers of CPS does not improve on the quadratic complexity of the mplus operation when nested left associatively. Moreover, the extra CPS layer causes moderate overhead compared to ordinary iterative-deepening depth-first search. Using two layers of Levels for breadth-first search blows up the memory requirements even more. Although we have run this specific benchmark with 2 GB initial heap, the memory requirements are so huge that reclaiming memory is sometimes a significant performance penalty. The large difference in run time between the $\mathit{pytriple}$ and the $\mathit{pytriple\_leq}$ benchmarks is suspicious. Probably, the slowdown is caused by limiting memory by the option -M2G.

The experiments suggest to use the monad $\mathit{CPS}\ \mathit{DiffList}$ if the search space is known to be finite and $\mathit{CPS}\ (\mathit{Bounded}\ \mathit{DiffList})$ if it is not. Although there is a moderate overhead of difference list compared to usual lists, the latter perform much worse in case of left associative uses of mplus. The memory requirements of breadth-first search prohibit its use in algorithms that require extensive search.

5 Final Notes

We have employed a continuation monad transformer to implement monads for non-determinism based on types that do not (need to) support monadic bind. Combining this approach with difference lists leads to the well-known two-continuation-based backtracking monad. Hinze has derived similar backtracking as monad transformer that adds backtracking functionality to an arbitrary base monad [Hin00]. Using different base types in our approach, we have found novel implementations of breadth-first search and iterative deepening depth-first search.

The latter strategies have also been implemented in Haskell by Spivey [Spi06] but not as instances of the $\mathit{MonadPlus}$ type class. Spivey uses a slightly different interface with an operation $\oplus$ for non-deterministic choice and an additional operation wrap to increase the search depth by one level. Our implementations of breadth-first search and depth-bounded search use a single operation choice that could be expressed as combination of $\oplus$ and wrap in Spivey's framework and allows us to implement both strategies in the $\mathit{MonadPlus}$ framework. Both implementations differ from the corresponding implementations given by Spivey due to the use of a continuation monad. Unlike Spivey's implementation, we can use difference list to represent levels for breadth-first search and don't need to return updated depth limits in depth-bounded search.

Similar to the asymptotic improvement of the mplus operation provided by the $\mathit{DiffList}$ type, monadic bind as defined for the CPS type improves the complexity of calls to $\gg=$ nested to the left. While some monads incur run-time quadratic in the number $\gg=$ calls nested to the left, the continuation-based implementation is linear [Voi08]. We have compared variations of the presented search strategies experimentally and found that the two-continuation-based backtracking monad outperforms the other strategies. Iterative deepening search, which requires only constant space, is also suitable for infinite search spaces.

Monads for non-determinism are usually expected to satisfy certain laws. Instances of $MonadPlus$ derived with the presented approach satisfy the monad laws [Com09a] by construction because the implementations of $return$ and $\gg\!=$ are always those of the continuation monad. Whether the derived instances satisfy laws for $MonadPlus$ [Com09b] depends on the employed instance of $Nondet$. The strategies presented in Section 3 do not satisfy the monoid laws of the $mzero$ and $mplus$ operations. However, manipulating a non-deterministic program w.r.t. these laws has no effect on which results are computed—it only affects their order.

Acknowledgements

Frank Huch proposed to implement depth-bounded search without returning updated depth limits and Janis Voigtländer provided valuable comments on a first draft of this paper.

References

[Com09a] The Haskell Community. `http://haskell.org/haskellwiki/Monad_Laws`, 2009.

[Com09b] The Haskell Community. `http://haskell.org/haskellwiki/MonadPlus`, 2009.

[Fis09] Sebastian Fischer. `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/level-monad`, 2009.

[Han07] Michael Hanus. Multi-paradigm Declarative Languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.

[Hin00] Ralf Hinze. Deriving backtracking monad transformers. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 186–197, New York, NY, USA, 2000. ACM.

[SHC95] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury, an Efficient Purely Declarative Logic Programming Language. In *In Proceedings of the Australian Computer Science Conference*, pages 499–512, 1995.

[Spi06] Michael Spivey. Algebras for combinatorial search. In *Workshop on Mathematically Structured Functional Programming*, 2006.

[Voi08] Janis Voigtländer. Asymptotic Improvement of Computations over Free Monads. In Christine Paulin-Mohring and Philippe Audebaud, editors, *Mathematics of Program Construction, Marseille, France, Proceedings*, volume 5133 of *LNCS*, pages 388–403. Springer-Verlag, July 2008.

[Wad95] Philip Wadler. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag.