The Monad.Reader/Issue2 /FunWithLinearImplicitParameters

From HaskellWiki

< The Monad.Reader | Issue2

This article needs reformatting! Please help tidy it up --WouterSwierstra 14:12, 9 May 2008 (UTC)

[attachment:Reflection.pdf PDF version of this article]

[attachment:Reflection.tar.gz Code from the article] [:IssueTwo/FeedBack /FunWithLinearImplicitParameters: Feedback]

Contents

1 Fun with Linear Implicit Parameters

1.1 Monadic Reflection in Haskell

by [:ThomasJ‰ger:Thomas J‰ger] for The Monad.Reader [:IssueTwo:Issue Two] BR *01.05.2005*

Abstract. Haskell is widely believed to be a purely functional language. While this is certainly true for Haskell98, GHC's various extensions can interplay in unforeseen ways and make it possible to write side-effecting code.

In this article, we take the next step of impure programming by implementing

Filinski's
`reflect`
and
`reify`
functions for a wide class of monads.

1.2 Introduction

The following sections provide a short introduction into the various concepts our implementation uses. You can download the implementation and the examples from the article [attachment:Reflection.tar.gz here], it has been successfully tested with ghc-6.2.2 and ghc-6.4. The examples of this article can be found

in the file
`Article.hs`
, the implementation of the library in
`Reflection.hs`
.

1.2.1 Shift and Reset

The
`shift`
and
`reflect`
control operators provide a way to manipulate

delimited continuations, which are similar to the undelimited continuation the

familiar

`call/cc`
uses, but more powerful. There are more detailed

descriptions available e.g. in Danvy & Filinski #ref1 1 and Shan #ref2 2; moreover, Dybvig, Peyton Jones, Sabry #ref3 3 give a unifying treatment of various forms of other "subcontinuations".

Instead of capturing an undelimited continuation as
`call/cc`
,
`shift`
only captures the subcontinuation/context up to the the next
`reset`
, and

reifies it into a function value. The result of the evaluation of the body then

becomes the result of the
`reset`
. For example in

```
#!syntax haskell
reset (1 + shift (\k -> k 1 + k 2)) :: Int
```

the context of
`shift`
is
`k = \x -> x + 1`
, so the expression evaluates to
`k 1 + k 2 = 2 + 3 = 5`
. The interpretation of
`shift`
and
`reset`
is very easy in the

continuation monad.

```
#!syntax haskell
-- An action in the continuation monad maps a continuation,
-- i.e the "rest" of the computation, to a final result of type r.
newtype Cont r a = Cont { runCont :: (a -> r) -> r }

instance Functor (Cont r) where {- ... -}
instance Monad (Cont r) where {- ... -}

-- NB. In the attached Article.hs file, these are called shiftC and resetC.
shift :: ((a -> r) -> Cont r r) -> Cont r a
shift e = Cont $ \k -> reset (e k)

reset :: Cont a a -> a
reset e = e `runCont` id

-- The above example written in monadic style
* Main> reset $ (1 +) `fmap` shift (\k -> return $ k 1 + k 2)
5
```

As we can see,
`reset e`
delimits all effects of
`e`
and returns a pure value;
`shift`
lets us explicitly construct the mapping from continuations to final results, so it is very similar to the data constructor
`Cont`
. Therefore
`shift`
and
`reset`
give us full control over the underlying continuation monad and are thereby strictly more expressive than
`call/cc`
, which is polymorphic in the answer type
`r`
. To treat the direct-style
`shift`
and
`reset`
safely in a typed

setting, it is necessary to express the answer type of the underlying continuation monad in the types. The Hindley-Milner type system cannot express this, but luckily, Haskell allows type information to be hidden in contexts, which provides our approach with full static type safety as opposed to Filinski's implementation in SML.

1.2.2 Monadic Reflection

Monadic reflection #ref4 4 enables us to write monadic code in direct style.

`reflect`
"reflects" a monadic value into a first-class value of our

language. The side effects can then be observed by "reifing" a value back into monadic form. For example,

```
#!syntax haskell
> reify (reflect [0,2] + reflect [0,1]) :: [Int]
and
> liftM2 (+) [0,2] [0,1]
```

both yield the same result, namely
`[0,1,2,3]`

In order to understand how monadic reflection can be implemented, we combine

the observation that
`shift`
and

reset
give us the full power over an

underlying continuation monad with an arbitrary answer type with Wadler's #ref5 5 observation that every monad can be embedded in the continuation

monad. So using a direct-style
shift
and
reset
, we can write

arbitrary monadic code in direct style.

Explicitly (but hiding the wrapping necessary for the ContT monad transformer), Wadler's transformation is as follows

```
#!syntax haskell
embed :: Monad m => m a -> (forall r. (a -> m r) -> m r)
embed m = \k -> k =<< m

project :: Monad m => (forall r. (a -> m r) -> m r) -> m a
project f = f return
```

Here,
`project . embed === id`
and the property of
`embed`
and
`project`
constituting monad morphisms between the monad
`m`
and the monad
`forall r. ContT m r a`
can easily be checked.

Translating these morphisms into direct style, we immediately arrive at

Filinski's
`reflect`
and
`reify`
operations

```
#!syntax haskell
reflect m = shift (\k -> k =<< m)
reify t = reset (return t)
```

Now let us have a closer look at the above example to see how it works operationally.

```
#!syntax haskell
e = reify (reflect [0,2] + reflect [0,1])
```

Substituting the definitions, this becomes

```
#!syntax haskell
e = reset (return (shift (\k -> k =<< [0,2]) + shift (\k -> k =<< [0,1])))
```

which simplifies to

```
#!syntax haskell
e = reset [shift (\k -> k 0 ++ k 2) + shift (\k' -> k' 0 ++ k' 1)]
```

Assuming left to right evaluation, the result of this expression is

`k 0 ++ k 2`
where
`k`
is bound to the subcontinuation

```
#!syntax haskell
k = \x -> reset [x + shift (\k' -> k' 0 ++ k' 1)]
```

Again, in this term,
`k'`
is bound to
`\y -> reset [x + y]`
, so
`k`
is the function

```
#!syntax haskell
k = \x -> [x + 0] ++ [x + 1] = \x -> [x,x+1]
```

Therefore, as we expected, the whole expression evaluates to

```
#!syntax haskell
e = k 0 ++ k 2 = [0,1] ++ [2,3] = [0,1,2,3]
```

1.2.3 Implicit Parameters

Implicit parameters #ref6 6 are GHC-specific type system extension providing dynamically bound variables. They are passed in the same way as type class dictionaries, but unlike type class dictionaries, their value can be changed for a subexpression. The types of the implicit parameters a function expects appear in type contexts which now make sense at arbitrary argument positions.

```
#!syntax haskell
addThree :: (?foo :: Int) => Int
addThree = 3 + ?foo

withFour :: ((?foo :: Int) => a) -> a
withFour x = let ?foo = 4 in x
```

```
* Main> withFour addThree
7
```

We see that implicit parameters act like a reader monad written in direct style. The commutativity of the reader monad ensures that the code still is referentially transparent (the monomorphic recursion issue aside that will be discussed below).

Linear implicit parameters #ref6 6 work very much like regular implicit parameters, but the type of the parameter is required to be an instance of the

class
`GHC.Exts.Splittable`
with the single method
`split :: a -> (a,a)`
. At each branching point of the computation, the

parameter gets split, so that each value is used only once. However, as we shall later see, this linearity is not enforced in all circumstances, with higher order functions and a certain class of recursive functions being the notable exceptions.

Possible uses are random number distribution, fresh name generation (if you do not mind the names becoming very long) or a direct-style !QuickCheck #ref7 7. In this article, they will be used to store a subcontinuation from

an enclosing
`reset`
. The syntax is exactly the same as in the implicit case with the
`?`
replaced by
`%`
. We give a small example

illustrating their intended use.

```
#!syntax haskell
import qualified System.Random as R

instance Splittable R.StdGen where split = R.split

randInts :: R.StdGen -> (Int, Int, Int)
randInts gen = let %gen = gen in (rand, rand, rand) where
 rand :: (%gen :: R.StdGen) => Int
 rand = fst $ R.random %gen
* Main> print . randInts =<< R.getStdGen
(-1305955622,-1639797044,-945468976)
```

As in the implicit case, the semantics of linear implicit parameters can be described in terms of a "monad", which, however, does not obey the monad laws in any nontrivial case.

```
#!syntax haskell
newtype Split r a = Split { runSplit :: r -> a }

instance Functor (Split r) where
```

```
 f `fmap` Split x = Split $ f . x

instance Splittable r => Monad (Split r) where
 return x = Split $ const x
 Split x >>= f = Split $ \s ->
 let (s1,s2) = split s in f (x s1) `runSplit` s2

toSplit :: ((%foo :: r) => a) -> Split r a
toSplit x = Split $ \r -> let %foo = r in x

fromSplit :: Split r a -> ((%foo :: r) => a)
fromSplit (Split f) = f %foo
```

The ability to freely transform between "monadic" and "implicit" style is often very helpful, e.g. to work around GHC's limitation that signature contexts in a mutually recursive group must all be identical.

1.2.4 Unsafe Operations

The code below uses two unsafe operations #ref8 8. We briefly discuss which conditions must be checked in order to ensure that they are used in a "safe" way.

```
#!syntax haskell
unsafePerformIO :: IO a -> a
unsafeCoerce# :: a -> b
```

The
`unsafePerformIO`
function executes an
`IO`
action and returns the

result as a pure value. Thus, it should only be used if the result of the action does not depend on the state of the external world. However, we do not demand that the result of the computation be independent of the evaluation order. Furthermore, we must be aware that the compiler may inline function

definitions, so that two invocations of
`unsafePerformIO`
might be unexpectedly shared or duplicated. The
`{-# NOINLINE foo #-}`
pragma can

be used to forbid inlining in such cases.

The
`unsafeCoerce#`
function is used to convert a value between two types that

are known to be equal although the type system cannot proof this fact. If the types do not match, its behavior is undefined; usually, the program will crash or return a wrong result.

1.2.5 Dynamic Exceptions

In addition to exceptions that only print an error message, the Hierarchical

Libraries provide the
`throwDyn`
and
`catchDyn`
functions that throw and catch

exceptions of an arbitrary instance of the class Typeable. However, there is a tricky aspect of exceptions because of Haskell's laziness. Consider

```
#!syntax haskell
* Main> print =<< evaluate ([1,throwDyn "escape"])
 `catchDyn` \"escape" -> return [2]
[1,*** Exception: (unknown)
```

Here the evaluation of the list only determines whether the list is empty, but the list is inspected when the expression is printed, and thus the exception

escapes the
`catchDyn`
exception handler.

When all thrown exception have to be caught, we must evaluate the expression fully before handling the exception, which can

be ensured with the
`DeepSeq`
#ref9 9 class.

```
#!syntax haskell
infixr 0 `deepSeq`, $!!

class DeepSeq a where
 deepSeq :: a -> b -> b

($!!) :: (DeepSeq a) => (a -> b) -> a -> b
f $!! x = x `deepSeq` f x
```

Not all types can be made an instance of
`DeepSeq`
. In particular, functions with an infinite domain and
`IO`
actions cannot be fully evaluated in a

sensible way.

1.3 Implementation

This section discusses the implementation of the monadic reflection library. It safely be skipped, especially the first two subsections are very technical.

1.3.1 Basic Declarations

`k :-> v`
is just an abstract representation of a finite map from k to v, The type
`Position`
will be used to store the context of the evaluation, so

it should have the property that different sequences of applications of

`leftPos`
and
`rightPos`
to an
`initPos`
yield different values. A
`Cell`
stores a value of arbitrary type. The most interesting declaration is that of
`Prompt`
. The field
`position`
saves the position of the current expression relative to the next enclosing reset,
`prompt`
is the expression this next enclosing
`reset`
computes,
`facts`
stores the subexpressions that already have been assigned a value, and
`promptID`
will

be used for exception handling.

```
#!syntax haskell
infixr 9 :->

lookup :: Ord k => (k :-> v) -> k -> Maybe v
insert :: Ord k => (k :-> v) -> k -> v -> k :-> v
empty :: k :-> v

leftPos :: Position -> Position
rightPos :: Position -> Position
initPos :: Position

type Facts = Position :-> Cell

data Cell = forall a. Cell a deriving Typeable

data Prompt r = Prompt {
 position :: Position,
 prompt :: Direct r r,
 facts :: Facts,
 promptID :: Unique
}

newPrompt :: Facts -> Direct r r -> Prompt r

instance Splittable (Prompt r) where
 split p = (p {position = leftPos pos},
 p {position = rightPos pos}) where
 pos = position p

type Direct r a = (%ans :: Prompt r) => a
```

1.3.2 Shift and Reset

`shift`
first saves the
`Prompt`
and checks if this
`shift`
has already been assigned a value using the
`facts`
dictionary. If so, it just returns that value, otherwise, the outer
`reset`
should return the value of
`f`
applied to the subcontinuation from the
`shift`
to the
`reset`
. The subcontinuation we pass to
`f`
creates a new copy of the
`Prompt`
on every invocation, updates the
`facts`
dictionary with the additional information that instead of the current
`shift`
, the value
`x`
should be returned, and finally executes the
`prompt`
computation of the enclosing
`reset`
. In order to pass the result of
`f`
up to the next
`reset`
, we use exception handling, the unique ID of the
`Prompt`
ensures that it is handled at the right place; the value, although known to be of type
`r`
is put in a
`Cell`
because we do not know whether
`r`
is an instance of the class
`Typeable`
. Now all
`reset`
has to do is evaluate the expression with a fresh
`Prompt`
, and return the thrown value instead if an exception is caught.

This gets a little more complicated because we need to be able to handle the

effects of nested
`resets`
.

```
#!syntax haskell
shift :: ((a -> r) -> Direct r r) -> Direct r a
shift f :: Direct r a =
 let ans :: Prompt r
 ans = %ans
 in case lookup (facts ans) (position ans) of
 Just (Cell a) -> unsafeCoerce# a
 Nothing -> throwDyn . (,) (promptID ans) . Cell . f $ \x ->
 let %ans = newPrompt
 (insert (facts ans) (position ans) (Cell x))
 (prompt ans)
 in prompt ans

reset :: DeepSeq r => Direct r r -> r
reset e :: r = let %ans = newPrompt empty res in res where
 res :: Direct r r
 res = unsafePerformIO $ do
 let catchEsc e' = evaluate (id $!! e') `catchDyn`
 \err@(i, Cell result) ->
 if i == promptID %ans
 then catchEsc $ unsafeCoerce# result
 else throwDyn err
 catchEsc e
```

It is interesting to observe that in case of the error monad, this code uses

the
`IO`
monad's exception handling mechanism to propagate the error.

Finally, we need to check the unsafe features are used in a safe way as

described above. The
`unsafeCoerce#`
calls are always coercing to type
`r`
and it is clear that always the same
`r`
is in scope which we are ensuring using the
`i == promptID`
check.
`unsafePerformIO`
is only

used for a "pure exception handling", which destroys purity, but still satisfies the weaker condition that the behavior does not depend on the outside world, which is essential here, as we rely on the property that a computation performs exactly the same steps when rerun.

1.3.3 Reflection and Reification

With working
`shift`

and
`reset`
functions, we can now turn to monadic

reflection primitives. We first consider the case of the continuation monad.

1.3.3.1 Reflecting the Cont Monad

```
#!syntax haskell
reflectCont :: Cont r a -> Direct r a
reflectCont (Cont f) = shift f

reifyCont :: DeepSeq r => Direct r a -> Cont r a
reifyCont e = Cont $ \k -> reset (k e)
```

As an example, we lift the function
`callCC`
from
`Control.Monad.Cont`

to direct-style.

```
#!syntax haskell
callCC' :: DeepSeq r => ((a -> b) -> Direct r a) -> Direct r a
callCC' f = reflectCont $ callCC $ \c -> reifyCont $ f $ reflectCont . c
```

However, the
`call/cc`
operation can be implemented much more nicely using only two
`shift`
s, as in

```
#!syntax haskell
callCC' :: ((forall b. a -> b) -> Direct r a) -> Direct r a
callCC' f = shift $ \k -> k $ f (\x -> shift $ \_ -> k x)
```

In both versions, the expression

```
#!syntax haskell
reset (callCC' (\k x -> k (x+)) 5) :: Int
```

correctly evaluates to
`10`
. It is a nice exercise to do this in Haskell's

continuation monad; but be warned that it is a little harder than the above direct-style version.

1.3.3.2 Reflecting Arbitrary Monads

Now, implementing
`reflect`

and
`reify`
is easier than in Filinski's implementation in SML, because the stronger static guarantees of our
`shift`
and
`reset`
functions eliminate the need for unsafe coercion functions.

```
#!syntax haskell
-- Type alias for more concise type signatures of direct-style code.
type Monadic m a = forall r. Direct (m r) a

reflect :: Monad m => m a -> Monadic m a
reflect m = shift (\k -> k =<< m)

reify :: (DeepSeq (m a), Monad m) => Monadic m a -> m a
reify t = reset (return t)
```

1.4 Interface

For quick reference, we repeat the type signatures of the most important library functions.

```
#!syntax haskell
type Direct r a = (%ans :: Prompt r) => a
shift :: ((a -> r) -> Direct r r) -> Direct r a
reset :: DeepSeq r => Direct r r -> r

type Monadic m a = forall r. Direct (m r) a
reflect :: Monad m => m a -> Monadic m a
reify :: (DeepSeq (m a), Monad m) => Monadic m a -> m a
```

1.5 Resolving Ambiguities

The use of linear implicit parameters comes with a few surprises. The GHC manual #ref6 6 even writes

1. 1. quote

```
So the semantics of the program depends on whether or not foo has a type
signature. Yikes!
You may say that this is a good reason to dislike linear implicit parameters
and you'd be right. That is why they are an experimental feature.
```

However, most of the problems can be circumvented quite easily, and the property that the meaning of a program can depend on the signatures given is actually a good thing.

1.5.1 Recursive Functions

Indeed, omitting a type signature can sometimes result in a different behavior. Consider the following code, where

```
shift (\k -> k n)
```
and
```
n
```
should behave identically.

```
#!syntax haskell
-- Without the explicit signature for k GHC does not infer a
-- sufficiently general type.
down 0 = []
down (n+1) = shift (\(k::Int -> [Int]) -> k n): down n
* Main> reset (down 4)
[3,3,3,3] -- wrong!
```

GHC considers the function
```
down
```
to be monomorphically recursive, but in fact the recursive call to
```
down
```
should be in a different context (with the implicit parameter bound to a different value), so
```
down
```
should

actually be polymorphically recursive. This is semantically different and ensures the linearity. We can persuade GHC to treat it correctly by giving the function an explicit signature.

```
#!syntax haskell
down' :: Int -> Direct [Int] [Int]
{- ... -}
* Main> reset (down' 4)
[3,2,1,0] -- right!
```

Furthermore, we have to watch out for a GHC bug #ref10 10 that appears to happen when expressions with differently polymorphic linear implicit parameter constraints are unified. In the above example, this occurs when

```
k
```
's explicit type signature is dropped and the signature of
```
down
```
is not generalized to
```
Int -> Direct r [Int]
```
.

1.5.2 Higher order functions

Implicit parameters are particularly tricky when functions using implicit parameters are passed to higher order functions. Consider the following example.

```
#!syntax haskell
-- The prelude definition of the function map
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

```
foo :: [[Int]]
foo = reify (map f [1,2,3]) where
 f :: Int -> Monadic [] Int
 f x = reflect [-x,x]
* Main> foo
[[-1,-1,-1],[1,1,1]] -- wrong!
```

The first surprise is that this code type checks at all: The type of the

function
`f`
is
`Int -> Monadic [] Int`
but in order to be passed to
`map`
, the function
`f`
must have the different type
`Monadic [] (Int -> Int)`
.

GHC pushes contexts at covariant argument positions as far to the left as possible using a technique called for-all-hoisting #ref6 6, which is of course sensible for type class constraints and implicit parameters, but destroys the linearity, which seems bad even in the motivating examples of random number or fresh name generation, and is only OK in the !QuickCheck example. So we always have to watch out for effectful functions that are passed as parameters, but at least we can copy the implementation of the higher order functions we want to use.

```
#!syntax haskell
map' :: (a -> Direct r b) -> [a] -> Direct r [b]
{- Implementation as above -}
foo = reify (map' f [1,2,3]) where {- ... -}
* Main> foo
[[-1,-2,-3],[-1,-2,3],[-1,2,-3],[-1,2,3],[1,-2,-3],[1,-2,3],[1,2,-3],
[1,2,3]] -- right!
```

1.5.3 The Monomorphism Restriction

What should the expression

```
#!syntax haskell
reify (let x = reflect [0,1] in [x,x+2,x+4])
```

evaluate to? Two possibilities come to mind: Either we choose a value for the

variable
`x`
first, and then evaluate the lists
`[x,x+2,x+4]`
or we view
`x`
as the reflected list

`[0,1]`
and the choice whether
`x`
stands for
`0`
or
`1`
is made whenever
`x`
it is evaluated. It is

immediately clear how both variants can be achieved in monadic style.

```
#!syntax haskell
* Main> do x <- [0,1]; return [x,x+2,x+4]
[[0,2,4],[1,3,5]]
* Main> let x = [0,1] in sequence [x,(+2) `fmap` x, (+4) `fmap` x]
[[0,2,4],[0,2,5],[0,3,4],[0,3,5],[1,2,4],[1,2,5],[1,3,4],[1,3,5]]
```

In direct style, this is even easier, but the meaning of our code now depends on the type signature.

```
#!syntax haskell
* Main> reify (let x :: Int; x = reflect [0,1] in [x,x+2,x+4])
[[0,2,4],[1,3,5]]
* Main> reify (let x :: Monadic [] Int; x = reflect [0,1] in [x,x+2,x+4])
[[0,2,4],[0,2,5],[0,3,4],[0,3,5],[1,2,4],[1,2,5],[1,3,4],[1,3,5]]
```

It is important that we give a real type signature:

`x :: Int = reflect [0,1]`
does not make any difference!

This is a nice and very natural way to describe both situations, but the answer to the question which one GHC chooses when no signature is given is less satisfactory: It depends on the status of the flag

`-f(no)monomorphism-restriction`
. With the monomorphism "restriction" #ref11 11 turned on,
`x`
must have

a monomorphic type, so the first situation applies, without the restriction

`x`
gets the most general type which leads to the second behavior. In my

opinion, it would be nice if there were a flag that, in order to give the programmer a chance to disambiguate his code, causes a warning to be emitted whenever the monomorphism restriction kicks in; a similar warning has been proven useful to detect numeric defaulting.

1.6 Examples

We now present some examples reflecting the
`Cont`
and
`[]`
monads.

1.6.1 Lazy Evaluation

The use of monads in Haskell models an impure language with call-by-value semantics. This is not surprising as one motivation for the use of monads is the need to do IO. For IO, evaluation order is important and call-by-value

makes evaluation order easier to reason about. For the
`IO`
monad this certainly the right decision, and if desired, the
`unsafeInterleaveIO`
function can be used to execute
`IO`
operations lazily.

But such a lazy monadic behavior would be practical for other monads, too: The list monad is very susceptible to space leaks and unnecessary recomputation. The reflected list monad, however, is often closer to the desired behavior, as the following examples suggest.

```
#!syntax haskell
-- Lazy repeat, Prelude.repeat would allow the side effect
-- of the argument to take place only once
repeat' :: Direct r a -> Direct r [a]
repeat' x = x:repeat' x
* Main> take 3 `fmap` sequence (repeat [1,2::Int])
<< Does not terminate. >>
* Main> reify (take 3 $ repeat' (reflect [1,2::Int]))
[[1,1,1],[1,1,2],[1,2,1],[1,2,2],[2,1,1],[2,1,2],[2,2,1],[2,2,2]]
* Main> fst `fmap` liftM2 (,) [1,2::Int] [3,4::Int]
[1,1,2,2]
* Main> reify (fst (reflect [1,2::Int], reflect [3,4::Int]))
[1,2]
* Main> reify (fst $!! (reflect [1,2::Int], reflect [3,4::Int]))
[1,1,2,2]
```

The last expression shows that we can easily revert to the eager version by adding appropriate strictness annotations.

1.6.2 Filtering Permutations

As a typical problem where the lazy behavior of our implementation is advantageous, we consider a small combinatorial example: Find all permutations of

```
#!latex
$(1,2,4,...,2^{n-1})$
```

such that all the sums of the initial sequences of the permutations are primes.

```
#!syntax haskell
-- NB. This section's example code can be found in the files Perms.*.
-- _very_ simple primality test.
isPrime :: Int -> Bool
isPrime n = n >= 2 && all (\k -> n `mod` k /= 0)
 (takeWhile (\k -> k*k <= n) $ 2:[3,5..])

-- check if all the initial sums are primes.
goodPerm :: [Int] -> Bool
goodPerm xs = all isPrime (scanl1 (+) xs)
```

If we want to solve the problem in Haskell, we need to make a big compromise: Either we take the easy road and generate a list of the permutations and then

`filter`

the good ones, which is unfortunately very slow because *all*

permutations must be checked even if it already turns out after inspecting a few list elements that no permutation starting this way can have the property.

Alternatively, we can hand-optimize the algorithm by performing the construction of the permutation step-wise and interleaving the primality checks appropriately. In our example, this is not really hard and the list monad is a great help, but it feels low-level, error-prone and lacks modularity. We would like the declarativity of the first approach while retaining the speed improvements the lazy checking provides.

So, should we to switch to another language? An obvious candidate is curry #ref12 12, a lazily evaluated hybrid functional-logic language with a very Haskell-like syntax and feel. Curry allows nondeterministic functions to be written by simply declaring the function multiple times; however, the nondeterminacy cannot be expressed on the type level. Using monadic reflection, we can do something very similar as follows.

```
#!syntax haskell
-- nondeterministic choice
(?) :: DeepSeq a => Monadic [] a -> Monadic [] a -> Monadic [] a
x ? y = reflect (reify x `mplus` reify y)

-- nondeterministically select a permutation
permute :: [Int] -> Monadic [] [Int]
permute [] = []
permute xs = y: permute ys where
 y::Int; ys::[Int]
 (y,ys) = select xs

select :: [Int] -> Monadic [] (Int,[Int])
select [] = reflect []
select (x:xs) = (x,xs) ? second (x:) (select xs) where
 -- a special case of Control.Arrow.second
 second f (x,y) = (x,f y)
```

Now we only need to ensure that the computation fails when the permutation does not have the desired property.

```
#!syntax haskell
```

```
solve :: Int -> Monadic [] [Int]
solve n = if goodPerm xs then xs else reflect [] where
 xs :: [Int]
 xs = permute $ map (2^) [0..n-1]
* Main> reify (solve 17)
[[2,1,4,1024,512,16,8,65536,128,4096,32,16384,32768,256,8192,64,2048],
 [2,1,4,1024,512,16,2048,16384,8192,65536,32768,64,32,256,128,4096,8]]
```

The relative performance of the different approaches is not surprising: The manual Haskell solution (GHC) is the fastest, the Curry solution (Muenster Curry) is about six times slower while the solution using monadic reflection is another four times slower (and gets slightly worse for larger values of

`n`

), since a lot of recomputation is implied by the way

`shift`

and

`reset`

are implemented. Finally, the naÔve solution would probably take

years to finish.

1.7 Further Ideas

This section discusses some further directions in which the ideas of this article might be extended.

1.7.1 Denotational Semantics

The relationship between laziness and direct-style continuation effects, despite often following the intuition, needs some further clarification. For that purpose, I wrote two interpreters of a simple untyped combinator language, which use a continuation-like monad and the monadic reflection library, respectively. They can be checked for coincidence using !QuickCheck tests generating type-checking expressions for the language. The monad

the interpreter is built upon is an

`ST`

monad augmented with continuations of answer type

`Int`

using the

`ContT`

transformer.

```
#!syntax haskell
newtype Eval s a
 = Eval { runEval :: ContT Int (ST s) a }
 deriving (Functor, Monad)
```

The interpreter maps the source language's expressions into the following universal type.

```
#!syntax haskell
```

```
type U s = Eval s (Ref s `Either` U' s)

data U' s
 = Int { runInt :: Int }
 | Fun { runFun :: U s -> U s }
 | List { runList :: Maybe (U s, U s) }

newtype Ref s = Ref { unRef :: STRef s (U' s `Either` U s) }
```

So an
`U s`
is either a reference or a value of type
`U' s`
; references either point to a thunk of type
`U s`
or to an evaluated value of type
`U' s`
. Laziness is provided by two functions of the following types.

```
#!syntax haskell
-- Delays a computation
delay :: U s -> U s
-- Force evaluation of a reference to a normal form.
force :: U s -> Eval s (U' s)
```

Details can be found in the [attachment:Reflection.tar.gz tarball] provided with this article. The distribution also contains two interpreters for a strict version of the language, which can be more straightforwardly implemented using the plain continuation monad and, in case of the direct-style interpreter, some strictness annotations.

1.7.2 A Lightweight Notation for Monads

Haskell's do-notation is often criticized being too verbose, especially for commutative monads; and the process of transforming pure functions into monadic style because some (possibly deeply nested) function needs some effects is tedious and error-prone.

GHC already has special support for the (commutative) reader monad, through implicit parameters. This special rÙle of the reader monad might be justified by additional properties this monad has, for example that there are

isomorphisms of type
`m (a -> b) -> a -> m b`
and
`m (a, b) -> (m a, m b)`
whose inverses are given by
`\f x -> f `ap` return x`
and
`liftM2 (,)`
, respectively.

Also, special tools #ref13 13 are being developed that automatically transform a function from direct into monadic style, but this process requires arbitrary decisions where to apply effects, e.g. it is unclear if

a function of type
`Int -> Bool`
should be monadified to a function of type
`Monad m => m Int -> m Bool`
or
`Monad m => Int -> m Bool`
, as

both make sense in different circumstances.

As we showed in this article, Haskell's type system is almost ready to express these differences on the type level; the only remaining problem is that forall-hoisting [6] changes the meaning of expressions. On the other hand, because of the interaction with laziness, keeping the semantics of the library described in this article would result in a rather complicated translation, as we saw in the last section. In order to get rid of this obscurity, one might imagine a type-directed translation which translates (pseudo-code)

```
#!syntax haskell
reflect :: m a -> (<m> => a)
reify :: Monad m => (<m> => a) -> m a

foo :: <[]> => Int
foo = reflect [0,2] + reflect [0,1]

bar :: [Int]
bar = reify foo
```

more strictly into

```
#!syntax haskell
foo :: [Int]
foo = (+) `fmap` [0,2] `ap` [0,1]

bar :: [Int]
bar = foo
```

However, this contradicts Haskell's philosophy to make invocation of effects as explicit as possible, and would probably be considered an "underkill". Moreover, it would require a decent solution to the monomorphism restriction problem.

1.8 Conclusion

Do not take this too seriously: Our code heavily relies on unsafe and experimental features; time and space usage are increased by the suboptimal encoding of continuations and the recomputations; and the number of supported

monads is limited by the
`DeepSeq`
requirement.

However, we provided a framework with strong static guarantees in which it is

easy to experiment with the unfamiliar

`shift`
and
`reset`
operators,

and we learned that GHC Haskell's type system goes well beyond Hindley-Milner and it is almost ready for an impure language where effects are declared explicitly on the type level.

More importantly, it is great fun to abuse just about every unsafe feature of (GHC) Haskell, to create an impure sublanguage with monadic effects.

1.9 Acknowledgments

I would like to thank the GHC team for this great compiler with its many fascinating extensions.

I also want to thank Peter Eriksen, Cale Gibbard and Don Stewart for proof-reading the article and their valuable suggestions, as well as Brandon Moore and Autrijus Tang for their advice on the references.

1.10 References

Anchor(ref1) [1] Olivier Danvy and Andrzej Filinski. "A Functional Abstraction of Typed Contexts". *DIKU. DIKU Rapport 89/12*. July 1989. Available online: http://www.daimi.au.dk/~danvy/Papers/fatc.ps.gz

Anchor(ref2) [2] Chung-chieh Shan. "Shift to Control". *2004 Scheme Workshop*. September 2004. Available online: http://repository.readscheme.org/ftp/papers/sw2004/shan.pdf

Anchor(ref3) [3] R. Kent Dybvig, Simon Peyton-Jones, and Amr Sabry. "A Monadic Framework for Subcontinuations". February 2005. Available online: http://www.cs.indiana.edu/~sabry/papers/monadicSubcont.ps

Anchor(ref4) [4] Andrzej Filinski. Representing monads. *In Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on* Principles of Programming Languages, Portland, Oregon, pages 446--457. Available online: http://citeseer.ist.psu.edu/filinski94representing.html

Anchor(ref5) [5] Philip Wadler. "The essence of functional programming". *Invited talk, 19'th Symposium on Principles of Programming Languages, ACM* Press. *January 1992. Available online:* http://homepages.inf.ed.ac.uk/wadler/papers/essence/essence.ps

Anchor(ref6) [6] The GHC Team. "The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.4". BR Linear Implicit Parameters: http://haskell.org/ghc/docs/6.4/html/users_guide/type-extensions.html#implicit-parameters BR Implicit Parameters: http://haskell.org/ghc/docs/6.4/html/users_guide/type-extensions.html#linear-implicit-parameters BR Forall-Hoisting: http://haskell.org/ghc/docs/latest/html/users_guide/type-extensions.html#hoist

Anchor(ref7) [7] Koen Claessen and John Hughes. "!QuickCheck: An Automatic Testing Tool for Haskell". http://www.cs.chalmers.se/~rjmh/QuickCheck/

Anchor(ref8) [8] Simon Peyton Jones. "Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell". *In "Engineering theories of software construction, ed Tony Hoare, Manfred* Broy, Ralf Steinbruggen, IOS Press, ISBN 1 58603 1724, 2001, pp47-96. Available online: http://research.microsoft.com/Users/simonpj/papers/marktoberdorf/mark.pdf

Anchor(ref9) [9] Dean Herington. "Enforcing Strict Evaluation". Mailing list post. http://www.haskell.org/pipermail/haskell/2001-August/007712.html

Anchor(ref10) [10] Thomas J‰ger "Linear implicit parameters: linearity not enforced". Mailing list post. http://www.haskell.org/pipermail/glasgow-haskell-bugs/2005-March/004838.html

Anchor(ref11) [11] Simon Peyton Jones [editor] "The Revised Haskell Report". 2002. Section, 4.5.5, "The Monomorphism Restriction". http://www.haskell.org/onlinereport/decls.html#sect4.5.5

Anchor(ref12) [12] Michael Hanus [editor] "Curry. An Integrated Functional Logic Language". Available online: http://www.informatik.uni-kiel.de/~mh/curry/papers/report.pdf

Anchor(ref13) [13] "Monadification as a Refactoring". http://www.cs.kent.ac.uk/projects/refactor-fp/Monadification.html