January 2000

A recursive do for Haskell : Design and Implementation

Levent Erkak

John Launchbury

A Recursive do for Haskell: Design and Implementation

Levent Erkök John Launchbury

Oregon Graduate Institute of Science and Technology

Abstract

Certain programs making use of monads need to perform recursion over the values of monadic actions. Although the do-notation of Haskell provides a convenient framework for monadic programming, it lacks the generality to support such recursive bindings. To remedy this problem, we propose an extension to Haskell's do-notation and describe its translation into the core language.

Computing Review Subject Categories: Formal definitions and theory (D.3.1), Language constructs and features (D.3.3).

Keywords: Haskell, monads, recursion, mfix, do-notation.

1 Introduction

Suppose you are designing an interpreter for a language that has `let` expressions for introducing local bindings. Operationally, the expression `let v = e in b` denotes the same value as `b` where `e` is substituted for all free occurrences of the variable `v`. The abstract syntax of your language might include:

```
data Exp = ...
         | Let Var Exp Exp
```

Assuming the language is applicative, the natural choice for implementation would be the environment monad. In this setting, the section of the interpreter that handles `let` expressions might look like:

```
eval (Let v e b) =
       do ev <- eval e
          inExtendedEnv (v, ev) (eval b)
```

where `inExtendedEnv` is a non-proper morphism of the environment monad extending the environment with the binding $\{v \mapsto ev\}$ before passing it on. This approach yields a very satisfactory implementation.

Notice that our `let` bindings are not recursive: The variable `v` is not known in the expression `e`. Consider what happens if we lift this restriction. All we need is a way to extend the environment with the value of `v` when we evaluate the expression `e`. That is, we want to write:

```
eval (Let v e b) =
       do ev <- inExtendedEnv (v, ev) (eval e)
          inExtendedEnv (v, ev) (eval b)
```

Unfortunately this is not valid Haskell: The variable `ev` is undefined when it is referenced in the first generator. The problem is that do-expressions of Haskell suffer from a similar problem that we are trying to solve in our toy language: Just as our tiny language can not express recursive bindings in `let`, the do-notation of Haskell can not express recursion over the values of monadic actions. Here, the function `inExtendedEnv` performs a monadic action (that of extending the environment and passing it on to its second argument), but in doing so it depends on a value that it defines, i.e. the value of the variable `ev`.

Having failed in our naive approach, how can we implement the recursive `let`? Assuming `Val` represents the type of values that our language can produce and the following declaration for the environments:

```
data Env a = Env ([(Var, Val)] -> a)
```

we can write the `Let` case of our interpreter as:

```
eval (Let v e b) =
        Env (\env ->
               let Env f = eval e
                   ev    = f ((v,ev):env)
                   Env g = eval b
               in g ((v,ev):env))
```

Although it works, this solution is quite annoying. First of all, we had to reveal how environments are actually implemented. Worse, our code will only work with that particular implementation: any change in the representation of environments will require change(s) in the interpreter. The code is no longer easy to understand or maintain: Almost all benefits of using a monad based implementation is lost. The failed approach that used recursive bindings in the do-notation had none of these problems.

Fortunately there is a way out. Recently, we have shown that such recursive bindings make sense in a variety of monads satisfying certain requirements [2]. In particular, a certain fixed point operator, called mfix, must be available for the monad in which we want to express recursion over the results of monadic actions. The research reported in our earlier work mainly concentrated on theoretical issues, such as axiomatization of the required recursive behavior, and the demonstration of satisfactory definitions for various monads. It also described a naive translation for a recursive do-notation.

The current paper, on the other hand, concentrates on the issues from a language design point of view. We describe the translation for the new do-notation that might be employed in a real Haskell compiler. It turns out that there

are several design choices involved, and we explain why we prefer the proposed solutions in detail. We hope that this paper will fulfill two purposes: First of all, it will serve as a Haskell proposal for a new semantics for do-expressions. Secondly, it will provide a guideline for Haskell implementors (or thrill seekers), in case the new translation ever gets adopted in Haskell.

2 Recursive monads and the new do-notation

We first recall the definition of recursive monads [2]:

Definition 2.1 *Recursive Monads.* A monad m is recursive if there is a function mfix :: $\forall a.(a \to m\ a) \to m\ a$ satisfying the axioms given in Figure 1.

From a programming point of view, we will need a new semantics for the do-notation that will allow recursive bindings. For clarity, we will call the recursive version "the μdo-notation", and we will write recursive do-expressions using the keyword μdo. (In actual program text, we will use `mdo`.) Whenever we refer to the do-notation, we will mean the currently available notation in Haskell that does not allow recursive bindings.

A μdo-expression is just like an ordinary do-expression, except the variables being bound in generators are visible throughout the entire body (rather than the textually following part). A μdo-expression can be can be naively translated into a do-expression as follows:

$$
\begin{array}{l}
\mu\texttt{do}\ p_1 \leftarrow e_1 \\
\quad \ldots \\
\quad p_n \leftarrow e_n \\
\quad e
\end{array}
\quad\Longrightarrow\quad
\begin{array}{l}
\texttt{mfix}\ (\lambda \tilde{}BV.\ \texttt{do}\ p_1 \leftarrow e_1 \\
\qquad\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad\qquad\quad p_n \leftarrow e_n \\
\qquad\qquad\qquad\qquad\quad v \leftarrow e \\
\qquad\qquad\qquad\qquad\quad \texttt{return}\ BV) \\
\qquad \gg\!= \lambda\ BV.\ \texttt{return}\ v
\end{array}
$$

where BV stands for the k-tuple consisting of all the variables occurring in all the binding patterns $p_1 \ldots p_n$ plus the brand new variable v. The variables occurring in these patterns may not be multiply bound: neither in the same pattern, nor in different patterns.

In this translation, we say that a μdo-expression is well-typed if the underlying monad is recursive and the resulting code is well-typed. We will refer to this translation as the naive translation.

As an example, here is the translation for the interpreter code given in the previous section:

```
eval (Let v e b) =
       mfix (\~(ev, f) ->
              do ev <- inExtendedEnv (v, ev) (eval e)
                 f  <- inExtendedEnv (v, ev) (eval b)
                 return (ev, f))
         >>= \(ev, f) -> return f
```

This translation is well-typed as long as the underlying environment monad is declared to be recursive with a suitable definition of mfix. (Further details for the environment monad and the definition of mfix can be found in [1].)

3 Let bindings

The do-notation of Haskell allows polymorphic let bindings. How can we accomodate them within the μdo-notation? An obvious extension suggests the following translation:

$$
\begin{array}{l}
\mu do\ \ldots 1\ldots \\
\quad \texttt{let}\ p_1 = e_1 \\
\qquad \ldots \\
\quad\ \ p_n = e_n \\
\quad \ldots 2\ldots \\
\quad e
\end{array}
\quad\Longrightarrow\quad
\begin{array}{l}
\texttt{mfix}\ (\lambda\ \tilde{}BV.\ \texttt{do}\ \ldots 1\ldots \\
\qquad\qquad\qquad\quad \texttt{let}\ p_1 = e_1 \\
\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad p_n = e_n \\
\qquad\qquad\qquad\quad \ldots 2\ldots \\
\qquad\qquad\qquad\quad v \leftarrow e \\
\qquad\qquad\qquad\quad \texttt{return}\ BV) \\
\qquad\qquad \gg\!= \lambda\ BV.\ \texttt{return}\ v
\end{array}
$$

The translation is similar to what we had before, except now the variables bound in $p_1 \ldots p_n$ appear in BV as well. The intuition is that a let bound variable will be visible anywhere in a μdo-expression.

Unfortunately, this idea leads to unfortunate translations. Consider the following example:

```
mdo z <- return (f 2 z)
    y <- return (f 'a' y)
    let f x y = x
    return ()
```

which will be translated to:

```
mfix (\(~(z, y, f, v)) ->
         do z <- return (f 2 z)
            y <- return (f 'a' y)
            let f x y = x
            v <- return ()
            return (z, y, f, v))
    >>= \(z, y, f, v) -> return v
```

The translated code will *not* type-check: The function f, which was used polymorphically, has become monomorphic. In fact, the situation is even worse: Referring to the schematic translation above, the let bound variables in patterns $p_1 \ldots p_n$ will be monomorphic in the code section represented by `...1...`, but they will be polymorphic in the code section `...2...` and in expressions $e_1 \ldots e_n$. This is quite bizarre.

One solution to this problem would be to restrict let-bound variables to be visible only in the textually following section of the μdo-expression. This restriction, however, is not in the spirit of μdo in the first place: any defined variable should be visible in the entire body.

A second solution would be to lift let-generators to the top in a μdo-expression. As a concrete example, the above code can be translated to:

```
mfix (\(~(z, y, v)) ->
         do let f x y = x
            z <- return (f 2 z)
            y <- return (f 'a' y)
            v <- return ()
            return (z, y, v))
     >>= \(z, y, v) -> return v
```

This translation will type-check just fine. But, are we justified in moving let-generators to the top? Unfortunately not. Moving let expression to the top might change the termination behavior. Consider the expression:

```
t :: Maybe Int
t = mdo x <- f x
        let f x = Just 4
        return x
```

If we lift let-generators to top, we get:

```
t' :: Maybe Int
t' = mdo let f x = Just 4
         x <- f x
         return x
```

$$\text{mfix}\ (\text{return} \cdot h) = \text{return}\ (\text{fix}\ h) \quad (1)$$

$$\text{mfix}\ (\lambda x.a \gg\!= f\ x) = a \gg\!= \lambda y.\text{mfix}\ (\lambda x.f\ x\ y) \quad (2)$$

$$\text{mfix}\ (\lambda\tilde{}(x,\_).\text{mfix}\ (\lambda\tilde{}(\_,y).f\ (x,y))) = \text{mfix}\ f \quad (3)$$

Figure 1: Axioms for mfix. In axiom 2, x is not free in a.

When run, t yields $\bot$, while t' computes to `Just 4`. The reason is that the introduction of f before the recursive binding provides additional information that is used in the fixed-point computation. Abstractly, moving let-generators around within a μdo-expression corresponds to the parametricity law from [2], namely:

$$\begin{aligned}&\text{mfix}\ (\lambda x.f\ x \gg\!= \text{return} \cdot h)\\ &\quad = \text{mfix}\ (\lambda x.\text{return}\ (h\ x) \gg\!= f) \gg\!= \text{return} \cdot h\end{aligned}$$

This law requires a strict h for equality. Notice that f is not strict in our example.

Although moving let-generators to the top can be viewed as an optimization increasing termination, we refrain from doing so since we can not guarantee that we can do it all the time: Consider a situation where we use recursive bindings and moving let-generators to the top improves termination. If we ever rearrange the expression so that we cannot move the let-generators anymore, (by creating a nested μdo or by just manually converting a let-generator to an equivalent `return` form), the expression may no longer terminate as often as it did before. This is not a particularly desirable situation: A perfectly valid rearrangement of the code should not fail to work just because an optimization no longer applies. For instance, consider t'', defined as:

```
t'' :: Maybe Int
t'' = mdo x <- f x
          f <- return (\x -> Just 4)
          return x
```

Intuitively, both t and t'' should compute the same value. If our translation optimizes t to t', then t would produce `Just 4`, but the computation of t'' will not terminate. Hence, an otherwise correct transformation might cause non-termination.

The solution we adopt is to require let bindings to be monomorphic in a μdo-expression. That is, `let` becomes just a syntactic sugar within μdo, translated as[1]:

```
let p1 = e1                (p1,...,pn) <- return (let p1 = e1
    ...          ⟹                                    ...
    pn = en                                          pn = en
                                                  in (p1, ... pn))
```

A function binding is translated similarly:

```
let f x y = x     ⟹     f <- return (λ x y. x)
```

This idea easily extends to more complicated forms of function definitions as well. For instance:

```
q :: Maybe (Int, Int)
q = mdo let len []     = 0
            len (x:xs) = 1 + len xs
        return (len [1,2,3], len [1,2])
```

can be treated as:

```
q :: Maybe (Int, Int)
q = mdo len <- return (let len []     = 0
                           len (x:xs) = 1 + len xs
                       in len)
        return (len [1,2,3], len [1,2])
```

Notice that this translation guarantees monomorphic use of let-generators. For instance, the translated code will be rejected by the type-checker if the last statement of `q` is changed to:

```
return (len [1,2,3], len "hi")
```

using the function `len` polymorphically.

This approach gives us a uniform and simple design. If a polymorphic let-definition is required, one should use the standard let-expressions of Haskell, rather than the let-generator, which will create its own scope with polymorphic names, as intended. For instance, our very first example should be written as:

```
mdo let f x y = x
    in mdo z <- return (f 2 z)
           y <- return (f 'a' y)
           return ()
```

which makes the intended use of f much more clear. (The only syntactic drawback is the need for an extra level of indentation.)

We expect this restriction to be negligible in practice. Such let-generators in do-expressions are generally used for giving a name to a common pure expression in the code to follow, and such expressions are rarely polymorphic.[2] Given that there is a way to create polymorphic pure values (by using a usual let-expression), we consider that the simplicity of this design far outweighs the generality we might obtain by a much more complicated translation scheme, as we briefly explore in the next section.

3.1 An excursion into types

The problem we have faced with let-generators is hardly new. The main issue boils down to the fact that the usual Hindley-Milner type system is not expressive enough for our purposes. Although all values are first class and we have a notion of parametric polymorphism, the combination of these two ideas is not available: polymorphic values are not first class [3].

[1] Irrefutable and lazy patterns will require special attention in forming the final tuple, as the result will not be valid Haskell. We ignore these issues as they are mere syntactic technicalities.

[2] To see how important polymorphic let-bindings within the do-notation, we have recently polled the Haskell mailing list, the primary discussion medium for discussing Haskell-related issues on the Internet. The consensus was that such polymorphic let-generators are hardly ever used in practice and even if needed, there is always an obvious way to rewrite the expression without using them. We consider this as an indication that the monomorphism restriction is hardly an issue for let-generators. Also, a quick look at the Nofib benchmark suite reveals that polymorphism in let-generators is not an essential tool in practice.

In a more expressive typing scheme, such as System F, we wouldn't have faced these problems. For instance, consider the following (admittedly artificial) μdo-expression:

```
t :: Maybe (Int, Int)
mdo ones <- return (1:ones)
    as   <- return ('a':as)
    l1   <- return (len (take 5 ones))
    l2   <- return (len (take 5 as))
    let len []     = 0
        len (x:xs) = 1 + len xs
    return (l1, l2)
```

When translated, we'll have:

```
mfix (\~(ones, as, l1, l2, len, v) ->
    do ones <- return (1:ones)
       as   <- return ('a':as)
       l1   <- return (len (take 5 ones))
       l2   <- return (len (take 5 as))
       let len []     = 0
           len (x:xs) = 1 + len xs
       v <- return (l1, l2)
       return (ones, as, l1, l2, len, v))
  >>= \(ones, as, l1, l2, len, v) -> return v
```

When loaded into Hugs, we get:

```
ERROR: Instance of Num Char required
```

Since the function `len` is λ-bound, its first use on the list `ones` causes the type-checker to believe that it has type `[Int] -> Int`, and the latter use with type `[Char] -> Int` is simply rejected. What we wanted to say, of course, is:

```
mfix (\~(ones, as, l1, l2,
         len :: forall a. [a] -> Int, v) ->
    do ones <- return (1:ones)
       as   <- return ('a':as)
       l1   <- return (len (take 5 ones))
       l2   <- return (len (take 5 as))
       let len []     = 0
           len (x:xs) = 1 + len xs
       v <- return (l1, l2)
       return (ones, as, l1, l2, len, v))
  >>= \(ones, as, l1, l2, len, v) -> return v
```

But this code is not typeable within the current type system employed by Haskell. (Not even the extensions provided by Hugs is enough to type this application: a tuple can not have a `forall` type element in it.)

Is there a solution within what is currently available in Haskell implementations? Hugs allows data type declarations with polymorphic fields, using the so-called first-class polymorphism idea [3]. Using this extension, we can code the above translation as:

```
data Args = A { ones :: [Int],
                as   :: [Char],
                l1   :: Int,
                l2   :: Int,
                len  :: forall a. [a] -> Int,
                v    :: (Int, Int)
              }

q :: Monad m => Args -> m Args
q a = do ones <- return (1:(ones a))
         as   <- return ('a':(as a))
         l1   <- return ((len a) (take 5 ones))
         l2   <- return ((len a) (take 5 as))
         let len []     = 0
             len (x:xs) = 1 + len xs
         v <- return (l1, l2)
         return (A ones as l1 l2 len v)
```

Notice that we have created a data type, `Args`, with polymorphic fields, and the original function is altered to process values from and out of this new data type.

To test mfix on the function q, we must pick a particular recursive-monad. Here is a little test with the `Maybe` monad:

```
t :: Maybe (Int, Int)
t = do a <- mfix q
       return (v a)
```

Here's the value of t:

```
Main> t
Just (5, 5)
```

Just as we expected. Notice that the polymorphic nature of the fixed-point will be retained. To see this, consider:

```
r = let Just a = mfix q
        f      = len a
    in (f [1,2,3], f "hi")
```

When executed, we get:

```
Main> r
(3,2)
```

showing that *len* is polymorphic.

As we have stated, this solution is not standard Haskell. Even if it was, it is quite questionable whether having polymorphic let-generators is worth all this complication and effort. Notice that we have not described an explicit translation schema in general. The above example is completely hand written and it can be quite painful to actually implement it in a real compiler. At this point, we prefer let-generators within μdo-expressions to be monomorphic, leaving the door open for further study, possibly using more exotic type systems.

4 A tour of μdo

In this section, we consider a number of ideas that will refine our naive translation.

4.1 Recursive variables

The basic idea behind the naive translation is to create a function of all the variables that are bound in a μdo-expression, which will then be handed to the function mfix. If there are k variables that are bound in the μdo-expression, then the function that is passed to mfix will have a $k+1$-tuple as its argument. For instance, consider the following μdo-expression, and the function created by the naive translation for it:

```
                                  λ~(a, b, c, v).
μdo a ← f a                           do a ← f a
    b ← g a          ⟹                   b ← g a
    c ← h a b                            c ← h a b
    e c                                  v ← e c
                                         return (a, b, c, v)
```

Intuitively, there is no need to carry around the variables b and c: They are not used before being bound within the body of the do-expression. That is, the following function will work just fine:

```
λ~(a, v).  do a ← f a
              b ← g a
              c ← h a b
              v ← e c
              return (a, v)
```

(Notice that we can't leave the variable v out: its value will be projected out after the application of mfix.)

This observation yields the first refinement: The k-tuple BV should only contain those variables that are referenced before defined in a μdo-expression.

4.2 Segmentation

Consider the following μdo-expression, which creates two infinite lists (consisting of 1's and 2's respectively), and announces their creation:

```
μdo putStr "forming a list of 1s"
    ones ← return (1:ones)
    putStr "forming a list of 2s"
    twos ← return (2:twos)
    return (ones, twos)
```

Our translation would produce:

```
mfix (λ~(ones, twos, v).
          do putStr "forming a list of 1s"
             ones ← return (1:ones)
             putStr "forming a list of 2s"
             twos ← return (2:twos)
             v ← return (ones, twos)
             return (ones, twos, v))
 >>= λ (ones, twos, v). return v
```

But this translation is quite unsatisfactory: The only recursion we need is in independently computing the lists `ones` and `twos`. From an intuitive point of view, recursion needs only be performed over sections of the code that actually need it. This suggests the following translation:

```
do putStr "forming a list of 1s"
   ones ← μdo ones ← return (1:ones)
              return ones
   putStr "forming a list of 2s"
   twos ← μdo twos ← return (2:twos)
              return twos
   return (ones, twos)
```

where the inner μdo-expressions will further be translated accordingly. This is analogous to an optimization performed by Haskell compilers for compiling ordinary `let` expressions, where the bindings that are mutually dependent are grouped together. In the case of `let`, this brings efficiency (no unnecessary knots need to be tied) and it enhances the polymorphic types of bound variables [5]. In our case, we increase the number of calls to mfix, but each mfix has a smaller piece of code to work on, hence, we might expect a gain in efficiency.

However, there is a deeper reason why we favor this translation. There are cases when the segmentation based translation will produce values while the naive version fails to terminate. As we explained in [2], the segmentation idea corresponds to the right shrinking law, which tells us when we are allowed to shrink the scope of an μdo-expression from the bottom:

$$\begin{aligned}&\text{mfix } (\lambda\tilde{}(x,y).f\ x \gg\!= \lambda z.g\ z \gg\!= \lambda w.\text{return } (z,w))\\ &\qquad \sqsubseteq \quad \text{mfix } f \gg\!= \lambda z.g\ z \gg\!= \lambda w.\text{return } (z,w)\end{aligned}$$

While some monads satisfy right shrinking as an equality, (identity, state, reader, output, etc.), some monads don't (maybe, lists, trees). Hence, performing segmentation will limit the scope of mfix calls to minimal segments, possibly improving the termination behavior. As a concrete example, consider the function:

```
g :: [Int] -> Maybe Int
g [x] = Nothing
g _   = Nothing
```

This function will return `Nothing` if its input can successfully be pattern matched against `[x]`. In particular, it will produce $\bot$ for the input $1 : \bot$. Now consider:

```
mdo xs <- Just (1:xs)
    g xs
```

We would expect the value of this expression to be `Nothing`. Unfortunately, if we do not perform segmentation, the translation will be:

```
mfix (\~(xs, v) -> do xs <- Just (1:xs)
                      v  <- g xs
                      return (xs, v))
     >>= \(xs, v) -> return v
```

which will evaluate to $\bot$. As we have discussed in [2], when the list xs is computed we expect to get the chain $\{\bot,\ 1 : \bot,\ 1 : 1 : \bot,\ \ldots\}$, but applying g to these values before feeding them back to the list producer will produce $\bot$ for the second element, hence short-circuiting the evaluation to $\bot$. If, on the other hand, we perform segmentation, we will get:

```
do xs <- mfix (\~(xs, v) -> do xs <- Just (1:xs)
                               v <- return xs
                               return (xs, v))
           >>= \(xs, v) -> return v
   g xs
```

which will be evaluated to `Nothing` as expected.

Hence, the segmentation idea serves two purposes. First, if we have a huge μdo-expression where only small parts of it have recursive dependencies, the recursive computation will take place only over those parts, rather than over the entire body. Secondly, and somewhat unexpectedly, monadic actions might interfere with values of bindings in unexpected ways, and segmentation will prevent such problems when the interference is not intended.

4.3 Exported variables

The final refinement to the translation is about an optimization based on the observation that not all variables need to be known in an enclosing environment when a segment is formed. For instance, consider:

```
mdo x <- f x y
    y <- g x y
    h x
```

When segments are formed, we get:

```
do (x, y) <- mdo x <- f x y
                 y <- g x y
                 return (x, y)
   h x
```

Obviously, the variable y is not needed to form the result, i.e. the following would suffice:

```
do x <- mdo x <- f x y
            y <- g x y
            return x
   h x
```

Although this is not a significant optimization, the produced code will at least be easier to compile. Furthermore, it reveals more of the dependencies in the input code, which might possibly trigger further optimizations in the later phases of compilation.

5 Type checking μdo-expressions

In this section, we discuss how we can type check μdo-expressions. First, we fix the syntax of μdo-expressions for the purposes of the rest of the paper. We assume the non-terminal e ranges over all Haskell expressions and we extend it with μdo expressions: (The meta-variable p ranges over patterns and `mdo` is a new keyword)

$$
\begin{array}{lcl}
e & ::= & \mu do \\
 & | & \ldots \textit{ other expressions in Haskell} \\
\mu do & ::= & \texttt{mdo}\ bs \\
bs & ::= & s^*\ e \\
s & ::= & p \leftarrow e \\
 & | & e \\
 & | & \texttt{let}\ a\ a^* \\
a & ::= & p = e
\end{array}
$$

where x^* denotes zero or more repetitions of x. The non-terminal s denotes statements, the building blocks of the μdo-notation. Furthermore, the variables bound in patterns within a μdo-expression may not be repeated: neither in different patterns, nor in the same pattern.

As usual, we will assume that a single expression e in a generator position in μdo is interpreted as the generator $\_ \leftarrow e$. To further simplify the matters, we will also assume let-generators are turned into their `return` equivalents as described in Section 3. That is, the non-terminal s simply becomes:

$$
s \quad ::= \quad p \leftarrow e
$$

Once the simplifications described above are made, type checking μdo-expressions is quite straightforward. Let $V(p)$ be the set of variables that are bound by a pattern p. To type "$\mu do\ \{p_i \leftarrow e_i\}\ e$" in a typing environment Γ, we first compute the set $BV = \cup_i V(p_i)$. Each $V(p_i)$ must be distinct, corresponding to the fact that variables can not be repeated in bindings. (We will discuss this in more detail in Section 7.1.) Let

$$\Gamma' = \Gamma + \{v_k : \tau_k \mid v_k \in BV\}$$

where we introduce a fresh τ_k for each bound variable v_k. Now, the typing rule simply becomes:

$$\frac{\Gamma' \vdash e_i : m\ \tau_i \qquad \Gamma' \vdash p_i : \tau_i \qquad \Gamma' \vdash e : m\ \tau}{\Gamma \vdash \mu do\ \{p_i \leftarrow e_i\}\ e : m\ \tau}$$

with the side condition that m must belong to the `MonadRec` class defined as:

```
class Monad m => MonadRec m where
   mfix :: (a -> m a) -> m a
```

6 The translation for μdo-expressions

In this section, we try to formalize the ideas and describe the translation of μdo-expressions into the core language in detail. We start by giving some definitions regarding variables and statements in μdo-expressions.

6.1 Some definitions

In these definitions, we assume that the let-generators are de-sugared into their `return` forms.

Definition 6.1 *Defined variables.* The variables defined by a generator $p \leftarrow e$ are those variables that appear in the pattern p. A sequence of bindings define variables that are defined by any statement in the sequence.

Definition 6.2 *Used variables.* The variables used by a generator $p \leftarrow e$, are those that appear free in e. A sequence of statements use those variables used by any statement in the sequence.

Definition 6.3 *Free variables.* The free variables of a sequence of statements are those that are used but not defined in the sequence.

Definition 6.4 *Recursive variables.* The recursive variables of a sequence of statements are those that are used *before* defined in that segment. Notice that being defined is an essential prerequisite: if the segment does not define that variable, it's not recursive.

Definition 6.5 *Connected statements.* A statement s is connected to a textually following statement s', if any of the following hold:

- s' defines a variable that is used by s,
- s' textually appears in between s and s'', where s is connected to s''.

The second condition can be considered as interval closure. Notice that, unlike a usual let-expression, we cannot reorder the bindings in a μdo-expression: Order does matter in performing side effects. Hence, if two statements are connected, we are to forced package them together with all other statements that textually appear in between them to ensure that the order of effects are preserved.

Definition 6.6 *Segments.* A sequence of statements within a μdo-expression is maximally connected if no statement following the sequence is connected to any statement in the sequence. Maximally connected sequences of statements are called segments. The segments of a μdo-expression can be computed as follows:

- The first statement in the body of an μdo-expression belongs to a segment.
- The furthest statement in the μdo-expression that the first statement is connected to, and all the other statements in between, form the segment. If there is no such statement, then this first statement forms a segment by itself.
- Once a segment is found, the next statement in the body of the μdo-expression (if any) starts a new segment.

Notice that the number of segments are bound by the number of statements in a μdo-expression.

Definition 6.7 *Exported variables of a segment.* The exported variables from a segment are those variables that are defined in the segment and used in any of the textually following segments.

6.2 Translation algorithm

We describe the algorithm step by step using the following schematic running example:

```
μdo <a b> ← <c d>          s0
   <e>    ← <f>            s1
   <g>    ← <h>            s2
   <f>    ← <a>            s3
   <i j>  ← <i e>          s4
   <j g>                   s5
```

where $\texttt{<}v_1 \ldots v_n\texttt{>}$ stands for any pattern binding variables $v_1 \ldots v_n$ on the left hand side of a generator and for any expression freely referencing variables $v_1 \ldots v_n$ on the right hand side. Notice that the actual patterns/expressions are not important for our purposes.

Segmentation Step: Starting with the first statement, form the segments as described in Definition 6.6.

To perform this step, we will need the defined and used variables of each statement. Luckily, for our running example, these sets are obvious:

D_0	=	$\{a, b\}$	U_0	=	$\{c, d\}$
D_1	=	$\{e\}$	U_1	=	$\{f\}$
D_2	=	$\{g\}$	U_2	=	$\{h\}$
D_3	=	$\{f\}$	U_3	=	$\{a\}$
D_4	=	$\{i, j\}$	U_4	=	$\{i, e\}$
D_5	=	$\emptyset$	U_5	=	$\{j, g\}$

Applying the computation given in Definition 6.6, we get four segments: $S_0 = \{s_0\}$, $S_1 = \{s_1, s_2, s_3\}$, $S_2 = \{s_4\}$ and $S_3 = \{s_5\}$.

Analysis step: For each segment do the following: Compute recursive variables of the segment (definition 6.4), call this set R. If R is empty, then this segment does not need fixed-point computation, leave it untouched. If R is not empty, then we will replace this segment with a single μdo expression as follows: First determine the exported variables of this segment (definition 6.7). Let this set be E. Then create the expression

```
return (v1, ..., vk)
```

where $v_1 \ldots v_k$'s are the elements of E. (If E is empty, we'll have the expression `return ()`.) Attach this expression to the end of the segment. Mark this segment as `RECURSIVE` for future processing.

Returning to our example, here are the sets R and E for each segment, notice that we need E only when R is non-empty:

R_0	=	$\emptyset$			
R_1	=	$\{f\}$	E_1	=	$\{e, g\}$
R_2	=	$\{i\}$	E_2	=	$\{j\}$
R_3	=	$\emptyset$			

Since only R_1 and R_2 are non-empty, we need to add a return statement to them for their exported variables, and mark them as `RECURSIVE`. That is, we add the statement `return (e,g)` to S_1 and `return j` to S_2.

Translation step: At this step, we are left with a number of segments, some of which are marked `RECURSIVE` by the previous step. For each marked segment, do the following:

- Create a brand new variable v,
- Modify the final expression (created by the previous step), so that its value will be bound to v,
- Create a tuple corresponding to the R set for this segment, add v to this tuple as well. Call this tuple RT. Also create the tuple corresponding to the set E, call it ET.
- Form the expression:

```
ET ← mfix (λ~RT. do ...
                    ...
                    v ← return ET
                    return RT)
   >>= λ RT. return v
```

Notice that every segment that was marked `RECURSIVE` becomes a single generator. Returning to our example, we create the following generator for segment S_1:

```
(e,g) ← mfix (λ~(f, v). do <e> ← <f>
                           <g> ← <h>
                           <f> ← <a>
                           v   ← return (e, g)
                           return (f, v))
          >>= λ(f, v). return v
```

And for S_2, we create:

```
j ← mfix (λ~(i, v). do <i, j> ← <i, e>
                       v ← return j
                       return (i, v))
          >>= λ(i, v). return v
```

Notice that, if there are no recursive bindings, each segment will contain a single statement, and no segment will be marked `RECURSIVE`. Furthermore, since every μdo-expression is required to have a final expression (which does not bind any variables), the last segment will always be a singleton non-recursive segment containing this final expression. In our example, S_3 is this final segment.

Finalization step: Now, concatenate all segments and form a single do-expression out of them. For our example, we obtain:

```
do <a b> ← <c d>
   (e,g) ← mfix (λ~(f, v). do <e> ← <f>
                              <g> ← <h>
                              <f> ← <a>
                              v   ← return (e, g)
                              return (f, v))
                 >>= λ(f, v). return v
   j     ← mfix (λ~(i, v). do <i, j> ← <i, e>
                              v ← return j
                              return (i, v))
                 >>= λ(i, v). return v
   <j g>
```

Again, if no recursive bindings are present, the algorithm will just leave the input untouched.

De-sugaring step: Now, we are left with nothing but non-recursive do-expressions. We apply the well known translation algorithm (described in [5]) to get rid of the do. The resulting expression is now in the core language.

7 Other issues

In this section, we briefly discuss some issues related to the integration of μdo-notation in Haskell.

7.1 Unified do-notation

Although we have used a separate keyword `mdo`, we do not believe that there is any need for retaining both `do` and `mdo` as distinct constructs. In the same way that Haskell's `let` plays the role of `letrec`, we believe that the do-notation should be changed to capture our translation. The compiler, upon analyzing the body of the `do`, should perform the appropriate translation depending on whether recursion is indeed used.

Of course, this brings along questions of compatibility: Will old do-expressions retain their meaning? The second mfix axiom guarantees us that this is the case [2]. There are two minor incompatibilities, however. The first is mentioned above—let-generators become monomorphic. The second problem is about the current syntax for do-expressions in Haskell, which allow repeated variables in binding patterns. A new binding simply shadows the earlier one. If we allow repetitions in the μdo-notation, however, the translation would not treat them as independent. Furthermore, a repeated variable might change its type in the do-notation, and this will fail to type check for the μdo-notation. More importantly, one might expect that repeated variables will provide a way of constraining the values that they might take in the μdo-notation, which is not what the translation implies. Hence, even if the translation goes through, this might lead to misunderstandings.[3] Therefore, the syntax of the new μdo-notation explicitly prohibits a variable from being repeated in different patterns (repetition within the same pattern is disallowed following the usual Haskell convention).

Luckily, it's not a common practice in the Haskell community to repeat variable names in bindings, and in most cases the type-checker should be helpful in locating any problems. In case the translation goes through, the value of the do-expression will likely be "wrong enough" to alert the programmer.

7.2 A new type class

To accommodate for the recursive-monads a `MonadRec` subclass of should be added to the standard prelude:

```
class Monad m => MonadRec m where
   mfix :: (a -> m a) -> m a
```

with the understanding that the mfix axioms are proof obligations on the user for each `MonadRec` instance declaration. In this way, whenever a do-expression is used recursively, the translation will place a call to the function `mfix`, which places a `MonadRec` instance requirement on the underlying monad. The type system will then be able to warn the users appropriately in case an error is detected. The prelude can also supply the instance declarations for the `Maybe` and `List` monads [2]. The current IO extension libraries contain the mfix functions for the internal `ST` and `IO` monads (the functions `fixST` and `fixIO` respectively). These definitions should also be moved into `MonadRec` instance declarations for the `ST` and `IO` monads.

[3] In a similar vein, it can be argued that repetitions should not have been allowed in the old do-notation either. List comprehensions become especially horrible: `f x = [x | x <- [x..4], x <- [x..8]]` is a confusing (yet legal) Haskell function.

7.3 Generators as final expressions

Recall that the do-notation of Haskell always requires a final expression. We keep the same restriction for μdo-notation. However, it might be argued that we relax this requirement and allow naming the final expressions value, in case it is needed in earlier expressions. For instance, the expression:

```
mdo { x <- Just (take 5 (1:x)) }
```

would produce `Just [1,1,1,1,1]`. Although attractive, we think requiring a final expression is a better choice for two reasons. First of all, the syntax should be similar to the non-monadic version, i.e. the usual `let` expressions of Haskell. (Notice that `let` always requires a final expression.) It's also not clear what `mdo {}` means, i.e. we lack a base case. The other alternative, requiring that there should be at least one generator in a μdo, is less attractive. The second reason comes from an observation about commutative monads: In a commutative monad, it does not matter which order the bindings are performed. In a μdo, however, moving the last generator around will change the meaning of the whole expression, which is not a desirable situation.

On the other hand, if such μdo expressions are found to be very practical in the Haskell community, the syntax can be changed to allow for this possibility. Ignoring other details, our translation for this case will look like:

$$\begin{array}{lcl} \mu\texttt{do}\ p_1 \leftarrow e_1 & & \mu\texttt{do}\ p_1 \leftarrow e_1 \\ \quad \ldots & \Longrightarrow & \quad \ldots \\ \quad p_n \leftarrow e_n & & \quad v@p_n \leftarrow e_n \\ & & \quad \texttt{return}\ v \end{array}$$

Notice the use of the "as" pattern. The translation algorithm will proceed just as before on this new form.

8 Current Status

We have a implemented the naive version of the translation by modifying the source code of the Hugs system. The web page `http://www.cse.ogi.edu/PacSoft/projects/muHugs` contains the software and the downloading instructions along with other research material.

In this simple implementation, no type checking is performed on μdo-expressions, hence occurrences of let expressions are not required to be monomorphic, and the associated typing problems are ignored. Also, let-generators cannot define functions whose definitions span multiple lines (such as the function `len` we have given in Section 3), each line will be treated as starting a new definition. Furthermore, one uses the keyword `mdo` to use the μdo-notation: The use of keyword `do` does not trigger any translation. This implementation also allows naming the final expressions value, as described in the previous section.

We have a prototype implementation of the full translation described in this paper, working on a simple subset Haskell. We plan to integrate the translation into a future version of Hugs.

9 Related Work

As far as the implementation is concerned, the only directly related work we know of took place within the context of the O'Haskell programming language. O'Haskell is a concurrent, object oriented extension of Haskell designed for addressing issues in reactive functional programming [4]. One application of O'Haskell is in programming layered network protocols. Each layer interacts with its predecessor and successor by receiving and passing information in both directions. In order to connect two protocols that have mutual dependencies, one needs a recursive knot-tying operation. Since O'Haskell objects are monadic, recursive monads are employed in establishing connections between objects. To facilitate for this operation, O'Haskell extends the do-notation with a keyword `fix`, whose translation is a simplified version of ours. This extension arose from a practical need in the O'Haskell work and it was not particularly designed to meet a general need.

10 Conclusions

In this paper, we have described how to extend the do-notation of Haskell to allow for recursive bindings using the ideas given in [2]. We have started with a naive translation and refined it using various ideas to obtain a final translation strategy. It is our hope that the μdo-notation will replace the do-notation of Haskell in the future and this work will serve as a guide for Haskell implementors in integrating the new translation into their compilers and interpreters.

11 Acknowledgements

We are thankful to Ross Paterson and Mark P Jones for pointing out the first-class polymorphism idea for handling let-generators and clarifying the issues involved. We are also grateful to the members of the OGI PacSoft Research Group for valuable discussions.

The research reported in this paper is supported by the National Science Foundation (CCR-9970980).

References

[1] Erkök, L., and Launchbury, J. Recursive monadic bindings: Technical development and details. Tech. Rep. CSE-00-011, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, June 2000.

[2] Erkök, L., and Launchbury, J. Recursive monadic bindings. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '00)* (2000, to appear.).

[3] Jones, M. P. First-class polymorphism with type inference. In *Proceedings of the Twenty Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)* (1997).

[4] Nordlander, J. *Reactive Objects and Functional Programming*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1999.

[5] Peyton Jones, S. L., and Hughes, J. (Editors.) Report on the programming language Haskell 98, a non-strict purely-functional programming language. Available at: `http://www.haskell.org/onlinereport`, Feb. 1999.