A Recursive do for Haskell: Design and Implementation

Levent Erkök, John Launchbury

Oregon Graduate Institute
Department of Computer Science and Engineering
20000 NW Walker Rd
Beaverton, OR 97006-1999 USA

Technical Report No. CSE 00-014
August 24, 2000

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

$$\begin{aligned}
\text{mfix}\,(\text{return}\cdot h) &= \text{return}\,(\text{fix}\,h) && (1)\\
\text{mfix}\,(\lambda x.a \gg\!= f\ x) &= a \gg\!= \lambda y.\text{mfix}\,(\lambda x.f\ x\ y) && (2)\\
\text{mfix}\,(\lambda\tilde{}(x,\_).\text{mfix}\,(\lambda\tilde{}(\_,y).f\ (x,y))) &= \text{mfix}\ f && (3)
\end{aligned}$$

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

When run, t yields $\perp$, 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:

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

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]:

$$\begin{array}{lcl}
\texttt{let}\ p_1\ \texttt{=}\ e_1 & & (p_1,\ldots,p_n) \leftarrow \texttt{return (let}\ p_1\ \texttt{=}\ e_1\\
\quad\ldots & \Longrightarrow & \qquad\qquad\qquad\qquad\quad \ldots\\
\quad p_n\ \texttt{=}\ e_n & & \qquad\qquad\qquad\qquad\quad p_n\ \texttt{=}\ e_n\\
 & & \qquad\qquad\qquad\quad \texttt{in}\ (p_1,\ \ldots\ p_n))
\end{array}$$

A function binding is translated similarly:

$$\texttt{let f x y = x} \quad\Longrightarrow\quad \texttt{f} \leftarrow \texttt{return}\ (\lambda\ x\ y.\ x)$$

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])
```

[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.

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].

[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.

```
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:

$$\begin{array}{rl} & \text{mfix}\ (\lambda^{\sim}(x,y).f\ x \gg\!= \lambda z.g\ z \gg\!= \lambda w.\text{return}\ (z,w)) \\ \sqsubseteq & \text{mfix}\ f \gg\!= \lambda z.g\ z \gg\!= \lambda w.\text{return}\ (z,w) \end{array}$$

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

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>               $s_0$
    <e>   ← <f>                 $s_1$
    <g>   ← <h>                 $s_2$
    <f>   ← <a>                 $s_3$
    <i j> ← <i e>               $s_4$
    <j g>                       $s_5$
```

where $\langle v_1 \ldots v_n \rangle$ 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:

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

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 ($v_1$, ..., $v_k$)
```

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:

$$
\begin{array}{lcllcl}
R_0 & = & \emptyset & & & \\
R_1 & = & \{f\} & E_1 & = & \{e, g\} \\
R_2 & = & \{i\} & E_2 & = & \{j\} \\
R_3 & = & \emptyset & & &
\end{array}
$$

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>
```

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.