A Recursive do for Haskell

Levent Erkök John Launchbury

OGI School of Science and Engineering, OHSU

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. In this paper, we describe an enhanced translation schema for the do-notation and its integration into Haskell. The new translation allows variables to be bound recursively, provided the underlying monad comes equipped with an appropriate fixed-point operator.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.3 [**Programming Languages**]: Language Constructs and Features—*control structures, recursion*

General Terms

Languages, Design

Keywords

Haskell, monads, do-notation, value recursion

1 Introduction

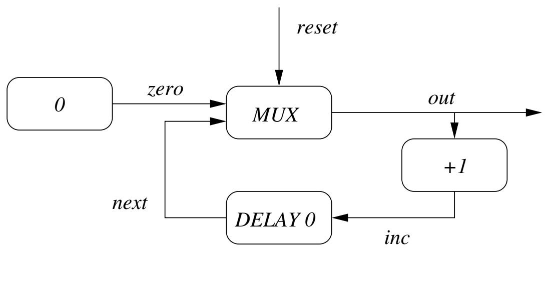
Recursive specifications are ubiquitous in the functional paradigm. While **let** (and **where**) constructs of Haskell provide a convenient notation for expressing recursive bindings in pure computations, the do-notation stops short of providing a similar facility in the monadic world. (Recall that a variable bound in a do-expression is visible only in the *textually following* generators, with no provision for cyclic definitions [18].) The aim of this paper is to bridge this gap, describing a new translation schema for the do-notation that allows variables to be bound recursively.

Let us consider a motivating example to familiarize ourselves with the notion of recursive monadic bindings: hardware modeling using monads. The aim is to model circuit elements such as *and*, *or* gates, multiplexers, registers, delays, etc., which can be combined in various ways to form bigger circuits. One way to solve this problem is to use ordinary lists as signals, and list processing functions as circuit elements, following the model used in the Hawk and Lava hardware description languages [3, 15]. Another alternative is to use monads, abstracting away from any particular representation of signals and circuits. As argued by Launchbury et. al. [14], the monadic approach allows "elaborations", i.e., by simply changing the underlying monad we can obtain different interpretations of the same circuit, a highly desirable feature in a hardware modeling language. For instance, we can generically specify a half-adder simply by:

```
halfAdd       :: Signal Bool → Signal Bool →
                     Circuit (Signal Bool, Signal Bool)
halfAdd i1 i2 = do sum   ← xor i1 i2
                   carry ← and i1 i2
                   return (sum, carry)
```

By picking appropriate instantiations for the *Signal* data type and the *Circuit* monad, (and corresponding implementations of *xor*, and *and* gates), we can simulate, dump a wire-by-wire description, or render this description of the half-adder in another language (such as VHDL) for further processing. This is the strength of the monadic approach.

Unfortunately, the monadic solution has its own shortcomings. Most importantly, it is not clear how to model circuits with feedback loops. Consider how we might model the following counter [14]:



The aim is to increment the value of the *out* line by 1 at each clock tick, dropping down to 0 whenever the *reset* line goes high. Following the half-adder example, we would like to model this circuit as follows:

$$\begin{array}{lll} \mathit{counter} & :: & \mathit{Signal\ Bool} \rightarrow \mathit{Circuit}\ (\mathit{Signal\ Int}) \\ \mathit{counter\ reset} & = & \textbf{do}\ \mathit{next} \leftarrow \mathit{delay}\ 0\ \mathit{inc} \\ & & \quad\ \ \mathit{inc}\ \ \leftarrow \mathit{lift1}\ (+1)\ \mathit{out} \\ & & \quad\ \ \mathit{out}\ \leftarrow \mathit{mux\ reset\ zero\ next} \\ & & \quad\ \ \mathit{zero} \leftarrow \mathit{lift0}\ 0 \\ & & \quad\ \ \mathit{return\ out} \end{array}$$

Note that the description we have given corresponds to the circuit diagram very closely. Alas, this definition is not valid Haskell. The variable *inc* used in the first line is unbound. Of course, we intended *inc* to refer to the signal produced in the next generator, but that information is lost in the do-expression. The variables *inc*, *out*, and *next* depend on each other cyclically, corresponding to the feedback loop in the circuit diagram, making serialization impossible.

As the reader might have already guessed, the solution is to use an appropriate fixed-point operator, tying the recursive knot implied by the cyclic dependency [3, 5, 7, 14]:

$$\begin{array}{l} \mathit{counter} \quad :: \mathit{Signal\ Bool} \rightarrow \mathit{Circuit}\ (\mathit{Signal\ Int}) \\ \mathit{counter\ reset} = \\ \quad \mathit{mfix}\ (\lambda\tilde{}(\mathit{next}, \mathit{inc}, \mathit{out}, \mathit{zero}). \\ \qquad \textbf{do}\ \mathit{next} \leftarrow \mathit{delay}\ 0\ \mathit{inc} \\ \qquad\quad\ \ \mathit{inc}\ \ \leftarrow \mathit{lift1}\ (+1)\ \mathit{out} \\ \qquad\quad\ \ \mathit{out}\ \leftarrow \mathit{mux\ reset\ zero\ next} \\ \qquad\quad\ \ \mathit{zero} \leftarrow \mathit{lift0}\ 0 \\ \qquad\quad\ \ \mathit{return}\ (\mathit{next}, \mathit{inc}, \mathit{out}, \mathit{zero})) \\ \qquad \gg\!= \ \lambda(\mathit{next}, \mathit{inc}, \mathit{out}, \mathit{zero}).\ \mathit{return\ out} \end{array}$$

(A note on notation: Except for some typographical improvements, we will use the Haskell syntax exclusively. For instance, we write Haskell's `\f -> \g -> \x -> (f . g) x` as $\lambda f.\lambda g.\lambda x.(f \cdot g)\ x$. Other changes should be obvious.)

The function *mfix*, known as a *value recursion* operator, performs the required recursive computation. As we will briefly review in the next section, such operators exist for a variety of monads; the most well known examples being the functions *fixIO* and *fixST* for the internal IO and state monads of Haskell [5, 8].

Regarding the use of such looping operators, Claessen writes [3]:

> *...loop combinators are unfortunate because they introduce extra clutter in the code that is hard to motivate.*

That is indeed the case, as evidenced by the difference between the specifications of the half-adder and the counter circuits given above. Compared to the half-adder, the specification of the counter carries a lot of extra baggage, making it hard to understand and maintain. (Note that binders can be arbitrary patterns in general, as in "$\mathit{Just}\ x \leftarrow f\ x$", making the situation even worse.) It is interesting to note that the lack of syntactic support for recursive bindings in the do-notation was one of the main reasons why Hawk and Lava languages abandoned the monadic approach in the first place, despite all its advantages. The translation we will introduce in this paper will handle such recursive definitions automatically, without bothering programmers with the details of the necessary plumbing.

The remainder of this paper is organized as follows: Section 2 reviews value recursion operators [5]. Section 3 describes the basic translation idea, investigating associated problems and design choices. The translation algorithm is given in Section 4. We have already implemented the new translation in the Hugs interpreter, Section 5 reports on the current status of this implementation. Section 6 presents some example uses of the recursive do-notation. Finally, Section 7 summarizes the related work and concludes.

2 Value recursion

The use of *mfix* to tie the recursive knot in monadic computations is similar to the handling of recursive bindings in usual let-expressions. For the sake of clarity, let us use the keyword **letrec** in the following discussion to indicate that a binding can be recursive, and **let** otherwise. In the pure world, we have the following operational equivalence:

$$\begin{array}{l} \quad \textbf{letrec}\ x = e\ \textbf{in}\ e' \\ \equiv \textbf{let}\ x = \mathit{fix}\ (\lambda x.\ e)\ \textbf{in}\ e' \\ \equiv (\lambda x.\ e')\ (\mathit{fix}\ (\lambda x.\ e)) \end{array}$$

Correspondingly, in the monadic world we expect to find a fixed-point operator *mfix* such that:

$$\begin{array}{l} \quad \textbf{mdo}\ \{\ x \leftarrow e;\ e'\ \} \\ \equiv \textbf{do}\ \{\ x \leftarrow \mathit{mfix}\ (\lambda x.\ e);\ e'\ \} \\ \equiv \mathit{mfix}\ (\lambda x.\ e) \gg\!= \ \lambda x.\ e' \end{array}$$

where we use the keyword **mdo**[1] when a binding can be recursive (i.e., x can appear free in e). This is the basic translation idea behind the recursive do-notation.

We use the term *value recursion* to capture this notion of recursion, and the term *value recursion operator* to refer to corresponding fixed-point operators. Briefly, a value recursion operator for a given monad m is a function:

$$\mathit{mfix} :: (\alpha \rightarrow m\ \alpha) \rightarrow m\ \alpha$$

which is subject to the following three semantic properties [5]:

1. *Strictness:* For a function $f :: \tau \rightarrow m\ \tau$, $\mathit{mfix}\ f$ is $\perp$ exactly when f is strict:
$$f\ \perp_\tau = \perp_{m\,\tau} \ \Leftrightarrow\ \mathit{mfix}\ f\ = \perp_{m\,\tau}$$
2. *Purity:* For pure functions, *mfix* should agree with the usual fixed-point operator *fix*:
$$\mathit{mfix}\ (\mathit{return} \cdot h) = \mathit{return}\ (\mathit{fix} \cdot h)$$
where $h :: \tau \rightarrow \tau$.
3. *Left shrinking:* Computations that do not involve the fixed-point variable can be pulled out of the fixed-point loop from the left hand side of a $\gg\!=$:
$$\begin{array}{l} \mathit{mfix}\ (\lambda x.\ q \gg\!= \ \lambda y.\ f\ x\ y) \\ \qquad = q \gg\!= \ \lambda y.\ \mathit{mfix}\ (\lambda x.\ f\ x\ y) \end{array}$$
where x does not appear free in q. The types involved are: $q :: m\ \sigma$, and $f :: \tau \rightarrow \sigma \rightarrow m\ \tau$.

Among these three basic requirements, the left-shrinking law requires special mention. Left-shrinking guarantees that the new translation schema for the do-notation will agree with the already existing translation, in case there are no recursive bindings. That is, considering the basic translation schema given above, the left-shrinking law guarantees that the following two expressions will be equivalent:

$$\begin{array}{ll} \textbf{mdo}\ x \leftarrow A \qquad\qquad & \textbf{do}\ x \leftarrow A \\ \qquad B & \qquad \textbf{mdo}\ B \end{array}$$

[1]The closest we can get to μ**do** using ASCII.

provided the code in block A does not make use of the variable x, or any variable defined in the block B. If B does not have any recursive bindings either, we can push **mdo** further down, eventually eliminating it altogether. Hence, the new translation will produce the same results as do-notation, provided there are no recursive bindings. The left-shrinking law we have given above captures this equivalence formally.

Having given a left-shrinking law, the reader might wonder about a corresponding right-shrinking law as well. That is, we might expect being able to pull out computations that do not make use of the recursion variable from the right hand side of a $\gg\!=$. Symbolically:

$$\begin{aligned} &\mathit{mfix}\ (\lambda(x,\ \_).\ f\ x \gg\!= \ \lambda z.\ g\ z \gg\!= \ \lambda w.\ \mathit{return}\ (z,\ w)) \\ &\quad = \mathit{mfix}\ f \gg\!= \ \lambda z.\ g\ z \gg\!= \ \lambda w.\ \mathit{return}\ (z,\ w) \end{aligned}$$

where $f :: \alpha \to m\ \alpha$, $g :: \alpha \to m\ \beta$. Note that g only depends on the result of f, but not the recursion variable, and is lifted out of the *mfix* loop. Unfortunately, right shrinking law is *not* satisfiable for a wide range of monads, including *list*, *maybe*, *IO*, and the strict state monads of Haskell. (In general, one can show that if the $\gg\!=$ operator of a monad m is strict in its first argument, then no value recursion operator for m can satisfy the right shrinking property [5].) As we will see in Section 3.2, this limitation plays an essential role in the design of the recursive do-notation.

3 The basic translation and design guidelines

For clarity, we will refer to the recursive version of the do-notation as the *mdo-notation*, and continue using the keyword **mdo**. Whenever we refer to the do-notation, we mean the currently available notation in Haskell that does not allow variables to be bound recursively.

Generalizing the idea introduced in the previous section, one might naively translate mdo-expressions as follows:

```
                                  mfix (λ~BV. do p1 ← e1
mdo p1 ← e1                                      ..
    ...                 ⟹                        pn ← en
    pn ← en                                      return BV)
    e                                ≫=  λBV. e
```

where BV stands for the tuple consisting of all variables occurring in patterns $p_1 \ldots p_n$, making sure that bound variables are visible throughout the entire body. The lazy match, obtained by ~, is essential in avoiding strictness problems.

However, there are a number of problems raised by the schema above. First of all, do-expressions in Haskell can use let-generators to introduce polymorphic bindings for pure expressions [18]. It is not clear how such bindings can be integrated into this translation. Similarly, in an ordinary do-expression, variables can be bound repeatedly, later bindings shadowing earlier ones. When bindings can be recursive, shadowing becomes problematic. Also, the use of a single *mfix* to handle recursion over the entire body of an mdo-expression may induce poor termination properties whenever the right-shrinking law fails (see Section 3.2). Intuitively, recursion should only be performed over generators that depend on each other cyclically, leaving the rest untouched. Furthermore, we would like to address these issues within the boundaries of the "syntactic-sugar" approach. That is, the translation should produce only valid (well-formed and well-typed) Haskell code. This approach keeps the extension simple, providing a smooth transition.

To summarize, the basic design guidelines for the mdo-notation are:

- Syntactic agreement with the do-notation: Programmers familiar with the do-notation should have no trouble using the recursive version.
- Semantic agreement with the do-notation: To the extent possible, valid do-expressions should also be valid mdo-expressions, with their meanings preserved.
- Segmentation: Calls to *mfix* should be isolated to recursive segments only, leaving the non-recursive parts out of the fixed-point computation. As we will see, segmentation is essential because extending the scope of recursion can give poorer results for those monads that fail to satisfy the right shrinking law.
- Pure syntactic sugar: The translation should only produce well-formed and well-typed Haskell code.

In the remainder of this section, we address these issues, refining the basic translation scheme as we go along.

3.1 Let generators

The do-notation of Haskell allows let-generators, with the following translation [18]:

```
do let p1 = e1                 let p1 = e1
       ...                         ...
       pn = en        ⟹            pn = en
   stmts                       in do stmts
```

The variables bound in $p_1 \ldots p_n$ can be polymorphically typed. In mdo-expressions, these variables should be visible throughout the entire body, suggesting the translation:

```
                          mfix (λ~BV. do stmts1
mdo stmts1                                let p1 = e1
    let p1 = e1                               ...
        ...                                   pn = en
        pn = en        ⟹                  stmts2
    stmts2                                return BV)
    e                              ≫=  λBV. e
```

where the variables bound in $p_1 \ldots p_n$ will appear in BV as well. Unfortunately, the resulting code is not guaranteed to be well-typed. To illustrate, consider the expression:

```
mdo z ← f 2 y
    y ← f 'a' z
    let f x _ = return x
    return (f y z, f z y)
```

which gets translated into the following ill-typed expression:

```
mfix (λ~(z, y, f).
        do z ← f 2 y
           y ← f 'a' z
           let f x _ = return x
           return (z, y, f))
    ≫= λ(z, y, f). return (f y z, f z y)
```

Since f is λ-bound, it becomes monomorphically typed, making its use at two different types illegal. In fact, the situation is even worse: Referring to the schematic translation above, let-bound variables in patterns $p_1 \dots p_n$ will have monomorphic types over $stmts_1$ and e, while they will retain their polymorphic typings over $stmts_2$ and $e_1 \dots e_n$. This situation is quite bizarre. Unfortunately, there is no easy solution to this problem. Since the tuple BV is λ-bound, the variables that appear in it will be monomorphically typed when we attempt to type check the body of the do-expression and the final expression e.

How should we deal with this problem? Clearly, it is unacceptable to ban let-generators completely because they are quite useful in practice. (Requiring let-bound variables to be visible only in the textually following generators would also be wrong.) An alternative is to go slightly beyond Haskell 98, using records with polymorphically typed fields [12]. Rather than using tuples, we can package the arguments into a record with polymorphic fields, retaining the polymorphic typings of let-bound variables. However, the resulting translation is overly complicated (as we need to perform type inference during the translation), making it extremely hard to formalize and automate [6]. One might also argue that we can go beyond the "syntactic-sugar" approach, i.e., let the translation produce ill-typed code, provided we can come up with special typing rules for mdo-expressions. We will not pursue this option here, however, in order to be able to keep the translation as simple as possible. (We will get back to this point in Section 4.3.)

The solution we adopt is to require let bindings to be monomorphic in mdo-expressions. That is, **let** becomes just a syntactic sugar within **mdo**, translated as:

```
let p1 = e1                BV ← return (let p1 = e1
    ...          ⟹                          ...
    pn = en                                  pn = en
                                          in BV)
```

where BV is the tuple corresponding to the variables bound in $p_1 \dots p_n$. This idea easily extends to more complicated forms of function definitions as well. For instance:

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

translates into:

```
mdo len ← return (let len []     = 0
                      len (x:xs) = 1 + len xs
                  in len)
    return (len [1,2,3], len [])
```

Note that we do not commit to a specific monomorphic type for *len*. As long as *len* is used consistently at a single monomorphic type, the translation will be well-typed.

We expect this restriction to be negligible in practice. Such polymorphic let-generators are hardly ever used in practice, and experience suggests that there is almost always an obvious way to rewrite the required polymorphic bindings using an explicit let-expression, avoiding the whole problem. Therefore, we believe that the simplicity of this design far outweighs any generality that might be obtained by more complicated translation schemas.

Remark 3.1 It might help programmers if monomorphic let-bindings were visually distinguishable from polymorphic ones. In a recent paper, Hughes argues that the syntax of let-expressions should be extended to allow explicit monomorphic bindings, suggesting the use of the symbol `:=` to tell them apart from polymorphic ones [10]. If this idea ever gets adopted in Haskell, let-generators in mdo-expressions can be restricted to use `:=` as well, emphasizing the fact that they will be monomorphically typed.

3.2 Segmentation

Consider the following mdo-expression, which creates two infinite lists consisting of 1's and 2's respectively.

```
mdo putStr "all 1s"
    ones ← return (1 : ones)
    putStr "all 2s"
    twos ← return (2 : twos)
    putStr "done"
```

The translation yields:

```
mfix (λ~(ones, twos).
          do putStr "all 1s"
             ones ← return (1 : ones)
             putStr "all 2s"
             twos ←  return (2 : twos)
             return (ones, twos))
    ≫= λ(ones, twos). putStr "done"
```

The resulting code is quite unsatisfactory. The only recursion we need is in independently computing the lists *ones* and *twos*, suggesting a translation of the form:

```
do putStr "all 1s"
   ones ← mdo ones ← return (1 : ones)
              return ones
   putStr "all 2s"
   twos ← mdo twos ← return (2 : twos)
              return twos
   putStr "done"
```

where the inner mdo-expressions will further be translated accordingly. This process is analogous to the handling of ordinary let-expressions in Haskell, where mutually dependent bindings are grouped together to enhance types of bound variables [18]. In our case, all variables are λ-bound, i.e., monomorphic, so typing is not an issue. However, we still need segmentation to avoid the unwanted interference from trailing computations. As an example, let

```
checkSingle     :: [Int] → IO ()
checkSingle [x] = putStr "singleton"
checkSingle _   = putStr "not−singleton"
```

and consider the following expression:

```
mdo xs ← return (1 : xs)
    checkSingle xs
    return ()
```

The translation yields:

```
mfix (λxs. do xs ← return (1 : xs)
              checkSingle xs
              return xs)
     ≫= λxs. return ()
```

where the call to *mfix* will invoke the Haskell library function *fixIO* :: $(\alpha \to IO\ \alpha) \to IO\ \alpha$, the value recursion operator for the IO monad [8]. Intuitively, when executed we expect this expression to print "`not-singleton`", as the value of *xs* should simply be the infinite list of 1's. Alas, the translation will diverge! The reason is simply that the pattern matching in *checkSingle* is too strict for the computation to proceed, failing the match immediately. However, with segmentation, we will get the code:

```
do xs ← mfix (λxs. return (1 : xs))
   checkSingle xs
   return ()
```

which will happily print "`not-singleton`", avoiding the unintended interference. (Interestingly, if the final "*return* ()" is omitted, the original translation will work as well, since the call to *checkSingle* will be the final expression, automatically pushed outside the *mfix* loop. Just adding "*return* ()" should not change the result, pointing out the need for segmentation.) Note that this problem will arise whenever the right-shrinking law fails, which is the case for many practical monads of interest. (See Section 2 for details.)

3.3 Shadowing

The current syntax of do-expressions allows variable names to be bound repeatedly, later bindings shadowing earlier ones. One could accommodate such bindings in the mdo-notation as well, by automatically renaming them. As a design choice, however, we reject this possibility. Although shadowing might be convenient at times, it is also a constant source of bugs. Since bound variables are visible throughout the entire body in an mdo-expression, allowing repetitions is much more likely to cause confusion. Therefore, we disallow shadowing in mdo-expressions. (This design choice also implies that the scoping rules for mdo-expressions are the same as those for let and where expressions, providing a consistent view of scoping in Haskell's binding constructs, both pure and monadic.)

4 Translation of mdo-expressions

We now present an algorithm to translate mdo-expressions to core Haskell.

4.1 Preliminaries

In the following discussion, we assume that let-generators are already de-sugared into their *return* equivalents, as described in Section 3.1. We use the meta-variable p to range over patterns, v over variables, and e over expressions.

Definition 4.1 (*Defined variables.*) A generator $p \leftarrow e$ defines the variables that appear in the pattern p. If the generator is of the form e, i.e., without any binding patterns, then it defines no variables. An mdo-expression m defines a variable v, if v is defined in a generator of m.

Definition 4.2 (*Used variables.*) A defined variable v is used in a generator $p \leftarrow e$ if v occurs free in e. (And similarly when there is no binding pattern.)

Definition 4.3 (*Recursive variables.*) Let m be an mdo-expression, and v be a used variable of m. Let g be the generator that defines v. The variable v is recursive if it is either used by g itself, or by a generator of m that appears textually before g.

Remark 4.4 Every defined variable comes from a distinct generator, due to the no-repetition requirement. Furthermore, only defined variables can be used, and only used variables can be recursive. That is, for an arbitrary mdo-expression, we have:

$$\text{Recursive Variables} \subseteq \text{Used Variables} \subseteq \text{Defined Variables}$$

Definition 4.5 (*Dependent generators.*) A generator g is dependent on a textually following generator g', if

- g' defines a variable that is used by g,
- or, g' textually appears in between g and g'', where g is dependent on g''.

Remark 4.6 The second condition in the above definition can be considered as interval closure. Note that, unlike a usual let-expression, we cannot reorder the generators: Order does matter in performing side effects. Hence, if a generator is dependent on another, we are forced to package them together with all the generators in between.

Definition 4.7 (*Segments.*) A segment of a given mdo-expression is a minimal sequence of generators such that no generator of the sequence depends on an outside generator. As a special case, although it is not a generator, the final expression in an mdo-expression is considered to form a segment by itself.

Remark 4.8 To compute the segments, it suffices to start with the first generator of an mdo-expression, and search for the last generator that it depends on (Definition 4.5). If such a generator exists, we add all the generators up to and including it to the segment. This process is repeated for each and every one of the generators in the segment, until we cannot add any new generators. Once a segment is found, the very next generator starts a new segment. Note that the number of segments is bounded above by the number of generators in the mdo-expression, plus one for the segment corresponding to the final expression.

Definition 4.9 (*Free variables of a segment.*) Let m be an mdo-expression, v be a defined variable, and s be a segment of m. We say that v is free in s if (i) v appears free in the right hand side of a generator of s, and (ii) v is defined in a segment textually preceding s.

Definition 4.10 (*Exported variables of a segment.*) A variable that is defined in a segment is exported if it is free in any of the textually following segments.

4.2 The translation algorithm

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

```
mdo {a b} ← {c d}        s0
    {e}   ← {f}          s1
    {g}   ← {h}          s2
    {f}   ← {a}          s3
    {i j} ← {i e}        s4
    {j g k}              s5
```

where $\{v_1 \ldots v_n\}$ stands for a pattern that binds the variables $v_1 \ldots v_n$ on the left hand side of a generator, and for an expression whose free variables are $v_1 \ldots v_n$ on the right hand side. Note that the actual patterns or expressions are not important for our purposes. For instance, the generator s_3 uses the variable a, and defines f. Generator s_2 defines g, but does *not* use h, since h is not defined in this expression. For our purposes, it is nothing but a constant. Similar remarks apply to the variables c,d and k as well.

Segmentation step: Starting with the first generator, form the segments as described in Remark 4.8.

To perform this step, we will need the defined (D_i) and used variables (U_i) of each generator s_i. Luckily, for our running example, these sets are obvious:

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

To compute the segments, we start with s_0. Since s_0 does not use any variables, it cannot depend on any further generators, and hence forms a segment by itself. The next generator to consider is s_1, which uses the variable f. Since f is defined by s_3, we have to package everything in between, i.e., s_1, s_2, and s_3 together. Since none of these generators depend on s_4 or s_5, we stop the iteration, forming the second segment. It is not hard to see that s_4 and s_5 form the next two segments by themselves. Therefore, we obtain the following four segments:

$$S_0 = \{s_0\},\ S_1 = \{s_1,s_2,s_3\},\ S_2 = \{s_4\},\ S_3 = \{s_5\}$$

Analysis step: For each segment S_i do the following: For each variable v defined in the segment, determine whether it is recursive (Definition 4.3). Collect all recursive variables of the segment S_i in the set R_i. If R_i is empty, this segment does not need fixed-point computation, leave it untouched. If R_i is not empty, compute the exported variables of the segment, E_i, and mark this segment as *recursive* for future processing. Returning to our example, we have:

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

Since only R_1 and R_2 are non-empty, we mark S_1 and S_2 as recursive; other segments are left untouched. (Note that the last segment can never be recursive.)

Translation step: At this point, 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 the tuples *ET* and *RT* corresponding to the sets E and R. If E is empty, *ET* will be the empty tuple. Create and add a brand new variable v to the tuple *RT* as well.
- Form the generator:

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

  where the dotted lines are filled with the generators of the segment.

Note that segments that are marked recursive by the previous step are turned into a single generator, while non-recursive segments are left untouched.[2] Returning to our example, we create the following generator for 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 the following for S_2:

```
j ← mfix (λ~(i, v). do {i j} ← {i e}
                       v     ← return j
                       return (i, v))
            ≫= λ(i, v). return v
```

The other two segments, S_0 and S_4, are left untouched.

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 k}
```

Remark 4.11 If there are no recursive bindings present to start with, the algorithm we have described will just leave the input untouched (except for replacing the keyword **mdo** by **do**). That is, the left shrinking property is automatically applied by the algorithm to get rid of unnecessary calls to *mfix*. (See Section 2 for details.)

Desugaring step: Now we are left with a non-recursive do-expression, and we can apply the standard translation to replace the **do** with explicit ≫='s, completing the translation [18].

4.3 Type checking mdo-expressions

To accommodate for the overloading of the name *mfix*, we simply add the following type class to Haskell:

```
class Monad m ⇒ MonadRec m where
    mfix :: (α → m α) → m α
```

Intuitively, an mdo-expression is well-typed if its translation produces a well-typed Haskell expression. In order to perform type-inference, a type judgement of the form:

$$\frac{\Gamma' \vdash e_i : m\,\tau_i \qquad \Gamma' \vdash p_i : \tau_i \qquad \Gamma' \vdash e : m\,\tau}{\Gamma \vdash \textbf{mdo}\,\{p_i \leftarrow e_i\}\; e : m\,\tau}$$

[2]Depending on the sets E and R, several other improvements are possible in forming the required generator. For instance, if E is a subset of R, then we do not need a new variable. We skip a detailed discussion of these improvements here, as they are not essential to our discussion.

suffices, with the side condition that m must belong to the *MonadRec* class. In this rule, Γ' is obtained by extending Γ with the variables defined in the given mdo-expression. Each such variable is assigned a monomorphic type variable to begin with. (For simplicity, we assume all generators have the form $p \leftarrow e$.) The only special care is needed in handling let-generators, which can be typed similarly to normal let-expressions. To ensure that variables introduced by let-generators are monomorphic, it suffices to leave out the generalization step in the type inference algorithm for let-bound variables [6, 13].

As we have promised in Section 3.1, let us reconsider the typing of let-generators, aiming to find a solution that would allow polymorphic bindings. In fact, it is arguable that we should have a more liberal scheme, where normal bindings can be polymorphic as well. For instance, there is no reason why the following expression should be ill-typed:

```
poly :: Maybe ([Bool], [Int])          -- ill−typed
poly = do nil ← return []
          return (True : nil, 1 : nil)
```

However, *poly* is not a well-typed Haskell expression, since *nil* is required to be monomorphic. Of course, we cannot allow polymorphic typings arbitrarily, as illustrated by the infamous ML-typing problem [20], coded here in Haskell:

```
do rf ← newSTRef (λx. x)
   writeSTRef rf (λx. x + 1)
   f  ← readSTRef rf
   return (f True)
```

Following the previous example, we might think that *rf* might be assigned the type $\forall\alpha.\ STRef\ s\ (\alpha \rightarrow \alpha)$, which leads to disaster. So, it seems that the *maybe* monad is mild enough that generalization is acceptable, but the state monad is not. It is beyond the scope of this paper to investigate exactly when one might allow generalization, but we conjecture that it is safe to do so in the following two cases:

- For any variable, provided the underlying monad is completely definable in Haskell, and not built on top of one of the internal state or IO monads,
- Or, variables bound by the let-generators, regardless of what the underlying monad is.

Since checking for the first condition seems to be rather expensive, we might settle for allowing generalization in let-bound variables only, which coincides with the treatment of let-generators in the current do-notation. (Such a solution would be similar to ML's value restriction, where only "syntactically distinguishable" values are typed polymorphically [20].) Of course, a more detailed study is needed before such an approach can be adopted. We leave the exploration of this idea for future work.

5 Current status

We have already implemented the new translation fully in the February 2001 release of the Hugs interpreter [11, 19]. We hope to integrate the translation into a future version of GHC [9] as well. In order to avoid any possible confusion, the new translation is triggered only with the keyword **mdo**, and the Hugs interpreter should be started with the `-98` flag. Programs using the mdo-notation should import the module *MonadRec*, which contains the declaration of the *MonadRec* class, and instances of *mfix* for the *maybe*, *list*, *IO* and state monads (both strict and lazy) of Haskell. (See the appendix for a listing of the *MonadRec* module.)

The new translation is relatively straightforward to implement; our implementation in Hugs required around 600 lines of additional C code (including comments), which constitutes about 1% of the whole Hugs source base.

6 Examples

In this section, we will present a couple of examples illustrating the use of the new mdo-notation.

The *repmin* problem is concerned with the replacement of all the numbers in a binary tree by their minimum. The challenge is to do so in a single pass [2, 4]. In 1984, Richard Bird devised a beautiful solution to this problem, exploiting laziness and cyclic definitions:

```
data Tree α = L α | B (Tree α) (Tree α)
              deriving Show

copy             :: Tree Int → Int → (Tree Int, Int)
copy (L a)     m = (L m, a)
copy (B l r) m = let (l′, ml) = copy l m
                     (r′, mr) = copy r m
                 in (B l′ r′, ml ‘min‘ mr)

repmin   :: Tree Int → Tree Int
repmin t = let (t′, m) = copy t m in t′
```

Here's an example run:

```
Main> repmin (B (L 11) (B (L 2) (L 3)))
B (L 2) (B (L 2) (L 2))
```

The single pass solution is achieved by the clever use of recursion in the let-expression of the function *repmin*. By the virtue of the recursive binding, the function *copy* simultaneously computes and replaces all the leaves with *m*, the minimum value in the tree.

Benton and Hyland take the problem one step further [1]. What if we also want to print the values stored in the nodes during this single traversal as well? It is easy to modify *copy* to achieve this effect:

```
copyPrint :: Tree Int → Int → IO (Tree Int, Int)
copyPrint (L a)     m = do print a
                           return (L m, a)
copyPrint (B l r) m = do (l′, ml) ← copyPrint l m
                         (r′, mr) ← copyPrint r m
                         return (B l′ r′, ml ‘min‘ mr)
```

But, it is not clear at all how to modify *repmin* accordingly. Obviously, the attempt:

$$copyPrint\ t\ m \gg\!= \lambda(t', m).\ return\ t'$$

is flawed, since *m* is no longer recursively bound! We need to tie the recursive knot with an appropriate value recursion operator. The mdo-notation comes to the rescue:

```
repminPrint   :: Tree Int → IO (Tree Int)
repminPrint t = mdo (t′, m) ← copyPrint t m
                    return t′
```

In this particular case, the appropriate operator is the one for the IO monad, i.e., the function *fixIO*. The class mechanism will automatically substitute the required instance.

Here is another variation of the *repmin* problem, demonstrating the use of the mdo-notation for the *list* monad. (See the appendix for the corresponding *MonadRec* instance declaration.) Consider the data type:

$$\textbf{data}\ Exp\ =\ C\ Int\ \mid\ A\ Exp\ Exp$$

representing simple arithmetic expressions formed out of integer constants and additions. The problem is to find all possible pair-swaps of a given expression. A swapping is defined to be the exchange of any two constants, not necessarily distinct. (For instance, the only possible swapping of 1 is 1, while that of $1+2$ are $1+2$, $2+1$, $2+1$, and $1+2$. The two $2+1$'s are considered different, corresponding to the swappings of 1–2 and 2–1. It is easy to see that an expression with n constants will have n^2 swappings, one for each pair of constants.) Solving the swappings problem is not a terribly hard task. Here, we present a particularly neat solution, illustrating the use of value recursion for the *list* monad:

$$\begin{array}{lll}
replace & :: & Int \to Exp \to [(Exp, Int)] \\
replace\ x\ (C\ y) & = & [(C\ x,\ y)] \\
replace\ x\ (A\ l\ r) & = & [(A\ l'\ r,\ y) \mid (l',\ y) \leftarrow replace\ x\ l] \\
 & & +\!\!+\ [(A\ l\ r',\ y) \mid (r',\ y) \leftarrow replace\ x\ r]
\end{array}$$

$$\begin{array}{lll}
pairSwaps & :: & Exp \to [Exp] \\
pairSwaps\ e & = & \textbf{mdo}\ (e',\ m) \leftarrow replace\ n\ e \\
 & & \phantom{\textbf{mdo}}\ (e'',\ n) \leftarrow replace\ m\ e' \\
 & & \phantom{\textbf{mdo}}\ return\ e''
\end{array}$$

The call *replace x e* creates copies of *e*, where each copy has one of its constants replaced by *x*. Each replaced constant is returned along with the corresponding copy. (If there are *n* constants in *e*, the call to *replace* will return *n* copies.) For instance:

$$\begin{array}{lll}
replace\ 1\ 2 & \Longrightarrow & [(1,\ 2)] \\
replace\ 1\ (2\ +\ 3) & \Longrightarrow & [(1\ +\ 3,\ 2),\ (2\ +\ 1,\ 3)]
\end{array}$$

The function *pairSwaps* makes two successive calls to *replace*, threading the input expression through. The first call replaces each constant with *n* (yet to be computed), determining the respective values for *m*. The second call completes the swapping by substituting *m*'s, and by computing the values of *n* needed in the first call. Each pairing of *m* and *n* corresponds to a possible swapping. The cyclic dependence between *m* and *n* achieves the required swapping quite neatly.

Here is an example run for the input $(1+2)+3$, using appropriate functions for parsing and printing:

```
Main> display (pairSwaps (parse "(1 + 2) + 3"))
[(1 + 2) + 3, (2 + 1) + 3, (3 + 2) + 1,
 (2 + 1) + 3, (1 + 2) + 3, (1 + 3) + 2,
 (3 + 2) + 1, (1 + 3) + 2, (1 + 2) + 3]
```

Note that the diagonal corresponds to the swappings of the literals with themselves, (i.e., 1–1, 2–2, and 3–3), simply repeating the original expression.

Further example uses of the mdo-notation can be found in the first author's thesis [5, Chapter 9], and online on the web [19].

7 Related work and Conclusions

Predating our work, the need for recursive bindings in the do-notation was also discussed in the framework of the O'Haskell language, a concurrent, object-oriented extension to Haskell [16]. O'Haskell extends the do-notation with a variety of new features. With regard to recursion, O'Haskell provides a special keyword **fix**, providing a way to specify a block of generators with mutual dependencies. The translation for fix-blocks is a simpler version of ours: No segmentation is performed and let-generators are not allowed. The translation seems to allow shadowing, but that appears to be an oversight, rather than a conscious design decision. The addition of the **fix** keyword to the do-notation in O'Haskell arose from practical programming needs; the syntax and the translation was not designed to meet a general need.

Paterson's arrow-notation supports recursive bindings as well, provided the underlying arrow comes equipped with a *loop* operator [17]. Similar to O'Haskell, mutually dependent generators are explicitly marked, using the keyword **rec**. No segmentation is performed on recursive blocks. Currently, let-generators are not supported in the arrow-notation, but the addition of such bindings seems straightforward. We note that all variables become λ-bound after the translation in the arrow-notation, forcing monomorphic types. Hence, regardless of the support for recursive bindings, let-generators will suffer from the monomorphism problem in the arrow-notation.

Recalling our design goals for the mdo-notation, we can conclude that our translation fulfills its purpose. To review briefly, we have aimed for syntactic and semantic agreement with the do-notation, segmentation for grouping minimally dependent sequences of statements together, and preservation of the syntactic-sugar status. Our translation achieves all these goals, except for syntactic agreement for a relatively small set of do-expressions. Since let-generators become monomorphic and shadowing is no longer allowed, any do-expression using these features will be rejected. However, we believe that neither of these restrictions will cause serious problems in practice. Also, if desired, the typing problem might be remedied by devising a solution along the lines we have described in Section 4.3.

It is our belief that Haskell should have just one version of the do-notation. Just like let-expressions, do-expressions should be capable of expressing both recursive and non-recursive bindings. (The type system will insist on the *MonadRec* instance only when recursive bindings are used.) However, such a change will potentially break existing programs, due to the minor incompatibilities mentioned above. Therefore, a separate notation (using the keyword **mdo**) has been adopted for the time being, possibly switching to the new translation in a future version of the Haskell standard.

8 Acknowledgements

We are thankful to Jeff Lewis of Galois Connections for helping out with the implementation of the mdo-notation. Jeff not only took time to explain us the internals of Hugs, but also helped with the actual coding process as well. We are also grateful to Mark P. Jones and other 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).

9 References

[1] Benton, N., and Hyland, M. Traced premonoidal categories (Extended Abstract). In *Fixed Points in Computer Science Workshop, FICS'02* (2002).

[2] Bird, R. S. Using circular programs to eliminate multiple traversals of data. *Acta Informatica 21* (1984), 239–250.

[3] Claessen, K. *Embedded languages for describing and verifying hardware*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 2001.

[4] de Moor, O. An exercise in polytypic program derivation: *repmin*. Unpublished manuscript. URL: `web.comlab.ox.ac.uk/oucl/work/oege.demoor/pubs.htm`, 1996.

[5] Erkök, L. *Value recursion in monadic computations*. PhD thesis, OGI School of Science and Engineering, OHSU, Portland, Oregon, 2002.

[6] Erkök, L., and Launchbury, J. A recursive `do` for Haskell: Design and implementation. Tech. Rep. CSE-00-014, Oregon Graduate Institute School of Science and Engineering, Department of CSE, OHSU, August 2000.

[7] Erkök, L., and Launchbury, J. Recursive monadic bindings. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00* (September 2000), ACM Press, pp. 174–185.

[8] Erkök, L., Launchbury, J., and Moran, A. Semantics of *fixIO*. In *Fixed Points in Computer Science Workshop, FICS'01* (September 2001).

[9] GHC web page. URL: `www.haskell.org/ghc`.

[10] Hughes, J. Global variables in Haskell. Draft paper. URL: `www.cs.chalmers.se/~rjmh/Globals.ps`.

[11] Hugs (Haskell Users Gofer System) web page. URL: `www.haskell.org/hugs`.

[12] 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).

[13] Jones, M. P. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop* (1999).

[14] Launchbury, J., Lewis, J., and Cook, B. On embedding a microarchitectural design language within Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99)* (1999), pp. 60–69.

[15] Matthews, J., Cook, B., and Launchbury, J. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages* (1998), IEEE Computer Society Press, pp. 90–101.

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

[17] Paterson, R. A new notation for arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming, ICFP'01, Florence, Italy* (September 2001), ACM Press, pp. 229–240.

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

[19] Recursive monadic bindings web page. URL: `www.cse.ogi.edu/PacSoft/projects/rmb`.

[20] Wright, A. K. Simple imperative polymorphism. *Lisp and Symbolic Computation 8*, 4 (1995), 343–355.

Appendix

The following listing contains the module *MonadRec*, containing the *MonadRec* class declaration and appropriate instances for various monads.

```
--------------------------------------------------------------------------
-- MonadRec.hs
--
-- Suitable for use with "recursive monadic bindings"
--       http://www.cse.ogi.edu/PacSoft/projects/rmb
--------------------------------------------------------------------------

module MonadRec (MonadRec(mfix)) where

import qualified LazyST
import qualified ST
import IOExts

-- The MonadRec class definition
class Monad m ⇒ MonadRec m where
    mfix :: (α → m α) → m α

-- Instances of MonadRec:

-- Maybe:
instance MonadRec Maybe where
    mfix f = let a = f (unJust a) in a
             where unJust (Just x) = x

-- List:
instance MonadRec [] where
    mfix f = case fix (f · head) of
                  []     → []
                  (x:_)  → x : mfix (tail · f)
          where fix   :: (α → α) → α
                fix f = let a = f a in a

-- IO:
instance MonadRec IO where
    mfix = fixIO

-- Lazy State:
instance MonadRec (LazyST.ST s) where
    mfix = LazyST.fixST

-- Strict State:
instance MonadRec (ST.ST s) where
    mfix = ST.fixST
```