Reification in Haskell

Bernard Pope and Lee Naish

The University of Melbourne, Victoria 3010, Australia,
{bjpop,lee}@cs.mu.oz.au,
WWW home page: http://www.cs.mu.oz.au/{~bjpop,~lee}

Abstract. In this paper we investigate a limited form of reification in Haskell. We incrementally develop a meta-representation with a view to expressing the lazy evaluation and cyclic sharing of first-order values. We implement most of our facility within Haskell for portability and make extensive use of Haskell's type classes. We show how this meta-representation can be derived for algebraic data types, and illustrate its use through a generic printing facility and a re-evaluation scheme. We also briefly explore a means for converting meta-representations back into their object level values.

1 Introduction

What is reflection? In the most general (philosophical or cognitive) sense it is the process of self-awareness. To quote Cantwell Smith [12], in his seminal work on reflection and semantics in Lisp:

> [Reflection is] the ability of an agent to reason not only introspectively, about itself and internal thought processes, but also externally, about its behaviour and situation in the world.

How then do these abstract notions of reason, thought and behaviour relate to formal languages and computation? Often we use the term *meta-programming* to denote the practice of reflection in programming languages. Roughly speaking, meta-programming is the process of writing programs that manipulate (parts of) other programs. The program that is being manipulated is called the *object program* and the program that does the manipulation is called the *meta-program*. Given a certain level of expressiveness, it may be possible to write the object program and the meta-program in the same language, such a language is said to be reflective. Meta-programming allows a number of interesting programming techniques, including self interpretation, generic programming and execution monitoring (for example debugging and tracing).

In this paper we focus on one particular meta-programming problem: how to represent first-order data generated during the execution of a Haskell program. Our requirements are twofold, and unfortunately somewhat at odds with each other. On the one hand we want a representation of data that is easily related to the syntactic description from the program text. On the other hand we want to represent certain operational characteristics of our data which do not have a syntactic counterpart, a necessary requirement because we are representing data as part of a *computation*. Our resulting meta-representation models a typical semantic domain of first-order values, extended to describe the computational notions of partial evaluation and sharing.

To put this in perspective, consider the generic print function: one that can provide a meaningful print representation of any value from the object language. By meaningful, we mean something which unambiguously denotes the underlying semantic content of the value being printed. Such a function is particularly useful for diagnostic applications (such as debugging) where arbitrary values from the running program may need to be displayed. One approach to implementing this function is to derive it directly from the internal (machine oriented) representation of values at runtime. For languages like Haskell this has several detractors. For example, due to the abstract nature of the language, the internal representation of values may be hard to reconcile with the syntactic view that the programmer is most likely to have in mind. Indeed the internal representation of values is not directly visible from within Haskell, and so the function must be implemented as a primitive. This reduces the portability of the function since it is reliant on the way a given language implementation represents values during execution. An alternative, and more promising approach, is to implement such a function within the language itself, and derive the the representation of values from their syntactic description.

In this paper we incrementally improve a simple meta-representation for first-order values which captures certain operational characteristics of those values. We show how type classes can be used to provide a generic means for converting object level values into the meta-representation, and describe two impure extensions to the language that facilitate the representation of partial evaluation and cyclic sharing. Using declarative debugging as a motivating example, we describe how our meta-representation can be used as the basis for a generic printing facility and a re-evaluation strategy. We consider automating some of the work by means of class instance derivation. We also consider an extension of the meta-representation that allows for meta-values to be mapped safely into their corresponding object level value. Finally we consider some related work in the field, and conclude. The paper assumes considerable familiarity with non-strict functional programming languages, in particular Haskell, and type classes.

2 A simple meta-representation

We begin with a simple representation of first-order values. We will use this as a basis for more useful and expressive representations later in the paper. We call our meta-representation of values *Meta*, and implement the representation with the following Haskell data type:

```
data Meta = Apply String [Meta]
```

The *Meta* type is simply an abstract structure for certain types of object values: applications of constructors to zero or more argument values (nullary constructors simply have no arguments). The *Meta* type provides us with a homogeneous representation which will prove to be useful when we want to write generic functions over values from many different types (for example the print facility mentioned in the introduction). For clarity of presentation we use strings of characters to represent the names of data constructors, although any enumerable type would be sufficient. We assume applications will be saturated, and thus encode the arguments to a constructor in a list for convenience. To simplify the description of nullary constructors we provide the function *nullCon*:

$$nullCon\ x\ =\ Apply\ x\ [\,]$$

A table of Haskell values and their resulting meta-representations is given below:

expression	meta-representation
True	*nullCon* "True"
Just False	*Apply* "Just" [*nullCon* "False"]
(*False*, 3.5 :: *Float*)	*Apply* "," [*nullCon* "False", *nullCon* "3.5"]
[( )]	*Apply* ":" [*nullCon* "( )", *nullCon* "[ ]"]

What values can we capture with the *Meta* type? The *Meta* type can represent any built in type that is composed of values of nullary constructors (such as *Int*s), and product types, such as tuples and those introduced by the **data** keyword. All the arguments to data constructors must be first-order. Functions are not represented. Deriving canonical representations of functions which are also meaningful to the programmer is difficult. Our method for deriving representations of first-order values is based on type classes and pattern matching. Type classes and pattern matching are not sufficiently expressive to deconstruct arbitrary functional values. The representation of functions also introduces the the issue of representing variables, which further complicate matters due to the scoping of identifiers in the language. If we implement a meta-syntax for functions based on lambda abstractions, then we lose the object-level identifiers that programmers assign to the definition of functions, which are essential for some applications such as debugging. Our difficulty with functions also precludes the deconstruction of partially applied data-constructors, even though it would appear that the meta-representation could accommodate them. A method for obtaining meaningful print-representations of functional values for the purposes of debugging is described in Pope and Naish [10]. In that work a type-directed program specialisation is performed to distinguish between different instances of higher-order polymorphic functions. The transformation wraps higher-order arguments in a data structure that contains both the original functional value and a string representation of that value based on the program text. Our *Meta* type is similar to the semantic domain of Meta-ML [11]. Note that Meta-ML also encodes a syntactic domain which includes a representation of lambda abstractions and variables. Meta-ML provides a more expressive form of meta-programming than the one described in this paper. Much of the extra power of Meta-ML is derived from the provision of a meta-interpreter, which is an additional level of complexity that we would rather avoid.

3 Reification: converting values into meta-representations

Reification is the materialization of an abstract concept. In the context of meta-programming, this translates to the process of mapping values from the object program (which are abstract) into structures in the meta-program (which are concrete). In practical terms we want a function that can convert first-order values of arbitrary type into a value of type *Meta*. We could proceed to define such a function for each type in our program (for example: '$reifyInt :: Int \rightarrow Meta$', '$reifyBool :: Bool \rightarrow Meta$', and so on). Aside from being a tedious task, this approach will encounter difficulties when we consider polymorphic data types. We would prefer a function which is polymorphic in the type of its argument. Parametric polymorphism will not suffice. Consider a function '$reify :: a \rightarrow Meta$'. This type scheme disallows any distinction to be made based on the structure of the argument, which is problematic since we require such structural information to convert values reliably into the meta-representation. Type classes provide a solution by allowing a polymorphic function whose implentation is dependent on the type at which it is applied dynamically.

We introduce a type class *Reify* which collects all the Haskell types that can be converted to the *Meta* type:

```
class Reify a where
  reify :: a → Meta
```

An example instance declaration for the $[a]$ type:

```
instance (Reify a) ⇒ Reify [a] where
  reify x = case x of
              [ ]      → nullCon "[ ]"
              (y : ys) → Apply ":" [reify y, reify ys]
```

The existence of the type class does not guarantee that our resulting meta-representations will faithfully reflect the underlying values which they represent. For this we need one more ingredient: automatic instance derivation. We do not want to place the onus of instantiating the type class on the programmer: they can get it wrong. Instead we want the compiler to automatically generate instance declarations based on the syntactic description of types introduced with the **data** keyword. This facility is already available for some standard classes from the Haskell Prelude, we show how this can easily be extended in section 7. For built in types such as *Int*, instance declarations must be provided by hand.

4 Non-strict evaluation

So far we have described a kind of reification by evaluation. Whereby the evaluation of our meta-representation forces the evaluation of the underlying value. In more abstract terms, if we denote diverging computations with the value $\perp$, then: $reify \perp = \perp$. In a strict language this would not be an issue, since the argument expression to *reify* would be fully evaluated before the body of *reify* is executed. In a non-strict language such as Haskell, it may be the case that the expression to *reify* is unevaluated, or partly evaluated. There are some circumstances when we would like to reify a value without effecting its state of evaluation, for example post-mortem debugging environments such as HOOD [3] and Buddha [9, 8]. To capture the fact that some parts of a value may be unevaluated when it is reified, we extend our definition of the *Meta* type with the constructor *Uneval* to close the meta-representation at unevaluated branches.

What do we mean by non-strict semantics? Is Haskell non-strict or lazy? The language definition says that Haskell is non-strict [5]. This is a weaker requirement than saying that the language is lazy: lazy evaluation is an implementation of non-strict semantics, however others are possible, for example lenient evaluation which is described in Tremblay [13].[1] For the purposes of this paper we assume a lazy evaluation semantics for Haskell, in the spirit of Launchbury [7], which is representative of the current Haskell implementations, though not demanded by the language definition.

Consider the program:

```
main = do
         let list  = map id [1 .. ]
         let len   = length (take 3 list)
         print len
```

At the end of the call to *print* the variable *list* will be only partially evaluated. We will represent its state of evaluation with the meta-representation:

```
Apply ":" [Uneval, Apply ":" [Uneval, Apply ":" [Uneval, Uneval] ] ]
```

In the above example the values of the expressions denoted by *Uneval* were not needed to compute the final result, and hence remain unevaluated under a lazy evaluation scheme.

We need to modify the *reify* function so that it can convert a value into a meta-representation, but avoid expanding unevaluated components of the value. To do this we need to know when a value has been computed. Under the lazy semantics of Launchbury [7], an expression is evaluated if it is in *weak head normal form* (whnf). There are two disjoint situations when a value is considered in whnf: when it is a function; or when the outermost data constructor of the value has been computed. Since we are only interested in first-order values, the second of the two cases is all we need to consider.

The underlying state of evaluation of a value cannot be determined from within pure Haskell code. Therefore we introduce a primitive into the language with the following name and type:

```
whnf :: a → Bool
```

[1] Tremblay [13] classifies Haskell as a lazy language despite the title of the language report: *Haskell 98: A Non-strict, Purely Functional Language.*

The function is impure (not referentially transparent) because its value depends on the context of when it is applied.[2] We modify the definition of *reify* such that all non-whnf components of values are represented by the *Uneval* constructor. An example instance declaration for the $[a]$ type:

```
instance (Reify a) ⇒ Reify [a] where
  reify x
     = if whnf x then
          case x of
            [ ] → nullCon "[ ]"
            (y : ys) → Apply ":" [reify y, reify ys]
       else
          Uneval
```

The use of *whnf* causes the *reify* function to be unsafe. In practice we would provide both the safe version described in section 3, and the unsafe version described presently. We have implemented the primitive *whnf* in two implementations of Haskell: version 5.02 of the Glasgow Haskell Compiler[3] and the February 2000 version of the HUGS interpreter[4]. For GHC a modest extension to the implementation is required, and for HUGS an implementation can be based on an existing internal interface library.

5 Cyclic structures

Non-strict programming languages allow for the definition and manipulation of recursively defined data structures [13]. For example, the potentially infinite list of ones can be denoted recursively as '$ones = 1 : ones$'. Under the lazy semantics of Launchbury such a value is represented with a cyclic structure, graphically illustrated in figure 1.

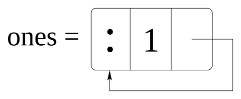

Fig. 1. A graphical view of a cyclic recursively defined value: an infinite list of ones

Sharing inside data structures is not observable within Haskell. In other words, the presence or absence of sharing in the underlying representation does

[2] It is possible to express the contextual dependency of *whnf* via the IO monad of Haskell, such that $whnf :: a \to IO\ Bool$. Indeed it may be beneficial to lift *reify* into the IO monad as well. However, the ramifications of this design are not immediately clear, and so it is left for future consideration.

[3] www.haskell.org/ghc

[4] www.haskell.org/hugs

not effect the denotational meaning of the program. Of course the operational meaning of the program may vary greatly depending on the amount of sharing that occurs (see Launchbury [7] for examples). The conversion to *Meta* described above does not preserve sharing in the object-level value. More importantly, where a finite graph may be used to represent the underlying value (via cyclic sharing), an infinite tree will be used to represent the meta-value. Whether this turns out to be a problem will depend on the possible uses of the meta-representation. If we are just printing values, then the infinite tree representation might be satisfactory, providing we implement some sort of non-strict print mechanism (to avoid non-termination in the print function: iterative deepening would suffice). However, there are uses of the meta-representation that rely on cycles being represented in the meta-value, for example the re-evaluation technique that we discuss in section 6.

The difficulty and danger of detecting sharing (and more specifically cycles) within Haskell data structures is explored thoroughly in Claessen and Sands [1]. The standard approach to detecting cycles in a graph is to traverse nodes and *remember* all the nodes that have been seen before. A cycle is detected when a node is encountered that has been traversed previously. This is difficult in Haskell because the language does not support any form of object identity, which precludes the recognition of previously traversed nodes.

One approach to adding object identity to the language is to say "a and b are identical if (and only if) the memory location of a is the same as the memory location of b", which is often called *pointer equality*. This definition interacts badly with the underlying semantics of the language, in particular with some garbage collecting techniques that may relocate objects during collection. Improvements to naive pointer equality have been proposed. The Glasgow Haskell Compiler[5] implements *stable pointers*, which are abstract references whose value is invariant with the underlying address of the object to which they refer. The downside of using references to detect cycles is the cost of searching through the history of references at each node that is visited.

An alternative cycle detecting algorithm is to uniquely mark nodes as they are visited. A cycle is detected when a new node is encountered which has already been marked. This technique requires the ability to destructively update nodes as they are encountered. This works well in a serial system, where it is guaranteed that there will only be one marker at any given time. Marking nodes from within a non-strict language allows for race conditions similar to what we would expect from a language with co-routining. The avoidance of race conditions in marking nodes is possible if the traversal of the graph proceeds fully until either an unevaluated component or a cycle is detected.

Cycle detection is an area that requires more work. None of the above methods are highly portable across implementations of Haskell, although it seems inevitable that extensions to the language are required. Presuming that cycles can be detected, we propose to extend the *Meta* type to include the constructor

[5] www.haskell.org/ghc

Cycle to mark cyclic components of the underlying representation of a value. Possible meta-representation for the infinite list *ones* include:

$$\mathit{Apply}\ \text{“:”}\ [\mathit{nullCon}\ \text{“1”},\ \mathit{Cycle}]$$

which indicates the presence of a cycle, but does not contain a cycle itself, or:

$$\mathbf{let}\ x\ =\ \mathit{Apply}\ \text{“:”}\ [\mathit{nullCon}\ \text{“1”},\ \mathit{Cycle}\ x]\ \mathbf{in}\ x$$

which indicates, and also contains a cycle by extending the *Cycle* constructor to expect an argument of type *Meta*.

6 Uses of the meta-representation

In this section we illustrate two uses of our meta-representation. Our motivation is derived from the development of a declarative debugging system for Haskell, described in Pope [9]. The debugger constructs a tree (called an *evaluation dependence tree* or EDT) which abstractly describes the evaluation of a given program. The nodes in the EDT correspond to function applications that were made during the evaluation of the program. The EDT is traversed in search of erroneous function applications. The correctness of each application is determined by an *oracle* (typically a person) who knows the intended meaning of each function in the program. The oracle judges the correctness of an application based on a textual representation of the arguments and result of the application. The debugger must store (representations of) values of arbitrary type in the nodes of the EDT and display those values to the oracle if required. The display of values must be meaningful to the oracle and must preserve the lazy evaluation properties of the original program.

For first-order values produced by the program, the *Meta* type and *reify* method are well suited to providing a printing mechanism that satisfies the requirements of the debugger. In particular, the arguments and results of applications are converted to type *Meta* before being stored in the nodes of the EDT. A simple means for converting *Meta* values to text is described by the *text* function below:

$$\begin{array}{l}
\mathit{text}\ \mathit{Uneval}\ =\ \text{“\_”} \\
\mathit{text}\ \mathit{Cycle}\ =\ \text{“cycle”} \\
\mathit{text}\ (\mathit{Apply}\ \mathit{con}\ \mathit{args}) \\
\quad =\ \text{“(”}\ +\!\!+\ \mathit{con}\ +\!\!+\ \mathit{concatMap}\ (\lambda\ \text{arg}\ \rightarrow\ \text{“ ”}\ +\!\!+\ \mathit{text}\ \mathit{arg})\ \mathit{args}\ +\!\!+\ \text{“)”}
\end{array}$$

The partially evaluated expression called '*list*' from section 4 would be displayed as: `(: _ (: _ (: _ _)))`. In practice we would specialise the printing of lists and tuples for clearer presentation.

The EDT is obtained by program transformation. Each function in the debugged program is transformed such that it returns a pair containing its original result and an EDT node representing an application of that function. The transformed program is then executed and the first component of the pair of the topmost application is demanded, thus forcing an execution equivalent to using the *original* (untransformed) program. The second component of the topmost application forms the root of the EDT, which is then traversed lazily in search of erroneous applications. This basic transformational approach suffers from prohibitive space usage because the EDT precludes the garbage collection of intermediate data structures created during the evaluation of the program.

Naish and Barbour [8] propose to solve the space leak introduced by the transformed program by initially only creating the top few levels of the EDT. The lower levels of computation are provided by calls to the original code of the program, thus avoiding the generation of a complete EDT for the execution. When the leaves of the partial EDT are encountered, the code for the transformed sub–computation at that point is called to further (partially) expand the EDT at the leaf. Thus some space is saved at the cost of re-evaluation. In a non-strict language, re-evaluation of expressions is difficult because the extent that an expression is evaluated depends on its context. Our meta-representation of values records the extent to which they were previously evaluated. We can use this information to re-evaluate expressions up until, but not exceeding, the extent to which they were previously evaluated. We define the following class *ReEval*, which collects all the (first-order) types from which values can be re-evaluated:

```
class ReEval a where
  reEval :: a → Meta → ( )
```

An example instance declaration for the $[a]$ type:

```
instance (ReEval a) ⇒ ReEval [a] where
  reEval _ Uneval = ( )
  reEval _ Cycle = ( )
  reEval [ ] (Apply "[ ]" [ ]) = ( )
  reEval (x : xs) (Apply ":" [x′, xs′])
      = case reEval x x′ of
            ( ) → case reEval xs xs′ of
                      ( ) → ( )
```

The *reEval* function always returns the unit value, denoted in Haskell by the constructor '( )'. Thus it would appear that *reEval* doesn't do anything constructive. From a purely functional point of view this is true. However, *reEval* does cause an important computational side-effect. In essence it causes its first argument to be evaluated to the extent of some previous evaluation, which we assume was of the same expression (this assumption can be guaranteed by the debugging system). The role of the *Meta* argument is to capture the extent of the previous evaluation, and halt the re-evaluation at the same point (or where cycles have been detected). It is essential for the re-evaluation mechanism to respect the lazy evaluation and cyclic sharing of expressions in order to preserve the termination properties of the original program that is being debugged.

7 Discussion and extensions

As was pointed out in section 3 we would like an automatic way for instances of the *Reify* class to be generated by the compiler. Intuitively this seems possible due to the fact that the rules for reification follow the syntactic description of algebraic types. Winstanley [14] describes a system called DrIFT for deriving instance declarations for arbitrary type classes based on a set of pre-processing rules. This is exactly the type of derivation technique that we require for the *Reify* class (and indeed also for the *ReEval* class). In figure 2 we outline the rules for deriving instances of the *Reify* class for first-order algebraic data-types.

```
For each data declaration:

    $d\ v_1\ \ldots\ v_n = c_1\ t_{1_1}\ \ldots\ t_{1_p} \mid \ldots \mid c_y\ t_{y_1}\ \ldots\ t_{y_r}$

Introduce a corresponding instance declaration:

    instance ($Reify\ v_1$, ... , $Reify\ v_n$) $\Rightarrow$ $Reify\ (d\ v_1\ \ldots\ v_n)$ where
      $reify\ x$
          = if $whnf\ x$ then
              case $x$ of
                $c_1\ u_{1_1}\ \ldots\ u_{1_p} \rightarrow Apply$ "$c_1$" [$reify\ u_{1_1}, \ldots, reify\ u_{1_p}$]
                ⋮
                $c_y\ u_{y_1}\ \ldots\ u_{y_r} \rightarrow Apply$ "$c_y$" [$reify\ u_{y_1}, \ldots, reify\ u_{y_r}$]
            else
              $Uneval$
```

Fig. 2. Algorithm for deriving instances of the *Reify* class.

So far we have only discussed the reification of values. We have not considered the converse operation of reflection[6]. There may be circumstances when we wish to convert a meta-representation of a value back into the value. The natural approach is to use type classes as we have done for reification.

A problem arises if we consider reflecting the meta-values *Uneval* and *Cycle*. How do we convert these into a meaningful object value? The only option seems to convert them to the undefined value (often called bottom). This is less than satisfactory, and would imply that the important equality '$reflect \circ reify = id$' does not always hold. The necessary insight for a solution comes from two places. Firstly, Laufer and Odersky [6] embed the object level value in the meta-representation. This means that during reification, the object value is stored in the meta-representation, and during reflection it is selected from the meta-representation. The main shortcoming of this work is that they define their

[6] Note the apparent confusion in terminology. *Reflection* can be taken to mean either the general activity of self reference or the more specific activity of mapping meta-values to values. For the rest of this paper we take the second meaning.

reification over a language where every value is of the same type (the SK combinator calculus). The second ingredient of our solution comes from Dornan [2], who uses existentially quantified types to allow the embedding of values of many different types into the same meta-representation. Unfortunately he requires dynamic types (via type tests at runtime) to extract object values from the meta-representation in a safe manner.

The following extension of the *Meta* type illustrates how Dornan's use of existential types could be used to allow the embedding of object level values in the meta-representation (ignoring cycles for simplicity):

```
data Meta a = Apply String [Arg]
            | Uneval a
data Arg    = forall a . Reify a ⇒ Arg a
```

Two things are worth noting. Firstly, the introduction of the new type *Arg*. *Arg* provides a wrapper for the arguments to a data constructor, which are stored unchanged in the meta-representation. We employ an existentially quantified type in the definition of *Arg* to allow values of any type to be stored in the meta-representation. In order for the value to be reified at a later point we constrain the existential variable to be a member of the *Reify* class. Without this constraint, there would be no possible way to manipulate the constructor arguments once they are stored in the meta-representation, which would be futile. Secondly, *Meta* takes a type parameter 'a', which is supplied as an argument to the *Uneval* constructor. This allows us to embed unevaluated components (often called *closures* or *thunks*) of a data structure directly in the meta-representation.

We modify the *Reify* class definition as follows:

```
class Reify a where
  reify :: a → Meta a
```

and add a new class for reflection:

```
class Reflect a where
  reflect :: Meta a → a
```

An instance of the type [a] for the *Reify* class:

```
instance Reify a ⇒ Reify [a] where
  reify x  = if whnf x then
                case x of
                  [ ] → nullCon "[ ]"
                  (y : ys) → Apply ":" [Arg y, Arg ys]
             else
                Uneval x
```

A curious aspect of this design is that the arguments to the list constructor remain un-reified in the meta-representation. The ability to reify these arguments at a later stage is guaranteed by the constraint on the existentially quantified variable in the definition of the type *Arg*. An instance of the type $[a]$ for the *Reflect* class:

```
instance Reflect [a] where
  reflect (Uneval x) = x
  reflect (Apply “[ ]” [ ]) = [ ]
  reflect (Apply “:” [Arg y, Arg ys])
        = (:) ((coerce y) :: a) ((coerce ys) :: [a])
```

Note the requirement of the unsafe function ‘*coerce* :: $a \rightarrow b$’, whose role is to convince the type checker that we may safely re-construct a list from its meta-representation, which requires unpacking values which were existentially quantified. In principle, such unrestrained type coercions are to be avoided in strongly typed languages. We justify their use in this particular application on the grounds that values of type *Meta* may only be constructed by compiler generated instances of the *Reify* class. In such circumstances the coercion is safe. With this new reflection capability we can write odd programs like this:

```
main = do
          let list = map id [1..100] :: [Int]
          print (length (take 3 list))
          let metalist = reify list
          printMeta metalist
          print (list == reflect metalist)
```

Presuming a suitable definition of *printMeta*, the output of this program is:

```
3
(: _ (: _ (: _ _)))
True
```

8 Related work

A means for displaying the evaluation of certain expressions in a running Haskell program is provided by the Haskell Object Observation Debugger (HOOD) by Gill [3]. A type class is used to implement an overloaded function called *observe*. This function has the type of the identity function, and so it can freely be inserted into any expression without changing the type of the program. However, calls to the observe function cause a side-effect to occur which is not visible from within the Haskell program. This side-effect is a log record of reduction steps for the expression being observed. The log information is written as XML to a file which can later be viewed through an interactive graphical browser. Observe will only record the actual parts of a data structure that were evaluated during a given program execution, and so it accurately reflects structures which were only partly evaluated due to lazy evaluation. Interestingly, it can capture the evaluation of functions, which are represented as sets of input and output values (but only for calls that were made during a given execution of the program). In essence it provides a means for displaying selected values from a given program execution. One of the main advantages of HOOD is that it is easily combined into an existing program and it only relies on a few commonly implemented extensions to Haskell, and thus is quite portable across Haskell implementations. The main disadvantage of HOOD is the prohibitive cost of logging reductions when many expressions need to be observed, such as the case in an interactive debugging or tracing system.

Dornan [2] describes a system for reification and reflection which is quite close to the one we have presented in this paper. His object language is Haskell and his meta-representation captures the same type of values as ours: first-order data structures. He uses an overloaded function for reification, although the overloading is not provided by type classes, though he does discuss this as a possibility. He uses dynamic types for reflection and describes an implementation of dynamic types via run-time tags. Existentially quantified types are used to encapsulate values of arbitrary type within the meta-representation. The prime motivation for his work is a flexible means for storing and communicating Haskell data-structures. His meta-representation does not capture the lazy evaluation nor cyclic sharing of the underlying object value.

Hinze and Peyton Jones [4] propose a considerable extension to the type class mechanism of Haskell to help support *generic programming*. Generic programming is one possible application of meta-programming, whereby functions can be defined over many different types by generalising their structure. The main contribution of Hinze and Peyton Jones is a means for providing default instance declarations for type classes which abstract the underlying structure of algebraic data types (sums, unit types and products). The outcome is a flexible means for deriving instances of type classes for any user defined type. This is more powerful than the ad-hoc deriving mechanism of the current Haskell definition, and safer than the pre-processing derivation mechanism of Winstanley [14]. Some of the generic programming capacity of this extension to the type class system can be obtained by defining functions over the meta-representation that we have presented in this paper.

9 Conclusion

The use of type classes for meta-programming is not a new idea. A survey of related literature will reveal their extensive use particularly with respect to the manipulation and representation of data values, for example [3, 4, 6, 14]. This paper has explored the use of type classes for providing an overloaded reification facility whose implementation can be derived from the syntactic description of algebraic data types. A simple meta-representation has been proposed, which gives a homogeneous structure for first-order values. Extensions of the basic meta-representation and the provision of some impure primitives allow for the reification of partial evaluation and cyclic sharing. The main benefit of our approach is simplicity: much of it can be implemented directly in Haskell, which is a boon for portability. The main downside of our simple approach is that we cannot model functional values at the meta-level. Certain aspects of our design have been influenced by the needs of a declarative debugging system [9]. It is interesting to consider what other application domains there are for such a meta-programming facility.

References

1. Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Asian Computing Science Conference*, pages 62–73, 1999.
2. C. Bentley Dornan. *Type-Secure Meta-Programming*. PhD thesis, Faculty of Engineering, University of Bristol, Bristol, United Kingdom, 1998.
3. A. Gill. Debugging haskell by observing intermediate data structures. Technical report, University of Nottingham, 2000. In Proceedings of the 4th Haskell Workshop, 2000.
4. R. Hinze and S. Peyton Jones. Derivable type classes, 2000. Submitted to the Haskell Workshop 2000.
5. S. Peyton Jones and J. Hughes (editors). Haskell 98: A non-strict, purely functional language. www.haskell.org/onlinereport, 1999.
6. K. Laufer and M. Odersky. Self-Interpretation and Reflection in a Statically Typed Language. In *OOPSLA Workshop on Reflection and Metalevel Architectures*, September 1993.
7. J. Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993.
8. L. Naish and T. Barbour. Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications*, 18(1):401–408, 1996.
9. B. Pope. Buddha: A declarative debugger for Haskell. Technical Report 98/12, The Department of Computer Science and Software Engineering, The University of Melbourne, 1998.
10. B. Pope and L. Naish. Specialisation of higher–order functions for debugging. To appear in the proceedings of the International Workshop on Functional and (Constraint) Logic Programming, Kiel, Germany, September 2001.
11. T. Sheard and J. Hook. Type safe meta-programming, 1994. Manuscript, Oregon Graduate Institute.
12. B. Cantwell Smith. Relfection and semantics in lisp. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, Salt Lake City, Utah, 1984.
13. G. Tremblay. Lenient evaluation is neither strict nor lazy. *Computer Languages*, 26(1):43–66, 2001.
14. N. Winstanley. A type-sensitive preprocessor for haskell, 1997. In Glasgow Workshop on Functional Programming, Ullapool, 1997.