Simple and Compositional Reification of Monadic Embedded Languages

Functional Pearl

Josef Svenningsson    Bo Joel Svensson

Chalmers University of Technology
{josefs,joels}@chalmers.se

Abstract
When writing embedded domain specific languages in Haskell, it is often convenient to be able to make an instance of the Monad class to take advantage of the do-notation and the extensive monad libraries. Commonly it is desirable to compile such languages rather than just interpret them. This introduces the problem of monad reification, i.e. observing the structure of the monadic computation. We present a solution to the monad reification problem and illustrate it with a small robot control language. Monad reification is not new but the novelty of our approach is in its directness, simplicity and compositionality.

Categories and Subject Descriptors    D.3.2 [Programming Languages]: Language Classifications—Applicative (functional) languages; D.3.4 [Programming Languages]: Processors—Code generation

Keywords    Compiling, embedded language, reification, monads

1. Introduction
Benny is a computer science student working in a project involving programming robots in a low-level imperative language. However, Benny has a budding interest in functional programming using Haskell and has read “The Haskell school of expression” [6]. He gets the idea to implement a language for robot control embedded in Haskell. Benny realises that the capabilities of the robot hardware are very similar to those of the robots that Hudak evaluates graphically in a grid world. There is however a very important difference; the embedded language needs to be compiled into some form that is understood by the robot.

Guided by the capabilities of the target robot, Benny designs an API for robot programming. The robot can perform operations such as move that steps the robot forward and turn left and right. The robot also has the capability to execute program loops and conditionals and it has a forward facing sensor with which it can query the world.

The API that Benny designs is simple:

move :: Program ()
turnLeft :: Program ()
turnRight :: Program ()
sensor :: Program Bool
cond :: Program Bool -> Program Bool -> Program ()
while :: Program Bool -> Program () -> Program ()

Now, Benny wants to express robot programs using the Haskell do notation. The motivation behind this is the imperative look and feel of the operations he identified and the potential to use all the control structures in the Control.Monad library. For this, a Monad instance is needed.

To be able to create a first-order representation of a computation described using monads, Benny needs to solve the problem of reifying monads. This is the story of how Benny came up with a particularly simple solution to this problem.

1.1 Example programs
Benny is quite happy with his language design and decides to write some programs to try it out.
The first program is called sMove, which implements a safe move operation. This operation can be executed by the robot even though it is facing an obstacle, without risk of harming robot or obstacle.

sMove :: Program ()
sMove = cond sensor turnRight move

The second program moves the robot forward until it stands directly in front of an obstacle such as a wall.

moveToWall :: Program ()
moveToWall = while ((liftM not) sensor) move

Based on these examples, Benny is quite satisfied with the design of his language and turns to implementing it.

1.2 Implementation and data structures
Enthused by the prospect of compiling these programs to the target language and seeing some robot action, Benny goes to work on the data structures.

Since the language is going to be compiled, Benny realises that the booleans in his language cannot be regular Haskell booleans. The booleans needs to be replaced by boolean typed expressions.
type Name = String

data BoolE = Lit Bool
| Var Name
| (:||:) BoolE BoolE
| (:&&:) BoolE BoolE
| Not BoolE

Following this, the operations that should go into the Program data type feel straightforward.

data Program a where
  Move :: Program ()
  TurnLeft :: Program ()
  TurnRight :: Program ()
  Sensor :: Program BoolE
  Cond :: Program BoolE -> Program ()
  While :: Program BoolE
  - > Program ()
  -> Program ()

Benny continues by naively adding constructors for Return and Bind to the Program data type.

data Program a where
  ...
  Return :: a -> Program a
  Bind :: Program a
  - > (a -> Program b)
  -> Program b

Then he writes down the Monad instance.

instance Monad Program where
  return = Return
  (>>=) = Bind

Proud of his accomplishments, Benny sends his Haskell module in an email to Prof. Björn. Benny knows that Björn is teaching an introductory Haskell course and should be able to provide feedback.

1.3 Problem statement
Meanwhile in Professor Björn’s Office

Prof. Björn notifies an email from Benny in his inbox and opens it.

Dear Professor Björn

I am CS student working in a robot control project. Usually, we program our robots in C but I have developed an interest in Haskell programming and thought it’d be natural to try implementing an EDSL. I know you teach an FP course and thought I would ask for your input. Attached to this mail is a file containing an outline of the data types I want to use. Does this look sensible to you?

Thank you
Benny

Prof. Björn opens the attachment and takes a look at the data type. He particularly notices the constructors Return and Bind and the Monad instance. He shakes his head at Benny’s naiveté and writes an email back:

Hello Benny

I’m afraid your implementation of the monadic primitives can never work. The constructor Return can take any arbitrary value that has nothing to do with the language you’re designing. These values may be strings, binary trees or higher-order functions from zygomorphisms to histomorphisms. The same problem goes for Bind; it has to be able to handle arbitrary values. It cannot possibly work! This is a very difficult problem and such a naive solution is bound to fail.

Prof. Björn
Dept. of Computer Science and Engineering
Chalmers University of Technology

While waiting for feedback from Björn, Benny has carried on trying to implement a compiler for his language.

When Björn’s email does arrive, Benny is confused. He feels he has already managed to compile the Program data type into a representation closer to that which is executed by the robot hardware.

He writes an apprehensive email back to Prof. Björn.

Hello Prof. Björn

Thanks for your feedback. But I think it does work! Attached to this email you find a file containing my attempt at compiling the Program data type into a representation closer to that which is executed by our robots.

I have also attached two example programs that you can compile and then run in the graphical simulator (see figure 1)

Benny

2. Compilation of the monadic robot EDSL

Björn receives Benny’s latest email

Björn looks through the Compiler module and finds a data type describing a first-order representation of the robot language.

data Prg = PMove
| PTurnRight
| PTurnLeft
| PSensor Name
| PCond BoolE Prg Prg
| PWhile Name Prg Prg
| PSeq Prg Prg
| PSkip
| PAbsign Name BoolE

From the Prg data type, Björn presumes that the PSeq constructor will be the result of compiling Bind and that PSensor binds a variable which can be used in PAbsign and PWhile. The conclusion is that the representation looks sensible, but he is very interested in seeing the Program a -> Prg transformation. The compile function assumes a splittable name supply with the interface described in figure 2.
spiralIn :: Int -> Program ()
spiralIn 0 = return ()
spiralIn n = do
    replicateM_ 2 $ do
        replicateM_ n move
        turnLeft
    spiralIn (n-1)

followWall :: Program ()
followWall = while (return true) $
    cond checkLeft $ move
    turnLeft
    s <- Sensor
    TurnRight
    return s

Figure 1. The spiralIn and followWall programs sent from Benny to Björn in an email

collinateSupply :: NameSupply
split2 :: NameSupply -> (NameSupply,NameSupply)
split3 :: NameSupply -> (NameSupply,NameSupply,NameSupply)
supplyValue :: NameSupply -> Int

Figure 2. The interface of the splittable name supply used in the implementation of the compile functions

runCompile :: Program a -> Prg
runCompile prg = snd $ compile s prg
    where
        s = newNameSupply

compile :: NameSupply -> Program a -> (a, Prg)
    where
        s1,s2,s3 = split3 s
        (b,bp)  = compile s1 b
        (a,p1)  = compile s2 p1
        (a2,p2) = compile s3 p2
        v = supplyValue s
        nom = "v" ++ show v
        (s1,s2,s3) = split3 s
        s = compile s prg
        v = supplyValue s
        nom = "v" ++ (show $ supplyValue s3)
        nvp = (wp' 'PSeq' PAssign nom b)
        s = compile s (Return a) = (a,PSkip)
        where
            (s1,s2,s3) = split3 s
            (b,bp)  = compile s1 b
            (a,p1)  = compile s2 p1
            (a2,p2) = compile s3 p2
            (s1,s2,s3) = split3 s
            s = compile s (Bind pa f) = (b, prg1 'PSeq' prg2)
                where
                    (s1,s2,s3) = split3 s
                    (a,prg1) = compile s1 a
                    (b,prg2) = compile s2 (f a)

Björn studies the code in disbelief, saves the module and tries it out on some examples. The code does indeed seem to work and Björn’s disbelief is replaced with enthusiasm; this appears to be a simple and convenient way to reify monads.

Björn sends an email to Benny, inviting him to a meeting.

3. Meeting in Björn’s office

Björn: Hello, come in.
Benny: Thanks.

Björn: So, about this robot language! You gave me a bit of a surprise there. I was entirely sure that what you did was impossible.

Benny: Oh, but I just did the first thing that came to mind. There was no deep thought behind it.

Björn: The problem is when you try to reify the bind constructor of your representation.

Benny: I didn’t realise it was a problem.

Björn: But it is! Your solution is that you are careful about the return types of all operations in your language. They are either of unit type or some type which can be guaranteed to be reified.

Benny: Yes, now that you mention it, I have seen that style used for defining DSLs.

Björn: Having constructors like Sensor return BoolE instead of Bool is a crucial part of why your method works. It allows the compile function to be written in such a way that it can generate the return values statically and pass them as arguments to Bind. In a sense you’re doing evaluation at the same time as compilation. There’s just enough evaluation to remove the monadic constructs. It’s a very neat trick!

Benny: Thank you.

Björn: Another way to think about your trick is that you are using a writer monad to evaluate your language and produce the first-order syntax tree as a side effect. Then you could have used a state monad transformer on top of that for the fresh name supply. But that’s a stylistic choice.

Benny: You seem to be making a big deal out of this. I just did what I thought was the most straightforward thing to do.

3.1 Related work

Björn: Others have solved this problem before but I have never seen a solution as simple as yours and was quite surprised that it works. For example, in this paper [Björn shows Benny the paper [7]], the authors use a continuation monad to be able to reify the monadic constructs of their language. If they...
were to implement your robot language they would use the same Program data type as you do. However, they would not expose that to the user of the language. Instead they would create a type like this:

\[
data \ P a =
\ P (\forall r. \ (a \to \text{Program} r) \to \text{Program} r))
\]

BJÖRN: The type P is a continuation monad so the monad instance comes naturally. Operations on P are defined using Bind like this:

\[
\text{while} :: P \text{BoolE} \to P () \to P ()
\]
\[
\text{while } c \ b =
\ P (\lambda k \to \text{While} (\text{runP} c) (\text{runP} b) \text{ `Bind` } k)
\]
\[
\text{cond} :: \text{BoolE} \to P () \to P () \to P ()
\]
\[
\text{cond } b \ t \ e =
\ P (\lambda k \to \text{Cond} b (\text{runP} t) (\text{runP} e) \text{ `Bind` } k)
\]

BENNY: I see. Continuations are quite magical to me. I could never have come up with that technique. How do they deal with transforming higher-order programs to first-order programs?

BJÖRN: That’s a good question. The transformation to first-order programs is dealt with by a library called Syntactic. It is described in a separate paper [Björn pulls out the paper [2] from a pile of papers]. It’s hard to do an apples to apples comparison between Syntactic and your technique. Syntactic solves a much bigger problem than what you’re doing so it is naturally more complicated.

BENNY: Ok.

BJÖRN: There are also the papers [5, 8] which solves the problem in a manner that is more similar to yours. They also have explicit constructors Bind and Return but give them a slightly different type which guarantees that all applications of Bind are normalised, so that all Bind are right-associated. Unexpectedly, this allows them to make instances of the Monad class but at the same time constrain the arguments of Bind. Their technique is more complicated than yours, but just as with the case of Syntactic, they solve a more general problem. However, there is one particular thing I like about your technique.

BENNY: What is that?

BJÖRN: Your technique is compositional.

3.2 Compositionality

BENNY: What does it mean that my technique is compositional?

BJÖRN: The compile function is compositional because there is one case for each constructor and each case deals with exactly one constructor. In particular, the constructors Return and Bind are handled completely separately from all other language constructs.

BENNY: And that is good?

BJÖRN: Absolutely! Composition definitions are nice because they imply that there is no weird semantical interaction between the constructs as they are defined independently. But it also means that it should be possible to factor out the constructors Return and Bind into a data type of their own. Then, it could be combined with other data types using techniques like Data Types a la Carte [9] or CompData [3]. This way, a language can be designed piece by piece. You can then select the set of pieces required for a particular task or that suits a particular brand of robots.

BENNY: Oh! Interesting.

3.3 The Monad laws

BJÖRN: There is still one problem with your method though.

BENNY: What’s that?

BJÖRN: When we make instances of the monad class we expect certain laws to hold. [Björn writes the laws on his whiteboard]

\[
m \ggg \text{return} = m \\text{eval (m >>= \text{return}) = eval m}
\]
\[
\text{return } a \ggg f = f a \\text{eval (\text{return } a >>= f) = eval (f a)}
\]
\[
(m \ggg f) \ggg g = m \ggg (\lambda a \to f a \ggg g) \\text{eval ((m >>= f) >>= g) = eval (m >>= (\lambda a \to f a >>= g))}
\]

BJÖRN: These laws clearly don’t hold for your instance. Take the first law for instance. The left hand side will have extra Bind and Return constructors compared to the right hand side.

BENNY: Hmm. I hadn’t really thought about that. But adding an extra return at the end of a computation shouldn’t make a difference in my implementation.

BJÖRN: So, you’re saying that your implementation actually obeys the first monad law?

BENNY: Yes, at least when I run my programs I will never see any difference between the left hand side and the right hand side.

BJÖRN: Aha, so what you’re saying is that if we compare the semantics of programs rather than comparing the programs themselves then we get some useful laws. [Björn scribbles some new laws on the whiteboard]

\[
\text{eval (m >>= \text{return}) = eval m}
\]
\[
\text{eval (\text{return } a >>= f) = eval (f a)}
\]
\[
\text{eval ((m >>= f) >>= g) = eval (m >>= (\lambda a \to f a >>= g))}
\]

BJÖRN: These laws are morally the same as the monad laws, especially if we don’t let the user of the robot language ever compare terms in the language.

BENNY: Yes, that captures my intuition very well.

BJÖRN: Ok, good. Can you prove these equations?

BENNY: No, I don’t have any experience proving programs correct.

BJÖRN: Well, it shouldn’t be that difficult. Let me see what I can come up with.

[Byran scribbles frenetically on a piece of paper for a couple of minutes] Aha! Your technique is quite general. It can reify monads into any kind of structure which is a monoid. Return translates into the monoid unit and bind translates into the monoid operation. For example, in your compilation function it is important that the semantics of PSkip is the identity of the semantics of PSkip and that PSkip is associative.

BENNY: Ok, that sounds good. I take that as meaning the method is quite general?

BJÖRN: Yes, requiring a monoid is a very mild restriction. Look at the time! This was interesting, I got quite carried away. We must round off but please get back to me if you make any more progress on your robot language.

BENNY: Thanks very much for your time. Bye!

BJÖRN: Thank you.

4. Composing reifiable monadic languages

At Benny’s computer

Benny was really intrigued by the idea of compositionally building embedded languages and after having read the papers Björn showed him he decides to try his own approach to the problem.
Figure 3. Data types from CompData for composing languages.

Inspired by CompData he starts out by adding the code in figure 3 to his file. The data type :+: is used for composing languages, and the :+: typeclass provides coercions so that the programmer doesn’t have to worry about using the right sequence of the InjL and InjR constructors to inject terms into the composed language.

Benny then starts to add data types for the different language constructs in his robotic language. The different operations for the robot are pleasingly easy to add.

In order to test out these definitions, Benny next turns to implementing a compiler. He realises that the compiler now needs to be implemented as a class with one instance per compilable sub-language.

Benny can now write the type of his robotic language, built from independent pieces. It is noteworthy that the monadic expressions have to be added on top of all the other language constructs.

Given the definition of his language Benny can now write an injection function which helps writing smart constructors for all the different robot operations.

cond :: (CondOp :<: f) -> MonadExp f BoolE -> MonadExp f () -> MonadExp f ()
cond b p1 p2 = inject $ Cond b p1 p2

while :: (WhileOp :<: f) -> MonadExp f BoolE -> MonadExp f () -> MonadExp f ()
while pb p = inject $ While pb p

Things are starting to fall into place. The only remaining bit is to complete the compiler for monadic expressions.

instance (Compile (f (MonadExp f))) where
    compile s (Oper o) = compile s o
    compile s (Return a) = (a,PSkip)
    compile s (Bind m f) = (b,prg1 'PSeq' prg2)
        where (s1,s2) = split2 s
            (a,prgl) = compile s1 m
            (b,prg2) = compile s2 (f a)

Now, Benny has all the functions he needs to compile a simple example program.

test1 :: Robot ()
test1 = do
    move
    turnL
    move
    turnL

Compiling the program above gives the expected result.

PSeq PMove (PSeq PTurnLeft (PSeq PMove PTurnLeft))

Benny decides to contact Björn again to show him what he has done.

Dear Prof. Björn

I have attached a file which demonstrates that the monadic constructs can be factored out and compiled separately, just as you suggested.

Thanks
Benny

Fascinating! You’ve managed to factor out the monadic operations so that they can be compiled once and for all. Users of your library don’t have to be concerned at all with the semantics of bind and return. Very nice!

I note that the data type for monadic operations cannot be composed as freely as the other operations, but instead has to be applied separately as a final step. Your solution is very similar to how variable binding is handled in the Syntactic library [1].

Prof. Björn
Dept. of Computer Science and Engineering
Chalmers University of Technology

5. Epilogue

This paper shows a simple method of implementing monadic EDSLs. It shows a naïve approach to the monad reification problem which has the additional benefit of being compositional. In section 4, the language is reimplemented in an extensible way and the compiler function is shown explicitly to be compositional by treating sub parts of the language separately.

The example language used throughout the story is small and quite limited. But the technique presented in this paper does scale up to larger languages and more complicated language constructs. It is currently used in the implementation of Obsidian, an EDSL for general purpose GPU programming [4]. Obsidian has language constructs which are considerably more advanced than the robot language is this paper. One example is the sequential for-loop which is higher-order (the type parameter t is not relevant to the current discussion):

SeqFor :: EWord32 -> (EWord32 -> Program t ()) -> Program t ()

There does, however, seem to be restrictions on what kind of language constructs the reification technique presented in this paper can deal with. For instance, the functions from the MonadPlus class has resisted our attempts at adding them to an EDSL in the same way as we’ve added Return and Bind. The exact limits of the expressiveness of our method is currently unknown to us.

Acknowledgments

We would like to thank Emil Axelsson for very valuable help during the implementation of the methods used in this pearl. We thank Mary Sheeran for suggesting the names Björn & Benny. The ICFP reviewers provided many helpful suggestion which has improved the paper.

This research has been funded by the Swedish Foundation for Strategic Research (which funds the Resource Aware Functional Programming (RAW FP) Project) and by the Swedish Research Council.

References