Typed Logical Variables in Haskell

Koen Claessen
koen@cs.chalmers.se

Peter Ljunglöf
peb@cs.chalmers.se

August 9, 2000

Abstract

We describe how to embed a simple typed functional logic programming language in Haskell. The embedding is a natural extension of the Prolog embedding by Seres and Spivey [16]. To get full static typing we need to use the Haskell extensions of quantified types and the ST-monad.

1 Introduction

Over the last ten to twenty years, there have been many attempts to combine the flavours of logic and functional programming [3]. Among these, the most well-known ones are the programming languages Curry [4], Escher [13], and Mercury [14].

Curry and Escher can be seen as variations on Haskell, where logic programming features are added. Mercury can be seen as an improvement of Prolog, where types and functional programming features are added. All three are completely new and autonomous languages.

Defining a new programming language has as a drawback for the developer to build a new compiler, and for the user to learn a new language. A different approach which has gained a lot of popularity the last couple of years is to *embed* a new language in another language, called the *host language*. Haskell has been shown to be extremely well-suited for this purpose in various areas [8, 2, 1].

The embedding approach has an obvious other advantage. Programs in the embedded language are first class citizens in the host language, and can therefore be generated by a program in the host language.

Our aim is to embed logic programming features in Haskell. To this end, several different approaches have been taken, noticably by Seres and Spivey, who embed a language of predicates over terms in Haskell [16, 17], and by Hinze, who shows how to describe backtracking efficiently and elegantly [5, 6]. Our approach combines these ideas and adds something new: the terms of the embedded program are, in contrast to Seres' and Spivey's approach, *typed*.

The resulting embedded language has several limitations however. First of all, there are some syntactic drawbacks in the sense that some predicate definitions can not be as elegantly described in Haskell as in Prolog, because of the special syntax that Prolog has for logical variables and unification pattern matching. Second, real implementations of logical programming have many specialised search strategies.

The rest of the paper is organized as follows. In section 2, we present a summary of Seres' and Spivey's work on embedding Prolog in Haskell. In Section 3, we generalize their work to use a monad. In Section 4, we show how we can use the ST-monad to deal with user-defined datatypes in a typed setting. In Section 5 we conclude.

2 Embedding Prolog in Haskell

Silvija Seres and Mike Spivey have embedded traditional logic programming in a functional framework [16, 17]. This section is essentially a summary of their embedding.

2.1 Logical Variables

The embedding consists of a datatype `Term` for terms, a type `Pred` of predicates, a unification predicate $(\doteq)$, the connectives $(\wedge, \vee)$ and the existential quantifier $(\exists)$:

```
(≐)      :: Term → Term → Pred
(∧),(∨)  :: Pred → Pred → Pred
(∃)      :: (Term → Pred) → Pred
```

The type `Term` for terms could be anything, but must have the possibility of being an uninstantiated variable. Here we use a type of binary trees, with `Var i` being a reference to the variable with the unique identifier i:

```
data Term = Var Id | Atom String | Nil | Term ::: Term
```

In the embedding we use lists to simulate backtracking, but this is a rather arbitrary choice — there are lots of other possibilities. So we define a type `Backtr`, which in this case is equivalent to lists. One thing which we use, is that all the possible variants for this search type are monads, more specifically monads with both a zero (written as `mzero`) and a plus (written as (+++)).

The type `Pred` of predicates is a function that takes a computation state and gives a stream of new states. A computation state `State` holds the current values of the logical variables (called substitutions), and a stream of uninstantiated variables.

```
type Backtr a = [a]
type Pred     = State → Backtr State
type State    = (Subst, [Term])
type Subst    = [(Id, Term)]
```

The unification predicate $(\doteq)$ uses a standard unification function `unify` :: `Term` → `Term` → `Subst` → `Backtr Subst`, which takes two terms and a substitution and returns either failure (represented by the zero of the backtracking monad) or a new substitution (represented by the unit):

```
(≐) :: Term → Term → Pred
a ≐ b = λ(sub,vs) → do sub' <- unify a b sub ; return (sub',vs)
```

The connective (∧) maps the second predicate onto the results of the first, concatenating the resulting list of lists and (∨) is simply concatenation of results:

```
(∧),(∨) :: Pred → Pred → Pred
p ∧ q = λst → p st ≫= q
p ∨ q = λst → p st ++ q st
```

Finally the quantifier (∃) applies the given function on a fresh variable, taken from the state:

```
(∃) :: (Term → Pred) → Pred
∃p = λ(sub,v:vs) → p v (sub,vs)
```

We can also define the success predicate `true` and the failure predicate `false` easily:

```
true, false :: Pred
true  = λst → return st
false = λst → mzero
```

With these definitions we get the standard depth-first search behaviour of e.g. Prolog. The definitions of `Pred`, (∧) and (∨) can also be changed to get other search strategies, e.g. breadth-first search, as shown in [17].

2.2 A Running Example

To get the feel of how to use this embedding, we give an example which we will continue to develop during the course of the paper. The example problem is to go from one node to another in a directed acyclic graph, returning the path travelled.

In Prolog, the standard solution is as follows. If we suppose that the graph is stored in the database as a predicate `edge(X,Y)` meaning that `X→Y` is an edge in the graph, the predicate `path(X,Y,Nodes)` can be defined as:

```
path(X,X,[X]).
path(X,Z,X:Nodes) :- edge(X,Y), path(Y,Z,Nodes).
```

In our embedding, we must specifically declare the logical variables using (∃), which makes the program more verbose, but the idea is the same:

```
path x z nodes =
      x≐z ∧ nodes≐x ::: Nil
    ∨ ∃(λnodes' → nodes≐x ::: nodes' ∧ ∃(λy → edge x y ∧ path y z nodes'))
```

Now we only need to know how to define a suitable graph, so here is an example of a directed acyclic graph with the five nodes a–e and seven edges, written in Prolog:

```
edge(a,b).  edge(a,d).  edge(b,c).  edge(b,d).
edge(c,d).  edge(c,e).  edge(d,e).
```

The direct translation of this predicate into the embedding becomes very verbose and unreadable:

```
edge :: Term → Term → Pred
edge x y = ( x≐Atom "a" ∧ y≐Atom "b" ) ∨ ( x≐Atom "a" ∧ y≐Atom "d" )
         ∨ ( x≐Atom "b" ∧ y≐Atom "c" ) ∨ ( x≐Atom "b" ∧ y≐Atom "d" )
         ∨ ( x≐Atom "c" ∧ y≐Atom "d" ) ∨ ( x≐Atom "c" ∧ y≐Atom "e" )
         ∨ ( x≐Atom "d" ∧ y≐Atom "e" )
```

But since we are embedding this program, we can use Haskell to *construct* this rather boring looking program, by making a translation from a list representation of the graph to the `edge` predicate:

```
edge :: Term → Term → Pred
edge x y = foldr (∨) false [ x≐Atom a ∧ y≐Atom b | (a,b) ← graph ]
  where
    graph = [ ("a","b"), ("a","d"), ("b","c"), ("b","d")
            , ("c","d"), ("c","e"), ("d","e") ]
```

3 A Monad for the Embedding

An observing reader may have noticed that the types and functions we have defined so far fit nicely in a monadic framework. To be more precise, the `Pred` type is isomorphic to an instance of the backtracking state monad BS:

```
newtype BS a = BS (State → Backtr (State,a))
type    Pred = BS ()
```

This monad has the following definitions of bind (≫=), return, zero and (+++):[1]

```
instance Monad BS where
    return a   = BS (λst → return (st,a))
    BS f ≫= k  = BS (λst → do (st',a) ← f st
                               let BS g = k a in g st')

instance MonadPlus BS where
    mzero          = BS (λst → mzero)
    BS f +++ BS g  = BS (λst → f st +++ g st)
```

After some thought we can see that (∧), (∨), `true` and `false`, correspond to (≫), (+++), `return ()` and `mzero` respectively. The unification (≐) is almost the same as before, and the existential quantifier can easily be defined using the auxiliary function `free`, which computes a new, unbound variable.

```
(≐) :: Term → Term → Pred
a ≐ b = λ(sub,vs) → do sub' ← unify a b sub ; return ((sub',vs),())

free :: BS Term
free = λ(sub,v:vs) → return ((sub,vs),v)

(∃) :: (Term → BS a) → BS a
∃p = do a ← free ; p a
```

[1] Observe that the `return`, (≫=), `mzero` and (+++) on the right hand side of the definitions apply to the `Backtr` monad, not the BS monad.

3.1 Revising the Example

Since the connectives, failure and the quantifier have the monadic counterparts ($\gg$), `mplus`, `fail` and `free`, we can reprogram our example into a more monadic form:

```
path :: Term → Term → Term → Pred
path x z nodes = do x≐z
                    nodes≐x:::Nil
               ∨ do nodes' ← free
                    nodes≐x:::nodes'
                    y ← free
                    edge x y
                    path y z nodes'
```

Now we can try this program on the problem to find a suitable path between the nodes a and e:[2]

```
Main> solve (path (Atom "a") (Atom "e") (Var "Xs"))
yes, Xs=a:::b:::c:::d:::e:::Nil
yes, Xs=a:::b:::c:::e:::Nil
yes, Xs=a:::b:::d:::e:::Nil
yes, Xs=a:::d:::e:::Nil
no (more) solutions
```

3.2 Functional Logic Programming

A monadic computation always returns a result, so why not use that in our embedding? Then we have an embedding of a logic programming language with results. Revising our example, we can view the `path` predicate as taking only the two nodes as arguments, and *returning* the path. This simplifies the definition somewhat:

```
path :: Term → Term → BS Term
path x z = do x≐z
              return (x:::Nil)
         ∨ do y ← neighbour x
              nodes ← path y z
              return (x:::nodes)
```

This definition uses the predicate `neighbour`, which gives the neighbours of the argument:

```
neighbour :: Term → BS Term
neighbour x = do y ← free ; edge x y ; return y
```

If we now try our example we get the following:

```
Main> solve (path (Atom "a") (Atom "e"))
a:::b:::c:::d:::e:::Nil
```

[2]The `solve` function is a small interpreter around the `BS` monad which displays the results in Prolog-style.

```
a ::: b ::: c ::: e ::: Nil
a ::: b ::: d ::: e ::: Nil
a ::: d ::: e ::: Nil
no (more) solutions
```

We can freely mix the predicate style, monadic style and functional logic style. Note however that we have not established full functional logic programming. This would for example require unification on function types, something we have only been able to do using a simple, *poor man's*, version. One could argue however that the expressivity that we do provide is the most used and the most interesting aspect of functional logic programming.

4 Typed Logical Variables

We have made an embedding of a simple functional logic programming language in a pure functional language. Our aim is now to transfer Haskell's strong typing into the embedding, instead of using the universal type `Term` for terms.

4.1 Polymorphic State

So far, the embedding uses an explicit state monad, using one type `State`, holding substitutions. If we want to use several different term datatypes at the same time, and it is unknown in advance which term datatypes we are going to use, this is a problem.

The problem can be solved if we move out of Haskell 98, and use one of the extensions available in GHC and Hugs: the ST-monad [12]. With the ST-monad, one has the capability to safely create, read and update reference cells of arbitrary type.

Another element in the embedding that is dependent on a certain term datatype is unification. A simple solution is to overload the unification operation, so that every term datatype can represent its own unification algorithm.

To add non-determinism to the ST-monad, in the form of backtracking, we can not simply use lists anymore. Instead, we introduce a backtracking monad transformer. This is just a variant on the continuation monad transformer, allowing for multiple answers instead of one.

```
newtype BacktrT m a = BTT (∀ans. (a → m [ans]) → m [ans])
```

A monad transformer allows lifting actions from the underlying monad in the new monad. There is also an operation that takes a lifted action back to the underlying monad.

```
lift :: Monad m ⇒ m a → BacktrT m a
lift m = BTT (λk → do a ← m ; k a)

lower :: Monad m ⇒ BacktrT m a → m [a]
lower (BTT f) = f (λa → return [a])
```

So finally, the definition of the new monad `LP`, for *Logic Programming*, allowing multiple term datatypes, is:

```
type LP s = BacktrT (ST s)
```

For efficiency purposes, we actually use Hinze's backtracking monad transformer [6], but we believe that the current definition is easier to understand.

4.2 Reading and Writing in the LP-monad

The logical variables used in a predicate are now implemented using the references of the ST-monad.

```
type LPRef = STRef
```

To unify and create uninstantiated variables we have to be able to read, write and create new references in the LP-monad. The reading and creating of references is just lifting of the corresponding functions in the ST-monad:

```
newLPRef :: a → LP s (LPRef s a)
newLPRef = lift ∘ newSTRef

readLPRef :: LPRef s a → LP s a
readLPRef = lift ∘ readSTRef
```

For writing references, we cannot simply lift the corresponding writing action in the ST-monad, since we have to take care of the backtracking as well. Writing a value to a reference has to be undone when we backtrack.

```
writeLPRef :: LPRef s a → a → LP s ()
writeLPRef ref a = BTT (λk → do a' ← readSTRef ref
                                writeSTRef ref a
                                ans ← k ()
                                writeSTRef ref a'
                                return ans)
```

Before we write the new value, we read the old value, which we restore after the branch `k` of the backtracking tree is done.

4.3 Logical Variables as References

A logical variable is either uninstantiated or it has a value of some kind. This sounds much like the `Maybe` type, and we use that to define variables:

```
type Var s a = LPRef s (Maybe a)
```

Now we define some types, for example atoms and lists:

```
data Atom s   = VarA (Var s (Atom s))   | Atom String
data List s a = VarL (Var s (List s a)) | Nil | a ::: List s a
```

Note that we need to use different constructors for the variables of each type,[3] so we cannot define a general form of the `free` predicate anymore. Therefore we declare the type class `Free` as:

```
class Free s a | a -> s where
  free :: LP s a
```

Note that we use the non-standard Haskell extension of multiple parameter type classes with functional dependencies here [11]. This is possible because all instantiations of a will contain an s. With this we can define the `free` predicate for each of our types:

```
instance Free s (Atom s) where
  free = VarA `liftM` newLPRef Nothing

instance Free s (List s a) where
  free = VarL `liftM` newLPRef Nothing
```

Overloading `free` has the extra advantage that we can create free pairs, triples, etc. as well:

```
instance (Free s a, Free s b) ⇒ Free s (a,b) where
  free = liftM2 (,) free free
```

4.4 Unification

Unifying two instances of a term datatype poses the same problem as with `free`, so we simply take the same solution; we overload unification.[4]

To minimize the work everytime a new datatype is declared, we split the work the unification algorithm has to do into two parts: variables and constructors. Therefore, we introduce a type class `Unify` containing two operators, `isVar`, used to check if a term happens to be a variable, and `unify`, used to unify two terms which are both not variables.

```
class Unify s a | a -> s where
  isVar :: a → Maybe (Var s a)
  unify :: a → a → LP s ()
```

The unification operator ($\doteq$) is defined in terms of these operations. Before we define it however, we introduce a helper function `unifyVar`, which unifies a variable and a constructor:

```
unifyVar :: Unify s a ⇒ Var s a → a → LP s ()
unifyVar ref a = do mb ← readLPRef ref
                    case mb of
                      Nothing → writeLPRef ref (Just a)
                      Just b  → a ≐ b
```

The unification algorithm can now be implemented as follows.

[3]This could be very much simplified if we had subtyping as in the language O'Haskell [15].
[4]There exist a general polytypic solution to this problem [9, 10], which takes a similar approach.

```
(≐) :: Unify s a ⇒ a → a → LP s ()
a ≐ b = case (isVar a, isVar b) of
          (Just var1, Just var2) | var1 == var2 → true
          (Just var , _        )               → unifyVar var b
          (_        , Just var )               → unifyVar var a
          _                                    → unify a b
```

It first deals with the special cases where at least one argument is a variable, and hands the other cases over to the user-defined operation `unify`. The instances for `Unify` of atoms and lists look like this:

```
instance Unify s (Atom s) where
  isVar (VarA var) = Just var
  isVar _          = Nothing

  unify (Atom s1) (Atom s2) | s1 == s2 = true
  unify _         _                    = false

instance Unify s a ⇒ Unify s (List s a) where
  isVar (VarL var) = Just var
  isVar _          = Nothing

  unify Nil      Nil      = true
  unify (a ::: as) (b ::: bs) = a ≐ b ∧ as ≐ bs
  unify _        _        = false
```

Note that the definitions of `isVar` almost look identical.

4.5 Getting out of the LP-Monad

To be able to extract a result from the LP-monad we have to define a run function that simply calls the run function of the ST-monad:

```
runLP :: (∀s. LP s a) → [a]
runLP m = runST (lower m)
```

This function can only be used when a does not depend on s. Both the `Atom` and the `List` types (as well as any type for logical variables) depend highly on s, because they can contain a variable constructor in the term. So, we define a conversion that takes away all variables from a term. To do this, we define a helper function which converts the contents of a variable. This only succeeds if the variable is actually instantiated.

```
variable :: (a → LP s b) → Var s a → LP s b
variable convert var = do Just a ← readLPRef var ; convert a
```

Here is how we can convert atoms and lists. The list conversion function takes as a parameter the conversion functions of its elements.

```
atom :: Atom s → LP s (Atom ())
atom (Atom s)   = return s
```

```
atom (VarA var) = variable atom var

list :: (a → LP s b) → List s a → LP s (List () b)
list elt Nil        = return Nil
list elt (a ::: as)    = liftM2 (:::) (elt a) (list elt as)
list elt (VarL var) = variable (list elt) var
```

Note that the result types use `()` as the state type, to indicate that these datatypes do not contain any variables.

We might be tempted to overload these functions, but we do not do this because it is not clear that we always want to convert the datatypes to their variable-less counter parts. For example, Haskell already has a perfectly fine list datatype, and here is how we could convert to it:

```
list' :: (a → LP s b) → List s a → LP s [b]
list' elt Nil        = return []
list' elt (a ::: as)    = liftM2 (:) (elt a) (list' elt as)
list' elt (VarL var) = variable (list' elt) var
```

Similarly, we might convert atoms directly to strings.

4.6 The Return of the Example

The only thing that we need to change in our standard example is the types for the predicates. The code is exactly the same as in Section 3.2, but the predicates now have the following types:

```
path      :: Atom s → Atom s → LP s (List s (Atom s))
neighbour :: Atom s → LP s (Atom s)
```

We get a type error if we try to evaluate the predicate `path (Atom "a") (Atom "b" ::: Nil)`, whereas without the ST-monad the call would just fail.

We can try to find the paths between a and e by using an adapted `solve` function, which is now implemented using `runLP`.

```
Main> solve (path (Atom "a") (Atom "e") >>= list atom)
a ::: b ::: c ::: d ::: e ::: Nil
a ::: b ::: c ::: e ::: Nil
a ::: b ::: d ::: e ::: Nil
a ::: d ::: e ::: Nil
no (more) solutions

Main> solve (path (Atom "a") (Atom "e") >>= list' atom)
[a,b,c,d,e]
[a,b,c,e]
[a,b,d,e]
[a,d,e]
no (more) solutions
```

5 Conclusions and Future Work

We have succeeded in embedding a simple typed functional logic language in Haskell, without extending the language with other than already well-accepted features, such as multiple parameter type classes (without overlapping instances), local universal quantification in types, and the ST-monad.

It takes some work to add a datatype to be used in the embedding. Some of this has to do with the fact that every datatype has to contain a special constructor for variables. One way of solving this is to define recursive datatypes using explicit fixpoints. This takes away some of the work when implementing a new datatype. However, the types get a lot more complicated and become unmanageable when dealing with more complicated types than regular datatypes. Another big help would be a polytypic programming tool [9, 7].

The resulting language is rather naive in several ways. First of all, the syntax of the programs is often more clumsy than the way you could write it in a dedicated logic programming language. This could be solved by adding some syntactic sugar.

Second, the search strategies are not as fancy as the ones one can find in some implementations of logic programming. Some of these could be implemented in our embedding, such as breadth-first searching and some features which make it possible to unify more cyclic structures. Others require a meta-level view of the program, such as indexing in most Prolog implementations.

Third, common implementations of Prolog have some interesting and practically useful extensions. Some of these are assert/retract in most Prolog implementations, and bb_get/bb_put in Sicstus Prolog, which are used to handle global variables that survive a failure. None of these are part of our embedding today, but we believe that most of them are not too hard to add.

Efficiency is often not the first aim of an embedding, but is sometimes desirable. Hinze's implementation of backtracking is quite fast [5], but the unification algorithm we implemented could probably be done better. For example, we could use optimizations in the spirit of the well-known union-find algorithm.

The techniques we used have been shown quite useful in implementing embeddings of other domains as well. For example, in ongoing unrelated work, we have used the ST-monad to implement an embedded language for describing state transition diagrams over variables of arbitary types using a similar method as the one we describe here.

For future work, we want to implement many of these proposed improvements and extensions. By doing this, we hope to find a good basis for a nice semantics for logic programming and its well-known extensions. Also, we would like to investigate how we can make the embedding more practical, by using it in more realistic programs.

References

[1] Koen Claessen. An embedded language approach to hardware description and verification, September 2000. Licentiate Thesis, Chalmers University of Technology, Gothenburg, Sweden.

[2] Conal Eliott. An embedded modeling language approach to interactive 3d and multimedia animation. In *IEEE Transactions on Software Engineering*, June 1999.

[3] Michael Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19 & 20:583–628, 1994.

[4] Michael Hanus. Curry: An integrated functional logic language, February 2000. The current report describing the language Curry, http://www.informatik.uni-kiel.de/~curry/report.html.

[5] Ralf Hinze. Prological features in a functional setting - axioms and implementations. In *Third Fuji Int. Symp. on Functional and Logic Programming*, pages 98–122, April 1998.

[6] Ralf Hinze. Deriving monad transformers. Technical Report IAI-TR-99-1, Institut für Informatik III, Universität Bonn, January 1999.

[7] Ralf Hinze. Polytypic functions over nested datatypes. *Discrete Mathematics and Theoretical Computer Science*, 3(4):159–180, 1999.

[8] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4):196, December 1996.

[9] Patrik Jansson. *Functional Polytypic Programming*. PhD thesis, Chalmers Univerisity of Technology and Göteborg University, June 2000.

[10] Patrik Jansson and Johan Jeuring. Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, September 1998.

[11] Mark P. Jones. Type classes with functional dependencies. In *9th European Symposium on Programming*, Berlin, Germany, 2000. Springer-Verlag LNCS 1782.

[12] John Launchbury and Simon Peyton Jones. Lazy functional state threads. In *Proc ACM Programming Languages Design and Implementation*, Orlando, 1994. ACM.

[13] John W. Lloyd. Programming in an integrated functional and logic programming language. *Journal of Functional and Logic Programming*, 3, March 1999.

[14] The Mercury project homepage. http://www.cs.mu.oz.au/research/mercury/.

[15] Johan Nordlander. *Reactive Objects and Functional Programming*. PhD thesis, Dept. of Computer Science, Chalmers University of Technology, 1999.

[16] Silvija Seres and Michael Spivey. Embedding Prolog in Haskell. In *Haskell Workshop*, Paris, France, September 1999.

[17] Silvija Seres, Michael Spivey, and Tony Hoare. Algebra of logic programming. In *Int. Conf. Logic Programming*, November 1999.