Haskell Rules: Embedding Rule Systems in Haskell

Steve Kollmansberger Martin Erwig
Oregon State University
{kollmast,erwig}@eecs.oregonstate.edu

Abstract

We present a domain-specific embedded language that allows the definition of rule systems in Haskell. As one particular example, we consider the modeling of type systems, which is an important part of programming language design. Type systems are most conveniently described using rule systems. Our approach is well integrated into Haskell's type system and thus facilitates the convenient modeling of type systems and language semantics in general. We also demonstrate how our DSEL allows functional-logic programming in Haskell.

Our system generalizes previous work by operating on user-defined data types, taking advantage of static typing, representing rules as functions, and allows a creative integration of logical variables into data structures which generalizes two previous approaches. We describe a straightforward method for translating rule systems into our DSEL.

Keywords Rule Systems, Haskell, Language Prototyping, Domain-Specific Embedded Languages, Functional-Logic Programming, Generic Unification

1. Introduction

Modeling type systems is an important part of programming language design. Type systems are typically defined through rule systems. More generally, any static or dynamic language semantics can be represented using rule systems. Implementing these rule systems into actual code can present a significant challenge. Until recently, rule systems could be directly transcribed only into logic languages, such as Prolog [6] or Twelf [15]. To implement a rule system in a functional language without writing lots of tedious unification code one can switch to a functional-logic language, such as Mercury [20] or Curry [9], but this has the drawback of losing libraries and code already written in Haskell. Moreover, tools or development environments might not be reusable between Haskell and other languages. Alternatively, one could extend functional programming with logic features. However, this approach has several shortcomings. In particular, many implementations of "Prolog in Haskell" are untyped and do not allow a smooth integration between the logical and functional aspects. This paper presents a method for directly modeling rule systems in Haskell in a type-safe, integrated, and flexible fashion.

Performing rule-based computation requires two fundamental components: searching and unification. Thus, a simple backtracking monad is not sufficient. One approach to this problem was attempted by Spivey and Seres [21]. Their approach uses types such as `Predicate` and `Answer` to represent logical computations along with a set of foundational logic combinators. However, this approach requires explicit use of logical operators, whereas in rule system these tend to be implicitly represented by various rules (or) and various premises (and). In addition, atoms must be represented using the provided type `Term` and rules using the provided type `Predicate`. This is neither flexible nor functional. Although our system uses a similar representation in the backend, users of Spivey and Seres' system must deal explicitly with the `Term` type, while users of our system need not ever know it exists.

Spivey and Seres' work is extended by Claessen and Ljunglöf [5]. The latter provides a complete environment where logical programs can be defined and solved. However, defining custom functions to operate on logical types is not easy, as no delay functionality is provided. To use additional pre-defined list functions, such as `reverse`, these would have to be redefined in a logical representation or added to the library itself.

By contrast, we present a method which allows a near mechanical transformation of rules into code. This code is represented using a DSEL, that is, a domain-specific embedded language which takes full advantage of Haskell functions, syntax, and type checking, but which is not overly burdened with housekeeping concerns. Our system allows the user to choose their own data structures (not just lists) and use their own or built-in functions which operate on those structures (not just those we have provided).

Consider the rules defining a type system for lambda calculus. These rules define the relation between an environment, a term, and a type in the form $\Gamma \vdash e :: t$. In our approach, we encode the relation as a function. In this case, we choose to take an environment and a term as input and produce a type as output. In general, any relation must be broken into input and output portions before being represented in our system. This can be a weakness compared to general logic programming, as a new rule system must be generated for each relation of input and output types. However, for our purposes, we find this limitation to be not a problem. Each rule is then translated into a function, named after the rule, which takes a value (or tuple of values) and produces a monadic computation of the output type. For lambda calculus, we take a tuple of environment and term

`(Env, Expr)` and produce a type `U Type`. The `U` monad is discussed in detail in Section 3.

In this example, the environment is simply a list of identifiers and types. An expression can consist of a variable, an application, or a lambda abstraction. Types are limited to several base types, functions, and also logical variables (`LVar`), which are used for unification.

```
data Expr = Var String
          | App Expr Expr
          | Abs String Expr

data Type = TVar LVar
          | TInt
          | TBool
          | Fun Type Type

type Env = [(String, Type)]
```

To translate a rule, a function is created whose arguments match the input part of the conclusion Consider, for example, the rule for application in lambda calculus.

$$\frac{\textsc{App}\quad \Gamma \vdash e :: t' \to t \qquad \Gamma \vdash e' :: t'}{\Gamma \vdash e\ e' :: t}$$

We create the following function to represent this rule. In order to avoid a runtime pattern match failure, each rule which depends on matching an input pattern must have a catch-all case which can simply fail in the `U` monad using the `none` value. The `U` monad, described in detail later, is a non-deterministic backtracking monad which stores logical substitutions and a counter for logical variables. The `none` value is a synonym for `mzero`, which terminates the current branch of searching.

```
app (env, App e e') = do
          [t, t'] <- newVars 2
          (env, e) .>. Fun t' t
          (env, e') .>. t'
          return t
app _ = none
```

We observe that both premises are written using the `.>.` function which associates an input with a desired output.

```
(.>.) :: Judgment i o => i -> o -> U ()
```

The precise definition of `.>.` and `Judgment` will be given later.

Logical variables that are used are introduced using the `newVars` function. The `newVars` function returns a list of fresh variables. In this case, we ask for two fresh variables. Originally, we attempted to use lazy evaluation to return an infinite list of fresh variables, but since these are generated monadically, this was not successful. The final output part of the conclusion is then indicated at the end of the rule. The `do` notation is used to sequence monadic values, and the `return` function encapsulates a result value into the monad.

In contrast to many embeddings of logic programming in functional languages, our DSEL does not attempt to represent general logic programs, but takes a syntax-directed approach. That is, the application of a rule (or perhaps several) are decided by the current working value. Thus, in type inference, the application rule is tried only when an application is being considered.

For completeness, here are the rules for variables and abstraction, and the functions implementing them.

$$\frac{\textsc{Var}\quad \Gamma(v) = t}{\Gamma \vdash v :: t}$$

```
var (env, Var v) =
     case (lookup v env) of
          Just t -> return t
          Nothing -> none
var _ = none
```

$$\frac{\textsc{Abs}\quad \Gamma, v :: t' \vdash e :: t}{\Gamma \vdash \lambda v.e :: t' \to t}$$

```
abs (env, Abs v e) = do
          [t, t'] <- newVars 2
          ((v,t'):env, e) .>. t
          return (Fun t' t)
abs _ = none
```

As these examples show, our DSEL completely abstracts away the work of unification and backtracking. We allow user-defined data structures with logical variables to be generated and solved. But is this sufficient? Imagine if we want to add polymorphism to the above-shown lambda calculus. First, we will extend the type representation `Type` to include universal quantification.

```
data Type = ...
          | Forall LVar Type
```

We also add a `Let` constructor to expressions.

```
data Expr = ...
          | Let String Expr Expr
```

We need some way of handling type schemas. In this case, we can define a function which operates on the environment and a given type to return a type schema that quantifies free type variables.

```
gen :: Env -> Type -> Type
gen env t = foldr Forall t (fvt \\ fve)
        where fvt = nub $ findvars t
              fve = nub $ concatMap
                    (findvars.snd) env
```

The function `findvars` finds all variables in a type.

```
findvars :: Type -> [LVar]
findvars (Fun x y) = findvars x++findvars y
findvars (TVar x)  = [x]
findvars _         = []
```

We can then modify the variable rule to generate types from universally quantified fresh variables. To be able to do this, we need to take each `Forall` constructor and replace the variable it indicates with a new, fresh variable. The replacement is performed using the function `sub`, which takes a value, and searchs the value for instances of the second parameter, replacing them with the third parameter.

```
forall :: Type -> U Type
forall (Forall t x) = do
        t' <- newVar
        let x' = sub x (TVar t) t'
        forall x'
forall x = return x
```

However, we cannot simply use `forall` as-is. The type passed into `forall` may include logical variables that have not been instantiated. Thus, we need a version of `forall` which is aware of logical variables.

In order to perform generalization a type must be fully known. It is not possible to generalize a type that still has pending substitutions. The variables (placeholders) that have pending substitution do not relate to the variables in

the environment, and so the appropriate selection of variables to generalize is not possible while there are unresolved logical variables. For example, if we wish to generalize the type of the term $\lambda x.x$, we expect to have an initial type of the form $a \to a$. However, this type does not appear until after substitution. Initially, only some logical variable b is known, with substitutions in the monad that can be unified to produce the full type.

If we mistakenly assume that this logical variable represents the type, we would thus return the general type $\forall b.b$. When the variable rule is called on such a term, with unification not yet occurred, the variable b would be updated to a fresh variable, say, c, and the constraint of $a \to a$ would be lost.

In this way, a logical-aware version of `forall` will wait until unification has produced a type before substituting fresh variables. This delay can be done automatically using lifting functions provided by the DSEL. This lifting is noted by Chin, et al. to be "important in many applications" but "non-trivial" [4].

The function `ufM` (short for unary-function-monad) lifts a monadic function with one parameter to be delayed until unification has completed as much as possible. The implementation of these lifting functions with respect to delayed computations is discussed in Section 2.

```
forall' = ufM forall
```

We can now use `forall'` in the variable rule to allow polymorphism.

```
var (env, Var v) =
     case lookup v env of
          Just t -> forall' t
          Nothing -> none
```

Likewise, simply using the `gen` function as-is would result in premature generalization. We can lift this function to wait until the type is complete before applying `Forall` constructors.

```
gen' env = uf (gen env)
```

The environment need not be a parameter to the lifted function since it is not subject to unification. The function `uf` (unary-function) lifts a function of one parameter. The distinction between `uf`, `ufM`, and other lifting functions is based on the input and output types of the function to be lifted. Previously, we used `ufM` to lift the function `forall` which has the type `Type -> U Type`. In general, `ufM` will lift any function which has the type `a -> U a`. Likewise, `gen env` has the type `Type` $\to$ `Type`. A function of type `a -> a` is lifted using the `uf` function. We can now use the `gen'` function in the rule to allow let polymorphism.

$$\frac{\Gamma, v :: t' \vdash e' :: t' \qquad \Gamma, v :: gen(\Gamma, t') \vdash e :: t}{\Gamma \vdash \textbf{let } v = e' \textbf{ in } e :: t} \text{ LET}$$

Using the `gen'` function, we translate this rule in the same way as the others.

```
letr (env, Let v e' e) = do
          [t, t'] <- newVars 2
          ((v,t'):env, e') .>. t'
          c <- gen' env t'
          ((v, c):env, e) .>. t
          return t
letr _ = none
```

To complete the definition of the type system all the rules have to be collected in a list called `rules`.

```
rules :: [(Env, Expr) -> U Type]
rules = [var, app, abs, letr]
```

The DSEL then allows inference to be applied to some input value to produce a possibly empty list of output values. An empty result list would occur, for example, if a term was type incorrect. When, for example, we ask for the type of the term $\lambda x.x$.

```
> infer ([], Abs "x" (Var "x"))
[Fun (TVar (VarRep 2)) (TVar (VarRep 2))]
```

The `infer` function is provided by the DSEL and applies the given rules to an input value. In this case, the input value is the empty environment and the identity function. One output is produced, which tells us that the identity function has the type $a \to a$. We could also try applying the identity function to a variable which we define in the environment.

```
> infer ([("t", TBool)],
      App (Abs "x" (Var "x")) (Var "t"))
[TBool]
```

Using our rule DSEL, the type inference algorithm for lambda calculus, and indeed, many others, can be specified in a simple way, directly derived from the inference rules themselves with minimal overhead.

In the rest of this paper, we will first discuss the high-level design and use of the rule DSEL in Section 2. Next, we will describe some of the low-level details of the unification monad in Section 3. In Section 4, we will also show how experimental type system extensions can easily be modeled and modified using the rule DSEL. A functional-logical programming example is shown in Section 5. Related work will then be discussed in Section 6. Finally, we will give concluding words in Section 7.

2. The Rule DSEL

The Rule DSEL is built on three main components: a unification algorithm, a substitution solver, and a non-deterministic state monad. Several primitive operators are also provided to help the translation from rule systems directly into DSEL code. We describe the DSEL and its implementation using a simple application of finding free variables in a lambda expression. Although finding free variables can easily be done simply by using functions, it serves here to demonstrate the concepts underlying our approach. The semantic rules for free variables are shown in Figure 1.

$$\frac{}{v \vdash \{v\}} \text{ VAR} \qquad \frac{e_1 \vdash V \qquad e_2 \vdash V'}{e_1\ e_2 \vdash V \cup V'} \text{ APP} \qquad \frac{e \vdash V}{\lambda v.e \vdash V - \{v\}} \text{ ABS}$$

Figure 1. Free variable rules.

We first need a representation for expressions.

```
data Expr = Var String
          | App Expr Expr
          | Abs String Expr
```

The free variable predicate associates expressions with sets of variable names. The function representing the predicate will therefore produce a list of strings, each representing a variable name. For union we will simply use the list concatenation operator combined with `nub` to remove duplicates. Any data structure to be used as an output type, however, must support logical variables.

There are two main ways of storing a logical variable within a data type. The standard way is to add a constructor for the logical variables. However, if using built-in data types, such as lists, this can be awkward since a new version of the data type must be constructed.

```
data LList a = LCons a (LList a)
             | LNil
             | LVar LVar
```

This can be very tedious, especially if there are a number of different data types. An alternate approach, devised by Chin et al. [4], is to place the logical variables outside the original data structure.

```
data L a = LV LVar | V a
```

In the data structure `L`, `LV` indicates a logical variable, while `V` indicates a value. This approach has the advantage that built-in data types, such as lists, can be used without defining a whole new structure. However, it does not allow using logical variables inside a data structure, such as is required with type inference. Thus, both approaches have merit. Our system does not require one or the other, but supports both by allowing any user defined data structure to be used.

We choose to use the second representation (`L a`) for logical variables rather than creating a new list data type, because we don't need embedded logical variables (a list where some elements are explicitly given and others are logical) for this example. In such a case, the second representation is simpler. Sometimes, embedded logical variables are needed, such as in type inference. For example, consider the definition of types with variables. We can define $a \to b$ using `Fun (LVar ...) (LVar ...)` in the first representation because the logical variable constructor fits with the type expected by the `Fun` constructor. Such a specification, however, is not possible with the second representation. Thus, while the second representation may be convenient, it is sometimes insufficient to represent all desired terms.

The relationship of input and output types in rules is formalized with the type class `Judgment`, which allows a set of rules to be associated with a typed relationship. An n-ary relationship is represented with $n-k$ arguments and k result types. The user must therefore decide in advance which types in the relationship are part of the input and which are part of the output. This requirement is a critical difference between our system and general logic programming. In this example we have one input type (`Expr`) and one output type (`[String]`). Examples with more argument and result types will be shown later in the paper. The requirement `Unifiable` is shorthand for the various class memberships that the unification algorithm requires. The monad `U` (which stands for unification) is a non-deterministic state monad which contains current substitutions and delayed computations. The `rules` list of functions indicates all of the rules, each of which takes an input `i` and produces a monadic output. The types `i` and `o` together represent the n-ary relationship.

```
class (Eq a, Data a, Read a) => Unifiable a

class Unifiable o =>
      Judgment i o | i -> o  where
        rules :: [i -> U o]
```

In our example, there are three syntactic constructs to handle, so there will be three rules. For variables, we simply return the variable. However, since the result type is not simply a list, but a list wrapped in a logical structure, we have to account for this representation in constructing the value. We allow users to define an isomorphic relationship between a base data structure (which does not need to contain logical variables) and the non-logical subset of the extended version, which does.

```
class Isomorphic a o | a -> o where
        to :: a -> o
        from :: o -> a
```

Thus, we need only establish the isomorphic representation and then write the rules.

```
instance Isomorphic [a] (L [a]) where
        to = V
        from (V x) = x
```

In this case, we can convert from the simple version using a plain list to the logical-aware version using the function `to`. This function will continue to operate correctly even if we decide later to change the logical-aware representation. The first rule, `var`, simply indicates that a variable is itself free. The `to` function is used to transform the list into the chosen logical-aware representation.

```
var (Var x) = return (to [x])
```

For application, we acquire the free variables of the function and its parameter, and concatenate them together, removing duplicates. However, we need some way of performing list concatenation (`(++) :: [a] -> [a] -> [a]`) on lists ensconced within the logical structure. In other words, we need a function that takes parameters of type `L [a]` in this case, or `LList a` if we had chosen to add a constructor for the logical variables within the data type.

The `Isomorphic` class is used for automatically lifting functions on type `a` into those on type `o`, and embedding them into the monad. Since we have defined an isomorphism between `[a]` and `L [a]`, we can simply use the binary-function lifting operator `bf`. Each lifting function requires its parameters to be members of the `Isomorphic` type class. This function delays the function in the monad until substitutions are as resolved as possible, and then uses the isomorphic relationship to translate the parameters to the non-logical version and back again. In particular, the monad runs through the rules, collecting substitutions and delayed functions. All substitutions are then resolved as much as possible, then each function is applied in sequence, followed by any additional possible resolution of substitutions.

For example, imagine we are using the representation `L [Int]` and wish to concatenate two lists. We could easily have the following values:

```
l1 = LV (VarRep 1)
l2 = V [1,2,3]
```

The first is merely a logical variable, the second an instantiated list. Both are of type `L [Int]`. How can these two be concatenated? In their current form, this is simply not possible. If we were assured that there would be no logical variables, we could write a function which concatenated the lists.

```
lcat (V x) (V y) = V (x++y)
```

This is fine, except there is no such guarantee that logical variables will not be present. The way to attain it is to apply unifications to reduce logical variables, and then apply the concatenation. For example, perhaps there exists a substitution `VarRep 1` $\mapsto$ `[4,5]`. In this case, we would first use unification to reduce `l1` to `[4,5]` and then apply the function `lcat`. The lifting functions combines these operations, creating a new function (like `lcat`) based on the given function and performing unification to remove logical variables from the parameters. Lifting is needed whenever logical variables may be present in the input of a function.

Unification operates on a single structured representation (explained in the next section). The function `mkGS` converts any unifiable type into a structure of type `GS` (generic structure), which unification operates on. Lifting functions, in general, first transform the input types into a type `GS` using `mkGS`. Since these terms may include logical variables, the function cannot be applied to them. Instead, the function and its parameters are stored. This combination of a function and its parameters (which need to be unified before they are applied) is called a *package*, and is represented by the type `GFS` (generic function structure). As packages may contain functions which take different numbers of parameters, the parameters are represented as a list. This allows one type (`GFS`) to hold unary functions, binary functions, and so on.

```
newtype GFS = GFS ([GS] -> U GS, [GS])
```

In order to construct the function, all the parameters must be taken as a list, converted from `GS` back to their original type (using the `from` function), the function applied to them, and then the result converted back into the `GS` type.

In the implementation of `bf` shown below, the function `f'` cannot be immediately evaluated since `x'` and `y'` still may have logical variables and thus cannot be converted using `frGS`. Instead, the `bind` function takes the package (consisting of a function and its parameters) and stores it in the monad where it will be evaluated after the parameters have undergone unification. The DSEL provides several lifting functions (each for a different function type), and additional lifting functions can be written following this pattern. One such function we have seen before is `bf` (binary-function lift). Its implementation is shown below as an illustration. The complete implementation of all lifting functions is available in the source code [17].

```
bf :: (Isomorphic a o, Wrapable o, Unifiable o) =>
      (a -> a -> a) -> o -> o -> U o

bf f x y = do
   a' <- newVar
   bind (GFS (f', [x',y'])) a'
   return a'
   where   x' = mkGS x
           y' = mkGS y
           f' = (\[x,y]->return $
                mkGS $ to $ f
                (from $ frGS x)
                (from $ frGS y)
```

Continuing our example, we can define `.++.`, which can be used in place of `++`, and all the unification issues will be resolved automatically. Likewise, a similar lifting function for unary functions can be used to lift the `nub` function, which removes duplicates from a list.

```
(.++.) = bf (++)
nub' = uf nub
```

The only remaining obstacle to defining the application rule is the ability to bind the recursive results of the two subexpressions to logical variables. First, all logical variables to be used must be generated.

Within a rule, fresh logical variables are generated using `newVar`, or its plural version. It generates a monadic, new `LVar` (which is simply an integer wrapped in a special type for the unification algorithm's use).

```
class Wrapable o where
        wrap :: LVar -> o

newVar :: Wrapable o => U o
newVar = newLVar >>= return . wrap
```

The class `Wrapable` defines how a logical variable is stored within a data type. The `Wrapable` class can easily be instantiated for both versions of logical lists shown previously.

```
instance Wrapable (LList a) where
        wrap = LVar

instance Wrapable (L a) where
        wrap = LV
```

The application rule can then be defined by tying each subexpression to a variable (representing a list) and concatenating them. Here again lifting is clearly necessary, as otherwise `v` and `v'` would simply be variables. After unification, they become lists.

```
app (App e1 e2) = do
        [v,v'] <- newVars 2
        e1 .>. v
        e2 .>. v'
        (v .++. v') >>= nub'
app _ = none
```

Besides using functions and encapsulated logical variables in data structures, the main thrust of the DSEL is tying input values to output values. This is done using the function `suppose`, or `.>.` in infix. The `suppose` function first takes the input value and recursively applies the rules to it to find all possible outputs. Next, these outputs and the desired output are both converted to the unification type and unified.

If unification succeeds (meaning the desired output and actual output are compatible), the generated substitutions are added to the monad. Otherwise, `none` indicates that this branch is not successful.

```
suppose :: Judgment i o => i -> o -> U ()
suppose input output = do
        iv <- try input rules
        let iv' = mkGS iv
        let output' = mkGS output
        case gunify iv' output' of
                Just x -> merge x >> return ()
                Nothing -> none

(.>.) = suppose
```

Finally, for lambda abstractions, we consider the free variables in the lambda body, minus the bound variable. We also need to lift the list difference function `\\` using `bf`.

```
(.\\.) = bf (\\)

abs (Abs v e) = do
        v' <- newVar
        e .>. v'
        v' .\\. (to [x])
abs _ = none
```

We then complete the relationship between the input type `Expr` and the output type `L [String]` by defining an instance of the type class `Judgment`.

```
instance Judgment Expr (L [String]) where
        rules = [var, app, abs]
```

Once the rules are complete, the DSEL provides a "run" function `infer` which takes a given input and determines all successful outputs based on the rules. The definition of this and other functions can be found in the source code [17].

```
infer :: Judgment i o => i -> [o]
```

The `infer` function returns the empty list if no result can be constructed. For example, attempting to determine the type for a type-incorrect input would result in the empty list. Multiple results can be returned if there are multiple paths through the rule system which produce different valid results.

With `infer` we can determine the free variables of various expressions. The `V` constructor in the return values is due to the container for values and logical variables. In each case, one result is returned, showing the list of free variables (if any) in the expression.

```
vx = Var "x"
vy = Var "y"

> infer vx
[V ["x"]]

> infer (Abs "x" vx)
[V []]

> infer (App vx (Abs "x" (App vx vy)))
[V ["x","y"]]
```

3. Supporting Monad and Unification

The basis of the search and unification is the nondeterministic state transformer monad. This monad carries a function from initial state to final state. The monad also instantiates the `MonadPlus` class, which allows searching and termination of branches when a solution is not possible. Note that for simplicity, we are using a naive backtracking monad. For implementation purposes, a more efficent monad, such as the on given by Hinze [10], would be appropriate.

```
newtype NDSM s a =
        NDSM {unNDSM :: (s -> [(s,a)])}
```

The monad instance contains three pieces of state: a counter for producing new logical variables, a list of unification substitutions, and a list of delayed computations and their parameters. Recall that `GS` is the generic structure for any type and `GFS` holds lifted functions.

```
type U b =
    NDSM (LVar, [(GS,GS)], [(GS,GFS)]) b
```

Substitutions are generated using the `suppose` operator. For example, if we infer the type of $\lambda x.x$, we start with its representation `Abs "x" (Var "x")`. We first apply the Abs rule, which generates two new variables t and t'. Assuming that x has type t', we infer the type of x (represented by `Var "x"`) using the Var rule. The Var rule looks in the environment and finds that x has the type t'. The variable t' is bound (through the premise of the Abs rule) to t. The resulting type $t' \to t$ is returned. This leaves a final set of substitutions like `[(a, Fun t' t), (t, t')]` The unification algorithm reduces these substitutions and determines that the principal type (as given by `a`) is `Fun t' t'`, or $a \to a$.

The logical variable counter is kept through incrementing an integer every time a variable is handed out. We make `LVar` an instance of `Enum` so that the successor function can be used to generate new variables. The function `newLVar` increments the current max variable counter and returns the new variable.

```
newtype LVar = VarRep Int

instance Enum LVar where
        fromEnum (VarRep x) = x
        toEnum x = VarRep x

newLVar :: U LVar
newLVar = NDSM (\(i,x,z)->
          [((succ i,x,z),succ i)])
```

The list of substitutions is generated by the unification algorithm at each `suppose` and merged in by partial evaluation to ensure that the substitutions are in the most reduced form possible. Once all substitutions have been received (the input value fully handled by the rules) and all substitutions resolved as much as possible, the delayed functions are executed.

Consider the example of list concatenation. In this case, we wish to take two lists and generate a new list. However, the lists exist only in substitutions. For example, we may have the substitutions $\{$`a` $\mapsto$ `[1,2,3]`, `b` $\mapsto$ `[4,5,6]`$\}$ and the concatenation `a .++. b`. Since `a` and `b` are logical variables (of type `LVar`), they cannot be concatenated. Instead, unification must be applied, and then the result can be concatenated and bound to a new variable, such as `[(c, [1,2,3,4,5,6])`.

Delay of function calls is essential for two purposes. First, many functions (such as list concatenation) cannot

operate on logical variables—all logical variables must be instantiated so that the value can be converted to the isomorphic representation, which does not include logical variables. This can only happen when substitutions to fully instantiate a term are available. For every logical variable in a term, there must exist a substitution to replace it (perhaps transitively) with a non-logical value. Second, some operations (such as type generalization) would be too broad if applied to a logical variable, and must instead be applied to the most constrained possible instantiation. For example, if a substitution ties b to $a \to a$, and this type is to be generalized, it is important that unification occurs and the type $\forall a.a \to a$ be generated and not $\forall b.b$.

The result of the function is then paired with the original value and used to further evaluate substitutions. If at any time contradictory substitutions are detected, that branch of searching is terminated.

Once all delayed functions have been executed and each branch has produced its result, all results are grouped together and are returned as a list by the previously mentioned `infer` function.

The unification algorithm is quite standard, except that it must operate on a wide variety of data types. Normally, this could be handled with Haskell generics, as attempted in [16]. However, this approach has a critical weakness: Since we need to store a list of unification substitutions, all the substitutions must be of the same type. If a data structure contains items of a different type, the substitutions will not be homogeneous. Consider the following data structures.

```
data A = A1 Int | A2 B
data B = B2 Int
```

In this case, values of type `A` may contain subvalues of type `B`. A list of substitutions of type `[(A,A)]` would not be able to store a substitution for any values on type `B`. Many approaches, such as those discussed in the Introduction, force the user to adopt an untyped representation to deal with this problem. We want to allow such data types to be used automatically, and so generic programming is not suitable for our DSEL.

Instead, we automatically convert typed data structures to and from an untyped representation. This representation ensures that the structures will remain type correct since all external usages are protected by Haskell typing rules. We first note that, fundamentally, all data structures are either some primitive value (integer, character, etc.) or a constructor with zero or more parameters. We retain the name and infix status of a constructor. The latter is important for converting back to the type. In addition, we retain a marking for logical variables, which are recognized in the unification algorithm. The `GS` type need only support all primitive types, along with a method for representing logical variables and constructors, to be complete. The user does not need to add to this type to represent their data.

```
data GS = GSVar Int
        | GSCons CS [GS]
        | GSInt Int
        | GSChar Char
        | ...

data CS = CS {csInfix :: Bool,
              csName :: String}
```

This embedding of logical variables allows partially constructed instances that are not possible in some systems which require logical variables to be separate from the data types they represent, as discussed in [4].

To convert from a given data type to the general type `GS`, we use Haskell generics to traverse the structure.

```
mkGS :: Data a => a -> GS
```

In order to convert back, however, we cannot use Haskell generics since more than one type may be involved. Therefore, we construct a string representation which is then read back into the data type. This is where the infix status of the constructor is involved. Lists are handled as a special case since they use a non-standard representation. The definition of `frGS` is given in the source code [17].

```
frGS :: Read a => GS -> a
```

We then apply the standard unification algorithm to values of type `GS` to attain substitutions.

```
gunify :: GS -> GS -> Maybe [(GS,GS)]
```

The substitution solver takes a list of unification substitutions and attempts to determine a canonical instance for each logical variable, that is, it attempts to resolve all open unifications. If this is possible, the substitutions are compatible and the solver returns the list of canonical instances. Otherwise, the solver fails.

4. An Application: Type Change Inference

In this section, we discuss an application of the rule DSEL in language prototyping. We show how to extend the lambda calculus type inference system to permit type error corrections to be inferred automatically. We then show an alternative approach to this problem and how it is very simple to transform the code to match the new approach.

Since the earliest automated type inference algorithms, researchers have attempted to improve type error messages. The primary problem has been that type errors reflect a compiler's understanding and may not necessarily point out what change a programmer should make. In many cases, the error messages include terminology specific to the type-inference algorithm and are distant from the actual code the programmer is attempting to debug. McAdam indicated that type errors should be more like feedback from a spell checker—offering concrete corrections without requiring understanding of the inner workings of the type system [14].

One approach to solving this problem is to model a type system that accepts all programs and returns not only a type but a set of changes which would make the program correct in the traditional Hindley/Milner system. This can be accomplished by returning a pair consisting of a type and a type-change expression. A type-change expression consists of an expression and two types. These three together mean to change the given expression from one type to any expression of the other type. Since these change expressions will be part of the output, they will also need to support logical variables.

```
data Expr = ...
          | Chg Expr Type Type
          | EVar LVar
```

Although we are adding only one (non-logical variable) constructor to `Expr`, an entirely new set of rules must be created to output a type and an expression instead of just a type.

We can model these rules on the original Hindley/Milner inference rules, extending them to include δ, an expression with changes. The formal syntax of change expressions is shown in Figure 2.

$$\delta \ ::= \ \delta \ \delta \mid \lambda v.\delta \mid \textbf{let } v = \delta \textbf{ in } \delta \mid \delta :: t \rightsquigarrow t$$

Figure 2. Syntax of type-change expressions.

For each Hindley/Milner rule, we also consider how it could "go wrong". For example, the VAR rule only works if the variable is in the environment. Therefore, we introduce a second rule, UNDEF, for the case when the variable is not in the environment. The application rule can go wrong if the function doesn't have a function type, or if the parameter doesn't match the function type. Therefore, we add two rules, PAR and ARG, which suggest changing the function and its argument, respectively. The rules define judgments of the form $\Gamma; \delta \vdash e :: t$ which say that given an environment Γ and a type-change expression δ we can derive a type correct expression e that has type t. The complete rules are shown in Figure 3.

The UNDEF rule also requires that we add a constructor for the undefined type to `Type`.

```
data Type = ...
          | Undef
```

In order to represent this system in our DSEL, we first need to determine the input/output relation we wish to use. The parameters in the rule system are environment, type-change expression, expression, and type. Of these, we will provide as input the environment and the expression, and receive as output the change expression and its type.

We then need to convert functions (or other non-recursive uses) into Haskell code and lift if necessary. Lifting is needed whenever a parameter may include a logical variable. In the above example, there are three functions: $\Gamma(v)$, which looks up a variable in the environment, $v \notin dom(\Gamma)$ which determines that a variable is not present in the environment, and *gen*, which we have previously discussed.

Determining whether or not a variable is present and finding its value can be accomplished with the function `lookup`. We can use both cases to represent VAR and UNDEF with a single rule function.

```
var (env, va@(Var v)) =
    case lookup v env of
        Just t -> do
                t' <- forall' t
                return (va,t')
        Nothing -> do
                t <- newTVar
                return (Chg (Var v) Undef t, t)
var _ = none
```

The `Just` case is essentially the same as the original type inference rules, except that it returns both the expression and its type. If the variable is not found in the environment, a change expression is generated, along with an unbound type. The unbound type will be constrained by its use, and this will be propagated automatically into the change expression.

The application rule introduces a twist: Although we do not have any changes to the application, there may be changes needed in either subexpression. Therefore, the expressions cannot be returned as they are—instead, logical variables must be used. There are two separate lists of variables generated, one being variables for types (`t` and `t'`) and the other being variables for expressions (`d` and `d'`). Since these variables have different types (`Type` and `Expr`, respectively), we must generate them separately.

```
app (env, (App e e')) = do
        [t',t] <- newVars 2
        [d',d] <- newVars 2
        (env, e) .>. (d, (Fun t' t))
        (env, e') .>. (d', t')
        return (App d d', t)
app _ = none
```

The rule system also states two ways this could "go wrong", as indicated in the PAR and ARG rules. In order to enforce the inequality constraint, we can use the DSEL-provided not equals operator, `./=.`, which for logical applications means "cannot be unified". For the ARG rule, we assert that the argument may be of some arbitrary type that does not match the function parameter. In that case, we change the argument to match.

```
arg (env, (App e e')) = do
        [t,t',t''] <- newVars 3
        [d,d'] <- newVars 2
        (env, e) .>. (d, (Fun t' t))
        (env, e') .>. (d', t'')
        t' ./=. t''
        return (App d (Chg d' t'' t'), t)
arg _ = none
```

The PAR rule starts out the same—it assumes the same premises, only returning a different conclusion. This time the change brings the function in line with the argument.

```
par (env, (App e e')) = do
        [t,t',t''] <- newVars 3
        [d,d'] <- newVars 2
        (env, e) .>. (d, (Fun t' t))
        (env, e') .>. (d', t'')
        t' ./=. t''
        return (App (Chg d
          (Fun t' t) (Fun t'' t)) d', t)
par _ = none
```

A similar transformation is permitted for the LET rule—we can change the definition to an arbitrary new type using the DEF rule.

```
def (env, (Let v e' e)) = do
        [t,t',t''] <- newVars 3
        [d,d'] <- newVars 2
        ((v,t'):env, e') .>. (d', t'')
        m <- gen' t'
        ((v, m):env, e) .>. (d, t)
        t'' ./=. t'
        return (Let v (Chg d' t'' t') d,t)
def _ = none
```

The LET rule appears similarly, without the change in types. Since `let` is a reserved word in Haskell, we name the function for this rule `letr`.

$$\textsc{Var}\ \frac{\Gamma(v) = t}{\Gamma; v \vdash v :: t} \qquad \textsc{Undef}\ \frac{v \notin dom(\Gamma)}{\Gamma; v ::\rightsquigarrow t \vdash v :: t} \qquad \textsc{App}\ \frac{\Gamma; \delta \vdash e :: t' \to t \qquad \Gamma; \delta' \vdash e' :: t'}{\Gamma; \delta\ \delta' \vdash e\ e' :: t}$$

$$\textsc{Arg}\ \frac{\Gamma; \delta \vdash e :: t' \to t \qquad \Gamma; \delta' \vdash e' :: t'' \qquad t' \neq t''}{\Gamma; \delta\ (\delta' :: t'' \rightsquigarrow t') \vdash e\ e' :: t}$$

$$\textsc{Par}\ \frac{\Gamma; \delta \vdash e :: t' \to t \qquad \Gamma; \delta' \vdash e' :: t'' \qquad t' \neq t''}{\Gamma; (\delta :: t' \to t \rightsquigarrow t'' \to t)\ \delta' \vdash e\ e' :: t} \qquad \textsc{Abs}\ \frac{\Gamma, v :: t'; \delta \vdash e :: t}{\Gamma; \lambda v.\delta \vdash \lambda v.e :: t' \to t}$$

$$\textsc{Let}\ \frac{\Gamma, v :: t'; \delta' \vdash e' :: t' \qquad \Gamma, v :: gen(\Gamma, t'); \delta \vdash e :: t}{\Gamma; \textbf{let}\ v = \delta'\ \textbf{in}\ \delta \vdash \textbf{let}\ v = e'\ \textbf{in}\ e :: t} \qquad \textsc{Def}\ \frac{\Gamma, v :: t'; \delta' \vdash e' :: t'' \qquad \Gamma, v :: gen(\Gamma, t'); \delta \vdash e :: t \qquad t'' \neq t'}{\Gamma; \textbf{let}\ v = \delta' :: t'' \rightsquigarrow t'\ \textbf{in}\ \delta \vdash \textbf{let}\ v = e'\ \textbf{in}\ e :: t}$$

Figure 3. Type-change-constrained typing rules.

```
letr (env, (Let v e' e)) = do
        [t,t'] <- newVars 3
        [d,d'] <- newVars 2
        ((v,t'):env, e') .>. (d', t')
        m <- gen' t'
        ((v, m):env, e) .>. (d, t)
        return (Let v d' d,t)
letr _ = none
```

Abstraction has no particular point of change, except perhaps the body. Since there are no detailed constraints placed on the body, we let recursive rules do the work to make changes within the body of a lambda abstraction.

```
abs (env, (Abs v e)) = do
        [t,t'] <- newVars 2
        d <- newVar
        ((x,t'):env, e) .>. (d, t)
        return (Abs v d, Fun t' t)
abs _ = none
```

We then define a `Judgment` instance from environment and expression to expression and type.

```
instance Judgment (Env,Expr) (Expr,Type) where
        rules = [app, var, abs,
                 letr, arg, par, def]
```

We can then use this rule system to correct type errors in expressions. To illustrate the working of the system, we define a simple base environment with several constants.

```
baseenv = [("1", TInt), ("t", TBool),
        ("plus", Fun TInt (Fun TInt TInt)),
        ("not", Fun TBool TBool)]
```

The standard Hindley/Milner rule system would return only one output for a type-correct expression, and no outputs for a type-incorrect expression. The described type-change system may return several outputs for a type-incorrect expression, each representing a different way to correct the problem.

```
> infer (baseenv, (App (Var "not") (Var "1")))
[(not (1 :: Int ~> Bool),Bool),
 ((not :: Bool->Bool ~> Int->Bool) 1,Bool)]
```

For the expression `not 1`, the system recommends either replacing the `1` with a Boolean value, or replacing `not` with a function that accepts an integer.

The system also suggests multiple changes as needed to make an expression type correct. Consider the expression `plus t t`. In this case, either the `plus` needs to change, or *both* the parameters.

```
> infer (baseenv,
 (App (App (Var "plus") (Var "t")) (Var "t")))
[(plus (t :: Bool ~> Int)
  (t :: Bool ~> Int),Int),
 ...]
```

We used this implementation to experiment on various type errors. This system was found to work reasonably well. In some cases, however, it showed poor performance. After some discussion, we realized there is a simpler way to model type changes—instead of having a set of rules for all the possible "go wrong" scenarios, simply allow the variable rule to introduce either a change or not. This rule system has the advantage of being simpler (fewer rules), but the disadvantage of potentially introducing changes when they are not needed. Would this new representation improve or hurt performance?

Fortunately, our DSEL allows models to be constructed and modified quickly. We can easily alter the system shown so far to make changes only at variables. First, we need to add a new variable change rule that indicates a change of type.

$$\textsc{VChg}\ \frac{\Gamma(v) = t}{\Gamma; v :: t \rightsquigarrow t' \vdash v :: t'}$$

Since the DSEL automatically tries all rules, each variable reference will match two rules: the traditional variable rule Var and our new variable change rule VChg.

```
vchg (env, (Var x)) = case lookup x env of
        Just y -> do
                a <- newVar
                y' <- forall' y
                return (Chg (Var x) y' a, a)
        Nothing -> none
vchg _ = none
```

We can now easily model an alternate system allowing changes only at the variable level by removing other change rules (Par, Arg, and Def) and adding VChg.

```
instance Judgment (Env,Expr) (Expr,Type) where
        rules = [app, var, abs, letr, vchg]
```

As expected, this new system returns the same results as the original one. But what about performance? We found that

this reduced rule system performed better than the original type change inference system in some cases, but worse in others. A fast algorithm to predict which system is faster in any given case remains under development.

The point of this example is that our DSEL allowed us to easily prototype a type system. When a question of representation arose, we were easily able to consider the alternative approach and test the performance differences between them. Thus, the rule DSEL supported rapid prototyping of a type system based on logic rules.

The DSEL could also be used to represent typing rules for more complex languages, such as Haskell. Jones [12] gives a method for typing Haskell in Haskell. However, he is forced to re-introduce well-known methods of unification and substitution which are not central to the application. Using the DSEL, the author could represent the typing rules for Haskell directly without concern for the underlying mechanisms.

5. Functional-Logic Programming

It is also possible to represent some more general logical applications using a portion of the rule DSEL. Consider the locally defined global identifier pattern described by [1]. In this case, we want to manipulate graphs and always ensure that each node is uniquely labeled.

```
data Node = Node Int

data Edge = Edge Node Node

data Graph = Graph [Node] [Edge]
```

Imagine a simple graph of three nodes.

```
g1 = Graph [n1, n2, n3]
        [Edge n1 n2, Edge n3 n2, Edge n3 n3]
        where [n1,n2,n3] = map Node [1..3]
```

Now imagine we want to connect graph `g1` to another graph which also has nodes, say, 1 and 2. In that case, there would be an overlap of node labels, and the edges would become ambiguous. In order to prevent this, the graphs must be relabeled at composition time, requiring an expensive composition function. Any function which manipulates two graphs together would have to relabel the graphs to avoid a potential name collision.

Imagine, in particular, we want to connect two graphs together with a single arc between the first nodes in each graph. We would prefer to just define the function `connectGraphs` as follows:

```
connectGraphs (Graph ns1 es1) (Graph ns2 es2) =
        Graph (ns1++ns2)
              (Edge (head ns1) (head ns2):
               es1++es2)
```

Now if we try `connectGraphs g1 g1` we get a mess that does not correctly represent the resulting graph, due to the lack of relabeling. We can avoid this problem by using logical variables. We replace integer constant node identifiers by logical variables to allow them to be uniquely named automatically.

```
data Node = Node LVar
```

Thus, instead of a node being represented by `Node 1` or `Node 2`, it is now wrapped in a logical variable and generated monadically, so the first node will be `Node (VarRep 1)` and the second `Node (VarRep 2)`, and so on. Since this is a clumsy representation, we can define a `Show` instance for `Node`.

```
instance Show Node where
        show (Node (VarRep x)) = show x
```

Next we define a graph using the `newVars` function to label the nodes.

```
g2 = do [n1,n2,n3] <- newVars 3
        return (Graph [n1, n2, n3]
                [Edge n1 n2, Edge n3 n2,
                 Edge n3 n3])
```

We can then use the Haskell-provided monadic lifting function `liftM2` to automatically transform `connectGraphs` to operate on monadic graph values.

```
cg = liftM2 connectGraphs
```

At this point we have a function which produces a value of type `U Graph`. Somehow we need to extract the actual graphs. In a case where the `Judgment` class is not being used, we provide the function `eval` to extract results.

```
eval :: Unifiable o => U o -> [o]
```

Our node labels are thus guaranteed unique.

```
> eval (cg g2 g2)
[Graph [1,2,3,4,5,6] [Edge 1 4,Edge 1 2,Edge 3 2,
                      Edge 3 3,Edge 4 5,Edge 6 5,
                      Edge 6 6]]
```

6. Related Work

The combination of functional and logic programming is not new. Early attempts to simulate logical variables with functional languages used techniques such as higher-order functions, where a data structure would be represented as a function [3].

Later, specialized functional-logic programming languages emerged. Somogyi et al. introduced Mercury [20], a functional-logic language which is under continuous development and is designed for commercial use. Mercury focuses strongly on performance, static error detection, and large system development.

Curry is another language in the functional-logic paradigm, introduced by Hanus [7]. Curry focuses on tightly integrating the functional and logical aspects to form a single, seamless methodology.

Lloyd introduced another functional-logic language called Escher [13] whose functionality arises from the Gödel logic language. The goal of Escher is to provide a research platform for the combination of functional and logic development idioms.

Although these languages all use familiar syntactic constructs, for a Haskell programmer being faced with the problem of implementing rule systems, they still require porting applications to a new language. We would prefer to be able to take advantage of certain logical features within the host language of Haskell. Thus, a functional-logical language, while interesting, is orthogonal to our goals.

Jansson and Jeuring devise an extension to Haskell called poly-types which they use to demonstrate a generic unification algorithm [11]. This extension allows unification on

all user-defined types. The authors apply unification functions to provide generics, such as determining the subterms of a term of any type, and checking two terms for top-level equality. However, the authors do not attempt to extend the unification algorithm to logic programming.

Providing logical functionality within Haskell was attempted by Spivey and Seres in [21]. Their embedding of general logical computation showed that the fundamental constructions of logic could in fact be handled by Haskell. However, their implementation leaves much to be desired. Users are forced to use the types for `Term` and `Predicate` to represent data structures to be used in logical computations. These types can be extremely cumbersome. The paper shows an example of representing the list `[a,b]` as `Func "cons" [Func "a" [], Func "cons" [Func "b" [], Func "cons" [Func "[]" []]]]`. Each predicate must then be constructed from only a few basic predicates, such as `and`, `or`, and `exists`. The authors successfully showed that logical programming in Haskell, with the help of combinators and a unification algorithm, was *possible*, but not that it could be *pretty*.

The representation shown by Spivey and Seres has the disadvantages of being both clumsy and untyped. Claessen and Ljunglöf introduced a monadic version of their work which separated atoms and lists, and enforced the typing there-of [5]. This work is both cleaner and safer. However, it still limits the user to the explicitly given functions. It would be nice if logical variables could be introduced in arbitrary Haskell structures and manipulated with arbitrary Haskell functions, while still being type safe.

A hybrid approach was taken by Chin et al. in embedding constraint handling rules into Haskell [4]. Their approach shows how to embed logical variables in user-defined data structures. However, each data structure defined by the user requires the explicit definition of an instance of the type class `Unification`. They mention that automatic code generation could alleviate this requirement. Our system uses generics to automate unification. The authors also introduce embedding and projection, which is the same idea as our `Isomorphic` class. However, the authors do not allow Haskell functions to be used from within constraint rules, calling this "non-trivial". This work later evolved in a separate language called Chameleon, which extends Haskell with constraint handling rules [22].

Sheard and Pasalic introduce a unification algorithm which operates on data types representing generic terms and structures (`GT` and `S`, respectively) [18, 19]. However, their representation may be considered only partial; both `GT` and `S` are parameterized, thus they describe a structure containing values of some given type. The authors also require the user to provide functions to transform rich values into values of the given generic types. Our approach also uses a generic representation `GS` which retains both the structure and terms all the way down to the atomic level. This approach allows terms with mixed atomic types, such as `(Int,Bool)`, to be stored as easily as any other type. These transformations are performed automatically using derived instances of `Data` and `Read`.

As functional-logic programming continues to grow, many classes of problems can be effectively solved with both paradigms at hand. A number of functional-logic design patterns are introduced by Antoy and Hanus in [2] and [1].

Hanus introduces a declarative web scripting system in [8] using Curry. Most of the system is defined functionally, with occasional bits using logical variables. Unfortunately, Curry does not include an HTML library, so the author was forced to re-invent much of the wheel. Haskell already includes libraries for HTML and CGI. Thus, our system could provide the same service as the author's without re-inventing basic combinators. Although other authors are quick to "jump ship" and create new languages to support functional-logic programming, doing so discards the wide variety of libraries and support already extant for Haskell. By embedding logical variables as needed in Haskell, our system can bridge the gap and provide light-weight logical support to what are otherwise mostly functional programs.

Our system does not attempt to compete with logical heavyweights, like Mercury or Curry, in terms of efficiency or expressive power. It does, however, offer a straightforward environment for prototyping rule systems and performing simple logical computations.

7. Conclusion

This paper introduces a domain-specific embedded language for representing rule systems in Haskell. Modeling of the classic lambda calculus type system and an extension are both shown. Using the underlying search and unification monad to perform other kinds of functional-logic programming is also possible and demonstrated.

We present a system which allows unification on user-defined data types, and most significantly, automatic lifting of Haskell and user-defined functions to work on logical structures, previously considered to be a non-trivial problem. Our system integrates these features using a simple monadic representation that closely resembles formal semantic rules, making translating rule systems into code a nearly mechanical process.

Our system allows logical programming to be performed within Haskell and without forcing the user to use only certain predefined data types or functions. This allows a convenient integration of logic programming into the functional paradigm, without having to ever leave Haskell.

References

[1] Sergio Antoy and Michael Hanus. Functional Logic Design Patterns. In *Proc. of the 6th Int. Symp. on Functional and Logic Prog.*, pages 67–87, 2002. LNCS 2441.

[2] Sergio Antoy and Michael Hanus. Concurrent Distinct Choices. *Journal of Functional Programming*, 14(6):1–12, 2004.

[3] F. Warren Burton. A Note on Higher-Order Functions Versus Logical Variables. *Information Processing Letters*, 31(2):91–95, 1989.

[4] Wei-Ngan Chin, Martin Sulzmann, and Meng Wang. A Type-Safe Embedding of Constraint Handling Rules into Haskell. Technical report, National University of Singapore, 2003.

[5] K. Claessen and P. Ljunglöf. Typed Logical Variables in Haskell. In *Haskell Workshop*, 2000. Electronic Notes in Theoretical Computer Science, Vol. 41, No. 1.

[6] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[7] Michael Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th SIGPLAN-SIGACT Symp. on Principles of Prog. Lang.*, pages 80–93, 1997.

[8] Michael Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the 3rd Int. Symp. on Practical Aspects of Declarative Lang.*, pages 76–92, 2001.

[9] Hanus, M. and Kuchen, H. and Moreno-Navarro, J. J. Curry: A Truly Functional Logic Language. In *Proc. of ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[10] Ralf Hinze. Deriving Backtracking Monad Transformers. In *Proc. of Int. Conf. on Functional Programming*, pages 18–20, 2000.

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

[12] Mark Jones. Typing Haskell in Haskell. In *Proc. of the Haskell Workshop*, 1999.

[13] John W. Lloyd. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming*, 1999(3), 1999.

[14] Bruce McAdam. How to Repair Type Errors Automatically. *3rd Scottish Functional Programming Workshop*, 2001.

[15] Pfenning, Frank and Schürmann, Carsten. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In H. Ganzinger, editor, *Proc. of the 16th Int. Conf. on Automated Deduction*, pages 202–206, Trento, Italy, 1999. Springer-Verlag LNAI 1632.

[16] PuRE Project. Data.Unification Module. `http://www.di.uminho.pt/~joostvisser/software/UMinhoHaskellSoftware-1.0/Data.Unification.html`.

[17] Haskell Rules. Rule Systems Embedded into Haskell, 2006. `http://eecs.oregonstate.edu/~erwig/HaskellRules`.

[18] Tim Sheard. Generic Unification Via Two-Level Types and Parameterized Modules. In *Proc. of the 6th ACM SIGPLAN Int. Conf. on Functional Prog.*, pages 86–97, 2001.

[19] Tim Sheard and Emir Pasalic. Two-Level Types and Parameterized Modules. *Journal of Functional Programming*, 14(5):547–587, 2004.

[20] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

[21] J. M. Spivey and S. Seres. Embedding Prolog in Haskell. In E. Meijer, editor, *Haskell Workshop*, pages 25–38, 1999.

[22] Peter Stuckey and Martin Sulzmann. A Theory of Overloading. *ACM Trans. on Programming Lang. and Systems*, 25(6):1216–1269, 2005.