Lightweight Higher-Order Rewriting in Haskell

(Draft submitted to TFP 2015)

Emil Axelsson and Andrea Vezzosi
Chalmers University of Technology

Abstract. We present a generic Haskell library for expressing rewrite rules with a safe treatment of variables and binders. Both sides of the rules are written as typed EDSL expressions, which leads to syntactically appealing rules and hides the underlying term representation. Matching is defined as an instance of Miller’s pattern unification, which makes for efficient execution when rules are applied in a bottom-up fashion. The restrictions of pattern unification are captured in the types of the library, and we show by example that the library is capable of expressing useful simplifications that might be used in a compiler.

1 Introduction

Work on embedded domain-specific languages (EDSLs) has taught us many useful techniques for constructing terms: smart constructors for hiding the underlying representation of expressions, higher-order functions to represent constructs that introduce local variables, phantom types to give a typed interface to an untyped representation, etc. Unfortunately, these techniques are only applicable to term construction, not to pattern matching. Pattern matching is needed to examine expressions; for example in transformations, interpretation or compilation.

So, although EDSL users have a very nice interface for constructing expressions, EDSL implementors are confined to working with the underlying representation. This can lead to several problems:

– Type safety: If the representation is untyped, it is easy to cause type errors when transforming expressions.
– Verbosity: The representation may be inconvenient to work with, especially if it is based generic encodings, such as compositional data types [3].
– Scoping: When transforming expressions with binders, it is easy to cause variables to escape their scope.

Although solutions or partial solutions exist for all of these problems, we are not aware of any solution in Haskell that handles all of them at once. This paper addresses all three problems in a single generic Haskell library for rewrite rules. Our library is also efficient: the complexity of rule application is determined only by the size of the rule, when rules are applied bottom-up. However, the library is restricted to plain rewrite rules – it cannot be used to define arbitrary functions on expressions.
1.1 Running Example

As our running example, we will use the functional for loop in the Feldspar EDSL [2]:

\[
\text{forLoop} :: \text{Data Int} \rightarrow \text{Data s} \rightarrow (\text{Data Int} \rightarrow \text{Data s} \rightarrow \text{Data s}) \rightarrow \text{Data s}
\]

\text{Data} is Feldspar’s expression type which is parameterized by the type of the value the expression computes. The first argument to \text{forLoop} is the number of iterations; the second argument is the initial state; the third argument is the body which computes the next state given the loop index and the current state; the result is the final state of the loop.

We are interested in expressing the following simplification rules for \text{forLoop}:

- If the number of iterations is 0, the result is the initial state.
- If the body always returns the previous state, the result is the initial state.
- If the body does not refer to the previous state, it is enough to run the last iteration of the loop.

Furthermore, we would like to express these rules in a way that

- is independent of the representation of \text{Data},
- does not allow accidentally changing the type of the expression,
- does not require looking at concrete variable identifiers,
- does not allow creating an ill-scoped expression.

To illustrate the problem, we try to express the rewrite rules as cases in a Haskell function. Assuming \text{Data} is a simple recursive data type, with constructors lambda abstraction, variables, for loops, etc., we might express the first two rules as follows:

\[
\text{simplify (ForLoop (Int 0) init _)} = \text{init}
\]

\[
\text{simplify (ForLoop _ init (Lam i (Lam s (Var s')))) \mid s == s'} = \text{init}
\]

Even though the definition looks quite readable, it violates most of our requirements on rewrite rules: it leaks the representation of \text{Data}, does not guarantee well-typedness, and involves comparing variable names.

The third rule is trickier. Using pseudo-Haskell syntax, we want to rewrite an expression of the form

\[
\text{forLoop l init (\lambda i s \rightarrow body)}
\]

to

\[
\text{cond (l==0) init (body[l\rightarrow]}\text{)}
\]

provided \text{s} does not occur freely in \text{body}. The object-level function \text{cond} is used to return \text{init} when the length is 0 and otherwise return \text{body} with \text{l-1} substituted for \text{i}.

Trying to express this rule as a case in \text{simplify} reveals an additional problem of this style of rewriting. It is possible for \text{body} to contain free variables that are bound by lambdas in the pattern. In order to prevent these variables from
escaping their scope, either we need to check for their absence or we need to substitute for these variables on the right hand side. In the case of the third for loop rule, we need to check that $s$ does not occur freely in $\text{body}$ and we need to substitute an expression for $i$ on the right hand side. Either of these actions is very easy to forget.

As a preview of our solution, here is the third forLoop rule expressed using our library:

```haskell
rule_for3 len init body =
  forLoop (mvar len) (mvar init) (\i s \rightarrow body -$-$ i)
  \Rightarrow
  cond (mvar len \equiv 0) (mvar init) (body -$-$ (mvar len - 1))
```

Note the use of Haskell’s $\lambda$-abstraction to give the pattern for the loop body. In addition to being quite close to the desired syntax, the rule is also guaranteed to be well-typed and well-scoped.

### 1.2 Overview of the Paper

Section 2 presents the basics of our rewriting library restricted to first-order matching. Section 3 revisits the general problem of higher-order matching and gives an efficient algorithm for matching and rewriting based on Miller’s pattern unification. Section 4 shows how our library can be extended to support higher-order matching. Section 5 demonstrates the library using a simple version of the Feldspar EDSL.

The rewriting library is available on Hackage.

## 2 A Generic Library for Rewrite Rules

In this section we show a first-order version of our library. The higher-order version in section 4 is mostly an extension of the definitions in this section. Only the representation of meta-variables need modification.

A rule is a pair of a left hand side (LHS) and a right hand side (RHS):

```haskell
data Rule lhs rhs where
  Rule :: lhs a \rightarrow rhs a \rightarrow Rule lhs rhs
```

The parameters $\text{lhs}$ and $\text{rhs}$ are representations of the left and right hand sides of the rule. These representations are parameterized by the type of the corresponding expression, just like $\text{data}$ in section 1.1. The type parameter is existentially quantified, and the only thing we care about is that $\text{lhs}$ and $\text{rhs}$ have the same type parameter.

Rather than using fixed types for $\text{lhs}$ and $\text{rhs}$, we will express our rules using type classes. This will allow us to use many of the same functions to express both sides of a rule, even if the two sides will in the end have slightly different representations. Using type classes also allows us to extend the rule language.

with new constructs simply by adding additional class constraints. Essentially, we regard lhs and rhs as languages in the final tagless style [5].

The first classes we introduce are for meta-variables and wildcards:

```haskell
class MetaVar r where
    type MetaRep r :: * → *
    mvar :: MetaRep r a → r a

class WildCard r where
    _ :: r a
```

The function `mvar` introduces a meta-variable given a representation for it. The reason for making the `MetaRep` an associated type is to be able to disallow inspection of the representation of meta-variables. As long as we keep `r` abstract, `MetaRep r` will also be an abstract type. The method `_` (double underscore) of the `WildCard` class constructs a pattern that matches any term. As we will see, our implementation only allows wildcards on the LHS of a rule.

Next, we introduce a convenient short hand for rules:

```haskell
(⇒) :: lhs a → rhs a → Rule lhs rhs
(⇒) = Rule
infix 1 ⇒
```

Interestingly, we now have all the machinery we need to start expressing some rules for numeric operations. Each rule is given as a Haskell definition that takes the necessary meta-variables as arguments:

```haskell
-- 0 + X ⇒ X
rule_add x = 0 + mvar x ⇒ mvar x

-- X - X ⇒ 0
rule_sub x = mvar x - mvar x ⇒ 0

-- 0 * _ ⇒ 0
rule_mul = 0 * ⇒ (id :: r Int → r Int) 0
```

How is it that we can already write rules about numeric operations without even having given a representation for the LHS and RHS of rules? Looking at the inferred type of `rule_add` tells us what is going on:

```haskell
rule_add :: (MetaVar lhs, MetaVar rhs
            , MetaRep lhs ~ MetaRep rhs
            , Num (lhs a)
            ) ⇒ MetaRep lhs a → Rule lhs rhs
```

Since the rules are expressed entirely using type class operations (including those of the `Num` class), the type is polymorphic in `lhs` and `rhs`. But we see a number of constraints due to the way the operations are used. The constraints tell us that both sides have to support meta-variables, and whatever the representation of meta-variables is, it must be the same on both sides. Furthermore, `lhs` has to have a `Num` instance. The type parameter `a` to `MetaRep r` ensures that meta-variables are used at the same type on both sides of a rule.
The constrained identity function in \texttt{rule\_mul} is used to fix the type of the rule (i.e. the parameter to \texttt{lhs} and \texttt{rhs}). It is needed because \texttt{rule\_mul} does not take a meta-variable identifier as argument, so the numeric type does not show up in the type of the rule (except in the context):

\begin{verbatim}
rule_mul :: (WildCard lhs, Num (lhs Int), Num (rhs Int)) ⇒ Rule lhs rhs
\end{verbatim}

Of course, much more work is needed before we can actually do something with the above rules, but the rules themselves will not need any modifications. They can be used with our library as they stand.

### 2.1 Representation of Terms and Patterns

We want to place different restrictions on the different representations in our library:

– Meta-variables are allowed in rules, but not in the terms being rewritten.
– Wildcards are only allowed on the LHS, not on the RHS of rules.

However, all constructs of the object language should be available to use in the rules.

In order to maintain these restrictions while allowing maximal sharing between the representations, we make use of Data Types à la Carte [18]. The basic idea is to use a standard fixed-point data type parameterized by a base functor:

\begin{verbatim}
data Term f = Term { unTerm :: f (Term f) }
\end{verbatim}

\texttt{Term} is a recursive data type where each node is a value of the base functor \texttt{f}. By using different \texttt{f} types, we can represent terms of different signatures.

Sharing between different representations is achieved by expressing the base functor as a co-product of smaller functors. Co-products are formed by the \texttt{:+:} type, which can be seen as a higher-kinded version of the \texttt{Either} type:

\begin{verbatim}
data (f :+: g) a = Inl (f a) | Inr (g a)
\end{verbatim}

infixr :+: 

For example, given two functors representing numeric and logic operations

\begin{verbatim}
data NUM a = Int Int | Add a a | Sub a a | Mul a a 
| Mul a a deriving (Functor)

data LOGIC a = Bool Bool | Not a | And a a | Equal a a | Cond a a a 
| Cond a a a deriving (Functor)
\end{verbatim}

we can form expressions of numeric and logic operations by using their co-product as the base functor of a \texttt{Term}:

\begin{verbatim}
type Exp = Term (NUM :+: LOGIC)
\end{verbatim}
The concrete representations for left and right hand sides of rules are defined as follows:

```haskell
class LHS f a = LHS { unLHS :: Term (WILD :+: META :+: f) }
class RHS f a = RHS { unRHS :: Term (META :+: f) }

data WILD a = WildCard deriving Functor
data META a = Meta Name deriving Functor
type Name = Int
```

Both LHS and RHS are parameterized on a base functor `f` representing the signature of the language the rules operate on. LHS extends `f` with meta-variables and wildcards while RHS only extends `f` with meta-variables. Both LHS and RHS have an extra phantom type parameter `a` which is used to get a typed interface to rules.

We can now make instances of the classes introduced earlier:

```haskell
instance WildCard (LHS f) where
  _ = LHS $ Term $ Inl WildCard

instance MetaVar (LHS f) where
  type MetaRep (LHS f) = META
  mvar = LHS . Term . Inr . Inl . castMETA

instance MetaVar (RHS f) where
  type MetaRep (RHS f) = META
  mvar = RHS . Term . Inl . castMETA
```

Note that `META` is used in two roles here: (1) as the constructor for meta-variables in LHS and RHS, and (2) as the concrete instance of `MetaRep`. The function `castMETA` is used to convert between these two roles:

```haskell
castMETA :: META a -> META b
castMETA (Meta v) = Meta v
```

For example, in the instance `MetaVar (LHS f)`, we have `mvar :: META a -> LHS f a` and then `castMETA` is used at the concrete type

```haskell
castMETA :: META a -> META (Term (WILD :+: (META :+: f)))
```

Our library makes use of the Compdata package [3] for the implementation of `Term` and ` :+:`. Compdata provides many additional utilities for working representations based on `Term`.

### 2.2 Matching and Rewriting

We will now give a formal definition of the rewriting algorithm used in our library. The following grammar defines terms and rules:
A term $t$ is a tree with $f$ symbols at each node. A left hand side $l$ is a term extended with meta-variables and wildcards, and a right hand side $r$ is a term that is only extended with meta-variables.

The first-order version of our library is based on standard syntactic rewriting, as defined in Figure 1. The matching relation $l \overset{2}{\Rightarrow} t \leftrightarrow \sigma$ defines how matching a term against a pattern results in a list of mappings from meta-variables to sub-terms. Wildcards and meta-variables match any term, with the difference that matching against a meta-variable results in a mapping in the substitution. For symbols, matching is done recursively for the children, and the resulting substitutions are concatenated.

Rewriting is defined as matching a term against the LHS and applying the corresponding substitution to the RHS. We use $[\sigma]r$ to denote application of a substitution $\sigma$ to $r$. Since we allow non-linear patterns, where the same meta-variable occurs more than once, we also have to check that the substitution obtained from matching is consistent; i.e. that each given meta-variable only maps to equal terms.

The corresponding functions in our library are

match :: (Functor f, Foldable f, EqF f)
  ⇒ LHS f a → Term f → Maybe (Subst f)

substitute :: Functor f ⇒ Subst f → RHS f a → Maybe (Term f)

**Fig. 1.** First-order matching and rewriting.
The match function succeeds if and only if the LHS matches the term and all occurrences of a given meta-variable are matched against equal terms. The substitute function succeeds if and only if each meta-variable in the RHS has a mapping in the substitution. The EqF class comes from the Compdata package, and is used for comparing symbols.

Combining match and substitute gives us the rewrite function:

```haskell
rewrite :: (Functor f, Foldable f, EqF f) ⇒ Rule (LHS f) (RHS f) → Term f → Maybe (Term f)
rewite (Rule lhs rhs) t = do
    subst ← match lhs t
    substitute subst rhs
```

When working with lists of rewrite rules, we are often interested in trying the rules in sequence and picking the first one that applies. That is the purpose of applyFirst:

```haskell
applyFirst :: (Functor f, Foldable f, EqF f) ⇒ [Rule (LHS f) (RHS f)] → Term f → Term f
applyFirst rs t = case [t' | r ← rs, Just t' ← [rewrite r t]] of
    t':_ → t'
    _ → t
```

If no rule matches, applyFirst returns the original term.

The final function in our first-order rewriting library transforms a term by rewriting each node from bottom to top:

```haskell
bottomUp :: (Functor f, Foldable f, EqF f) ⇒ [Rule (LHS f) (RHS f)] → Term f → Term f
bottomUp rs = applyFirst rs . Term . fmap (bottomUp rs) . unTerm
```

Since each node is a functor value, we use fmap to recursively transform all children. Then we use applyFirst to rewrite the resulting term.

### 3 Higher-Order Matching

The library presented in Section 2 works well for first-order rules, such as `rule_add` from earlier. But in order to express simplification rules for the for loop in Section 1.1, we need to extend the library and the rewriting algorithm with support for higher-order terms and rules.

The matching algorithm from Figure 1 is purely syntactic. It obeys the following property, where \( = \) is syntactic equality:

\[
    l \overset{2}{=} t \Rightarrow \sigma \sigma[l] = t
\]

Higher-order matching [10,19], on the other hand, obeys the following semantic property, where \( t \equiv_{\alpha, \beta, \eta} u \) means that \( t \) and \( u \) have the same normal form up to

\[2\] The property is almost true: it holds if we replace all wildcards in \( l \) with unique meta-variables.
α-renaming:
\[ l \overset{\sigma}{\Rightarrow} t \Rightarrow [\sigma][l] \equiv_{\alpha,\beta,\eta} t \]

Substitution in the higher-order case must be capture-avoiding.

If we extend our rule language to higher-order rules, the third rule of the for loop in section 1.1 can be defined as follows:

\[
\text{forLoop } l \text{ init } \left( \lambda i.\lambda s. \text{body } i \right) \Rightarrow \text{cond } \left( \text{eq } l \ 0 \right) \text{ init } \left( \text{body } \left( \text{sub } l \ 1 \right) \right)
\]

We use the convention to write meta-variables using smallcaps. The symbols \( \text{forLoop}, \text{cond}, \text{eq} \) and \( \text{sub} \) represent for loops, conditions, equality and subtraction, respectively. We also treat numeric literals as predefined symbols.

Using normal syntactic matching semantics, the above rule would only match a for loop whose body binds exactly the variables \( i \) and \( s \), and where some expression is immediately applied to \( i \) inside the abstraction. However, using higher-order matching semantics, the pattern \( \lambda i.\lambda s. \text{body } i \) matches any expression with two enclosing \( \lambda \)-abstractions and a body that only refers to the first bound variable.

As an example, we match the term \( t_1 \) against the pattern \( l_1 \) defined as follows:

\[
t_1 = \text{forLoop } 10 \ 0 \ (\lambda x.\lambda y. \text{sub } x \ 2) \quad l_1 = \text{forLoop } l \text{ init } \left( \lambda i.\lambda s. \text{body } i \right)
\]

Despite the fact that \( \text{sub } x \ 2 \) is not an immediate application to \( x \), the pattern matches, and results in the substitution

\[
\sigma_1 = [ L \mapsto 10, \text{init} \mapsto 0, \text{body} \mapsto \lambda z. \text{sub } z \ 2 ]
\]

We check the result by applying \( \sigma_1 \) to \( l_1 \) and see that the result is equivalent to \( t_1 \):

\[
\llbracket \sigma_1 \rrbracket l_1 = \text{forLoop } 10 \ 0 \ (\lambda i.\lambda s. (\lambda z. \text{sub } z \ 2) \ i) \equiv_{\alpha,\beta,\eta} t_1
\]

An alternative to introducing a fresh variable \( z \) for \( \text{body} \) is to reuse the existing variable name \( x \). That would give the following substitution instead:

\[
\sigma_1 = [ \ldots, \text{body} \mapsto \lambda x. \text{sub } x \ 2 ]
\]

This result is just as valid as the previous one, and it has the advantage that the body \( \text{sub } x \ 2 \) does not need to be renamed.

An implicit side condition in higher-order matching is that the resulting substitution is not allowed to contain free variables that were not free in the original term. For example, the following term does not match \( l_1 \):

\[
t_2 = \text{forLoop } 10 \ 0 \ (\lambda x.\lambda y. \text{sub } x \ y)
\]

An attempt at a solution might be

\[
\sigma_2 = [ \ldots, \text{body} \mapsto \lambda x. \text{sub } x \ s ]
\]
This solution has $s$ as a free variable. However, $\sigma_2[l_1]$ is not equivalent to $t_2$, because substitution is defined to be capture-avoiding.

If we want to allow $s$ to occur in the body, we need to declare that by listing $s$ as one of the arguments to BODY:

$$l_2 = \text{forLoop } l \text{ init } (\lambda i. \lambda s. \text{BODY } i \ s)$$

Matching $t_2$ against this pattern results in the substitution

$$\sigma_3 = [\ldots, \text{BODY } \mapsto \lambda x. \lambda y. \text{sub } x \ y]$$

for which it holds that $\sigma_3[l_2]$ is equivalent to $t_2$.

### 3.1 Tractability

Higher-order matching is an instance of higher-order unification, with the difference that the latter permits meta-variables on both sides of the $\equiv$ relation. Higher-order unification is undecidable in general \[9\]. Higher-order matching is decidable, but its complexity class is at least NP-complete for second-order problems and upwards \[19\].

Miller identified a fragment for which unification is efficient, namely when each meta-variable is applied only to distinct object variables \[11\]. Note that $l_1$ and $l_2$ from before fall under this category, because BODY is only applied to the object variables $i$ and $s$. This restriction of the general problem is called the pattern fragment. The term “pattern” refers to the list of object variables that a meta-variable is applied to, and should not be confused with its use in the term “pattern matching”.

### 3.2 Rewriting Based on Pattern Unification

Matching for the pattern fragment can be done as a lightweight extension to the first-order algorithm presented in section 2.2.

Figure 2 shows the previous grammar extended with object variables and $\lambda$-abstraction. We ensure that terms and rules are in $\beta$-short normal form by making use of the so-called spine formulation \[6\] which disallows application of $\lambda$-abstractions. We do however allow general applications in the result after rewriting, which is why the production $t \overset{\beta}{\to} t'$ for terms is put in parenthesis. On the LHS, meta-variables can only be applied to object variables, while this restriction is not needed on the RHS.

A simplified higher-order matching algorithm is defined in Figure 3. The sym rule has been replaced with the atom rule, which covers both symbols and object variables. $\lambda$-abstractions are matched structurally. Meta-variables are matched simply by turning the list of arguments into a number of $\lambda$-abstractions, as we did previously in the for loop example. Like in that example, we also reuse the names $v_1 \ldots v_n$ in the lambda abstractions, which avoids having to rename variables in $t$.

The given algorithm is a bit simplified for presentation purposes:
Symbols

\( f \) a set of symbols with associated arities

Object variables \( v \)

Atoms \( a, b ::= v \mid f \)

Meta-variables \( M \)

Terms

\( t ::= a \bar{t} \mid \lambda v.t \mid (| t \bar{t} ) \)

LHS

\( l ::= a \bar{l} \mid \lambda v.l \mid M \bar{v} \mid _- \)

RHS

\( r ::= a \bar{r} \mid \lambda v.r \mid M \bar{r} \)

Rules

\( \rho ::= l \implies r \)

Fig. 2. Grammar for higher-order terms and rewrite rules in the pattern fragment.

\[
\begin{array}{ccc}
\text{WILDL} & \text{LAM} \\
\begin{array}{c}
\_ \coloneqq t \rightsquigarrow [] \\
\lambda v.l \coloneqq \lambda v.t \rightsquigarrow \sigma \\
a = b & l_1 \coloneqq t_1 \rightsquigarrow \sigma_1 & \ldots & l_n \coloneqq t_n \rightsquigarrow \sigma_n \\
a \bar{l}_1 \ldots \bar{l}_n \coloneqq b \bar{t}_1 \ldots \bar{t}_n \rightsquigarrow \text{concat}(\sigma_1 \ldots \sigma_n) \\
\text{freeVars}(\lambda v_1 \ldots \lambda v_n.t) = \emptyset & \text{META} \\
M \bar{v}_1 \ldots \bar{v}_n \coloneqq t \rightsquigarrow [M \mapsto \lambda v_1 \ldots \lambda v_n.t]
\end{array}
\end{array}
\]

Fig. 3. Simplified higher-order matching for the pattern fragment.

- It does not deal with \( \alpha \)-renaming.
- It does not allow any free variables in the substitution. As we saw earlier, we can allow free variables if they were already free in the original term.
- It assumes that \( \lambda \) is always matched against \( \lambda \). For example, the term \( \lambda v.f v \) will not match its \( \eta \)-reduced form \( f \), as it should.

The implementation in our library deals correctly with \( \alpha \)-renaming and free variables. The simplest way to deal with \( \eta \) conversion is to always \( \eta \)-expand subexpressions of function type to get terms in \( \eta \)-long normal form. Our matching algorithm currently does not do this; however, it is possible to define the user interface in such a way that partially applied atoms do not occur in practice. We will see how that is done in Section 5.

Once higher-order matching has been defined, higher-order rewriting is defined analogously to the \textit{rewrite} function in Figure 1. It should be noted that when substituting for meta-variables on the RHS, we may create \( \beta \)-redexes for meta-variables that have arguments. In our implementation, it is possible to choose whether to reduce those redexes immediately or leave them for later.

Matching according to the rules in Figure 3 is efficient in the sense that the number of recursive steps is bounded by the size of the pattern. The only source of inefficiency is the use of \textit{freeVars} in the \textit{META} rule. This function needs to traverse the whole term. However, it is a common scenario to apply rewrite
rules to each node from bottom to top in a term. When doing this, it is possible to cache the set of free variables in the nodes on the way up. This is how the implementation in our library works. The result is that the complexity of rule application is determined only by the size of the rule.

3.3 Most general solutions

There is one aspect of Miller’s pattern restriction that we do not enforce: meta-variables must only be applied to distinct object variables. This restriction is needed to ensure the existence of a most general unifier. The main reason we do not enforce it is that it is hard to capture this particular restriction in the types of the library.

For example, when matching $\lambda x. sub x 2$ against $\lambda y. body y y$, there are two possible solutions: $body \mapsto \lambda a. \lambda x. sub x 2$ and $body \mapsto \lambda x. \lambda a. sub x 2$. Our implementation will blindly give the result $body \mapsto \lambda x. \lambda x. sub x 2$, which is equivalent to the first solution. There is nothing wrong with either solution; the only problem is that picking one instead of the other is a bit arbitrary.

To avoid this problem, the library user must make sure to only apply meta-variables to distinct object variables.

4 Extending the Library to Higher-Order Rewriting

We will now show how to extend the first-order library from Section 2 to higher-order rewriting. LHS and RHS in Figure 2 permit application of meta-variables to objects of different kind. LHS only allows application to object variables, while RHS allows application to arbitrary terms. We reconcile these different requirements using the type `MetaExp` which represents meta-variables applied to a number of arguments:

```haskell
data MetaExp (r :: * → *) a where
    MVar :: MetaRep r a → MetaExp r a
    MApp :: MetaExp r (a → b) → MetaArg r a → MetaExp r b

type family MetaArg (r :: * → *) :: * → *
type family MetaRep (r :: * → *) :: * → *
```

The representation of the meta-variable is given by the type family `MetaRep` (corresponding to the associated type of the same name in Section 2), and the representation of the arguments is given by `MetaArg`. By using different `MetaArg` representations, we can enforce different requirements for meta-variable application in the LHS and RHS.

We introduce yet another type family which gives an abstract representation of object variables:

```haskell
type family Var (r :: * → *) :: * → *
```

We can now give the following `MetaArg` instances for LHS and RHS:
The first instance ensures that meta-variables can only be applied to object variables on the LHS, while the second instance permits arbitrary terms as meta-variable arguments on the RHS.

We redefine the `MetaVar` class with a single method that constructs an expression from a `MetaExp` value:

```haskell
class MetaVar r where
  metaExp :: MetaExp r a -> r a
```

```haskell
instance MetaVar (LHS f) -- see library source for details
instance MetaVar (RHS f) -- see library source for details
```

Introducing meta-variables using `MVar`, `MApp` and `metaExp` is quite cumbersome, so we provide a number of helper functions:

```haskell
mvar :: MetaVar r => MetaRep r a -> r a
mvar = metaExp . MVar

($$) :: MetaExp r (a -> b) -> MetaArg r a -> MetaExp r b
($-$) :: MetaVar r => MetaExp r (a -> b) -> MetaArg r a -> r b
(-$) :: MetaRep r (a -> b) -> MetaArg r a -> MetaExp r b
(-$-) :: MetaVar r => MetaRep r (a -> b) -> MetaArg r a -> r b

($$) = MApp
f $$ a = metaExp (MApp f a)
f $$ a = MApp (MVar f) a
f $$ a = metaExp (MApp (MVar f) a)
```

```
infixl 2 $$, $$-`, --$, -$-
```

The function `mvar` has the same type as in Section 2, and it introduces a meta-variable without any arguments. For meta-variables with arguments, we use the different application operators:

- `$$-` is used when there is only one argument.
- `$$-` is used for the first variable when there are more than one.
- `$$-` is used for the last variable when there are more than one.
- `$$-` is used for used for any but the first and last variable.

As an example, assume we have two meta-variables and two object variables of the following types (for some base functor `F`):

```haskell
m1 :: MetaRep (LHS F) Int
m2 :: MetaRep (LHS F) (Int -> Char -> Bool)
v1 :: Var (LHS F) Int
v2 :: Var (LHS F) Char
```

Then we can use them to form LHS terms like this:
4.1 Object variables and binders

The following type class is for object variables and binders:

\[
\text{class } \text{Bind } r \text{ where}
\]
\[
\text{var} :: \text{Var } r \ a \to r \ a
\]
\[
\text{lam} :: (\text{Var } r \ a \to r \ b) \to r \ (a \to b)
\]

The function \text{var} constructs a variable, and \text{lam} makes a \(\lambda\)-abstraction from a Haskell function. For example, the term \(\lambda x. x + 2\) is represented as follows:

\[
\text{lam } (\lambda x \to \text{var } x + 2)
\]

Note that the only way to construct a value of the abstract type \text{Var} is using \text{lam}. This ensures that \text{Var} faithfully represents object variables.

The concrete representation of object variables uses \text{VAR} which is a typed newtype wrapper around a name:

\[
\text{newtype VAR } a = \text{Var } \text{Name}
\]

\text{VAR} has the same double role as \text{META} in Section 2.1: it is both used to identify object variables and as a functor that represents a variable node in a term.

4.2 Rewriting

The functions that perform higher-order rewriting have slightly different types compared to those from Section 2.2. One difference is that the result of rewriting is a term where each node is annotated with its set of free variables. As discussed in Section 3.2, we need to cache the set of free variables in order to avoid quadratic complexity of bottom-up rewrites.

The function \text{bottomUp} has the following type:

\[
\text{bottomUp} :: \{ \ldots
\]
\[
, g \sim (f :\&: \text{Set Name})
\]
\[
\Rightarrow (\text{Term } g \to \text{Term } g \to \text{Term })
\]
\[
\to [\text{Rule } (\text{LHS } f) (\text{RHS } f)]
\]
\[
\to \text{Term } f
\]
\[
\to \text{Term } (f :\&: \text{Set Name})
\]

\text{Term } (f :\&: \text{Set Name}) is a term where each node is annotated with a set of names. The first argument is an application operator which is used when replacing applied meta-variables on the RHS of a rule. Taking this operator as an argument allows the user to choose whether to construct a redex or to reduce it right away.
4.3 Quantifying over Meta-Variables

Functions such as \texttt{bottomUp} take a list of rules as argument. But most rules are of the form of Haskell functions that take extra arguments corresponding to the meta-variables used. For example, the type of \texttt{rule.add} from Section 2 is

\[
\texttt{rule.add} :: (\texttt{MetaVar lhs}, \texttt{MetaVar rhs}, \texttt{Num (lhs \ a)}
\texttt{, MetaRep lhs \sim \texttt{MetaRep rhs}}
\Rightarrow \texttt{MetaRep lhs \ a} \rightarrow \texttt{Rule lhs rhs}
\]

The \texttt{Quantifiable} type class automates the task of providing fresh meta-variables to functions like \texttt{rule.add}:

\[
\texttt{class Quantifiable rule where}
\texttt{type RuleType rule}
\texttt{quantify' :: Name \rightarrow rule \rightarrow RuleType rule}
\]

\[
\texttt{quantify :: (Quantifiable rule, RuleType rule \sim \texttt{Rule lhs rhs})}
\Rightarrow \texttt{rule \rightarrow Rule lhs rhs}
\texttt{quantify = quantify' 0}
\]

\[
\texttt{instance Quantifiable (Rule lhs rhs) where}
\texttt{type RuleType (Rule lhs rhs) = Rule lhs rhs}
\texttt{quantify' _ = id}
\]

\[
\texttt{instance (Quantifiable rule, m \sim \texttt{MetaId a}} \Rightarrow \texttt{Quantifiable (m \rightarrow rule)} \texttt{where}
\texttt{type RuleType (m \rightarrow rule) = RuleType rule}
\texttt{quantify' i rule = quantify' (i+1) (rule (MetaId i))}
\]

The first instance is for rules that do not have any meta-variables to quantify over. The second instance recursively quantifies one meta-variable at a time. \texttt{MetaId} is the concrete representation of meta-variables.

Using \texttt{quantify}, we can easily package our rules in a list:

\[
\texttt{rules} = [\texttt{quantify$ \texttt{intArg \ rule.add}}
\texttt{, \texttt{quantify$ \texttt{intArg \ rule.sub}}
\texttt{, \texttt{quantify \ rule.mul}}
\]
\]

\[
\texttt{intArg :: (c \texttt{Int} \rightarrow a) \rightarrow (c \texttt{Int} \rightarrow a)}
\texttt{intArg = id}
\]

The function \texttt{intArg} is needed to resolve the type of the meta-variable, which would otherwise be ambiguous.

5 Case Study – Feldspar

The repository contains an example file\footnote{\url{https://github.com/emilaxelsson/ho-rewriting/blob/master/examples/Feldspar.hs}} inspired by Feldspar that makes use of the library. In this section, we will highlight the important parts of that implementation. The interested reader is encouraged to learn more by looking at the source code.
Feldspar’s expression type Data is defined as a newtype wrapper around a Term over the functor Feld:

```
```

```
newtype Data a = Data { unData :: Term Feld }
```

Feld is a sum of several smaller functors, where VAR, LAM and APP represent the constructs of the lambda calculus, NUM and LOGIC represent numeric and logic operations, and FORLOOP is the Feldspar-specific for loop.

Object variables are represented just as Feldspar expressions, which avoids the need to use the var function to introduce object variables:

```
type instance Var Data = Data
```

Since we want to be able to construct for loops in rules as well as ordinary Feldspar expressions, we introduce a type class for it:

```
class ForLoop r where
    forLoop_ :: r Int → r s → r (Int → s → s) → r s
```

The third argument to forLoop_ has function type, so it needs to be constructed by lam. The following higher-order function takes care of wrapping the body in lam:

```
forLoop :: (ForLoop r, Bind r) ⇒ r Int → r s → (Var r Int → Var r s → r s) → r s
```

```
forLoop len init body = forLoop_ len init (lam $ λi → lam $ λs → body i s)
```

Exposing functions like forLoop to the user instead of lam and forLoop_ ensures that λ-abstractions are only used at specific places. This solves the problem of matching lambdas that was mentioned in Section 3.2. Although restricting the use of lam may not be desired in general, it works well in a language like Feldspar which is essentially a first-order language with a few predefined higher-order symbols like FORLOOP.

Using forLoop, we can now express the three for loop simplification rules from Section 1.1:

```
rule_for1 init = forLoop 0 (mvar init) (λi s → 0) ⇒ mvar init
rule_for2 init = forLoop 0 (mvar init) (λi s → var s) ⇒ mvar init
rule_for3 len init body = forLoop (mvar len) (mvar init) (λi s → body -$- i)
    ⇒ cond (mvar len === 0) (mvar init) (body -$- (mvar len - 1))
```

The === operator is equality in this toy version of Feldspar.

A Feldspar simplifier is obtained from bottomUp by giving it a list of all the rules defined in this paper in a list:

```
simplify :: Data a → Data a
simplify = Data . stripAnn . bottomUp app rulesFeld . unData

rulesFeld = [ quantify rule_add, ..., quantify rule_for1, ... ]
```
The function `stripAnn` is used to remove the free variable annotations. The application operator `app` tells `bottomUp` to keep any redexes created by rewriting.

We have used `forLoop` to define rules, but we can also use it to write Feldspar expressions. Here is an example containing two for loops that can be simplified away:

```haskell
forExample :: Data Int → Data Int
forExample a = forLoop (a-a) a (λ i s → i*s+70) + forLoop a a (λ i s → i*i+100)
```

We simplify the expression by running

```haskell
<Main> unData $ simplify $ lam forExample
(Lam 2 (Add (Var 2) (Cond (Equal (Var 2) (Num 0)) (Var 2) (App (Lam 1 (Add (Mul (Var 1) (Var 1)) (Num 100))) (Sub (Var 2) (Num 1))))))
```

We see that rewriting removes both loops: the first one because its number of iterations is 0 (a-a), and the second one because its body does not refer to the previous state.

### 6 Related Work

One of the problems solved in this paper being able to use the same syntax both for pattern matching and construction and to hide the underlying representation of expressions. A more general solution to this particular problem is function patterns [1,7,16], which allow ordinary functions to be used inside patterns. When matching a term `t` against a function pattern, say `f p`, the inverse of `f` is used to compute a value to match against the argument `p`. Here, `f` can be a smart constructor whose purpose is to hide the representation of terms, or to give a typed interface using phantom types.

The recent `PatternSynonyms` extension in GHC allows the declaration of bidirectional patterns that can be used both for matching and construction. These can be seen as a restricted form of function patterns.

There may seem to be a similarity between function patterns and patterns with applied meta-variables in our library. In both cases we have patterns involving applied functions. However, there are important differences: First of all, function patterns allow ordinary functions inside patterns, while our library is only concerned with functions in some object language. Moreover, function patterns only allow using existing functions inside patterns; they cannot be used to synthesize function definitions the way that higher-order matching does.

Mohnen introduced context patterns for Haskell [12]. These are more similar to higher-order matching in that they allow meta-variables of function type (so matching can actually synthesize function definitions). However, a main difference to our work (again) is that context patterns involve actual Haskell functions rather than object-level functions.

Yokoyama et al. have made a library based on Template Haskell for higher-order rewriting of Haskell code [20]. Just like in our library, they restrict matching in the interest of efficiency. However, their restrictions are different: for example,
meta-variables can be applied to at most one argument, but this argument can be an arbitrary pattern. It remains to be investigated whether their restrictions are suitable for the kind of syntactic rewrites that we are interested in.

There has also been work on generic, first-order rewriting libraries for Haskell [14,8,4]. In particular, the work of Van Noort et al. [14] has similarities to our implementation: it uses an intensional representation of rules as data, and it generically extends data type representations with a constructor for meta-variables. The main differences are that their library works for any regular data type and that it does not support higher-order rewriting. The library by Felgenhauer, et al. [8] also has an intensional representation of rules, but uses a simpler term representation: a rose tree extended with meta-variables.

Higher-order pattern unification is at the core of systems that manipulate higher-order data like Twelf [17] and λProlog [13] or type reconstruction algorithms for dependently typed languages like Agda [15]. In such cases the unification problems that fall into the pattern fragment are solved immediately, while the others are suspended in hope that they will become tractable later when more meta-variables have been solved.

7 Future Work

When writing rules like rule_add below, the intention is that lhs and rhs should be abstract in order to disallow inspection of the meta-variable argument.

```plaintext
rule_add :: ( ... ) ⇒ MetaRep lhs a → Rule lhs rhs
rule_add x = 0 + mvar x =⇒ mvar x
```

However, rewriting functions like bottomUp accept rules with the concrete representations LHS and RHS. In order to make the library safer, we should require the arguments to rewriting functions to be polymorphic in their representations. We have not yet been able to make such a solution work together with quantify and constraints like Num (lhs a). But we are hopeful that it can be done.

Acknowledgements

This research was funded by the Swedish Foundation for Strategic Research (in the RAWFP project).

References


20. Yokoyama, T., Hu, Z., Takeichi, M.: Design and implementation of deterministic higher-order patterns