Lightweight Higher-Order Rewriting in Haskell

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 higher-order pattern unification and has the same complexity as first-order matching. 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 on generic encodings, such as compositional data types [4].
– 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. 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 for-loop in the Feldspar EDSL [3]. Feldspar is a Haskell EDSL for signal processing and numeric computations. It supports common functional programming idioms, such as `map` and `fold`, and generates high-performance C code from such programs.

One of the more low-level constructs in Feldspar is `forLoop`:

```
forLoop :: Data Int → Data s → (Data Int → Data s → Data s) → Data s
```

`Data` is Feldspar's expression type which is parameterized by the type of the value the expression computes. The first argument to `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 `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 `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 `Data` is a simple recursive data type, with constructors for lambda abstraction, variables, for-loops, etc., we might express the first two rules as follows:

```
simplify (ForLoop (Int 0) init _)                          = init
simplify (ForLoop _ init (Lam i (Lam s (Var s')))) | s == s' = init
```

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

The third rule is trickier. We want to rewrite an expression of the form

```
forLoop len init (λi s → body)
```

to

```
cond (len==0) init body'
```

where `body'` is `body` with `len-1` substituted for `i` and provided that `s` does not occur freely in `body`. The object-level function `cond` is used to return `init` when the length is 0 and otherwise return `body'`.

Trying to express this rule as a case in `simplify` reveals an additional problem of this style of rewriting: it is possible for `body` to contain free variables. 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 `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:

```
rule_for3 len init body =
    forLoop (meta len) (meta init) (λi s → body -$- i)
      ⟹
    cond (meta len === 0) (meta init) (body -$- (meta len - 1))
```

Note the use of Haskell's λ-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 a simple 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.[1] The code makes use of many Haskell extensions, including `TypeFamilies`, `GADTs`, `DerivingFunctor`, etc. Consult the GHC documentation[2] for more information on these extensions.

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 needs to be modified.

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

```
data Rule lhs rhs where
  Rule :: lhs a → rhs a → Rule lhs rhs
```

The parameters `lhs` and `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 `Data` in Section 1.1. The type parameter is existentially quantified, and the only thing we care about is that `lhs` and `rhs` have the same type parameter.

[1] http://hackage.haskell.org/package/ho-rewriting-0.2
[2] https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghc-language-features.html

Rather than using fixed types for `lhs` and `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 [6].

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

```
class MetaVar r where
  type MetaRep r :: * → *
  meta :: MetaRep r a → r a

class WildCard r where
  __ :: r a
```

The function `meta` 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:

```
(⟹) :: 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:[3]

```
-- 0 + X  ⟹  X
rule_add x = 0 + meta x  ⟹  meta x

-- X - X  ⟹  0
rule_sub x = meta x - meta x  ⟹  0

-- 0 * _  ⟹  0
rule_mul = 0 * __  ⟹  (0 :: _ Int)
```

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:

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

[3] Note the *partial* type annotation `(... :: _ Int)` in `rule_mul`. It is used to constrain the type parameter of the RHS without saying anything about the representation of the RHS. Partial type signatures require the recent `PartialTypeSignatures` extension to GHC. However, this extension is not strictly needed: an equivalent formulation of the RHS would be `(id :: r Int → r Int) 0`.

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 if they occur multiple times a rule.

The partial type annotation in `rule_mul` is used to fix the type of the rule (i.e. the parameter to `lhs` and `rhs`). It is needed because `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):

```
rule_mul :: (WildCard lhs, Num (lhs Int), Num (rhs Int)) ⇒ Rule lhs rhs
```

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 need 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 [21]. The basic idea is to use a standard fixed-point data type parameterized by a base functor:

```
data Term f = Term { unTerm :: f (Term f) }
```

`Term` is a recursive data type where each node is a value of the base functor `f`. By using different `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 `:+:` type, which can be seen as a higher-kinded version of the `Either` type:

```
data (f :+: g) a = Inl (f a)
                 | Inr (g a)

infixr :+:
```

For example, given two functors representing numeric and logic operations

```
data NUM a                      data LOGIC a
    = Int Int                       = Bool Bool
    | Add a a                       | Not a
    | Sub a a                       | And a a
    | Mul a a                       | Equal a a
  deriving (Functor)                | Cond a a a
                                  deriving (Functor)
```

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

```
type Exp = Term (NUM :+: LOGIC)
```

The concrete representations for left and right hand sides of rules are defined as follows:

```
newtype LHS f a = LHS { unLHS :: Term (WILD :+: META :+: f) }
newtype 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 a phantom type parameter `a` which denotes the type of the corresponding term. It is used to ensure that only well-typed left and right hand sides can be constructed.

We can now make instances of the classes introduced earlier:

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

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

instance MetaVar (RHS f) where
  type MetaRep (RHS f) = META
  meta = 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:

```
castMETA :: META a → META b
castMETA (Meta v) = Meta v
```

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

```
castMETA :: META a → META (Term (WILD :+: (META :+: f)))
```

Our library makes use of the Compdata package [4] for the implementation of `Term` and `:+:`. Compdata is a Haskell library based on Data Types à la Carte, and it provides many utilities for working with 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:

$$\frac{}{\_ \overset{?}{=} t \rightsquigarrow []}\ \textsc{wild} \qquad \frac{}{M \overset{?}{=} t \rightsquigarrow [M \mapsto t]}\ \textsc{meta}$$

$$\frac{f = g \quad l_1 \overset{?}{=} t_1 \rightsquigarrow \sigma_1 \quad \ldots \quad l_n \overset{?}{=} t_n \rightsquigarrow \sigma_n}{f\ l_1 \ldots l_n \overset{?}{=} g\ t_1 \ldots t_n \rightsquigarrow concat(\sigma_1 \ldots \sigma_n)}\ \textsc{symbol}$$

$$\begin{array}{lll} rewrite(l \Longrightarrow r, t) = [\![\sigma]\!]r & \text{where} & l \overset{?}{=} t \rightsquigarrow \sigma \\ & & consistent(\sigma) \end{array}$$

Fig. 1. First-order matching and rewriting.

Symbols	f, g	a set of symbols with associated arities
Meta-variables	M	
Terms	t	$::=\ f\ \vec{t}$
LHS	l	$::=\ f\ \vec{l} \mid M \mid \_$
RHS	r	$::=\ f\ \vec{r} \mid M$
Rules	ρ	$::=\ l \Longrightarrow r$

A term t is a tree where each node has a symbol f and zero or more sub-trees. 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 $l \overset{?}{=} t \rightsquigarrow \sigma$ defines how matching a term t against a pattern l results in a list σ 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 σ 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

```
type Subst f = [(Name, Term f)]  -- Substitution

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

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

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:

```
rewrite :: (Traversable f, EqF f)
        ⇒ Rule (LHS f) (RHS f) → Term f → Maybe (Term f)
rewrite (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`:

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

If no rule matches, `applyFirst` returns the original term.

Another strategy is to rewrite each node in a term from bottom to top:

```
bottomUp :: Functor f ⇒ (Term f → Term f) → Term f → Term f
bottomUp rew = rew . Term . fmap (bottomUp rew) . unTerm
```

The first argument to `bottomUp` is the node rewriter. Since each node is a functor value, we use `fmap` to recursively transform all children. Then we apply the node rewriter to the resulting term.

A top-down strategy is defined in a similar way; just apply the rewrite before the recursive call:

```
topDown :: Functor f ⇒ (Term f → Term f) → Term f → Term f
topDown rew = Term . fmap (topDown rew) . unTerm . rew
```

Typically, one is interested in combinations such as `bottomUp (applyFirst rs)`, which applies the first matching rule in the list `rs` to each node in a term.

3 Higher-Order Rewriting

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:[4]

$$l \overset{?}{=} t \rightsquigarrow \sigma \quad \Rightarrow \quad [\![\sigma]\!] l = t$$

Higher-order matching [11,22], on the other hand, obeys the following *semantic* property, where $t \equiv_{\alpha,\beta,\eta} u$ means that t and u reduce to the same term up to α-renaming:

$$l \overset{?}{=} t \rightsquigarrow \sigma \quad \Rightarrow \quad [\![\sigma]\!] l \equiv_{\alpha,\beta,\eta} t$$

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:

$$\mathit{forLoop}\ \textsc{len}\ \textsc{init}\ (\lambda i.\lambda s.\ \textsc{body}\ i) \implies \mathit{cond}\ (\mathit{eq}\ \textsc{len}\ 0)\ \textsc{init}\ (\textsc{body}\ (\mathit{sub}\ \textsc{len}\ 1))$$

We use the convention to write meta-variables using SMALLCAPS. The symbols $\mathit{forLoop}$, cond, eq and 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 $\lambda i.\lambda s.\ \textsc{body}\ i$ matches any expression with two enclosing λ-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:

$$\begin{array}{lcl} t_1 & = & \mathit{forLoop}\ 10\ 0\ (\lambda x.\lambda y.\ \mathit{sub}\ x\ 2) \\ l_1 & = & \mathit{forLoop}\ \textsc{len}\ \textsc{init}\ (\lambda i.\lambda s.\ \textsc{body}\ i) \end{array}$$

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

$$\begin{array}{lcl} \sigma_1 & = & [\ \textsc{len} \mapsto 10 \\ & & ,\ \textsc{init} \mapsto 0 \\ & & ,\ \textsc{body} \mapsto \lambda z.\ \mathit{sub}\ z\ 2\ ] \end{array}$$

We check the result by applying σ_1 to l_1 which gives a result equivalent to t_1:

$$[\![\sigma_1]\!] l_1 \ = \ \mathit{forLoop}\ 10\ 0\ (\lambda i.\lambda s.\ (\lambda z.\ \mathit{sub}\ z\ 2)\ i) \quad \equiv_{\alpha,\beta,\eta} \quad t_1$$

An alternative to introducing a fresh variable z for BODY is to reuse the existing variable name x. That would give the following substitution instead:

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

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

[4] The property is almost true: it holds if we replace all wildcards in l with unique meta-variables.

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:

$$t_2 \ = \ \mathit{forLoop}\ 10\ 0\ (\lambda x.\lambda y.\ \mathit{sub}\ x\ y)$$

An attempt at a solution might be

$$\sigma_2 \ = \ [\ \ldots,\ \textsc{body} \mapsto \lambda x.\ \mathit{sub}\ x\ s\ ]$$

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:

$$l_2 \ = \ \mathit{forLoop}\ \textsc{len}\ \textsc{init}\ (\lambda i.\lambda s.\ \textsc{body}\ i\ s)$$

Matching t_2 against this pattern results in the substitution

$$\sigma_3 \ = \ [\ \ldots,\ \textsc{body} \mapsto \lambda x.\lambda y.\ \mathit{sub}\ x\ y\ ]$$

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 $\overset{?}{=}$ relation. Higher-order unification is undecidable in general [10]. Higher-order matching is decidable, but its complexity class is at least NP-complete for second-order problems and upwards [22].

Miller identified a fragment for which unification is efficient, namely when each meta-variable is applied only to distinct object-language variables [12]. Note that l_1 and l_2 from before fall under this category, because BODY is only applied to the object-language variables i and s. This restriction of the general problem is called the *pattern fragment*. The term "pattern" refers to the list of object-language 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-language variables and λ-abstraction. We ensure that terms and rules are in β-short normal form by making use of the so-called spine formulation [7] which disallows application of λ-abstractions. We do however allow general applications in the result after rewriting, which is why the production $t\ \vec{t}$ for terms is put in parentheses. On

SYMBOLS	f	a set of symbols with associated arities
OBJECT VARIABLES	v	
ATOMS	a, b	$::= \ v \mid f$
META-VARIABLES	M	
TERMS	t	$::= \ a\ \vec{t} \mid \lambda v.t \quad (\mid t\ \vec{t})$
LHS	l	$::= \ a\ \vec{l} \mid \lambda v.l \mid M\ \vec{v} \mid \_$
RHS	r	$::= \ a\ \vec{r} \mid \lambda v.r \mid M\ \vec{r}$
RULES	ρ	$::= \ l \implies r$

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

$$\frac{}{\_ \overset{?}{=} t \rightsquigarrow []}\ \text{WILD} \qquad \frac{l \overset{?}{=} t \rightsquigarrow \sigma}{\lambda v.l \overset{?}{=} \lambda v.t \rightsquigarrow \sigma}\ \text{LAM}$$

$$\frac{a = b \quad l_1 \overset{?}{=} t_1 \rightsquigarrow \sigma_1 \quad \ldots \quad l_n \overset{?}{=} t_n \rightsquigarrow \sigma_n}{a\ l_1 \ldots l_n \overset{?}{=} b\ t_1 \ldots t_n \rightsquigarrow concat(\sigma_1 \ldots \sigma_n)}\ \text{ATOM}$$

$$\frac{freeVars(\lambda v_1 \ldots \lambda v_n.t) = \emptyset}{M\ v_1 \ldots v_n \overset{?}{=} t \rightsquigarrow [M \mapsto \lambda v_1 \ldots \lambda v_n.t]}\ \text{META}$$

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

the LHS, meta-variables can only be applied to object-language 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-language variables. λ-abstractions are matched structurally. Meta-variables are matched simply by turning the list of arguments into a number of λ-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:

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

The implementation in our library deals correctly with α-renaming and free variables. The simplest way to deal with η conversion is to always η-expand subexpressions of function type to get terms in η-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 *rewrite* function in Figure 1. It should be noted that when substituting for meta-variables on the RHS, we may create β-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 possible source of inefficiency is the use of $freeVars$ in the META rule. Our implementation avoids traversing the whole term when checking the free variables simply by caching the set of free variables for each node in a term. The result is that the complexity of rule application is determined only by the size of the rule – just like for first-order matching.

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-language 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-language 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 kinds. LHS only allows application to object-language 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:

```
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 MetaRep (r :: ∗ → ∗) :: ∗ → ∗
type family MetaArg (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-language variables:

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

We can now give the following `MetaArg` instances for `LHS` and `RHS`:

```
type instance MetaArg (LHS f) = Var (LHS f)
type instance MetaArg (RHS f) = RHS f
```

The first instance ensures that meta-variables can only be applied to object-language 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:

```
class MetaVar r where
  metaExp :: MetaExp r a → r a

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:

```
meta :: MetaVar r ⇒ MetaRep r a → r a
meta = 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 `meta` 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 argument when there are more than one argument.
`$-`	is used for the last argument when there are more than one argument.
`$$`	is used for used for any but the first and last arguments.

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

```
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:

```
meta m1        :: LHS F Int
m2 -$ v1 $- v2 :: LHS F Bool
```

4.1 Object-language variables and binders

The following type class is for object-language variables and binders:

```
class Bind r where
  var :: Var r a → r a
  lam :: (Var r a → r b) → r (a → b)
```

The function `var` constructs a variable, and `lam` makes a λ-abstraction from a Haskell function. For example, the term $\lambda x.\ x + 2$ is represented as follows:

```
lam (λx → var x + 2)
```

Note that the only way to construct a value of the abstract type `Var` is using `lam`. This ensures that `Var` faithfully represents object-language variables.

The concrete representation of object-language variables uses `VAR` which is a typed newtype wrapper around a name:

```
type instance Var (LHS f) = VAR
type instance Var (RHS f) = VAR

newtype VAR a = Var Name deriving Functor
```

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

The above `Var` instances both have `VAR` on the right hand side, but in Section 5 we will see an instance with a different right hand side.

The library uses a first-order term representation internally, despite the fact that `lam` has a higher-order type. This is possible due Axelsson and Claessen's technique for generating first-order terms from a higher-order interface [2].

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 make matching efficient.

The function `applyFirst` now has the following type:

```
applyFirst :: (..., g ∼ (f :&: Set Name))
           ⇒ (Term g → Term g → Term g)
           → [Rule (LHS f) (RHS f)]
           → Term g → Term g
```

`Term (f :&: Set Name)` is a term where each node is annotated with a set of names. The first argument to `applyFirst` 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.

In order to shield the user from the free-variable annotations, we provide the following function that turns a rewriter for annotated terms into one for non-annotated terms:

```
rewriteWith :: (..., g ∼ (f :&: Set Name))
            ⇒ (Term g → Term g) → Term f → Term f
```

A typical use of this function is

```
rewriteWith (bottomUp (applyFirst ...)) :: (...) ⇒ Term f → Term f
```

where `f` is a functor without annotation.

4.3 Quantifying over Meta-Variables

Functions such as `applyFirst` 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 `rule_add` from Section 2 is

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

The `Quantifiable` type class automates the task of providing fresh meta-variables to functions like `rule_add`:

```
class Quantifiable rule where
  type RuleType rule
  quantify' :: Name → rule → RuleType rule

quantify :: (Quantifiable rule, RuleType rule ∼ Rule lhs rhs)
         ⇒ rule → Rule lhs rhs
quantify = quantify' 0

instance Quantifiable (Rule lhs rhs) where
  type RuleType (Rule lhs rhs) = Rule lhs rhs
  quantify' _ = id

instance (Quantifiable rule, m ∼ MetaId a) ⇒ Quantifiable (m → rule) where
  type RuleType (m → rule) = RuleType rule
  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. `MetaId` is the concrete representation of meta-variables.

Using `quantify`, we can package our rules in a list of the type expected by `applyFirst`:

```
rules = [ quantify (rule_add :: _ Int → _)
        , quantify (rule_sub :: _ Int → _)
        , quantify rule_mul ]
```

Note the use of a partial type signature to constrain the type of the meta-variable, which would otherwise be ambiguous.

5 Case Study – Feldspar

The repository contains an example file[5] 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`:

```
type Feld = VAR :+: LAM :+: APP :+: NUM :+: LOGIC :+: FORLOOP

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

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

Since we want to be able to construct for-loops in rules as well as in ordinary Feldspar expressions, we overload the for-loop using a type class:

```
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

[5] https://github.com/emilaxelsson/ho-rewriting/blob/0.2/examples/Feldspar.hs

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 such as `FORLOOP`.

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

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

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

A Feldspar simplifier is obtained by applying the simplification rules bottom-up as follows:

```
simplify :: Data a → Data a
simplify = Data . rewriteWith (bottomUp (applyFirst app rulesFeld)) . unData
```

The application operator `app` tells `applyFirst` to keep any redexes created by rewriting, and `rulesFeld` is a list of all the rules defined in this paper.

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:

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

We simplify the expression by running

```
*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 the simplifier removes both loops: the first one because it never changes the state, and the second one because its body does not refer to the previous state.

6 Related Work

Function patterns One of the problems solved in this paper is 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* [1,8,17], 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 [13]. 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.

Rewriting libraries Yokoyama et al. have made a library based on Template Haskell for higher-order rewriting of Haskell code [23]. 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 [15,9,5]. In particular, the work of van Noort et al. [15] 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. [9] also has an intensional representation of rules, but uses a simpler term representation: a rose tree extended with meta-variables.

As a concrete example, here is the rule for addition with zero, expressed with our library and with the library by van Noort et al.:

```
rule_add x = 0 + meta x      ⟹  meta x  -- our library
rule_add x = Add (Num 0) x  ⟼  x       -- van Noort's library
```

`Add` and `Num` are constructors of the regular data type for expressions. It seems plausible that their library can be combined with smart constructors to get rules that look more like in our library (with the added benefit of type-safety, etc.). We see no direct reason why one could not also extend their library to higher-order rewriting, given a suitable generic representation of variables and binders.

Strategic rewriting libraries such as KURE [19] provide a rich set of strategies for building complex traversals of data. The main focus in this paper is on rewrite rules rather than strategies – `bottomUp` and `topDown` being the two main strategies presented. It would be interesting to extend our library with more strategies, or perhaps even combine it with an existing library for strategic rewriting.

Pattern unification Higher-order pattern unification is at the core of systems that manipulate higher-order data like Twelf [18] and λProlog [14] or type reconstruction algorithms for dependently typed languages such as Agda [16]. 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 Discussion and Future Work

The motivation behind the presented library is for it to be used in the implementation of Feldspar. However, Feldspar is currently based on a different term representation. Our future plan is to rewrite Feldspar so that it can use the rewriting library for optimizations.

A key aspect of the library is the algorithmic efficiency of higher-order rewriting: the complexity of rule application is bounded only by the size of the rule, just like for first-order rewriting. However, we have not yet tested how well the library performs in practice for large problems. This remains as future work.

The presented library makes it possible to express higher-order rewrite rules in a safe way using clean syntax. However, one disadvantage of the library is that the error messages can be quite confusing due to the heavy type-level machinery involved. This is a common problem of embedded DSLs, but it may be solvable by recent work on type error diagnosis [20].

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.

```
rule_add :: ( ... ) ⇒ MetaRep lhs a → Rule lhs rhs
```

But rewriting functions such as `applyFirst` 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 such as `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). The anonymous referees provided useful input.

References

1. Antoy, S., Hanus, M.: Declarative programming with function patterns. In: Hill, P.M. (ed.) Logic Based Program Synthesis and Transformation, LNCS, vol. 3901, pp. 6–22. Springer Berlin Heidelberg (2006)
2. Axelsson, E., Claessen, K.: Using circular programs for higher-order syntax: Functional pearl. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. pp. 257–262. ACM, New York, NY, USA (2013)
3. Axelsson, E., Claessen, K., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajda, A.: Feldspar: A domain specific language for digital signal processing algorithms. In: 8th IEEE/ACM International Conference on Formal Methods and Models for Codesign. pp. 169–178. IEEE (2010)
4. Bahr, P., Hvitved, T.: Compositional data types. In: Proceedings of the seventh ACM SIGPLAN workshop on Generic programming. pp. 83–94. ACM, New York, NY, USA (2011)
5. Brown, N.C., Sampson, A.T.: Matching and modifying with generics. In: Draft proceedings of Trends in Functional Programming. pp. 304–318 (2008)

6. Carette, J., Kiselyov, O., Shan, C.c.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. Journal of Functional Programming 19(5), 509–543 (Sep 2009)
7. Cervesato, I., Pfenning, F.: A linear spine calculus. Journal of Logic and Computation 13(5), 639–688 (2003)
8. Dévai, G.: Extended pattern matching for embedded languages. Annales Univ. Sci. Budapestiensis de Rolando Eötvös Nominatae, Sectio Comutatorica 36, 277–297 (2012)
9. Felgenhauer, B., Avanzini, M., Sternagel, C.: A Haskell library for term rewriting. CoRR abs/1307.2328 (2013), http://arxiv.org/abs/1307.2328
10. Goldfarb, W.D.: The undecidability of the second-order unification problem. Theoretical Computer Science 13(2), 225 – 230 (1981)
11. Huet, G.: Résolution d'équations dans les langages d'ordre 1, 2,..., ω. Ph.D. thesis, Université Paris VII (1976)
12. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. Journal of logic and computation 1(4), 497–536 (1991)
13. Mohnen, M.: Context patterns in Haskell. In: Kluge, W. (ed.) Implementation of Functional Languages. LNCS, vol. 1268, pp. 41–57. Springer Berlin Heidelberg (1997)
14. Nadathur, G., Miller, D.: An overview of Lambda-Prolog. Tech. Rep. MS-CIS-88-40, University of Pennsylvania, Department of Computer and Information Science (1988)
15. van Noort, T., Rodriguez Yakushev, A., Holdermans, S., Jeuring, J., Heeren, B., Magalhães, J.P.: A lightweight approach to datatype-generic rewriting. Journal of Functional Programming 20, 375–413 (7 2010)
16. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (September 2007)
17. Oosterhof, N., Hölzenspies, P., Kuper, J.: Application patterns. In: van Eekelen, M. (ed.) Trends in Functional Programming. pp. 370–382. Tartu University Press, Tallinn (September 2005)
18. Pfenning, F., Schürmann, C.: System description: Twelf – a meta-logical framework for deductive systems. In: Proceedings of the 16th International Conference on Automated Deduction. pp. 202–206. Springer-Verlag, London, UK, UK (1999)
19. Sculthorpe, N., Frisby, N., Gill, A.: The kansas university rewrite engine. Journal of Functional Programming 24, 434–473 (2014)
20. Serrano, A., Hage, J.: Feedback upon feedback. Presented at Trends in Functional Programming (2015), ftp://ftp-sop.inria.fr/indes/TFP15/TFP2015_submission_22.pdf
21. Swierstra, W.: Data types à la carte. Journal of Functional Programming 18, 423–436 (6 2008)
22. Wierzbicki, T.: Complexity of the higher order matching. In: Proceedings of the 16th International Conference on Automated Deduction, LNCS, vol. 1632, pp. 82–96. Springer Berlin Heidelberg (1999)
23. Yokoyama, T., Hu, Z., Takeichi, M.: Design and implementation of deterministic higher-order patterns (2005), http://takeichi.ipl-lab.org/yicho/YoHT05.pdf