A Lightweight Approach to Datatype-Generic Rewriting

Thomas van Noort

Alexey Rodriguez

Stefan Holdermans

Johan Jeuring

Bastiaan Heeren

Technical Report UU-CS-2008-020
July 2008

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

A Lightweight Approach to Datatype-Generic Rewriting

Thomas van Noort[1] Alexey Rodriguez[2] Stefan Holdermans[2] Johan Jeuring[2,3] Bastiaan Heeren[3]

[1]Inst. for Computing and Information Sciences, Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
[2]Department of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
[3]School of Computer Science, Open University of the Netherlands, P.O. Box 2960, 6401 DL Heerlen, The Netherlands

thomas@cs.ru.nl {alexey,stefan,johanj}@cs.uu.nl bastiaan.heeren@ou.nl

Abstract

Previous implementations of generic rewriting libraries have a number of limitations: they require the user to either adapt the datatype on which rewriting is applied, or the rewriting rules are specified as functions, which makes it hard or impossible to document, test, and analyse them. We describe a library that demonstrates how to overcome these limitations by defining rules in terms of datatypes, and show how to use a type-indexed datatype to automatically extend a datatype for syntax trees with a case for metavariables. We then show how rewrite rules can be implemented without any knowledge of how the datatype is extended with metavariables. We use Haskell, extended with associated type synonyms, to implement both type-indexed datatypes and generic functions. We analyse the performance of our library and compare it with other approaches to generic rewriting.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism

General Terms Design, Languages

Keywords datatype-generic programming, term rewriting

1. Introduction

Consider a Haskell datatype *Prop* for representing expressions of propositional logic:

```
data Prop = Var String | T | F | Not Prop
          | Prop :∧: Prop | Prop :∨: Prop
```

Now suppose we wish to simplify such expressions using the law of contradiction: $p \wedge \neg p \rightarrow F$. We could then encode this rule as a function and apply it to an expression using a bottom-up traversal function *transform*:

```
simplify :: Prop → Prop
simplify prop = transform andContr prop
  where
    andContr (p :∧: Not q) | p ≡ q = F
    andContr p                     = p
```

Although this definition is relatively straightforward, encoding rules as functions has a number of drawbacks. To start with, a rule is not a concise one-line definition, because we have to provide a catch-all case in order to avoid pattern-matching failure at runtime. Second, pattern guards are needed to deal with multiple occurrences of a variable, cluttering the definition. Lastly, rules cannot be analysed easily since it is hard to inspect functions.

One way to solve these drawbacks is to design specialized rewriting functionality. We could, for example, define a datatype to represent rewriting rules on propositions, and implement some associated rewriting machinery that supports patterns with metavariable occurrences. While the drawbacks mentioned above are solved, this solution has a serious disadvantage: there is a large amount of datatype-specific code. If we wanted to use rewriting on, say, a datatype representing arithmetic expressions, we would have to define the rewriting functions and the datatype for rules again.

In this paper, we propose a rewriting library that is generic in the datatype on which rules are applied. Using our library, the example above can be written as follows:

```
simplify :: Prop → Prop
simplify prop = transform (applyRule andContr) prop
  where
    andContr p = p :∧: Not p :⇝ F
```

The library provides *transform*, *applyRule*, and (:⇝) which are generic and in this case, instantiated with the type *Prop*. There is no constructor for metavariables. Instead, we use ordinary function abstraction to make the metavariable p explicit in the rule *andContr*, which is now a direct translation of the rule $p \wedge \neg p \rightarrow F$. This rule description no longer suffers from the drawbacks of the solution that was based on pattern matching.

Specifically, these are the contributions of our paper:

- Our library implements term rewriting using generic programming techniques within Haskell extended with associated type synonyms (Chakravarty et al. 2005), as implemented in the Glasgow Haskell Compiler (GHC). In order to specify rewrite rules, terms have to be extended with a constructor for metavariables used in rules. This extension is achieved by making the rule datatype a type-indexed datatype (Hinze et al. 2004). The rewriting machinery itself is implemented using well-known generic functions such as *map*, *crush*, and *zip*.
- We present a new technique to specify metalevel information without the need to modify the domain datatype definitions. We use it to make rule specifications in our library user-friendly: rules are defined using the original term constructors. This is remarkable because the values of type-indexed datatypes (which in our library are used to represent terms extended with metavariables) must be built from different constructors than those from the user program (*Prop*, in our example). Thus, we hide implementation details about the use of type-indexed types from the library's users. This technique can also be applied to other generic programs such as the Zipper (Huet 1997).

Besides these contributions, we think our rewriting library is an elegant example of how to implement type-indexed types in a lightweight fashion by means of associated type synonyms. Most

examples of type-indexed types (Hinze et al. 2004; Van Noort 2008) have been implemented using a language extension such as Generic Haskell (Löh 2004). Other examples of lightweight implementations of type-indexed types have been given by Chakravarty et al. (2008), and Oliveira and Gibbons (2005).

We are aware of at least one other generic programming library for rewriting in Haskell. Jansson and Jeuring (2000) implement a generic rewriting library in PolyP, an extension of Haskell with a special construct for generic programming. Our library differs in a number of aspects. First, we use no specific generic-programming extensions of Haskell. This is a minor improvement, since we expect that Jansson and Jeuring's library can easily be translated to plain Haskell as well. Second, we use a type-indexed data type for specifying rules. This is a major difference, since it allows us to generically extend a datatype with metavariables. In Jansson and Jeuring's library, a datatype either has to be extended by hand, forcing users to introduce new constructors, or one of the constructors of the original datatype is reused for metavariables. Neither solution is very satisfying, since either the rules have to be specified in the new datatype using different constructor names, or we introduce a safety problem in the library since an object variable might accidentally be considered to be a metavariable.

This paper is organised as follows. Section 2 continues the discussion on how to represent rewrite rules, and motivates the design choices we made in our generic library for term rewriting. We then present the interface of our library in Section 3 from a user's perspective, followed by a detailed description of the implementation of our library and the underlying machinery in Section 4. We compare the efficiency or our library to other approaches to term rewriting in Section 5. Finally, we discuss related work in Section 6, and we present our conclusions and ongoing research in Section 7.

2. Motivation

There are at least two techniques to implement rule-based rewriting in Haskell: "rules based on pattern matching" and "rules as values of datatypes".

2.1 Rules Based on Pattern Matching

The first technique encodes rewrite rules as Haskell functions, using pattern matching to check whether the argument term matches the left-hand side of the rule. If this is the case, then we return the right-hand side of the rule, or else, we return the term unchanged. For example, the rule $\neg(p \wedge q) \rightarrow \neg p \vee \neg q$ is implemented as follows:

```
deMorgan :: Prop → Prop
deMorgan (Not (p :∧: q)) = Not p :∨: Not q
deMorgan p               = p
```

We have to add a catch-all case, because otherwise a runtime failure is caused by an argument that does not match the pattern.

Rules containing variables with multiple occurrences in the left-hand side cannot be encoded as Haskell functions directly. Instead, such occurrences must be enforced in so-called pattern guards. For example, $p \vee \neg p \rightarrow T$ is implemented by:

```
exclMiddle :: Prop → Prop
exclMiddle (p :∨: Not q) | p ≡ q = T
exclMiddle p                     = p
```

Here we have replaced the second occurrence of p with a fresh variable q, but we have retained the equality constraint as a pattern guard ($p \equiv q$). Note that this requires the availability of an equality function for the type *Prop*.

In some situations, it is useful to know whether a rule has been successfully applied or not. We provide this information by returning the rewriting result in a *Maybe* value, at the expense of some additional notational overhead:

```
exclMiddleM :: Prop → Maybe Prop
exclMiddleM (p :∨: Not q) | p ≡ q = Just T
exclMiddleM p                     = Nothing
```

Still, encoding rules as functions is very convenient because we can directly use generic traversal combinators that are parameterised by functions. For example, the Uniplate library (Mitchell and Runciman 2007) defines *transform*,

```
transform :: Uniplate a ⇒ (a → a) → a → a
```

which applies its argument in a bottom-up fashion, and many other traversal combinators.

These combinators can, for example, be used to remove tautological statements from a proposition:

```
removeTaut :: Prop → Prop
removeTaut = transform exclMiddle
```

Pattern matching allows for rewrite rules to be encoded more or less directly as functions. Furthermore, on account of their functional nature, rules can be immediately combined with generic programming strategies in order to control their application. However, having rules as functions raises a number of issues:

- The rules cannot be observed easily because it is not possible to directly inspect a function in Haskell. There are several reasons why it would be desirable to observe rules, such as:
  - Documentation: The rules of a rewriting system can be pretty printed to generate documentation.
  - Automated testing: In general, a rule must preserve the semantics of the expression that it rewrites. A way to test this property is to randomly generate terms and check whether the rewritten version preserves the semantics. However, a rule with a complex left-hand side will most likely not match a randomly generated term, and hence it will not be tested sufficiently. If the left-hand side of a rule were inspectable, we could tweak the generation process in order to improve the testing coverage.
  - Inversion: The left-hand side and right-hand side of a rule could be exchanged, resulting in the inverse of that rule.
  - Tracing: When a sequence of rewriting steps leads to an unexpected result, you may want to learn which rules were applied in which order.
- The lack of nonlinear pattern matching in Haskell can become a nuisance if the left-hand side of a rule contains many occurrences of the same variable.
- It is tedious to have to specify a catch-all case when rules are encoded as functions. All rule definitions require this extra case.
- Haskell is not equipped with first-class pattern matching, so the user cannot write abstractions for commonly occurring structures in the left-hand side of a rule.

2.2 Rules as Values of Datatypes

Instead of using functions, we use a datatype to encode our rewrite rules, which makes left-hand sides and right-hand sides observable:

```
data RuleSpec a = a :⇝ a
```

In the case of propositions, we have to introduce a variation of the *Prop* datatype, which has an extra constructor *MetaVar* for metavariables:

```
data EProp = MetaVar String
           | EVar String | ET | EF | ENot EProp
           | EProp :∧: EProp | EProp :∨: EProp
```

This extended datatype *EProp* is used to define rewrite rules. Of course, we need to define a rewriting function specific to propositions that uses rules expressed by value of *EProp* to transform expressions:

```
rewriteProp :: RuleSpec EProp → Prop → Prop
```

Here we do not show the implementation of *rewriteProp*, but note that its implementation would be specific to propositions. If we want to specify rewrite rules that work on different datatype, then we also have to define new *rewrite* functions for those datatypes. With the specific rewrite function for propositions, for instance, we can use rules on propositions in the following way:

```
simplifyProp :: Prop → Prop
simplifyProp = transform (rewriteProp exclMiddle)
  where
    exclMiddle =
      MetaVar "p" :∨: ENot (MetaVar "p") :⇝ ET
```

In this definition, another problem becomes apparent. The way in which rules are specified is very inconvenient, because, rather than the type *Prop*, we are required to use the new datatype *EProp* with its different constructor names. Furthermore, this definition is not explicit about which metavariables are used. So, the rewrite function is responsible for verifying that each metavariable occurring in the right-hand side side of the rule is actually bound at the left-hand side.

3. Interface

The interface to our generic rewriting library is presented in Figure 1. We explain it briefly by means of several examples. We first specify a few rules to be used with the library:

```
notTrue    ::                    RuleSpec Prop
andContr   ::          Prop → RuleSpec Prop
deMorgan   :: Prop → Prop → RuleSpec Prop

notTrue        = Not T          :⇝ F
andContr  p    = p :∧: Not p    :⇝ F
deMorgan p q   = Not (p :∧: q)  :⇝ Not p :∨: Not q
```

Rule specifications represent metavariables as function arguments. For example, the *deMorgan* rule uses two metavariables, so they are introduced as function arguments p and q. Rules with no metavariables, such as *notTrue*, do not take any argument.

Rules must first be transformed to an internal representation before they can be used for rewriting. Let us first take a look at the functions $rule_0$, $rule_1$, and $rule_2$. These take a rule specification and return a rule. Each function handles rules with a specific number of metavariables. In principle, the user would have to pick the right function to build a rule, but in practice, it is more convenient to abstract over the number of metavariables by means of a type class. The library provides an overloaded function *rule* for building rules from specifications with any number of metavariables[1]:

```
notTrueRule, andContrRule, deMorganRule :: Rule Prop

notTrueRule    = rule notTrue
andContrRule   = rule andContr
deMorganRule   = rule deMorgan
```

The library defines two rewriting functions, namely *rewriteM* and *rewrite*. The first has a more informative type: if the term does not match against the rule, the monad is used to notify about the failure. The second rewriting function always succeeds, and it returns the term unchanged whenever the rule is not applicable. Both functions are straightforward to use with built rules. Consider the following expressions:

```
rewrite   notTrueRule  (Not T)
rewrite   andContrRule (Var "x" :∧: Not (Var "x"))
rewriteM deMorganRule (T :∧: F)
```

These expressions evaluate to *F*, *F*, and *Nothing* respectively. Sometimes, it is more practical to directly apply a rule specification, without calling the intermediate compilation step. The function *applyRule* can be used for this purpose:

```
applyRule deMorgan (Not (T :∧: Var "x"))
```

This expressions evaluates to `Not T :∨: Not (Var "x")`.

```
-- Rule specifications and rules (ADT)
data RuleSpec a = a :⇝ a
type Rule a

-- Build rules from specifications with 0, 1, and 2 metavariables
rule0 :: Rewrite a ⇒                RuleSpec a  → Rule a
rule1 :: Rewrite a ⇒ (      a → RuleSpec a) → Rule a
rule2 :: Rewrite a ⇒ (a → a → RuleSpec a) → Rule a

-- Build rules from specifications with any number of metavariables
rule :: (Builder r, Rewrite (Target r)) ⇒ r → Rule (Target r)

-- Application of rules to terms
rewriteM :: (Rewrite a, Monad m) ⇒ Rule a → a → m a
rewrite  :: Rewrite a            ⇒ Rule a → a → a

-- Application of rule specifications to terms
applyRule :: (Builder r, Rewrite (Target r))
          ⇒ r → Target r → Target r
```

Figure 1. Generic rewriting library interface

3.1 Representing the Structure of Datatypes

To enable generic rewriting on a datatype, a user must describe the structure of that datatype to the library. This is a process analogous to that of Scrap Your Boilerplate (Lämmel and Peyton Jones 2003) and Uniplate. These libraries require datatypes to be instances of the classes *Data* and *Uniplate* respectively.

In our library, the structure of a regular datatype is given by an instance of the type class *Regular*. Figure 2 shows the definition of *Regular* and the type constructors used to describe type structure. This type class declares *PF*, an associated type synonym of kind $* \longrightarrow *$ that abstracts over the immediate subtrees of a datatype and captures the notion of *pattern functor*. For example, Figure 3 shows the complete definition of the *Regular* instance for *Prop*. The associated type synonym *PF* corresponds to PolyP's type constructor *FunctorOf* (Jansson and Jeuring 1997; Norell and Jansson 2004). Its instantiation follows directly from the definition of a datatype. Choice amongst constructors is encoded by nested sum types (:+:); therefore, five sums are used above to encode six constructors. A constructor with no arguments is encoded by *Unit*; this is the case for the second and third constructors (*T* and *F*). Constructor arguments that are recursive occurrences (such as that for *Not*) are encoded by *Id*. Other constructor arguments are encoded by *K*, and, hence, the argument of *Var* is encoded in that way. Finally, constructors with more than one argument (like (:∧:) and (:∨:)) are encoded by nested product types (:*:).

[1] The associated type synonym *Target* returns the datatype to which the rule is applied. For example, *Target* (*Prop* → *RuleSpec Prop*) yields *Prop*. For more details, see Section 4.4.

```
class Functor (PF a) ⇒ Regular a where
  type PF a :: ∗ → ∗
  from      :: a      → PF a a
  to        :: PF a a → a
data K a r     = K a
data Id r      = Id r
data Unit r    = Unit
data (f :+: g) r = Inl (f r) | Inr (g r)
data (f :∗: g) r = f r :∗: g r
infixr 7 :∗:
infixr 6 :+:
```

Figure 2. *Regular* type class and view types

```
instance Regular Prop where
  type PF Prop = K String :+: Unit :+: Unit :+: Id
                 :+: Id :∗: Id
                 :+: Id :∗: Id
  from (Var s)   = Inl (K s)
  from T         = Inr (Inl Unit)
  from F         = Inr (Inr (Inl Unit))
  from (Not p)   = Inr (Inr (Inr (Inl (Id p))))
  from (p :∧: q) = Inr (Inr (Inr (Inr (Inl (Id p :∗: Id q)))))
  from (p :∨: q) = Inr (Inr (Inr (Inr (Inr (Id p :∗: Id q)))))
  to (Inl (K s))                                  = (Var s)
  to (Inr (Inl Unit))                             = T
  to (Inr (Inr (Inl Unit)))                       = F
  to (Inr (Inr (Inr (Inl (Id p)))))               = (Not p)
  to (Inr (Inr (Inr (Inr (Inl (Id p :∗: Id q)))))) = (p :∧: q)
  to (Inr (Inr (Inr (Inr (Inr (Id p :∗: Id q)))))) = (p :∨: q)
```

Figure 3. Complete *Regular* instance for *Prop*

The two class methods of *Regular*, *from* and *to*, relate datatype values with the structural representation of those values. Together, they establish a so-called *embedding-projection pair*: i.e., they should satisfy $to \circ from = id$ and $from \circ to \sqsubseteq id$ (Hinze 2000). More specifically, *from* transforms the top-level constructor into a structure value, while leaving the immediate subtrees unchanged. The function *to* performs the transformation in the opposite direction.

We want to stress here that, although the instance declaration for *Prop* may seem quite verbose, it follows directly and mechanically from the structure of the datatype. Defining instances of *Regular* could easily be done automatically, for example by using Template Haskell (Sheard and Peyton Jones 2002). Moreover, all that needs to be done to use the generic rewriting library on a datatype is declaring it an instance of *Regular* of *Rewrite* (see Section 4.5).

4. Implementation

In this section, we describe the implementation of the library. We proceed as follows. First, we explain how generic functionality is implemented for datatypes that are instances of *Regular*. In particular, we explain how to implement *map*, *crush*, and *zip* generically over pattern functors. Next, we present the implementation of generic rewriting using these functions. Finally, we describe how to support rule specifications that use the original datatype constructors.

4.1 Generic Functions

Generic functions are defined once and can be used on any datatype that is an instance of *Regular*. The definition of a generic function is given by induction on the types used to describe the structure of a pattern functor. In our library, generic function definitions are given using type classes.

```
class Functor f where
  fmap :: (a → b) → f a → f b
instance Functor Id where
  fmap f (Id r) = Id (f r)
instance Functor (K a) where
  fmap _ (K x) = K x
instance Functor Unit where
  fmap _ Unit = Unit
instance (Functor f, Functor g) ⇒ Functor (f :+: g) where
  fmap f (Inl x) = Inl (fmap f x)
  fmap f (Inr y) = Inr (fmap f y)
instance (Functor f, Functor g) ⇒ Functor (f :∗: g) where
  fmap f (x :∗: y) = fmap f x :∗: fmap f y
```

Figure 4. *fmap* definitions

4.1.1 Generic Map

Mapping over the elements of a pattern functor is defined by means of the *Functor* type class, as given in Figure 4. Here, the interesting case is the transformation of pattern functor elements, which are stored in the *Id* constructors. Note that this is a well-known way to implement generic functions, derived from how generic functions are implemented in PolyP.

Now, using *fmap*, we define the *compos* operator of Bringert and Ranta (2006), which applies a function to the immediate subtrees of a value. A pattern functor abstracts precisely over those subtrees, so we use *Regular* to define *compos* generically:

```
compos :: Regular a ⇒ (a → a) → a → a
compos f = to ∘ fmap f ∘ from
```

The function *compos* transforms the a argument into a value of the pattern functor, so that it can apply f to the immediate subtrees using *fmap*. The version of *compos* given here is equivalent to PolyP's *mapChildren* (Jansson and Jeuring 1998); we could also define monadic or applicative variants of this operator by generalising generic mapping in a similar fashion.

Generic *compos* can be used, for example, to implement the bottom-up traversal *transform*, used in the introduction:

```
transform :: Regular a ⇒ (a → a) → a → a
transform f = f ∘ compos (transform f)
```

4.1.2 Generic Crush

A *crush* is a useful fold-like operation on pattern functors. It combines all the a values within pattern-functor value into a b-value using a binary operator $(\oplus)$ and an initial b-value e. The definition of *crush* is given in Figure 5.

A common use of *crush* is collecting all recursive elements of a pattern-functor value:

```
flatten :: Crush f ⇒ f a → [a]
flatten = crush (:) []
```

If a datatype is an instance of *Regular*, we obtain the immediate recursive occurrences of a value by means of *children*:

```
children :: (Regular a, Crush (PF a)) ⇒ a → [a]
children = flatten ∘ from
```

4.1.3 Generic Zip

Generic *zip* over pattern functors is crucial to the implementation of rewriting. Figure 6 shows the definition of generic *zip* over pattern

```
class Crush f where
  crush :: (a → b → b) → b → f a → b
instance Crush Id where
  crush (⊕) e (Id x) = x ⊕ e
instance Crush (K a) where
  crush _ e _ = e
instance Crush Unit where
  crush _ e _ = e
instance (Crush a, Crush b) ⇒ Crush (a :+: b) where
  crush (⊕) e (Inl x) = crush (⊕) e x
  crush (⊕) e (Inr y) = crush (⊕) e y
instance (Crush a, Crush b) ⇒ Crush (a :*: b) where
  crush (⊕) e (x :*: y) = crush (⊕) (crush (⊕) e y) x
```

Figure 5. *crush* definitions

```
class Zip f where
  fzipM :: Monad m ⇒ (a → b → m c) → f a → f b → m (f c)
instance Zip Id where
  fzipM f (Id x) (Id y) = liftM Id (f x y)
instance Eq a ⇒ Zip (K a) where
  fzipM _ (K x) (K y)
    | x ≡ y       = return (K x)
    | otherwise = fail "fzipM: structure mismatch"
instance Zip Unit where
  fzipM _ Unit Unit = return Unit
instance (Zip a, Zip b) ⇒ Zip (a :+: b) where
  fzipM f (Inl x) (Inl y) = liftM Inl (fzipM f x y)
  fzipM f (Inr x) (Inr y) = liftM Inr (fzipM f x y)
  fzipM _ _ _ = fail "fzipM: structure mismatch"
instance (Zip a, Zip b) ⇒ Zip (a :*: b) where
  fzipM f (x1 :*: y1) (x2 :*: y2) =
    liftM2 (:*:) (fzipM f x1 x2) (fzipM f y1 y2)
```

Figure 6. *fzipM* definitions

functors, *fzipM*, which takes a function that combines the a and b values that are stored in the pattern functor structures. It traverses both structures in parallel and merges all occurrences of a and b values along the way. The function has a monadic type because the structures may not match in the case of sums (which corresponds to different constructors) and constant types (because of different values stored in K). However, the monadic type also allows the merging function to fail or to use state, for example.

There are some useful variants of *fzipM*, such as a zip that uses a non-monadic merging function:

```
fzip :: (Zip f, Monad m)
     ⇒ (a → b → c) → f a → f b → m (f c)
fzip f = fzipM (λx y → return (f x y))
```

and a partial generic zip that does not have a monadic return type:

```
fzip' :: Zip f ⇒ (a → b → c) → f a → f b → f c
fzip' f x y =
  case fzip f x y of
    Just res → res
    Nothing → error "fzip': structure mismatch"
```

4.2 Generic Rewriting

Now that we have defined basic generic functions, we continue with defining the basic rewriting machinery. Rules consist of a left-hand side and a right-hand side which are combined using the infix constructor (:⇝):

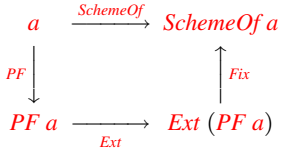
```
data RuleSpec a = a :⇝ a
lhs, rhs :: RuleSpec a → a
lhs (x :⇝ _) = x
rhs (_ :⇝ y) = y
```

4.2.1 Schemes

What is the type of rules that rewrite *Prop*-values? The type *RuleSpec Prop* is a poor choice, because such rules cannot contain metavariables. Our solution is to define a type that adds a metavariable case to a datatype. For this purpose, we define a type-indexed type *SchemeOf* for *schemes*. A scheme extends its argument with a metavariable constructor. Now, the type of rules that rewrite *Prop*-values is *RuleSpec* (*SchemeOf Prop*). We define a type synonym *Rule* such that this type can be written more concisely as *Rule Prop*:

```
type Rule a = RuleSpec (SchemeOf a)
```

The process of obtaining *SchemeOf a* for a given type a can be depicted as follows:



To define *SchemeOf a*, we assume that a is an instance of *Regular* and, hence, has an associated pattern functor *PF a*. Furthermore, we extend the pattern functor with a case for metavariables. The type of schemes of a is then defined as the fixed point of the now extended pattern functor.

To extend a pattern functor with a case for metavariables, it is, in essence, enough to introduce a sum to encode the choice between the extra metavariable case and a value from the original pattern functor:

```
type Ext f = K MetaVar :+: f
```

However, *Ext* extends a type with metavariables on the top-level only, whereas ee also have to allow metavariables to occur in subterms. In other words, the pattern functor has to be extended *recursively*. To this end, we introduce a type synonym *Scheme* that encodes the recursive structure of schemes by means of a type-level fixed-point operator *Fix*:

```
newtype Fix f = In{out :: f (Fix f)}
type Scheme f = Fix (Ext f)
```

A scheme for a given regular type is now defined in terms of the type-indexed type *PF*:

```
type SchemeOf a = Scheme (PF a)
```

Our use of type-indexed types is simpler than in other applications such as the Zipper, because there are no types defined by induction on the pattern functor.

It remains to define the type *MetaVar*. In our library, metavariables are defined by *Int*-values:

```
type MetaVar = Int
```

The choice for *Int* is rather arbitrary, and other representations could be used as well. In Sections 4.3 and 4.4 we show that the use of *MetaVar*-values can be made internal to the library and that the concrete representation of metavariables need not be exposed to the user.

To easily construct *Scheme*-values we define two helper functions, which construct a metavariable, or a pattern functor value, respectively.

```
metaVar :: MetaVar → Scheme f
metaVar = In ∘ Inl ∘ K

pf :: f (Scheme f) → Scheme f
pf = In ∘ Inr
```

Rewriting functions are often defined by case analysis on values of type *Scheme*. Therefore, we use a view on *Scheme* to conveniently distinguish metavariables from pattern functor values.

```
data SchemeView f = MetaVar MetaVar | PF (f (Scheme f))

schemeView :: Scheme f → SchemeView f
schemeView (In (Inl (K x))) = MetaVar x
schemeView (In (Inr r))     = PF r
```

We also define a function to embed a-values into their extended counterparts.

```
toScheme :: Regular a ⇒ a → SchemeOf a
toScheme = pf ∘ fmap toScheme ∘ from
```

Finally, we define a fold on *Scheme* values that applies its argument functions to either a metavariable or a pattern functor value.

```
foldScheme :: Functor f
           ⇒ (MetaVar → a) → (f a → a) → Scheme f → a
foldScheme f g scheme =
  case schemeView scheme of
    MetaVar x → f x
    PF r      → g (fmap (foldScheme f g) r)
```

4.2.2 Basic Rewriting

When rewriting a term with a given rule, the left-hand side of the rule is matched to the term, resulting in a substitution mapping metavariables to terms:

```
type Subst a = Map MetaVar (a, SchemeOf a)
```

We store both the original term a and the corresponding scheme *SchemeOf* a for efficiency reasons. This approach prevents the matched subterm from being converted multiple times since the right-hand side of a rule can contain multiple occurrences of a metavariable. Instead, each occurrence is instantiated just by selecting the second component from the substitution.

A substitution is obtained by the function *match*, which is passed a scheme and a value of the original type:

```
match :: (Regular a, Crush (PF a), Zip (PF a), Monad m)
      ⇒ SchemeOf a → a → m (Subst a)
match scheme term =
  case schemeView scheme of
    MetaVar x →
      return (singleton x (term, toScheme term))
    PF r      →
      fzip (,) r (from term) ≫=
      crush matchOne (return empty)
  where
    matchOne (term1, term2) msubst =
      do subst1 ← msubst
         subst2 ← match (apply subst1 term1) term2
         return (union subst1 subst2)
```

When we encounter a metavariable we return the singleton substitution mapping the metavariable to the original term and its representation as a scheme. Otherwise, we zip the two terms to check that the structures match. Then, we obtain a single substitution by matching each recursive occurrence and returning the union of the resulting substitutions. In the definition of *matchOne*, we apply the substitution obtained thus far to the first term using the function *apply*. This function enforces linear patterns by instantiating each metavariable for which there is a binding. This also guarantees that the resulting substitution does not overlap with the substitution obtained thus far, since multiple occurrences of a metavariable are replaced throughout the term. As a consequence, we can just return the union of the two substitutions.

The function *apply* is defined in terms of *foldScheme*:

```
apply :: Regular a
      ⇒ Subst a → SchemeOf a → SchemeOf a
apply subst = foldScheme findMetaVar pf
  where
    findMetaVar x = maybe (metaVar x) snd (lookup x subst)
```

When a metavariable is encountered, we lookup the corresponding term in the second component of the substitution. A pattern functor value is reconstructed using the function *pf*.

The function *inst*, which is similar to *apply*, instantiates each metavariable in a term and returns a value of the original datatype:

```
inst :: Regular a ⇒ Subst a → SchemeOf a → a
inst subst = foldScheme findMetaVar to
  where
    findMetaVar x =
      maybe (error "inst: unbound metavariable")
            fst
            (lookup x subst)
```

Again, we lookup the corresponding term in the substitution. Since we construct a value of the original datatype, unbound metavariables are not allowed and result in a runtime error. A pattern functor value is converted to a value of the original datatype using *to*.

Finally, we combine the functions defined for matching a term and instantiating a term in the definition of *rewriteM*:

```
rewriteM :: (Regular a, Crush (PF a), Zip (PF a), Monad m)
         ⇒ Rule a → a → m a
rewriteM r term =
  do subst ← match (lhs r) term
     return (inst subst (rhs r))
```

We match the left-hand side of the rule to the given term, resulting in a substitution. Then, we use this substitution to instantiate the right-hand side of the rule. Since the matching process might fail, a monadic computation is returned. A similar library function is *rewrite*, defined in terms of *rewriteM*:

```
rewrite :: (Regular a, Crush (PF a), Zip (PF a))
        ⇒ Rule a → a → a
rewrite r term = maybe term id (rewriteM r term)
```

This function always succeeds since the original term is returned when rewriting fails.

4.2.3 Example

At this point, we can already use generic rewriting to implement the example presented in the introduction. However, as we will see, the rewriting library interface is not yet very user-friendly.

Recall that a rule for rewriting propositions has type *Rule Prop*. Hence, we define the rule *andContr* using a scheme representation:

```
andContr :: Rule Prop
andContr = p `pAnd` pNot p :⇝ pF
  where
    p = metaVar 1
```

We abbreviate the structure encodings of *Prop* values in order to improve readability. Functions *pF*, *pNot*, and *pAnd* are defined as follows:

```
pF :: SchemeOf Prop
pF = pf (Inr (Inr (Inl Unit)))

pAnd ::
  SchemeOf Prop → SchemeOf Prop → SchemeOf Prop
pAnd p q = pf (Inr (Inr (Inr (Inl (Id p :*: Id q)))))

pNot :: SchemeOf Prop → SchemeOf Prop
pNot p = pf (Inr (Inr (Inr (Inl (Id p)))))
```

Except for the application of *pf*, structure values resemble those used in the *Regular* instance for *Prop*. In fact, it is possible to define *pVar*, *pT*, and *pF* more concisely by using *toScheme*:

```
pVar :: String → SchemeOf Prop
pVar s = toScheme (Var s)

pT :: SchemeOf Prop
pT = toScheme T

pF :: SchemeOf Prop
pF = toScheme F
```

Unfortunately, *pNot*, *pAnd*, and *pOr* cannot be defined similarly. Consider the following failed attempt for *Not*:

```
badPNot :: SchemeOf Prop → SchemeOf Prop
badPNot p = toScheme (Not p)
```

This definition is not correct because the constructor *Not* cannot take a *SchemeOf Prop* value as an argument. It follows that we cannot use *Not* and *toScheme*, and we are forced to define *pNot* redundantly using the structure value of *Not*.

There are two problems with using generic rewriting as presented so far. First, in order to write concise rules, the user will need to define abbreviations for the structure representations of constructors, such as *pF* and *pAnd*. Second, such abbreviations force the user into relating constructors to their structure representations yet again, even though this is already made explicit in the *Regular* instance for the datatype.

4.3 Metavariables as Function Arguments

We have demonstrated how generic term rewriting is implemented in our library. A second contribution of this paper is that we show how our library allows its users to specify rules using just the constructors of the original datatype definition. This is a clear improvement over the rewriting example above, because a user does not need to directly manipulate structure values. While such specifications are given using the original constructors, they still need to be transformed into *SchemeOf* values so that the rewriting machinery defined earlier can be used. We now discuss how to transform these rule specifications into the internal library representation of rules, or internal rules for short.

We start with the simpler case of rules that contain no metavariables. An example of such a rule is *notTrue*:

```
notTrue :: RuleSpec Prop
notTrue = Not T :⇝ F
```

Because the rule does not contain metavariables, it is sufficient to apply *toScheme* to perform the conversion:

```
rule0 :: Regular a ⇒ RuleSpec a → Rule a
rule0 r = toScheme (lhs r) :⇝ toScheme (rhs r)
```

In our system, rules with metavariables are specified by functions from terms to rules. For example, the rule *andContr* is modified as follows to represent the metavariable:

```
andContr :: Prop → RuleSpec Prop
andContr p = p :∧: Not p :⇝ F
```

A rule with one metavariable is represented by a function with one argument that returns a *RuleSpec* value, where the argument can be seen as a placeholder for the metavariable. This is convenient for programmers because metavariables are specified in the same way for different datatypes, rule specifications are more concise, scope checking is performed by the compiler, and implementation details such as variable names and the internal representation of terms remain hidden.

To use these specifications for rewriting, we have to transform them into the internal representation for rules. Consider the function that performs this transformation for rules with one metavariable:

```
rule1 :: (Regular a, Zip (PF a), LR (PF a))
      ⇒ (a → RuleSpec a) → Rule a
rule1 r = introMetaVar (lhs ∘ r) :⇝ introMetaVar (rhs ∘ r)
```

Function *introMetaVar* performs the transformation from a function on terms to a term extended with metavariables. The value *r* is composed to yield either the left-hand side or right-hand side of the rule. The context *LR* (*PF a*) provides support for extending a term with metavariables, we defer the discussion of this predicate for the moment.

The function passed to *introMetaVar* is not allowed to inspect the metavariable argument: the metavariable can only be used as part of the constructed term. As our library relies on this property, the user should not inspect metavariables in rule definitions. For example, the following rule is invalid:

```
bogusRule :: Prop → RuleSpec Prop
bogusRule (Var _) = T          :⇝ F
bogusRule p       = p :∧: Not p :⇝ F
```

Ideally, the restriction that metavariables are not inspected are encoded in the type system, so that rules like *bogusRule* are ruled out statically. However, to the best of our knowledge, it is impossible to enforce this restriction without changing the user's datatype.

A rule function that uses but does not inspect its metavariable argument has the property that if two different values are passed to that function, the resulting rule values will only differ at the places where the metavariable argument occurs. The function *insertMetaVar* exploits this property and inserts a metavariable exactly at the places where the two term representations differ:

```
insertMetaVar :: (Regular a, Zip (PF a))
              ⇒ MetaVar → a → a → SchemeOf a
insertMetaVar v x y =
  case fzip (insertMetaVar v) (from x) (from y) of
    Just str  → pf str
    Nothing → metaVar v
```

Function *insertMetaVar* traverses two pattern functor values in parallel using *fzip*, constructing a scheme (*SchemeOf a*) during the traversal. Whenever the two structures are different, a metavariable is returned. Otherwise, the recursive occurrences of the matching structures are traversed with *insertMetaVar v*.

We now define *introMetaVar*:

```
introMetaVar :: (Regular a, Zip (PF a), LR (PF a))
             ⇒ (a → a) → SchemeOf a
introMetaVar f = insertMetaVar 1 (f left) (f right)
```

Function *insertMetaVar* requires that the two term arguments are different at metavariable occurrences, so we apply *f* to two values, *left* and *right*, such that they cause the failure of *fzip*. In particular, we want the following property to hold:

$$\forall f.\ fzip\ f\ (from\ left)\ (from\ right) \equiv Nothing$$

Informally, this property states that *left* and *right* produce values of the same type with different top-level constructors (or the same top-level constructor but different values at a non-recursive position). This follows from the fact that *fzip* only fails for incompatible

```
class LR f where
  leftf  :: a → f a
  rightf :: a → f a
instance LR Id where
  leftf  x = Id x
  rightf x = Id x
instance LRBase a ⇒ LR (K a) where
  leftf  _ = K leftb
  rightf _ = K rightb
instance LR Unit where
  leftf  _ = Unit
  rightf _ = Unit
instance (LR f, LR g) ⇒ LR (f :+: g) where
  leftf  x = Inl (leftf x)
  rightf x = Inr (leftf x)
instance (LR f, LR g) ⇒ LR (f :*: g) where
  leftf  x = leftf  x :*: leftf  x
  rightf x = rightf x :*: rightf x
```

Figure 7. *leftf* and *rightf* definitions

pattern functor values, which represent the top-level constructors. It is important that *left* and *right* differ at the top-level (i.e., as soon as possible), otherwise metavariables would be inserted deeper than intended. Functions *left* and *right* are defined as follows:

```
left :: (Regular a, LR (PF a)) ⇒ a
left = to (leftf left)
right :: (Regular a, LR (PF a)) ⇒ a
right = to (rightf left)
```

These definitions use auxiliary functions *leftf* and *rightf* that generate a pattern functor value and convert it to the expected a using *to*. The arguments of *leftf* and *rightf* are the recursive occurrences, here we give the same argument in both cases (*left*) to emphasize that the difference must be at top-level. These auxiliary functions are defined generically, that is, by induction on the structure of the pattern functor. Figure 7 shows the definitions of both functions as methods of the type class *LR*. The constant case uses an additional type class, which is shown in Figure 8.

The property for *fzip* above is restated for *leftf* and *rightf* as follows:

```
∀f x y. fzip f (leftf x) (rightf y) ≡ Nothing
```

Note that the *LR* instances for *Unit* and *Id* do not satisfy the above property. We explain why we chose this suboptimal formulation. Ideally, we would prefer to enforce the above property statically for all pattern functors, and replace the *LR* instances for *Unit* and *Id* by weaker instances of the type class *One*, where *One* produces a single functor value. Then, we could write

```
instance (One f, One g) ⇒ LR (f :+: g) where ...
```

Unfortunately, the product case makes this scheme unrealistic:

```
instance (LR  f, One g) ⇒ LR (f :*: g) where ...
instance (One f, LR  g) ⇒ LR (f :*: g) where ...
```

To generate two different product values, at least one of the components should be inhabited by two different values. However, these two overlapping instances cannot be expressed in Haskell. So, we adopt the current simpler approach, where, to satisfy the property above, we require the pattern functor to contain at least one sum or one constant case. Furthermore, note that the current instance of sum cannot be given as, for example, *Inl* ⊥ and *Inr* ⊥, because the function *to* may evaluate to ⊥ when presented with such values.

```
class LRBase a where
  leftb  :: a
  rightb :: a
instance LRBase Int where
  leftb  = 0
  rightb = 1
instance LRBase [a] where
  leftb  = []
  rightb = [error "Should never be inspected"]
```

Figure 8. *leftb* and *rightb* definitions

We call *leftf* recursively in both sum methods, to emphasize that different sum choices are enough for *zip* to fail.

With a slight generalization we allow the specification of rules with two metavariables. For example, consider the following rule:

```
deMorgan :: Prop → Prop → RuleSpec Prop
deMorgan p q = Not (p :∧: q) :⇝ Not p :∨: Not q
```

To use the *deMorgan* rule with our rewriting machinery we need a variant of $rule_1$ that handles two metavariables:

```
rule2 :: (Regular a, Zip (PF a), LR (PF a))
      ⇒ (a → a → RuleSpec a) → Rule a
rule2 r =
  introMetaVar2 (λx y → lhs (r x y)) :⇝
  introMetaVar2 (λx y → rhs (r x y))
```

The real work is carried out by $introMetaVar_2$.

```
introMetaVar2 :: (Regular a, Zip (PF a), LR (PF a))
              ⇒ (a → a → a) → SchemeOf a
introMetaVar2 f = term1 `mergeSchemes` term2
  where
    term1 = insertMetaVar 1 (f left left) (f right left )
    term2 = insertMetaVar 2 (f left left) (f left  right)
```

As before, metavariables are inserted by *insertMetaVar*, but there is a slight complication: the rule specification now has two arguments rather than one. Because *insertMetaVar* inserts only one variable at a time, we need to handle the two metavariables separately. Suppose that we handle the first metavariable, then we need two terms to pass to *insertMetaVar*. We apply f to different arguments for the first metavariable, and we supply the same argument for the second metavariable. In this way, differences in the generated terms arise only for the first metavariable. The second metavariable is handled in a similar way.

At this point we have two terms, each containing either one of the metavariables. We use the function *mergeSchemes* to combine the two schemes into one that contains both metavariables.

```
mergeSchemes :: Zip f
             ⇒ Scheme f → Scheme f → Scheme f
mergeSchemes p@(In x) q@(In y) =
  case (schemeView p, schemeView q) of
    (MetaVar i, _) → p
    (_, MetaVar j) → q
    _              → In (fzip' mergeSchemes x y)
```

The generic function $fzip'$ is used to traverse both terms in parallel until a metavariable is encountered at either side. Then, the term at the other side can be discarded, for we are sure that it is just a *left*-value that was kept constant for the sake of handling one metavariable at a time.

4.4 Rules with an Arbitrary Number of Metavariables

Our technique for introducing metavariables in rules exhibits a clear pattern. It would be easy to define functions $rule_3$, $rule_4$, and so on to handle the cases for other fixed numbers of metavariables.

Alternatively, we can try and capture the general pattern in a class declaration and a suitable set of instance declarations. To this end, let us first examine this particular pattern more closely. In general, a rule involving n metavariables, for some natural number n, can be built by a function f that expects n arguments and produces a rule specification. To do so, we invoke the function n times, each time putting the value *right* into a specific argument position while keeping the value *left* in the remaining positions. Each of the n rule specifications obtained through this diagonal pattern are combined, by means of the function *insertMetaVar*, with a base value obtained by passing *left* values exclusively to f:

$$\underbrace{(f\ left\ left \cdots left)}_{base} \bowtie \underbrace{\begin{cases} (f\ right\ \ left\ \ \cdots\ \ left\ ) \\ (f\ \ left\ \ right\ \cdots\ \ left\ ) \\ \quad\vdots\quad\ \vdots\quad\ \ \vdots\quad\ \ddots\quad\ \vdots \\ (f\ \ left\ \ \ left\ \ \cdots\ right\ ) \end{cases}}_{diag}$$

In this way, we obtain n rules that are, by repeated merging, combined into the single rule originally expressed by f.

First, we define a type class *Builder* that contains the family of functions that is used to build rules:

```
class Regular (Target a) ⇒ Builder a where
  type Target a :: *
  base          :: a → RuleSpec (Target a)
  diag          :: a → [RuleSpec (Target a)]
```

Apart from the methods *base* and *diag* that provide values obtained from invocations of each *Builder*-function with *left* and *right* arguments, the class contains an associated type synonym *Target* that holds the type targeted by the rule under construction. Some typical instances of the resulting type family are:

```
type Target (                        RuleSpec Prop) = Prop
type Target (          Prop → RuleSpec Prop) = Prop
type Target (Prop → Prop → RuleSpec Prop) = Prop
```

It remains to inductively define instances for building rewrite rules with an arbitrary number of metavariables. The base case, for constructing rules that do not involve any metavariables, is straightforward: such rules are easily built from functions that take no arguments and immediately return a *RuleSpec*:

```
instance Regular a ⇒ Builder (RuleSpec a) where
  type Target (RuleSpec a) = a
  base x                   = x
  diag x                   = [x]
```

Given an instance for the case involving n metavariables, the case for $(n+1)$ metavariables is, perhaps surprisingly, not much harder:

```
instance (Builder a, Regular b, LR (PF b))
         ⇒ Builder (b → a) where
  type Target (b → a) = Target a
  base f              = base (f left)
  diag f              = base (f right) : diag (f left)
```

We now define a single function *rule* for constructing rewrite rules from functions over metavariable placeholders:

```
rule :: (Builder r, Zip (PF (Target r)))
     ⇒ r → Rule (Target r)
rule f = foldr1 mergeRules rules
  where
    mergeRules x y =
      mergeSchemes (lhs x) (lhs y) :⇝
      mergeSchemes (rhs x) (rhs y)
    rules          = zipWith (ins (base f)) (diag f) [1..]
    ins x y v      =
      insertMetaVar v (lhs x) (lhs y) :⇝
      insertMetaVar v (rhs x) (rhs y)
```

For instance, rules are now defined as follows:

```
notTrueRule, andContrRule, deMorganRule :: Rule Prop
notTrueRule    = rule (          Not T          :⇝ F)
andContrRule   = rule (λp     → p :∧: Not p    :⇝ F)
deMorganRule   = rule (λp q → Not (p :∧: q) :⇝
                              Not p :∨: Not q  )
```

To directly apply a rule constructed by a *Builder*, we compose *rule* and *rewrite*:

```
applyRule :: (Builder r, Rewrite (Target r))
          ⇒ r → Target r → Target r
applyRule = rewrite ∘ rule
```

In summary, the *Builder* class and its instances provide a uniform approach to constructing rewrite rules that involve an arbitrary number of metavariables. Note that rules with different numbers of metavariables are, once built, uniformly typed and can, for example, be straightforwardly grouped together in a list or in any other data structure. Building rules by repeatedly comparing the results of different invocations of functions comes with some overhead, but for the typical case where the rule definition is in a constant applicative form, this amounts to a one-time investment.

4.5 Polishing the Interface

We now have a library for generic rewriting. However, the types of high-level functions contain too much implementation detail. For instance, recall the type of *rewriteM*:

```
rewriteM :: (Regular a, Crush (PF a), Zip (PF a), Monad m)
         ⇒ Rule a → a → m a
```

Function *rewriteM* exposes implementation details such as calls to generic *crush* and generic *zip* in its type signature. We hide these by defining a type class synonym *Rewrite* as follows:

```
class (Regular a, Crush (PF a), Zip (PF a), LR (PF a))
   ⇒ Rewrite a
```

For instance, the type of propositions is made an instance of *Rewrite* by declaring

```
instance Rewrite Prop
```

Using *Rewrite*, the type signature of *rewriteM* becomes:

```
rewriteM :: (Rewrite a, Monad m) ⇒ Rule a → a → m a
```

Some of the constraints of *Rewrite* are superfluous to some generic functions in the library. For example, the function *rule* only requires an instance of *Zip*. It might appear that the additional constraints on *rule* restrict its use unnecessarily, but this is not a problem in practice because functors built from sums and products already satisfy those constraints.

5. Performance

We have measured execution times of our generic rewriting library, mainly to measure how well the generic definitions perform compared to hand-written code for a specific datatype. We have tested

our library with the proposition datatype from the introduction, extended with constructors for implications and equivalences. We have defined 15 rewrite rules, and we used these rules to bring the proposition to disjunctive normal form (DNF). This rewrite system is a realistic application of our rewriting library, and is very similar to the system that is used in an exercise assistant for e-learning systems (Heeren et al. 2008).

The library has been tested with four different strategies: such a strategy controls which rewrite rule is tried, and where. The strategies range from naive (i.e, apply some rule somewhere), to more involved strategy specifications that stage the rewriting and use all kinds of traversal combinators. QuickCheck (Claessen and Hughes 2000) is used to generate a sequence of random propositions, where the height of the propositions is bounded by five, and the same sequence is used for all test runs. Because the strategy highly influences how many rules are tried, we vary the number of propositions that has to be brought to disjunctive normal form depending on the strategy that is used. The following table shows for each strategy the number of propositions that are normalized, how many rules are successfully applied, and the total number of rules that have been fired:

strategy	terms	rules applied	rules tried	ratio
dnf-1	10,000	217,076	113,728,320	0.19%
dnf-2	50,000	492,114	22,716,336	2.17%
dnf-3	50,000	487,490	22,955,220	2.12%
dnf-4	100,000	872,494	19,200,407	4.54%

The final column shows the percentage of rules that succeeded: the numbers reflect that the simpler strategies fire more rules.

We compare the execution times of four different implementations for the collection of rewrite rules.

- **Pattern Matching (PM).** The first implementation defines the 15 rewrite rules as functions that use pattern matching. Obviously, this implementation suffers from the drawbacks that were mentioned in Section 2.1, making this version less suitable for an actual application. However, this implementation of the rules is worthwhile to study because Haskell has excellent support for pattern matching, which will likely result in efficient code.
- **Specialized Rewriting (SR).** We have also written a specialized rewriting function that operates on propositions. Because the *Prop* datatype does not have a constructor for metavariables, we have reused the *Var* constructor for this purpose, thus mixing object variables with metavariables.
- **Pattern-functor View (PV).** Here we implemented the rules using the generic functions for rewriting that were introduced in this paper. The instance for the *Regular* type class is similar to the declaration in Figure 3, except that a balanced encoding was used for the constructors of the *Prop* datatype to make it more efficient.
- **Fixed-point View (FV).** The last version also uses the generic rewriting approach, but it uses a different structural representation called the fixed-point view (Holdermans et al. 2006). Like the pattern-functor view, this view makes recursion explicit in a datatype, but additionally, every recursive occurrence is mapped to its structural counterpart. In this view, *from* and *to* would have the following types:

$$\begin{aligned} &\mathit{from} :: a \to \mathit{Fix}\ (\mathit{PF}\ a) \\ &\mathit{to} \quad :: \mathit{Fix}\ (\mathit{PF}\ a) \to a \end{aligned}$$

  We include this view in our measurements because such deep structure mapping is common in generic programs that deal with regular datatypes (Jansson and Jeuring 1997). Indeed, the fixed-point view was used in our library before we switched to the current structure representation.

All test runs were executed on an Apple MacBook Pro with a 2.2 GHz Intel Core 2 Duo processor, 2 Gb SDRAM memory, and running MacOS X 10.5.3. The programs were compiled with GHC version 6.8.3 with all optimizations enabled (using the -O2 compiler flag). Execution times were measured using the `time` shell command, and were averaged over three runs. The following table shows the execution time in seconds for each implementation of the strategies:

strategy	PM	SR	PV	FV
dnf-1	6.31	16.75	56.78	70.69
dnf-2	3.49	6.37	23.66	30.24
dnf-3	3.52	6.42	23.82	30.26
dnf-4	5.72	9.14	27.82	37.65

Because the strategies normalize a varying number of terms, it is hard to draw any conclusion from results of different rows. The table shows that the pattern-matching approach (PM) is significantly faster compared to the other approaches. The specialized rewriting approach (SR) adds observability of the rewrite rules, at the cost of approximately doubling the execution time. The two generic versions, when compared to the SR approach, suffers from a slowdown of a factor 3 to 4. This slowdown is probably due to the conversions from and to the structure representation of propositions. The approach with a fixed-point view (FV) is less efficient than using the pattern-functor view (PV) since the latter only converts a term when this is really required. This is reflected in the recursive embedding-projection pair for the FV in contrast to the non-recursive embedding-projection pair of the PV.

Execution time is consumed not only by the rewrite rules, but also by the strategy-controlled traversals over the propositions. These traversal functions, such as *transform* from the introduction, can of course be defined generically. The execution times presented before use specialized traversal functions for the *Prop* datatype, but we have repeated the experiment using the generic traversal definitions. Although traversal combinators are not the focus of this paper, it is interesting to measure how much impact this change has on performance. The following table shows the execution times for the different implementations that now make use of generic traversal combinators:

strategy	PM	SR	PV	FV
dnf-1	10.56	20.50	61.95	91.12
dnf-2	11.18	14.40	32.21	54.86
dnf-3	11.38	14.51	32.21	55.44
dnf-4	9.04	12.53	31.56	46.08

First, the observations made for the table presented earlier still hold. By comparing the two tables point-wise it can be concluded that the generic traversal functions are slower. The relative increase in execution time for the versions that already used generic definitions (that is, PV and FV) is, however, lower compared to the non-generic versions (PM and SR). Furthermore, the additional cost of making the traversal generic depends on the strategy being used, since the strategies use different combinators.

So what can be concluded from these tests? The tests do not offer spectacular new results, but rather confirm that observability of rules comes at the expense of loss in runtime efficiency. Furthermore, generic definitions introduce some additional overhead. The trade-off between efficiency and genericity depends on the application at hand. For instance, the library would be suitable for the online exercise assistant, because runtime performance is less important in such a context. We believe that improving the efficiency of generic library code is an interesting area for future research. By inlining and specializing generic definitions, and by applying par-

tial evaluation techniques, we expect to get code that is more competitive to the hand-written definitions for a specific datatype. Fusion techniques (Alimarine and Smetsers 2005; Coutts et al. 2007) can help to eliminate some of the conversions between a value and its structure counterpart.

6. Related Work

The generic rewriting library introduced in this paper can be viewed as a successor to the library given by Jansson and Jeuring (2000); we discussed the relation with this library in the introduction.

Libraries that provide generic traversal combinators, such as Strafunski (Lämmel and Visser 2002), Scrap Your Boilerplate (Lämmel and Peyton Jones 2003), Uniplate (Mitchell and Runciman 2007), a pattern for almost compositional functions (Bringert and Ranta 2006), and probably more, can be used to define rewrite rules in the form of functions. This has the disadvantages we describe in Section 2, the most important of which is that it is impossible to document, test, and analyse rewrite rules. The advantage of using functions instead of rules is that datatypes do not have to be extended with metavariables: we reuse program-level variables for metavariables in our rules.

The *match* function is a simple variant of generic unification functions (Jansson and Jeuring 1998; Sheard 2001). Our library cannot be easily generalised to perform unification and stay user-friendly, because the result of unification may contain values of datatypes extended with metavariables, and hence a user would have to deal with structure values. On the other hand, we could use two-level types as in the work from Sheard but require less work from the user, because most functionality can be obtained generically from the structure representation of the two-level types. Furthermore, we could easily adapt our library to use mutable variables, as in Sheard's work, to improve performance.

Brown and Sampson (2008) implement generic rewriting using the Scrap Your Boilerplate library. Rule schemes are described in a special purpose datatype that does not depend on the type of values being rewritten. In contrast to our system, rules are not typed and hence invalid rules are only detected at runtime. This system can handle rewriting of a system of datatypes while our library is limited to a single regular datatype. However, we know how to lift this restriction in a type-safe way, see Section 7.

There exist a number of programming languages built on top of the rewriting paradigm, such as ELAN (Borovanský et al. 2001), OBJ (Goguen and Grant 1997), and ASF+SDF (Van Deursen et al. 1996). Instead of built-in support for rewriting, we focus on how to support rewriting in a mainstream higher-order functional programming language by providing a library.

7. Conclusions and Further Work

We have shown the interface and implementation of a generic library for rewriting. Our library overcomes problems in previous generic rewriting libraries: users do not have to adapt the datatype on which they want to apply rewriting, they do not need knowledge of how the generic rewriting library has been implemented, and they can document, test, and analyse their rules. The performance of our library is not as good as that of hand-written datatype-specific rewriting functions, but we think the loss of performance is acceptable for many applications.

One of the most important limitations of the library described in this paper is that it only works for datatypes that can be represented by means of a fixed-point. Such datatypes are also known as regular datatypes. This is a severe limitation, which implies that we cannot apply the rewriting library to nested datatypes or systems of (mutually recursive) datatypes. Indeed, many real-world applications involve such systems: examples include systems of linear equations and the abstract syntax of expressions that may contain declarations that, in turn, may consist of expressions again. However, with some additional machinery we can overcome this limitation and so we have an implementation of the rewriting library that enables generic rewriting on systems of datatypes. The techniques used are rather sophisticated and apply to several other problems as well, such as the Zipper. A thorough explanation of the underlying ideas and implementation can be found in a dedicated paper (Rodriguez et al. 2008).

Our library requires that rules do not inspect their metavariable argument. For instance, we do not allow arbitrary function applications in the right-hand side of a rule, contrary to rules that are defined as functions with pattern matching. Another limitation of our library is that rules cannot have preconditions, for example, that a proposition matched against a metavariable is in disjunctive normal form. Extending rewrite rules with preconditions remains future work.

Currently, we are also working on generating test data generically. The left-hand side of a rewrite rule can be used as a template for test data generation to improve the testing coverage. We plan to use it to provide a testing framework for users of our library. This functionality can be easily added to our library: all it takes to define a new generic function is declaring a type class and providing a set of inductively defined instances.

Acknowledgments

This work was made possible by the support of the SURF Foundation, the higher education and research partnership organisation for Information and Communications Technology (ICT). Please visit `http://www.surf.nl` for more information about SURF.

This work has been partially funded by the Technology Foundation STW through its project on "Demand Driven Workflow Systems" (07729), and by the Netherlands Organisation for Scientific Research (NWO), through its projects on "Real-life Datatype-Generic Programming" (612.063.613) and "Scriptable Compilers" (612.063.406).

The authors would like to thank Chris Eidhof and Sebastiaan Visser for their work on testing rewrite rules by means of generic generation of test data, and Andres Löh for productive discussions on this work. Finally, the authors are indebted to the anonymous reviewers for their useful suggestions.

References

Artem Alimarine and Sjaak Smetsers. Improved fusion for optimizing generics. In Manuel V. Hermenegildo and Daniel Cabeza, editors, *Practical Aspects of Declarative Languages, 7th International Symposium, PADL 2005, Long Beach, CA, USA, January 10–11, 2005, Proceedings*, volume 3350 of *Lecture Notes in Computer Science*, pages 203–218. Springer-Verlag, 2005.

Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Christophe Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12(1): 69–95, 2001.

Björn Bringert and Aarne Ranta. A pattern for almost compositional functions. In John H. Reppy and Julia L Lawall, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16–21, 2006*, pages 216–226. ACM Press, 2006.

Neil C. C. Brown and Adam T. Sampson. Matching and modifying with generics. In Peter Achten, Pieter Koopman, and Marco T. Morazán, editors, *Draft Proceedings of the Ninth Symposium on Trends in Functional Programming (TFP), May 26–28 2008, Center Parcs "Het Heijderbos", The Netherlands*, pages 304–318, 2008. The draft proceedings of the symposium have been published as a technical report (ICIS-R08007) at Radboud University Nijmegen.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26–28, 2005*, pages 241–253. ACM Press, 2005.

Manuel M. T. Chakravarty, Gabriele Keller, Roman Leshchinskiy, and Simon Peyton Jones. Generic programming with type families, 2008. Work in progress.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18–21, 2000*, pages 268–279. ACM Press, 2000.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1–3, 2007*, pages 315–326. ACM Press, 2007.

Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore, 1996.

Joseph Goguen and Malcolm Grant. *Algebraic Semantics of Imperative Programs*. The MIT Press, Cambridge, Massachusetts, 1997.

Bastiaan Heeren, Johan Jeuring, Arthur van Leeuwen, and Alex Gerdes. Specifying strategies for exercises. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, *Intelligent Computer Mathematics, MKM 2008, Birmingham, UK, July 28–August 1, 2008, Proceedings*, volume 5144 of *LNAI*, pages 430–445. Springer-Verlag, 2008.

Ralf Hinze. Generic programs and proofs, 2000. Habilitationsschrift.

Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming*, 51(2):117–151, 2004.

Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In Tarmo Uustalu, editor, *Mathemathics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3–5, 2006, Proceedings*, volume 4014 of *Lecture Notes in Computer Science*, pages 209–234. Springer-Verlag, 2006.

Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

Patrik Jansson and Johan Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In Johan Jeuring, editor, *Proceedings Workshop on Generic Programming (WGP2000), July 6, 2000, Ponte de Lima, Portugal*, pages 33–45, 2000. The proceedings of the workshop have been published as a technical report (UU-CS-2000-19) at Utrecht University.

Patrik Jansson and Johan Jeuring. PolyP – a polytypic programming language extension. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15–17 January 1997*, pages 470–482. ACM Press, 1997.

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

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003), New Orleans, LA, USA, January 18, 2003*, pages 26–37. ACM Press, 2003.

Ralf Lämmel and Joost Visser. Typed combinators for generic traversal. In Shiriram Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19–20, 2002, Proceedings*, volume 2257 of *Lecture Notes in Computer Science*, pages 137–154. Springer-Verlag, 2002.

Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In Gabriele Keller, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, pages 49–60. ACM Press, 2007.

Thomas van Noort. Generic views for generic types. Master's thesis, Utrecht University, 2008.

Ulf Norell and Patrik Jansson. Polytypic programming in Haskell. In Philip W. Trinder, Greg Michaelson, and Ricardo Pena, editors, *Implementation of Programming Languages, 15th International Workshop, IFL 2003, Edinburgh, UK, September 8–11, 2003, Revised Papers*, volume 3145 of *Lecture Notes in Computer Science*, pages 168–184. Springer-Verlag, 2004.

Bruno C. d. S. Oliveira and Jeremy Gibbons. TypeCase: A design pattern for type-indexed functions. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell, Tallinn, Estonia, September 30, 2005*, pages 98–109. ACM Press, 2005.

Alexey Rodriguez, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. Technical Report UU-CS-2008-019, Utrecht University, 2008.

Tim Sheard. Generic unification via two-level types and parameterized modules. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Florence, Italy, September 3–5, 2001*, pages 86–97. ACM Press, 2001.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Pittsburgh, Pennsylvania, 2002*, pages 1–16. ACM Press, 2002.