

A Lightweight Approach to Datatype-generic Rewriting

Thomas van Noort

Alexey Rodriguez Yakushev

Stefan Holdermans

Johan Jeuring

Bastiaan Heeren

José Pedro Magalhães

Technical Report UU-CS-2010-008
March 2010

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 Noort¹, Alexey Rodriguez Yakushev²,
Stefan Holdermans², Johan Jeuring^{3,4},
Bastiaan Heeren⁴, and José Pedro Magalhães³

¹ Institute for Computing and Information Sciences, Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands, thomas@cs.ru.nl

² Vector Fabrics, Paradijslaan 28, 5611 KN Eindhoven, The Netherlands
{alexey,stefan}@vectorfabrics.com

³ Department of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands, {johanj,jpm}@cs.uu.nl

⁴ School of Computer Science, Open University of the Netherlands
P.O. Box 2960, 6401 DL Heerlen, The Netherlands, bastiaan.heeren@ou.nl

Term-rewriting systems can be expressed as generic programs parameterised over the shape of the terms being rewritten. Previous implementations of generic rewriting libraries require users to either adapt the datatypes that are used to describe these terms or to specify rewrite rules as functions. These are fundamental limitations: the former implies a lot of work for the user, while the latter makes it hard if not impossible to document, test, and analyse rewrite rules. In this report, we demonstrate how to overcome these limitations by making essential use of type-indexed datatypes. Our approach is lightweight in that it is entirely expressible in Haskell with GADTs and type families and can be readily packaged for use with contemporary Haskell distributions.

1 Introduction

Consider a Haskell datatype *Prop* for representing formulae of propositional logic,

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

and suppose we wish to simplify such formulae using the *principle of contradiction*:

$$p \wedge \neg p \rightarrow \perp.$$

Ideally, our formulation of this rewrite rule as an executable program is neither much longer nor much more complicated than this rule itself.

One approach is to encode the rule as a function and then to apply it to individual formulae using some bottom-up traversal combinator *transform*:

```

simplify :: Prop → Prop
simplify = transform contradiction
where
  contradiction (p :∧: Not q) | p ≡ q = F
  contradiction p                    = p.

```

Although this implementation is relatively straightforward, encoding rules by functions has a number of drawbacks. To start with, rules cannot be concise one-line definitions as we have to provide a catch-all case in order to avoid pattern-matching failures at run-time. Secondly, pattern guards (such as $p \equiv q$ in our example) are needed to deal with multiple occurrences of variables, cluttering the definition. Lastly, rules cannot be analysed easily since it is hard to inspect functions.

A way to overcome these drawbacks is to provide specialised rewriting functionality. That is, we can define a datatype representing rewrite rules on formulae and implement the machinery required for rewriting (e.g., functions for matching formulae against rules and substituting formulae for metavariables) on top of this datatype. While this does overcome the drawbacks mentioned above, this approach comes with a serious disadvantage: it requires a large amount of datatype-specific code. If our next task is to rewrite, say, arithmetic expressions, we have to define a new datatype for representing rewrite rules and a new implementation of all the rewriting machinery.

However, both the datatype for representing rules and the associated rewriting machinery can be determined from the type that is used to describe the terms being rewritten. Hence, there is an excellent opportunity for datatype-generic programming here. In this report, we seize this opportunity and present a rewriting library that is generic in the type of terms being rewritten. Using our library, the example above can be written as

```

simplify :: Prop → Prop
simplify = transform (rewriteWith contradiction)
where
  contradiction p = p :∧: Not p ↦ F.

```

The library provides *rewriteWith* and \mapsto , which are generic and, in this case, instantiated with the type of propositional formulae *Prop*. A noticeable aspect of our approach is that metavariables in rewrite rules, such as p in our example, are introduced through ordinary function abstraction in Haskell, allowing the user to define her rules in terms of the term type *Prop* rather than some dedicated type for representing rules over *Prop*. The body of the function *contradiction* is now a fairly direct transcript of the rule $p \wedge \neg p \rightarrow \perp$. As we will see, rewrite rules constructed with our library neither suffer from the drawbacks of the approach that uses pattern matching nor require large amounts of datatype-specific boilerplate code.

More specifically, the contributions of this report are the following:

- We present a library for term rewriting that is implemented using a simple design pattern (Section 4) for datatype-generic programming in Haskell extended with type families (Chakravarty et al., 2005b,a; Schrijvers et al., 2008). As such, our library is “lightweight” and can be used readily with recent versions of the Glasgow Haskell Compiler (GHC) ¹.
- To represent rewrite rules our library needs to extend the type that is used to describe the terms being rewritten internally with an extra constructor for metavariables (Section 5.2). This extension is constructed generically using a type-indexed datatype (Hinze et al., 2004). Distinct metavariables in a single rewrite rule can, in our approach, range over rewritable terms of different type (Section 5.1).
- Internally, the library implements rewriting in terms of generic functions for pattern matching (Section 5.4) and substitution (Section 5.3) over generically extended datatypes. These datatypes are, however, completely hidden from the user, who writes her rewrite rules using the constructors of the types of terms that are to be rewritten (Section 6).
- We compare the efficiency of our library to that of other approaches to term rewriting in Haskell (Section 10).

This report is based on a paper presented at the 2008 Workshop on Generic Programming (Van Noort et al., 2008). The present report includes several improvements over this previous work. Most prominently, while the library described in the WGP paper could only be used to generically rewrite values of regular datatypes, we now support generic rewriting for a strictly larger class of datatypes, including types from families of mutually recursive datatypes. Furthermore, we now detect ill-formed rewrite rules (Section 7) and facilitate guarded rewrite rules (Section 8) as well as heterogeneously typed metavariables (Section 5.1).

1.1 Road map

The remainder of this report is structured as follows. In Section 2, we discuss the two fundamental approaches to representing rewrite rules in Haskell. In Section 3 we present our proposal for a datatype-generic library for term rewriting from a user’s perspective.

Sections 4 to 6 deal with the implementation of our library’s main functionality. Section 4 showcases, through an example generic function, how datatype-generic functions are implemented in our library. Section 5 discusses how generic rewriting functionality is composed from more elementary generic functions for pattern matching and substitution, and shows how these functions are implemented. In Section 6, we demonstrate how the not so programmer-friendly representation of rewrite rules, used internally by the generic functions from Section 5, is hidden from the users of our library.

¹The library is dubbed `guarded-rewriting` and available on Hackage.

Sections 7 and 8 discuss additions to the core functionality. In Section 7, it is shown how nonsensical rewrite rules can be detected statically, i.e., without applying them. In Section 8, the library is extended with support for rewrite rules that have preconditions associated with them.

Section 9 discusses, as a case study, the use of our library in a realistic application. Section 10 presents the results of two performance benchmarks. Section 11 discusses related work; Section 12 concludes.

2 Representing rewrite rules

Before we present our approach to datatype-generic rewriting in Section 3, let us first have a more in-depth look at the two fundamental approaches to representing rewrite rules in Haskell that were already briefly discussed in the Introduction: the extensional approach (Section 2.1) and the intensional approach (Section 2.2).

2.1 Extensional representations

The extensional approach to representing rewrite rules encodes rules as Haskell functions, using pattern matching to check whether the argument term matches the left-hand side of the rule. If this is indeed the case, the right-hand side of the rule is returned; otherwise, the argument term is returned unchanged. For example, the rule

$$\neg(p \wedge q) \rightarrow \neg p \vee \neg q,$$

that is derived from one of De Morgan’s laws, is extensionally encoded as

$$\begin{aligned} deMorgan &:: Prop \rightarrow Prop \\ deMorgan (Not (p \wedge q)) &= Not p \vee Not q \\ deMorgan p &= p. \end{aligned}$$

Note the catch-all case that makes that arguments not matching the pattern $\neg(p \wedge q)$ do not result in run-time errors.

As Haskell lacks support for nonlinear patterns, rewrite rules containing metavariables with multiple left-hand-side occurrences cannot be written as functions directly. Instead, such variables are encoded by means of so-called *pattern guards*. For instance, a rule for the *principle of the excluded middle*,

$$p \vee \neg p \rightarrow \top,$$

in which the metavariable p occurs twice at the left-hand side, is implemented by

$$\begin{aligned} excludedMiddle &:: Prop \rightarrow Prop \\ excludedMiddle (p \vee Not q) \mid p \equiv q &= T \\ excludedMiddle p &= p, \end{aligned}$$

where the second occurrence of p is replaced by an occurrence of a fresh variable q and equality of p and q is enforced through the guard $p \equiv q$. Note that this encoding requires equality to be defined for values of type *Prop*.

In some applications of rewriting, it is useful to know whether or not a rewrite rule was applied successfully. This information can be made available, at the expense of some additional notational overhead, by wrapping the rewriting result in a *Maybe*-value:

$$\begin{aligned} \text{excludedMiddleM} &:: \text{Prop} \rightarrow \text{Maybe Prop} \\ \text{excludedMiddleM } (p \vee \text{Not } q) \mid p \equiv q &= \text{Just } T \\ \text{excludedMiddleM } p &= \text{Nothing}. \end{aligned}$$

Encoding rewrite rules in terms of Haskell functions allows for function-parameterised traversal combinators to be used directly in rewriting applications. As an example, the Uniplate library (Mitchell and Runciman, 2007) provides, amongst others, the combinator *transform*,

$$\text{transform} :: \text{Uniplate } \alpha \Rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha,$$

which applies its argument function in a bottom-up fashion in all recursive positions in a tree. Given a suitable *Uniplate*-instance for the type *Prop*, it is straightforward to use this combinator to remove certain classes of tautological clauses from propositional formulae:

$$\begin{aligned} \text{removeTautologies} &:: \text{Prop} \rightarrow \text{Prop} \\ \text{removeTautologies} &= \text{transform excludedMiddle}. \end{aligned}$$

However, even though Haskell’s pattern-matching facilities enable a more or less direct encoding of rewrite rules as functions and the interaction with traversal libraries comes almost for free, the extensional approach to representing rewrite rules raises some issues.

- Extensionally represented rules cannot be easily observed as in Haskell it is not possible to inspect functions. Still, there are several reasons why it is desirable to have observable rewrite rules:

Documentation: If rules are observable, they can be pretty-printed in order to generate documentation for a rewrite system.

Static checking: Observability of rules allows for checking whether a given set of rewrite rules constitutes a confluent and terminating rewrite system.

Automated testing: In most applications, a rule is expected to preserve the semantics of the term being rewritten. One way to test this property is to randomly generate terms, to rewrite these, and then to check whether the rewritten terms indeed have the same semantics as the original terms. However, a rewrite rule with a nontrivial left-hand side will most likely not match successfully against a randomly generated term. Hence, such rules are in danger of not getting tested sufficiently. If left-hand sides of rules are inspectable, term generation can be directed to produce matching terms more often, effectively improving test coverage.

Associativity- and commutativity-aware rewriting: Many domains, such as that of logical propositions, have associative and commutative operators. If the rewriting infrastructure is aware of this fact, rewrite rules can be specified more concisely and repetition can be avoided. With an intensional approach, this can be implemented by making the matching algorithm return all possible substitutions. In an extensional approach, the behavior of pattern matching is fixed and cannot be made aware of these operators.

Inversion: If the left-hand side and right-hand side of a rewrite rule can be accessed, these can be exchanged, resulting in the inverse of the rule.

Tracing: When a sequence of rewrite steps leads to an unexpected result, one may want to learn which rules were applied in which order.

- It is tedious to have to specify a catch-all case when rules are encoded as functions. All rule definitions require this extra case.
- The lack of nonlinear pattern matching in Haskell becomes a nuisance if left-hand sides of rules contain many occurrences of the same variables.
- As Haskell lacks first-class pattern matching, the user cannot easily abstract over commonly occurring structures in the left-hand sides of rewrite rules.

These issues can be overcome by switching to an intensional representation instead.

2.2 Intensional representations

In the intensional approach, rewrite rules are not encoded as functions, but as values of a datatype, so that the left- and right-hand sides of rules become observable:

```
data Rule  $\alpha$  = Rule{ lhs ::  $\alpha$ , rhs ::  $\alpha$  }.
```

Values of type *Rule α* are used to encode rewrite rules with left- and right-hand sides of type α . For example, rewrite rules for formulae of propositional logic can be expressed as values of type *Rule EProp*, where *EProp* is an extended version of the datatype *Prop* of propositional formulae with an extra constructor *Metavar* to represent metavariable occurrences in rewrite rules:

```
data EProp
  = EVar String | ET | EF | ENot EProp | EProp : $\odot$ : EProp | EProp : $\odot$ : EProp
  | Metavar String.
```

With values of type *Rule EProp* in place, we need to define rewrite functions that interpret these values as functions over propositions represented by *Prop*:

```
rewritePropWith :: Rule EProp  $\rightarrow$  Prop  $\rightarrow$  Prop.
```

Here we do not give an implementation of *rewritePropWith*, but note that its type (and thus its implementation) is specific to propositional formulae. If we want to implement

rewrite functionality that works on different datatypes, then we have to define new rewrite functions for these types.

With the proposition-specific rewrite function *rewritePropWith*, rules over propositional formulae can be written and used as in

```
removeTautologies :: Prop → Prop
removeTautologies = transform (rewritePropWith excludedMiddle)
  where
    excludedMiddle = Rule{ lhs = Metavar "p" :⊗: ENot (Metavar "p"), rhs = ET}.
```

An apparent inconvenience of this style of defining rules is that, rather than reusing the type *Prop* of terms being rewritten and the constructors *Not* and *:∨:*, to make provision for metavariables, one has to use the extended representation *EProp* and its constructors *ENot* and *:⊗:*.

3 Datatype-generic rewriting

In this section, we present the interface to our library for datatype-generic rewriting. In Sections 4 to 6, we zoom in at the concrete implementation of this interface.

The rewrite system that we present in this paper uses intensionally represented rewrite rules. As observed in the previous section, straightforward implementations of such rewrite systems suffer from two drawbacks: (1) they require a significant amount of datatype-specific code and (2) rewrite rules need to be expressed in terms of a new datatype obtained by extending the original datatype with a constructor for metavariables. Our system, however, is carefully designed to circumvent these drawbacks: (1) we provide a single implementation of rewriting that is generic in the type of terms being rewritten and (2) we completely hide the internal representation of rewrite rules from the user of our library.

More specifically, in our approach rewrite rules are specified in terms of *templates*:

```
closedWorldTemplate :: Template Prop
contradictionTemplate :: Prop → Template Prop
deMorganTemplate    :: Prop → Prop → Template Prop
closedWorldTemplate = Not T ↦ F
contradictionTemplate p = p :∧: Not p ↦ F
deMorganTemplate p q = Not (p :∧: q) ↦ Not p :∨: Not q.
```

Templates are constructed by means of an operator \mapsto ,

$$(\mapsto) :: \alpha \rightarrow \alpha \rightarrow \text{Template } \alpha,$$

which takes a left-hand side and a right-hand side of a type α and produces a template for rewrite rules on α . Note that both sides of a template are just values of the type of terms being rewritten. In particular, templates are expressed without need for an additional datatype providing for metavariables. Instead, metavariables are encoded as ordinary Haskell function arguments. The template for the De Morgan rule from

the example above, for instance, uses two metavariables which are introduced through function arguments p and q .

To prepare templates for use in our rewrite system, the user needs to synthesise *rules* from these. To this end, the library provides an overloaded function *synthesise* (defined in Section 6.3), that takes templates (or functions producing templates) for rewrite rules on some type α to values of type *Rule* α :

```
closedWorld, contradiction, deMorgan :: Rule Prop
closedWorld = synthesise closedWorldTemplate
contradiction = synthesise contradictionTemplate
deMorgan    = synthesise deMorganTemplate.
```

Here, values of type *Rule* α (with an implementation that differs slightly from the one given above; see Section 5) form the internal representation of rewrite rules on α in our library.

The generic rewrite functionality is now exposed through a pair of rewrite functions *rewriteWith* and *rewriteWithM*. The first,

$$\text{rewriteWith} :: \text{Rewritable } \alpha \Rightarrow \text{Rule } \alpha \rightarrow \alpha \rightarrow \alpha,$$

takes as arguments a rule over some rewritable type α (see Section subsec:making-terms-rewritable) and a value of type α , and attempts to apply the rule to the value. For example, the expression

$$\text{rewriteWith closedWorld (Not T)}$$

yields the formula F . If the second argument to *rewriteWith* does not match the left-hand side of its first argument, the value to be rewritten is returned unmodified; for instance,

$$\text{rewriteWith contradiction (Var "x" :& Not (Var "y"))}$$

yields $\text{Var "x" :& Not (Var "y")}$ as the argument term does not match a contradictory formula. To make a failed attempt at rewriting explicit in the value returned, the second generic rewrite function,

$$\text{rewriteWithM} :: (\text{Rewritable } \alpha, \text{Monad } \mu) \Rightarrow \text{Rule } \alpha \rightarrow \alpha \rightarrow \mu \alpha,$$

wraps its result in a monad μ . For example, instantiating μ with the *Maybe*-monad, the invocation

$$\text{rewriteWithM deMorgan (T :& F)}$$

results in the value *Nothing*, while

$$\text{rewriteWithM deMorgan (Not (T :& F))}$$

produces *Just (Not T :v: Not F)*.

As with other lightweight approaches to generic rewriting, such as Scrap Your Boilerplate (Lämmel and Peyton Jones, 2003) and Uniplate (Mitchell and Runciman, 2007), a small effort is required from the users of our library in order to prepare their datatypes for generic rewriting. In particular, they must describe the *structure* of their datatypes (Section 3.1) and make these datatypes instances of the type class *Rewritable* (Section 3.2).

3.1 Representing the structure of datatypes

In our library, the structure of datatypes is described through instances of a type class *Representable*:

```
class Representable  $\alpha$  where
  type Rep  $\alpha$  ::  $\star$ 
  from ::  $\alpha \rightarrow$  Rep  $\alpha$ 
  to   :: Rep  $\alpha \rightarrow$   $\alpha$ .
```

Here, *Rep* is a so-called *associated type synonym* (Chakravarty et al., 2005a). A type α is representable if it is isomorphic to its *generic representation type* *Rep* α ; the isomorphism is witnessed by a pair of functions *from* and *to* that convert between the type and its generic representation.

Base types, such as *Int*, *Float*, and *Char* form their own generic representations:

```
instance Representable Int   where type Rep Int   = Int   ; from = id ; to = id
instance Representable Float where type Rep Float = Float; from = id ; to = id
instance Representable Char where type Rep Char = Char ; from = id ; to = id.
```

Further generic representation types are composed from a fixed set of structure constructors. These include the nullary type constructor *Nil* and the binary type constructors *:+:* and *:::*, defined as:

```
infixr 6 :+:
infixr 5 :::
data Nil      = Nil
data  $\alpha$  :+:  $\beta$  = Inl  $\alpha$  | Inr  $\beta$ 
data  $\alpha$  :::  $\beta$  =  $\alpha$  :::  $\beta$ .
```

A given datatype’s representation type follows immediately from its structure. Choice amongst data constructors is encoded in terms of right-nested sums constructed by *:+:*. A data constructor itself is represented as a type-level list of its argument types, constructed by *:::* and *Nil*. (Note that, instead of the more common *sums-of-products* representation of datatypes (Jansson and Jeuring, 1997; Backhouse et al., 1999; Hinze, 2000), we use a list-like representation (Holdermans et al., 2006) as we want to make sure that constructor arguments are always encoded as the first operand of the constructor *:::*.) For example, Haskell’s *Maybe*-type, given by

```
data Maybe  $\alpha$  = Nothing | Just  $\alpha$ ,
```

is represented by the type $Nil \text{ :+: } (\alpha \text{ ::: } Nil)$ and we can write

```
instance Representable (Maybe  $\alpha$ ) where
  type Rep (Maybe  $\alpha$ ) =  $Nil \text{ :+: } (\alpha \text{ ::: } Nil)$ 
  from Nothing = Inl Nil
  from (Just  $x$ ) = Inr ( $x \text{ ::: } Nil$ )
  to (Inl Nil) = Nothing
  to (Inr ( $x \text{ ::: } Nil$ )) = Just  $x$ .
```

The type-class methods *from* and *to* form a so-called *embedding-projection pair* and are supposed to witness the isomorphism between a type and its generic representation “modulo undefinedness”, i.e., it should hold that $to \circ from = id$ and $from \circ to \sqsubseteq id$ (Hinze, 2000).

The functional programmer’s all-time favourite datatype, i.e., the type of cons-lists,

```
data [ $\alpha$ ] = [] |  $\alpha$  : [ $\alpha$ ],
```

is in our approach represented by $Nil \text{ :+: } (\alpha \text{ ::: } [\alpha] \text{ ::: } Nil)$, yielding the declaration

```
instance Representable [ $\alpha$ ] where
  type Rep [ $\alpha$ ] =  $Nil \text{ :+: } (\alpha \text{ ::: } [\alpha] \text{ ::: } Nil)$ 
  from [] = Inl Nil
  from ( $x : xs$ ) = Inr ( $x \text{ ::: } xs \text{ ::: } Nil$ )
  to (Inl Nil) = []
  to (Inr ( $x \text{ ::: } xs \text{ ::: } Nil$ )) =  $x : xs$ .
```

Note that the generic representation types of recursive datatypes are themselves non-recursive: *from* only converts the top-level constructor of a value into its generic representation and leaves all subtrees untouched.

For the type *Prop* of propositional formulae,

```
data Prop = Var String | T | F | Not Prop | Prop : $\wedge$ : Prop | Prop : $\vee$ : Prop,
```

we have

```
type Var = String ::: Nil
type T = Nil
type F = Nil
type Not = Prop ::: Nil
type And = Prop ::: Prop ::: Nil
type Or = Prop ::: Prop ::: Nil
```

as abbreviations for the generic representations of the alternatives and then

```

instance Representable Prop where
  type Rep Prop = Var :+: T :+: F :+: Not :+: And :+: Or
  from (Var x) = Inl (x :: Nil)
  from T       = Inr (Inl Nil)
  from F       = Inr (Inr (Inl Nil))
  from (Not p) = Inr (Inr (Inr (Inl (p :: Nil))))
  from (p :∧: q) = Inr (Inr (Inr (Inr (Inl (p :: q :: Nil))))))
  from (p :∨: q) = Inr (Inr (Inr (Inr (Inr (p :: q :: Nil))))))
  to (Inl (x :: Nil)) = Var x
  to (Inr (Inl Nil)) = T
  to (Inr (Inr (Inl Nil))) = F
  to (Inr (Inr (Inr (Inl (p :: Nil)))))) = Not p
  to (Inr (Inr (Inr (Inr (Inl (p :: q :: Nil)))))) = p :∧: q
  to (Inr (Inr (Inr (Inr (Inr (p :: q :: Nil)))))) = p :∨: q.

```

Instance declarations of *Representable* can be quite verbose, as in the case for *Prop*. However, these declarations are completely determined by the structure of the represented datatypes and can easily be derived automatically, for example by means of a Template Haskell program (Sheard and Peyton Jones, 2002). Moreover, all that needs to be done to use our library on a user-defined datatype, such as *Prop*, is declaring it an instance of *Representable*, *Typeable*, and *Rewritable*—and, as we will see next, instances of the latter two can be given almost effortlessly.

3.2 Making terms rewritable

The class *Rewritable* of types with rewritable values is given by

```

class (Representable  $\alpha$ , Typeable  $\alpha$ ,
       Eq (Rep  $\alpha$ ), Extensible (Rep  $\alpha$ ), Matchable (Rep  $\alpha$ ), Substitutable (Rep  $\alpha$ ),
       Sampleable (Rep  $\alpha$ ), Diffable (Rep  $\alpha$ ))  $\Rightarrow$ 
  Rewritable  $\alpha$ .

```

As this class does not have any methods or associated types, it is only introduced for its superclass constraints. These constraints encode the conditions that need to be fulfilled by a term type in order for its values to be rewritable.

Not only do we need an instance of *Representable*, we also require term types to be in the class *Typeable* that was originally introduced for use with the Scrap Your Boilerplate-library (Lämmel and Peyton Jones, 2003). Currently, *Typeable* is Haskell’s *de facto* standard API for reifying types at the value level and as such it is included in the base libraries that ship with the Glasgow Haskell Compiler. Recent versions of the GHC even provide support for automatically deriving instances of *Typeable* for user-defined datatypes.

The remaining superclass constraints on *Rewritable* place restrictions on the generic representations of term types and make specific parts of the generic rewriting machinery available for all instances of *Rewritable*. More specifically, each of these constraints

accounts for one generic function. As representation types are built from a limited set of type constructors, these constraints imply no additional burden on the user of our generic rewriting library. That is, all needed instances for the base types *Int*, *Float*, and *Char* and the representation constructors *Nil*, *:+:*, and *:::* are already provided by the library. The details behind these instances are discussed in the next sections: in Section 4 we give instances of the standard class *Eq* for our generic representation types; in Section 5 we give the definitions and instances of the custom classes *Extensible*, *Matchable*, and *Substitutable*, while Section 6 covers *Sampleable* and *Diffable*.

For now, we observe that, with the appropriate instances of *Representable* and *Typeable* in place, putting a term type in the class *Rewritable* reduces to a mere one-liner:

```
instance Rewritable Int
instance Rewritable Float
instance Rewritable Char
instance Rewritable α ⇒ Rewritable (Maybe α)
instance Rewritable α ⇒ Rewritable [α]
instance Rewritable Prop.
```

4 Generic equality

The previous section introduced the interface to our library for datatype-generic rewriting. Let us now turn to the concrete implementation of this interface.

In this section, we present an implementation of a type-indexed equality function. In the next section, this generic function is used in our implementation of generic pattern matching, but here it also serves as a neat example of the design pattern for lightweight type-indexed functions that we employ for all generic functions in our library. The general pattern for implementing generic functions, followed throughout this paper, is that we overload a given function f for all generic representation types and then derive a generic version f' that “ties the knot” and works for all types in *Rewritable*.

In our implementation, we rely on the class *Eq* from Haskell’s Standard Prelude to provide an interface for overloaded equality:

```
class Eq α where
  (≡), (≠) :: α → α → Bool
  x ≡ y = ¬ (x ≠ y)
  x ≠ y = ¬ (x ≡ y).
```

As the class *Rewritable* requires the generic representation types of all its instances to be in the class *Eq*, we can directly define an equality operator \equiv' that works for all types of rewritable terms:

```
(≡') :: Rewritable α ⇒ α → α → Bool
x ≡' y = from x ≡ from y.
```

To test two equally typed rewritable terms for equality, we convert them to their generic representations and then test these for equality.

It remains to declare instances of *Eq* for the types that appear in generic representations. The case for *Nil* is straightforward:

```
instance Eq Nil where
  Nil  $\equiv$  Nil = True.
```

For sums, we require the summands to be instances of *Eq* and test whether both generic representations have their origins in the same alternative. If so, both values are compared recursively; otherwise, we produce *False*:

```
instance (Eq  $\alpha$ , Eq  $\beta$ )  $\Rightarrow$  Eq ( $\alpha$  :+:  $\beta$ ) where
  Inl  $x$   $\equiv$  Inl  $y$  =  $x$   $\equiv$   $y$ 
  Inr  $u$   $\equiv$  Inr  $v$  =  $u$   $\equiv$   $v$ 
  _  $\equiv$  _ = False.
```

The case for *:::* is more delicate. In our encoding of a datatype's structure, the second type argument of *:::* is always another type-level list and so we can assume that this type argument is itself in *Eq* as well. The first type argument, however, can be any type and, hence, we cannot just assume it to be an instance of *Eq*. Instead, we require this type argument to be in *Rewritable*, so that we can use the operator \equiv' defined above to compare values of this type:

```
instance (Rewritable  $\alpha$ , Eq  $\beta$ )  $\Rightarrow$  Eq ( $\alpha$  :::  $\beta$ ) where
  ( $x$  :::  $xs$ )  $\equiv$  ( $y$  :::  $ys$ ) =  $x$   $\equiv'$   $y$   $\wedge$   $xs$   $\equiv$   $ys$ .
```

5 Matching and substituting

In the previous section, we demonstrated how generic functions are implemented in our library. We continue our exploration of the internals of the library by discussing the core functionality of our library: the implementation of the function *rewriteWith* and its monadic companion *rewriteWithM*.

These are implemented in terms of two generic functions *match'* and *substitute'*,

```
match'    :: (Rewritable  $\alpha$ , Mappable  $\Gamma$ , Monad  $\mu$ )  $\Rightarrow$ 
             Pattern  $\alpha$   $\Gamma$   $\rightarrow$   $\alpha$   $\rightarrow$   $\mu$  (Substitution  $\Gamma$ )
substitute' :: (Rewritable  $\alpha$ , Monad  $\mu$ )  $\Rightarrow$ 
                Substitution  $\Gamma$   $\rightarrow$  Pattern  $\alpha$   $\Gamma$   $\rightarrow$   $\mu$   $\alpha$ .
```

The type *Pattern* α Γ (see Section 5.2) is used in our library for the intensional representation of the left- and right-hand sides of rewrite rules over a term type α . Its type argument Γ is a so-called *metavariable environment*: a type-level list that encodes the types of the metavariables in a rewrite rule. Successfully matching a term against a left-hand-side pattern results in a substitution (Section 5.3) for the metavariables that

occur in the pattern. As pattern matching may fail, the function *match'* returns its result in a monad μ . This function requires the metavariable environment Γ involved to be in the class *Mappable* (defined in Section 5.1), which simply means that an empty substitution can be produced for Γ . Substitutions are partial maps from metavariables drawn from a given environment to matched subterms. Given such a substitution and a right-hand-side pattern, the generic function *substitute'* attempts to construct a new term value. This construction fails if the substitution is not defined for all metavariables that occur in the right-hand-side pattern, which explains the monadic result type of *substitute'*.

As metavariable environments are only of interest to the internals of our library, they are hidden from the user by wrapping the left- and right-hand-side patterns that constitute a rewrite rule in an existential type:

```
data Rule ::  $\star \rightarrow \star$  where
  Rule :: Mappable  $\Gamma \Rightarrow$  Pattern  $\alpha \Gamma \rightarrow$  Pattern  $\alpha \Gamma \rightarrow$  Rule  $\alpha$ .
```

Here, the existential type *Rule* is defined as a so-called *generalised algebraic datatype* or GADT (Xi et al., 2003; Peyton Jones et al., 2006).

Given the GADT *Rule* and suitable definitions of *match'* and *substitute'*, the monadic rewrite function *rewriteWithM* can be written as

```
rewriteWithM :: (Rewritable  $\alpha$ , Monad  $\mu$ )  $\Rightarrow$  Rule  $\alpha \rightarrow$   $\alpha \rightarrow$   $\mu \alpha$ 
rewriteWithM (Rule lhs rhs) x = do
  s  $\leftarrow$  match' lhs x
  substitute' s rhs.
```

That is, the term x is matched against the left-hand side *lhs* of a given rewrite rule. If the match is successful, the resulting substitution s is applied to the right-hand side *rhs* of the rewrite rule in order to produce the result term. An implementation for the nonmonadic rewrite function *rewriteWith* is obtained by instantiating the type of *rewriteWithM* with the *Maybe*-monad:

```
rewriteWith :: Rewritable  $\alpha \Rightarrow$  Rule  $\alpha \rightarrow$   $\alpha \rightarrow$   $\alpha$ 
rewriteWith rule x = case rewriteWithM rule x of
  Nothing  $\rightarrow$  x
  Just y  $\rightarrow$  y.
```

In the remainder of this section, we discuss the implementation of metavariables (Section 5.1), patterns (Section 5.2), substitutions (Section 5.3), and generic pattern matching (Section 5.4).

5.1 Typed metavariables

In our intensional representation of rewrite rules, we encode metavariables by *De Bruijn indices* (De Bruijn, 1972). Our implementation allows different metavariables to range over differently typed subterms. To enforce a type-safe use of metavariables, we adopt

the approach of [Pašalić and Linger \(2004\)](#) and implement metavariables as values of the GADT *Ref* of typed references:

```
data Ref ::  $\star \rightarrow \star \rightarrow \star$  where
  RZero :: Ref  $\alpha$  ( $\alpha$  ::  $\Gamma$ )
  RSucc :: Ref  $\alpha$   $\Gamma \rightarrow$  Ref  $\alpha$  ( $\beta$  ::  $\Gamma$ ).
```

Here, we use as metavariable environments Γ the heterogeneous lists constructed from *Nil*' and *::*' that we also use in generic representations. A value of type *Ref* α Γ then carries the Peano encoding of an index for an α -typed position in a heterogeneous list of type Γ . Note that such a value can never refer to an empty list, simply because the constructor types dictate that the lists contain at least one value.

As an example of the use of *Ref*, consider the function *deref* for dereferencing a typed reference to a value in a heterogeneously typed list:

```
deref :: Ref  $\alpha$   $\Gamma \rightarrow$   $\Gamma \rightarrow \alpha$ 
deref RZero ( $x$  ::  $xs$ ) =  $x$ 
deref (RSucc  $r$ ) ( $x$  ::  $xs$ ) = deref  $r$   $xs$ .
```

In the implementation of *match'* and *substitute'*, typed references are used as indices into heterogeneously typed partial maps:

```
data PMap ::  $\star \rightarrow \star$  where
  PNil :: PMap Nil
  PCons :: Rewritable  $\alpha \Rightarrow$  Maybe  $\alpha \rightarrow$  PMap  $\Gamma \rightarrow$  PMap ( $\alpha$  ::  $\Gamma$ ).
```

Values of type *PMap* Γ are partial maps from Γ -typed references to rewritable terms. Looking up a value in a partial map is implemented through the function *lookup*,

```
lookup :: Monad  $\mu \Rightarrow$  Ref  $\alpha$   $\Gamma \rightarrow$  PMap  $\Gamma \rightarrow \mu$   $\alpha$ 
lookup RZero (PCons Nothing  $s$ ) = fail "null reference"
lookup RZero (PCons (Just  $x$ )  $s$ ) = return  $x$ 
lookup (RSucc  $r$ ) (PCons  $mb$   $s$ ) = lookup  $r$   $s$ ,
```

that returns its result in a monad μ to provide for the case in which looking up fails. Since the types of the *RZero* and *RSucc* constructors ensure that the referenced partial map is nonempty, the definition of *lookup* does not require a case for *PNil*.

For the construction of partial maps of type *PMap* Γ , we require that Γ is a type-level list of rewritable-term types, so that *PNil* and *PCons* can be used to produce an initial, empty map. To this end, we make the list constructors *Nil* and *::* instances of a class *Mappable* that provides an empty-map constructor:

```
class Mappable  $\Gamma$  where
  empty :: PMap  $\Gamma$ 
instance Mappable Nil where
  empty = PNil
instance (Rewritable  $\alpha$ , Mappable  $\Gamma$ )  $\Rightarrow$  Mappable ( $\alpha$  ::  $\Gamma$ ) where
  empty = PCons Nothing empty.
```

Updating a rewritable term in a partial map involves destructing a typed reference and traversing the map until the appropriate position has been reached:

$$\begin{aligned} \text{update} &:: \text{Ref } \alpha \ \Gamma \rightarrow \alpha \rightarrow \text{PMap } \Gamma \rightarrow \text{PMap } \Gamma \\ \text{update } \text{RZero } x \ (\text{PCons } mb \ s) &= \text{PCons } (\text{Just } x) \ s \\ \text{update } (\text{RSucc } r) \ x \ (\text{PCons } mb \ s) &= \text{PCons } mb \ (\text{update } r \ x \ s). \end{aligned}$$

Singleton mappings are then constructed by updating a single term in an empty map:

$$\begin{aligned} \text{singleton} &:: (\text{Rewritable } \alpha, \text{Mappable } \Gamma) \Rightarrow \text{Ref } \alpha \ \Gamma \rightarrow \alpha \rightarrow \text{PMap } \Gamma \\ \text{singleton } r \ x &= \text{update } r \ x \ \text{empty}. \end{aligned}$$

Finally, two maps for the same environment Γ can be merged if they agree on their codomain:

$$\begin{aligned} (\oplus) &:: \text{Monad } \mu \Rightarrow \text{PMap } \Gamma \rightarrow \text{PMap } \Gamma \rightarrow \mu \ (\text{PMap } \Gamma) \\ \text{PNil} \oplus \text{PNil} &= \text{return } \text{PNil} \\ \text{PCons } \text{Nothing } s \oplus \text{PCons } \text{Nothing } s' &= \text{liftM } (\text{PCons } \text{Nothing}) \ (s \oplus s') \\ \text{PCons } \text{Nothing } s \oplus \text{PCons } (\text{Just } y) \ s' &= \text{liftM } (\text{PCons } (\text{Just } y)) \ (s \oplus s') \\ \text{PCons } (\text{Just } x) \ s \oplus \text{PCons } \text{Nothing } s' &= \text{liftM } (\text{PCons } (\text{Just } x)) \ (s \oplus s') \\ \text{PCons } (\text{Just } x) \ s \oplus \text{PCons } (\text{Just } y) \ s' &= \text{liftM } (\text{PCons } (\text{Just } x)) \ (s \oplus s') \\ &\quad | \ x \equiv y \\ &\quad | \ \text{otherwise} \qquad \qquad \qquad = \text{fail } \text{"merging failure"}. \end{aligned}$$

Here, liftM ,

$$\text{liftM} :: \text{Monad } \mu \Rightarrow (\alpha \rightarrow \beta) \rightarrow \mu \ \alpha \rightarrow \mu \ \beta,$$

is the function from Haskell's standard libraries that lifts a given unary function into an arbitrary monad. If, for at least one reference, the arguments of the monadic merge operator \oplus produce different terms, merging fails. As all terms contained in a partial map are of types in the class *Rewritable*, equality of terms can be tested by means of the generic equality test \equiv .

5.2 Generic patterns

Recall from the definition of the GADT *Rule* that the left- and right-hand sides of rewrite rules are represented by values of the type $\text{Pattern } \alpha \ \Gamma$, where α is the type of terms to be rewritten and Γ is a metavariable environment. The idea is to derive the definition of $\text{Pattern } \alpha \ \Gamma$ from the definition of α , much like in Section 2.2 the definition of *EProp* was derived from the definition of *Prop*, but without requiring the user to explicitly declare the pattern type. As pattern types are supposed to hold the same values as their corresponding term types, but additionally allow each subterm to be replaced by a metavariable, *Pattern* can be elegantly defined in terms of a so-called *type-indexed datatype*. A type-indexed datatype (Hinze et al., 2004) is a datatype that is defined by induction over the structure of generically representable types.

Here, we encode type-indexed datatypes as *datatype families* (Schrijvers et al., 2008). That is, we define a datatype family *Extended*,

data family *Extended* $\alpha :: \star \rightarrow \star$

with index α . A type *Extended* $\alpha \Gamma$ is to be interpreted as the type that is obtained from extending α with metavariables from Γ .

Instances of *Extended* are given for all representation constructors. These instances recursively introduce metavariable alternatives in all subterm positions in the generic representation of a type's structure, while duplicating the remainder of the structure. A pattern is then defined as either a duplicate of a type's structure with metavariable alternatives for all subterm positions or otherwise a metavariable of the appropriate type:

type *Pattern* $\alpha \Gamma = \text{Extended} (\text{Rep } \alpha) \Gamma \text{ :+} \text{ Ref } \alpha \Gamma$.

As values of base types do not contain subterms, the extension of these types amounts to mere duplication:

newtype instance *Extended Int* $\Gamma = \text{Int}' \text{ Int}$
newtype instance *Extended Float* $\Gamma = \text{Float}' \text{ Float}$
newtype instance *Extended Char* $\Gamma = \text{Char}' \text{ Char}$.

Note that in our library subterm positions in a type are encoded as elements of type-level lists. Hence, sums and lists are extended recursively with metavariable alternatives inserted for all list elements:

data instance *Extended Nil* $\Gamma = \text{Nil}'$
data instance *Extended* $(\alpha \text{ :+} \beta) \Gamma = \text{Inl}' (\text{Extended } \alpha \Gamma) \mid \text{Inr}' (\text{Extended } \beta \Gamma)$
data instance *Extended* $(\alpha \text{ :::} \beta) \Gamma = \text{Pattern } \alpha \Gamma \text{ :::}' \text{ Extended } \beta \Gamma$.

Because extended types contain at least the values of the original representation types (modulo renaming of constructors and redirections into sum types), converting from terms to patterns is straightforward. First, we declare a class *Extensible* of types that can be lifted into their extended counterparts,

class *Extensible* α **where**
extend $:: \alpha \rightarrow \text{Extended } \alpha \text{ Nil}$,

and then we define a generic extension function *extend'* for constructing patterns from terms:

extend' $:: \text{Rewritable } \alpha \Rightarrow \alpha \rightarrow \text{Pattern } \alpha \text{ Nil}$
extend' $x = \text{Inl} (\text{extend} (\text{from } x))$.

Note that a value of a type *Pattern* $\alpha \text{ Nil}$, due to the empty metavariable environment, is guaranteed to not contain any metavariables.

Lifting base types reduces to wrapping values in the extension constructors *Int'*, *Float'*, and *Char'*:

instance *Extensible Int* **where** *extend* = *Int'*
instance *Extensible Float* **where** *extend* = *Float'*
instance *Extensible Char* **where** *extend* = *Char'*.

Extension of the empty list involves converting from *Nil* to *Nil'*,

```
instance Extensible Nil where
  extend Nil = Nil',
```

while sums are extended recursively:

```
instance (Extensible  $\alpha$ , Extensible  $\beta$ )  $\Rightarrow$  Extensible ( $\alpha$  :::  $\beta$ ) where
  extend (Inl  $x$ ) = Inl' (extend  $x$ )
  extend (Inr  $y$ ) = Inr' (extend  $y$ ).
```

For **:::**, we require the first type argument to be rewritable, so that subterms can be lifted generically:

```
instance (Rewritable  $\alpha$ , Extensible  $\beta$ )  $\Rightarrow$  Extensible ( $\alpha$  :::  $\beta$ ) where
  extend ( $x$  :::  $xs$ ) = extend'  $x$  :::' extend  $xs$ .
```

The conversion from terms to patterns is used in Section 6 for the synthesis of rewrite rules from functions over term types.

5.3 Generic substitutions

Substitutions are just partial maps over a given metavariable environment:

```
type Substitution  $\Gamma$  = PMap  $\Gamma$ .
```

Applying a substitution then involves traversing a value of an extended type and replacing all metavariable occurrences by subterms drawn from the partial map in order to obtain a term representation:

```
class Substitutable  $\alpha$  where
  substitute :: Monad  $\mu$   $\Rightarrow$  Substitution  $\Gamma$   $\rightarrow$  Extended  $\alpha$   $\Gamma$   $\rightarrow$   $\mu$   $\alpha$ .
```

As looking up metavariables in partial maps may fail, *substitute* returns its result in a monad μ . To apply a substitution to a pattern, we distinguish between values of extended types and metavariables. In the former case, we use *substitute* to yield a representation and then convert this representation to a term by means of *to*. In the latter case, the metavariable is looked up in the partial map that represents the substitution:

```
substitute' :: (Rewritable  $\alpha$ , Monad  $\mu$ )  $\Rightarrow$  Substitution  $\Gamma$   $\rightarrow$  Pattern  $\alpha$   $\Gamma$   $\rightarrow$   $\mu$   $\alpha$ 
substitute'  $s$  (Inl  $e$ ) = liftM to (substitute  $s$   $e$ )
substitute'  $s$  (Inr  $r$ ) = lookup  $r$   $s$ 
```

Substitutions over extended base types are performed by stripping off the extension constructors:

```
instance Substitutable Int where substitute  $s$  (Inl'  $n$ ) = return  $n$ 
instance Substitutable Float where substitute  $s$  (Float'  $r$ ) = return  $r$ 
instance Substitutable Char where substitute  $s$  (Char'  $c$ ) = return  $c$ .
```

Similarly, for the empty lists of constructor arguments, we have

```
instance Substitutable Nil where
  substitute s Nil = return Nil.
```

Extended sum values are processed recursively and the obtained values are reinjected into the appropriate side of the original sum type:

```
instance (Substitutable  $\alpha$ , Substitutable  $\beta$ )  $\Rightarrow$  Substitutable ( $\alpha$  :+:  $\beta$ ) where
  substitute s (Inl e) = liftM Inl (substitute s e)
  substitute s (Inr e) = liftM Inr (substitute s e).
```

The instance for `:::` once more requires all elements in a list to be in the class *Rewritable* and invokes the generic function *substitute'* to apply substitutions to patterns:

```
instance (Rewritable  $\alpha$ , Substitutable  $\beta$ )  $\Rightarrow$  Substitutable ( $\alpha$  :::  $\beta$ ) where
  substitute s (pat ::: es) = liftM2 ( ::: ) (substitute' s pat) (substitute s es).
```

To lift the list constructor `:::` into a monad, this instance uses the standard function *liftM2*,

$$\text{liftM2} :: \text{Monad } \mu \Rightarrow (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \mu \alpha \rightarrow \mu \beta \rightarrow \mu \gamma,$$

for turning binary functions into monadic operations.

5.4 Generic pattern matching

Finally, let us consider how substitutions are constructed, namely, by generically matching term values against patterns. The required machinery breaks down in a class *Matchable* of representation types which can be matched against their recursively extended counterparts,

```
class Matchable  $\alpha$  where
  match :: (Mappable  $\Gamma$ , Monad  $\mu$ )  $\Rightarrow$  Extended  $\alpha$   $\Gamma \rightarrow \alpha \rightarrow \mu$  (Substitution  $\Gamma$ ),
```

and a top-level generic function *match'* for matching terms against either an extended representation or otherwise a top-level metavariable:

```
match' :: (Rewritable  $\alpha$ , Mappable  $\Gamma$ , Monad  $\mu$ )  $\Rightarrow$ 
  Pattern  $\alpha$   $\Gamma \rightarrow \alpha \rightarrow \mu$  (Substitution  $\Gamma$ )
match' (Inl e) x = match e (from x)
match' (Inr r) x = return (singleton r x).
```

If a term x is to be matched against an extended representation e , x is itself converted to a generic representation *from x* and matched by means of *match*. If x is matched against a metavariable r , a singleton substitution is constructed that maps r to x . Pattern-match failures are dealt with monadically.

Matching values of base type against extended base values requires an equality test. If this test succeeds, an empty substitution is produced; otherwise, a mismatch is reported:

```

instance Matchable Int where
  match (Int' n) n'
    | n ≡ n'    = return empty
    | otherwise = fail "pattern mismatch"

instance Matchable Float where
  match (Float' r) r'
    | r ≡ r'    = return empty
    | otherwise = fail "pattern mismatch"

instance Matchable Char where
  match (Char' c) c'
    | c ≡ c'    = return empty
    | otherwise = fail "pattern mismatch".

```

Provided that both the extended representation and the term representation are completely defined (i.e., do not diverge), matching is always successful for empty lists:

```

instance Matchable Nil where
  match Nil' Nil = return empty.

```

For values of sum types, we check whether the extended representation and the term representation encode the same alternative. If so, we proceed recursively; otherwise, matching fails:

```

instance (Matchable α, Matchable β) ⇒ Matchable (α :+: β) where
  match (Inl' e) (Inl x) = match e x
  match (Inr' e) (Inr y) = match e y
  match _ _                = fail "pattern mismatch".

```

For nonempty lists, we attempt to match the head x against a pattern pat by means of a call to the generic function $match'$ and the tail xs against extended representations es through a recursive call to $match'$. If both x and xs are matched successfully, the resulting substitutions are merged with the operator \oplus from Section 5.1:

```

instance (Rewritable α, Matchable β) ⇒ Matchable (α ::: β) where
  match (pat ::: es) (x ::: xs) = join (liftM2 (⊕) (match' pat x) (match es xs)).

```

As both matching and merging may fail, this gives rise to a nested monadic structure, which we flatten with a call to the function $join$,

$$join :: Monad \mu \Rightarrow \mu (\mu \alpha) \Rightarrow \mu \alpha,$$

from the standard libraries.

This completes our implementation of generic matching and substitution.

6 Synthesising rewrite rules

In the previous section, we have demonstrated how rewrite rules are, in our library, intensionally represented in terms of the type synonym *Pattern* and the type-indexed datatype *Extended*. Implementing patterns through generic types frees the user of the library from the burden of defining separate datatypes for representing the left- and right-hand sides of rewrite rules for various term types, but still allows us to enjoy the benefits of observable rules.

However, this use of generic types raises the question how the user is supposed to *define* her rewrite rules. Using `$` from the Haskell Prelude,

```
infixr 0 $
( $ ) :: (α → β) → α → β
f $ x = f x,
```

to avoid excessive use of parentheses, she could write the rewrite rule derived from the principle of contradiction as

```
contradiction :: Rule Prop
contradiction = Rule lhs rhs
  where
    lhs = Inl $ Inr' $ Inr' $ Inr' $ Inr' $ Inr' $ Inl' $
          (Inl $ Inr' $ Inr' $ Inr' $ Inl' $ Inr RZero ::: ' Nil') ::: '
          Inr RZero ::: ' Nil'
    rhs = Inl $ Inr' $ Inr' $ Inl' Nil',
```

but clearly this style of definition is not exactly user-friendly and, moreover, quite error-prone. Of course, the definition of so-called *smart constructors*, such as

```
(^') :: Extended Prop → Extended Prop → Extended Prop
p ^' q = Inl $ Inr' $ Inr' $ Inr' $ Inr' $ Inl' $ p ::: ' q ::: ' Nil'
```

make take away some of the burden, but these smart constructors then need to be defined for all types of rewriteable terms, defeating the very purpose of datatype-generic programming. Instead, our library facilitates the definition of rewrite rules in terms of the real constructors of the type of terms that are to be rewritten. The rule above, for example, can conveniently and concisely be written as

```
contradiction :: Rule Prop
contradiction = synthesise (λp → p :^: Not p ↔ F).
```

That is, rewrite rules are synthesised from functions that take placeholders for metavariables as arguments and produce values of the type of rewritable terms—in this case, *Prop*. This way, rewrite rules are specified in the same way for different term types, while the internal representation of the rules remains hidden from the user.

To synthesise rules from functions, we develop some more generic machinery. The idea is to instantiate each function parameter twice—each time with distinct term

values. For example, for the function $\lambda p \rightarrow p : \wedge : \text{Not } p \mapsto F$ that we used above, we could instantiate the parameter p first with the value T and then with the value F . The first instantiation then yields the left-hand side $T : \wedge : \text{Not } T$ and the right-hand F ; the second instantiation yields $F : \wedge : \text{Not } F$ and F . Next, we compare the obtained pairs of left- and right-hand sides to determine where metavariables are to be inserted. As, in our example, the produced left-hand sides differ in the left operand of $: \wedge :$ and in the argument of Not , an occurrence of some metavariable is inserted in these locations. The two right-hand sides are identical, so no metavariable occurrence will show up there.

In this section, we implement this scheme of producing rewrite rules generically. We first show how to generate pairs of distinct values for term types (Section 6.1). Then, we present a generic *diff* function that localises the positions in which metavariables are to be inserted (Section 6.2). Finally, a class of synthesiser types is given (Section 6.3).

6.1 Generic sampling

To produce pairs of distinct values for types in *Rewritable*, we define a class *Sampleable*,

```
class Sampleable  $\alpha$  where
  left  ::  $\alpha$ 
  right ::  $\alpha$ .
```

Instances of *Sampleable* are supposed to have their methods *left* and *right* produce values that differ in their top-level constructors. With instances of *Sampleable* declared for all generic representation types, functions *left'* and *right'* can be defined generically for all types of rewritable terms:

```
left', right' :: Rewritable  $\alpha \Rightarrow \alpha$ 
left'  = to left
right' = to right.
```

As always, appropriate instances for base types are straightforward to produce:

```
instance Sampleable Int   where left = 0   ; right = 1
instance Sampleable Float where left = 0   ; right = 1
instance Sampleable Char where left = 'a' ; right = 'b'.
```

For *Nil*, it is not possible to produce distinct *left* and *right* values:

```
instance Sampleable Nil where
  left = Nil
  right = Nil.
```

As a result, no metavariables are ever introduced in rules over types with only a single nonbottom value. (Note that this is by no means a fundamental limitation as meaningful rewrite rules for such types cannot be given anyway.) Sum types present us with the opportunity to actually produce values that are distinct in their top-level constructor. For *left*, we choose *Inl*, while for *right*, *Inr* is selected:


```

instance (Sampleable  $\alpha$ , Sampleable  $\beta$ )  $\Rightarrow$  Sampleable ( $\alpha$  :::  $\beta$ ) where
  left  = Inl left
  right = Inr left.

```

The values for *left* and *right* need to be finite, as infinite values will lead to nontermination of the generic *diff* function in Section 6.2. To guarantee termination, we require the leftmost constructor of a datatype to be nonrecursive, such that *left* always produces a finite value for this constructor. Then, we can use the same value in the definition of *left* and *right* as the top-level constructor distinguishes the values.

For **:::**, we have only one constructor at our disposal, so a distinction in top-level constructors is to be made at a deeper level:

```

instance (Rewritable  $\alpha$ , Sampleable  $\beta$ )  $\Rightarrow$  Sampleable ( $\alpha$  :::  $\beta$ ) where
  left  = left' ::: left
  right = right' ::: right.

```

6.2 Generic diff

To determine at which positions in a pattern metavariables are to be introduced, we require the ability to generically compute a “diff” between two patterns. If such a position is found, it depends on the type of the metavariable to be introduced whether or not a new pattern can be distilled from the differences between the pattern values compared. To this end, we require term types to be in the class *Typeable*, so that their types can be compared at run-time.

The class *Typeable* comes with an operation *gcast*,

$$gcast :: (Typeable\ \alpha, Typeable\ \beta) \Rightarrow \varphi\ \alpha \rightarrow Maybe\ (\varphi\ \beta),$$

that allows values of type $\varphi\ \alpha$ to be cast into values of type $\varphi\ \beta$ if and only if α and β are the same type. In our implementation of a generic diff, we attempt to cast values of type *Pattern* $\alpha\ \Gamma$ into values of type *Pattern* $\beta\ \Gamma$ with both α and β in *Typeable*. This means we need a cast operation that operates on the second-to-last argument of a type constructor rather than on the last argument. To obtain such an operation, we define a type *Flip* that swaps the last two arguments of a type constructor,

$$\text{newtype } Flip\ \varphi\ \alpha\ \beta = Flip\ (\varphi\ \beta\ \alpha),$$

and then define a function *gcast'* for casting second-to-last type arguments:

```

gcast' :: (Typeable  $\alpha$ , Typeable  $\beta$ )  $\Rightarrow$   $\varphi\ \alpha\ \gamma \rightarrow Maybe\ (\varphi\ \beta\ \gamma)$ 
gcast' x = case gcast (Flip x) of
  Nothing       $\rightarrow$  Nothing
  Just (Flip y)  $\rightarrow$  Just y.

```

Next, we define a class *Diffable* of representation types for which a diff can be computed:

```

class Diffable  $\alpha$  where
  diff :: Typeable  $\beta \Rightarrow$ 
         Extended  $\alpha$   $\Gamma \rightarrow$  Extended  $\alpha$   $\Gamma \rightarrow$  Maybe (Extended  $\alpha$  ( $\beta$  ::  $\Gamma$ )).

```

For each generic representation type α , the overloaded function *diff* takes two values of type *Extended* α Γ for some environment Γ and attempts to introduce a new β -typed metavariable at the deeper locations in which the two values differ. If the two values differ at top-level or at an inappropriately typed location, *diff* fails and produces *Nothing*.

Diffs for rewritable terms can now be computed by means of a generic function *diff'*:

```

diff' :: (Rewritable  $\alpha$ , Typeable  $\beta$ )  $\Rightarrow$ 
        Pattern  $\alpha$   $\Gamma \rightarrow$  Pattern  $\alpha$   $\Gamma \rightarrow$  Maybe (Pattern  $\alpha$  ( $\beta$  ::  $\Gamma$ ))
diff' (Inl  $e$ ) (Inl  $e'$ )           =
  case diff  $e$   $e'$  of Nothing  $\rightarrow$  gcast' (Inr RZero) ; Just  $e'' \rightarrow$  Just (Inl  $e''$ )
diff' (Inr  $r$ ) (Inr  $r'$ ) |  $r \equiv r' =$  Just (Inr (RSucc  $r$ ))
diff' _ _ _ _ _ _ _ _ _ _ _ _ _ = Nothing.

```

This generic function takes patterns over a type α as argument. If both patterns consist of values e and e' of an extended type, the overloaded *diff* function is used to compare e and e' . (Note that these values are produced by *left* and *right* from Section 6.1 and have to be unequal or finite as to ensure termination.) If *diff* successfully computes a combined value e'' of extended type, this value is wrapped in a pattern *Inl* e'' and returned. If *diff* fails, we attempt to insert a metavariable *RZero* of type β at top-level. As insertion of such a metavariable is only allowed if α and β are the same type, we use the function *gcast'* defined above to compare α and β at run-time. If both patterns are metavariable alternatives *Inr* r and *Inr* r' , we require r and r' to be the same metavariable and construct a corresponding metavariable *Inr* (*RSucc* r) in the extended environment β :: Γ . If r and r' are not equal, or if the two patterns are constructed from different alternatives, we produce *Nothing*.

It remains to give instances of *Diffable* for our generic representation constructors. As values of base types contain no subterms and can thus only differ at top-level, an implementation of *diff* for these types reduces to testing for equality:

```

instance Diffable Int where
  diff (Int'  $n$ ) (Int'  $n'$ )
    |  $n \equiv n' =$  Just (Int'  $n$ )
    | otherwise = Nothing

instance Diffable Float where
  diff (Float'  $r$ ) (Float'  $r'$ )
    |  $r \equiv r' =$  Just (Float'  $r$ )
    | otherwise = Nothing

instance Diffable Char where
  diff (Char'  $c$ ) (Char'  $c'$ )
    |  $c \equiv c' =$  Just (Char'  $c$ )
    | otherwise = Nothing.

```

The extension of *Nil* holds only a single value *Nil*, so *diff* for empty lists cannot fail:

```
instance Diffable Nil where
  diff Nil Nil = Just Nil.
```

For values of sum type, we compare the top-level constructors. If these are different, we produce *Nothing*; otherwise, comparison proceeds recursively:

```
instance (Diffable  $\alpha$ , Diffable  $\beta$ )  $\Rightarrow$  Diffable ( $\alpha :+: \beta$ ) where
  diff (Inl e) (Inl e') =
    case diff e e' of Nothing  $\rightarrow$  Nothing; Just e''  $\rightarrow$  Just (Inl e'')
  diff (Inr e) (Inr e') =
    case diff e e' of Nothing  $\rightarrow$  Nothing; Just e''  $\rightarrow$  Just (Inr e'')
  diff _ _ = Nothing.
```

Similarly, for $:::$, the comparison of two values *pat* $:::$ *es* and *pat'* $:::$ *es'* continues recursively underneath the constructor $:::$:

```
instance (Rewritable  $\alpha$ , Diffable  $\beta$ )  $\Rightarrow$  Diffable ( $\alpha ::: \beta$ ) where
  diff (pat  $:::$  es) (pat'  $:::$  es') =
    case (diff pat pat', diff es es') of
      (Just pat'', Just es'')  $\rightarrow$  Just (pat''  $:::$  es'')
      _  $\rightarrow$  Nothing.
```

6.3 Generic synthesis

With generic sampling and generic *diff* defined, we can now implement the synthesis of rewrite rules from functions over term types. These functions wrap the left- and right-hand sides of rules in values of a type *Template*,

```
data Template  $\alpha$  = Template  $\alpha$   $\alpha$ ,
```

of which the values simply constitute pairs of terms. For the concise definition of templates, we introduce an operator \mapsto :

```
infix 1  $\mapsto$ 
( $\mapsto$ ) ::  $\alpha \rightarrow \alpha \rightarrow$  Template  $\alpha$ 
lhs  $\mapsto$  rhs = Template lhs rhs.
```

Next, we define a class *Synthesiser* of types of which the values can be used to synthesise rewrite rules:

```
class Rewritable (Term  $\alpha$ )  $\Rightarrow$  Synthesiser  $\alpha$  where
  type Term  $\alpha$  ::  $\star$ 
  type Env  $\alpha$  ::  $\star$ 
  patterns ::  $\alpha \rightarrow$  (Pattern (Term  $\alpha$ ) (Env  $\alpha$ ), Pattern (Term  $\alpha$ ) (Env  $\alpha$ )).
```

Each instance α of *Synthesiser* has an associated type synonym *Term* α that gives the type of terms that are rewritten by a synthesised rewrite rule. Similarly, the associated type synonym *Env* α gives the term types over which the metavariables of a synthesised rule range. For example, a rewrite rule synthesised from a function of a type $\alpha \rightarrow \beta \rightarrow \text{Template } \gamma$ has two metavariables, ranging over values of types α and β , and is used to rewrite terms of type γ . Operationally, a value x of a type from *Synthesiser* can be used to produce a pair *patterns* x that contains the left- and right-hand-side components of a rewrite rule. Synthesis then reduces to combining these components in a *Rule*-value:

```
synthesise :: (Synthesiser  $\alpha$ , Mappable (Env  $\alpha$ ))  $\Rightarrow$   $\alpha \rightarrow$  Rule (Term  $\alpha$ )
synthesise x = let (lhs, rhs) = patterns x in Rule lhs rhs.
```

Instances of the class *Synthesiser* are defined inductively over the structure of function types. As a base case, we have an instance for *Template* α for any type α of rewritable terms:

```
instance Rewritable  $\alpha \Rightarrow$  Synthesiser (Template  $\alpha$ ) where
  type Term (Template  $\alpha$ ) =  $\alpha$ 
  type Env (Template  $\alpha$ ) = Nil
  patterns (Template lhs rhs) = (extend' lhs, extend' rhs).
```

Rewrite rules that are synthesised directly from templates over α operate on terms of type α and contain no metavariables. Left- and right-hand sides for these rules can be obtained simply by lifting template components into the type *Pattern* α *Nil* of patterns over α without variables, for which we use the generic function *extend'* defined in Section 5.2.

In the inductive step, we require, in order for a function type $\alpha \rightarrow \beta$ to be in the class *Synthesiser*, α to be a type of rewritable terms and β to be a type of synthesisers:

```
instance (Rewritable  $\alpha$ , Synthesiser  $\beta$ )  $\Rightarrow$  Synthesiser ( $\alpha \rightarrow \beta$ ) where
  type Term ( $\alpha \rightarrow \beta$ ) = Term  $\beta$ 
  type Env ( $\alpha \rightarrow \beta$ ) =  $\alpha ::$  Env  $\beta$ 
  patterns f =
    let (lhs, rhs) = patterns (f left')
        (lhs', rhs') = patterns (f right')
    in case (diff' lhs lhs', diff' rhs rhs') of
        (Just lhs'', Just rhs'')  $\rightarrow$  (lhs'', rhs'')
        _  $\rightarrow$  error "synthesis failure".
```

Function abstraction over α adds an α -typed metavariable to the environment *Env* β , but does not alter the type *Term* β of terms the synthesised rule operates on. Patterns of the left- and right-hand sides of the rewrite rule are constructed by applying the function twice (once on the value produced by *left'* and once on the value produced by *right'*) and then computing diffs from the obtained components, possibly introducing occurrences of a new metavariable that ranges over terms of type α . If diffs cannot be computed, synthesis fails with a run-time error—an issue to be discussed in more detail in the next section.

7 Detecting ill-formed rewrite rules

In the previous sections, we have shown the implementation of our library’s core functionality. In particular, we have shown how, although we use an intensional representation of rewrite rules internally, we allow the user to define rules in terms of functions over the datatypes of terms being rewritten. Due to this sugarcoating, additional verification of rewrite rules is required.

Consider, for example, the following rewrite rule over propositional formulae,

$$\begin{aligned} \text{funny} &:: \text{Rule Prop} \\ \text{funny} &= \text{synthesise } (\lambda n \rightarrow f \ n \mapsto T), \end{aligned}$$

where f is some function taking *Int*-values to values of type *Prop*:

$$f :: \text{Int} \rightarrow \text{Prop}.$$

It is unclear what the semantics of such a rewrite rule should be. That is, in a well-formed rewrite rule we expect metavariables to exclusively occur as constructor arguments, not as arguments to arbitrary functions. Using Haskell’s variables as placeholders for our metavariables means, however, that we cannot preclude such ill-formed rules and that we have to rely on the user not to construct nonsensical rules as the one above.

Another class of meaningless rewrite rules can be excluded by equipping our library with functionality for detecting their ill-formedness. Consider, for instance, the rule

$$\begin{aligned} \text{unbound} &:: \text{Rule Prop} \\ \text{unbound} &= \text{synthesise } (\lambda p \rightarrow T \mapsto T : \forall : p), \end{aligned}$$

in which the metavariable p on the right-hand side is not bound on the left-hand side of the rewrite rule, and,

$$\begin{aligned} \text{superfluous} &:: \text{Rule Prop} \\ \text{superfluous} &= \text{synthesise } (\lambda p \ q \rightarrow p : \forall : p \mapsto p), \end{aligned}$$

in which the metavariable q is superfluous since it is “declared” but not used at all in the rewrite rule. In general, we consider a rewrite rule well-formed if and only if all of its declared metavariables are bound in its left-hand side—and, interestingly, this notion of well-formedness can be checked for statically, i.e., without applying the rule.

To this end, we extend the library with a function *validate* that provides the user with an opportunity to verify the use of declared metavariables in rewrite rules:

$$\text{validate} :: \text{Rewritable } \alpha \Rightarrow \text{Rule } \alpha \rightarrow \text{Bool}.$$

This function is intended to be applied just after rule synthesis.

Validation is achieved by constructing a *use record* with a field for each metavariable, denoting its presence in the left-hand side of the rewrite rule:

```

data Record ::  $\star \rightarrow \star$  where
  RNil   :: Record Nil
  RCons :: Bool  $\rightarrow$  Record  $\Gamma \rightarrow$  Record ( $\alpha :: \Gamma$ ).

```

An initial blank record is created by setting each presence to *False*:

```

class Recordable  $\Gamma$  where
  blank :: Record  $\Gamma$ 
instance Recordable Nil where
  blank = RNil
instance Recordable  $\Gamma \Rightarrow$  Recordable ( $\alpha :: \Gamma$ ) where
  blank = RCons False blank.

```

We now require environments to be instances of the type class *Recordable* and, hence, a constraint is added to the constructor *Rule* from Section 5:

```

data Rule ::  $\star \rightarrow \star$  where
  Rule :: ( $\dots$ , Recordable  $\Gamma$ )  $\Rightarrow$  Pattern  $\alpha$   $\Gamma \rightarrow$  Pattern  $\alpha$   $\Gamma \rightarrow$  Rule  $\alpha$ .

```

A use record is updated by traversing the left-hand side of a rewrite rule and checking off each metavariable encountered:

```

class Validateable  $\alpha$  where
  record :: Extended  $\alpha$   $\Gamma \rightarrow$  Record  $\Gamma \rightarrow$  Record  $\Gamma$ .

```

Recall from Section 5 that a *Pattern* is either a value of a corresponding extended type or else a metavariable. In the former case, we traverse the extended term recursively, looking for metavariable occurrences; in the latter case we, check off the metavariable in the use record:

```

record' :: Rewritable  $\alpha \Rightarrow$  Pattern  $\alpha$   $\Gamma \rightarrow$  Record  $\Gamma \rightarrow$  Record  $\Gamma$ 
record' (Inl  $e$ )  $rec$  = record  $e$   $rec$ 
record' (Inr RZero) (RCons  $b$   $rec$ ) = RCons True  $rec$ 
record' (Inr (RSucc  $r$ )) (RCons  $b$   $rec$ ) = RCons  $b$  (record' (Inr  $r$ )  $rec$ ).

```

Traversing base-type values results in no change to the use record as base values cannot contain metavariables:

```

instance Validateable Int where record (Int'  $n$ ) = id
instance Validateable Float where record (Float'  $r$ ) = id
instance Validateable Char where record (Char'  $c$ ) = id.

```

Similarly, traversing *Nil* values results in the original record:

```

instance Validateable Nil where
  record Nil' = id.

```

Values of sum types are traversed by stripping their top-level constructor:

instance (*Validateable* α , *Validateable* β) \Rightarrow *Validateable* ($\alpha \text{ :+} \beta$) **where**
record (*Inl*' e) = *record* e
record (*Inr*' e) = *record* e .

For :: , we update the record by traversing the subterms and the pattern in sequence:

instance (*Rewritable* α , *Validateable* β) \Rightarrow *Validateable* ($\alpha \text{ ::} \beta$) **where**
record ($\text{pat} \text{ ::}' es$) = *record*' $\text{pat} \circ \text{record}$ es .

Note that since the record is only used to check off metavariable use, the order of the calls to *record*' and *record* plays no rôle.

Next, we add a superclass constraint for *Validateable* to the declaration of the class *Rewritable* from Section 3.2,

class (\dots , *Validateable* (*Rep* α)) \Rightarrow *Rewritable* α ,

and define a top-level function for validating rules:

validate :: *Rewritable* $\alpha \Rightarrow$ *Rule* $\alpha \rightarrow$ *Bool*
validate (*Rule* $lhs\ rhs$) = *check* (*record*' $lhs\ blank$)
where
check *RNil* = *True*
check (*RCons* $b\ rec$) = $b \wedge \text{check } rec$.

Starting with a blank record, *validate* records all occurrences of metavariables at the left-hand side of a rewrite rule and then verifies that all metavariables in the environment of the rule are checked off in the updated record.

8 Guarded rewriting

In the previous section, we have added some infrastructure for statically validating rewrite rules to the core functionality of our library. In this section, we further extend the library and add support for rewrite rules guarded by *preconditions*.

As an example, consider the datatype *Lam* of lambda-expressions,

data *Lam* = *Var* *String* | *Abs* *String* *Lam* | *App* *Lam* *Lam*,

and an accompanying function *fv* that produces the variables that appear free in a given lambda-expression:

$fv :: Lam \rightarrow [String]$.

Now suppose that we want to define a rewrite rule that implements *eta-reduction*:

$\lambda x. e\ x \rightarrow e$, if x not free in e .

That is, eta-reduction applies to expressions that match the pattern $\lambda x. e\ x$, but only if such an expression additionally fulfills the precondition that the variable x does not

appear free in the expression e . Using the extension presented in this section, such rewrite rules can, with our library, be written as in

```
etaReduction :: Rule Lam
etaReduction = synthesise (\x e → Abs x (App e (Var x)) ↦ e § x ∉ fv e).
```

Here, we synthesise a rule over lambda-expressions from a function that produces a template constructed with the operators \mapsto and \S . The latter adds a *guard* to the rewrite rule, i.e., a boolean expression that may refer to the metavariables abstracted over by the synthesiser function.

In order to implement preconditions, we extend our type *Rule* of rewrite rules with a component containing a guard:

```
data Rule :: * → * where
  Rule :: (Mappable Γ, Recordable Γ, Testable Γ) ⇒
         Pattern α Γ → Pattern α Γ → Guard Γ → Rule α.
```

In addition to the classes *Mappable* (cf. Section 5) and *Recordable* (cf. Section 7), metavariable environments used within rules are restricted to be instances of the class *Testable*, to be explained below. Guard types are defined inductively over the structure of metavariable environments. That is, we have a type family *Guard*,

```
type family Guard Γ :: *
```

with instances

```
type instance Guard Nil      = Bool
type instance Guard (α :: Γ) = α → Guard Γ.
```

A guard for a rewrite rule without metavariables is just a boolean expression. For rules that do have metavariables, a guard is a function that takes an argument of appropriate type for each metavariable and produces a boolean.

Given a substitution for a metavariable environment Γ (cf. Section 5.3), values of type *Guard Γ* can be tested in order to obtain a boolean that indicates whether the corresponding precondition is fulfilled. To this end, we define the type class *Testable* of environments for which guards are testable:

```
class Testable Γ where
  test :: Guard Γ → Substitution Γ → Bool.
```

For the empty-environment type *Nil*, the guard is itself already a value of type *Bool*, so testing can just discard the supplied substitution (which can only be constructed by *PNil* anyway):

```
instance Testable Nil where
  test b PNil = b.
```

For an environment $\alpha :: \Gamma$, the guard function is applied to the value that is to be substituted for the metavariable corresponding to α and the resulting guard for Γ is tested recursively:


```

instance Testable  $\Gamma \Rightarrow$  Testable ( $\alpha :: \Gamma$ ) where
  test f (PCons (Just x) s) = test s (f x)
  test f (PCons Nothing s) = error "test failure".

```

If no substitution value is available, the governing rewrite rule was ill-formed (cf. Section 7) and testing fails with a run-time error.

As the GADT *Rule* now requires all metavariable environments to be testable, enforcing preconditions is straightforward:

```

rewriteWithM :: (Rewritable  $\alpha$ , Monad  $\mu$ )  $\Rightarrow$  Rule  $\alpha \rightarrow \alpha \rightarrow \mu \alpha$ 
rewriteWithM (Rule lhs rhs grd) x = do
  s  $\leftarrow$  match' lhs x
  if test grd s then substitute' s rhs else fail "precondition failure".

```

If, for a given rule *Rule* *lhs rhs grd* and term *x*, *x* successfully matches against the left-hand side *lhs*, the resulting substitution *s* is tested against the guard *grd*. If the test succeeds, the substitution *s* and the right-hand side *rhs* are combined to produce a new term; otherwise, the rule does not apply and rewriting fails.

What remains is to adapt the synthesis of rules from templates and functions producing templates (cf. Section 6). Firstly, we extend templates with a boolean component:

```

data Template  $\alpha$  = Template  $\alpha$  Bool.

```

Next, we redefine and introduce the smart constructors \mapsto and \S , respectively:

```

infix 1  $\mapsto$ 
infix 0  $\S$ 
( $\mapsto$ ) ::  $\alpha \rightarrow \alpha \rightarrow$  Template  $\alpha$ 
lhs  $\mapsto$  rhs = Template lhs rhs True
( $\S$ ) :: Template  $\alpha \rightarrow$  Bool  $\rightarrow$  Template  $\alpha$ 
Template lhs rhs b  $\S$  b' = Template lhs rhs b'.

```

The class *Synthesiser* now gets an additional method *guard* that produces, for a synthesised rule, a guard of appropriate type:

```

class ...  $\Rightarrow$  Synthesiser  $\alpha$  where
  ...
  guard ::  $\alpha \rightarrow$  Guard (Env  $\alpha$ )

```

For rules synthesised directly from templates, this guard is just the boolean from the template:

```

instance ...  $\Rightarrow$  Synthesiser (Template  $\alpha$ ) where
  ...
  guard (Template lhs rhs b) = b.

```

For rules synthesised from functions, the guard is itself a function too:

instance $\dots \Rightarrow \text{Synthesiser } (\alpha \rightarrow \beta)$ **where**
 \vdots
 $\text{guard } f = \text{guard} \circ f.$

The function *synthesise*, finally, that turns synthesisers into rewrite rules simply puts guards in the right places in rules:

$\text{synthesise} :: (\text{Synthesiser } \alpha, \text{Mappable } (\text{Env } \alpha), \text{Testable } (\text{Env } \alpha)) \Rightarrow \alpha \rightarrow \text{Rule } (\text{Term } \alpha)$
 $\text{synthesise } x = \text{let } (\text{lhs}, \text{rhs}) = \text{patterns } x \text{ in Rule lhs rhs (guard } x).$

Note that the given implementation of guarded rewrite rules has one obvious drawback: preconditions are encoded extensionally rather than intensionally and are therefore not observable. This reintroduces some of the problems mentioned in Section 2. Most prominently, when pretty-printing rewrite rules, the rendering of preconditions will pose a problem. The other issues listed in Section 2 are, however, of lesser importance. To what extent rewrite rules are still suitable for automated testing, strongly depends on how often preconditions apply: only if preconditions are rarely fulfilled, the generation of appropriate test data may be problematic. For inversion and tracing, nonobservability of preconditions plays no limiting rôle.

9 A case study: solving arithmetic equations

The previous section completed our exploration of our generic library for term rewriting. In this and the next section, we evaluate our approach. In this section, we present (part of) a small case study of a more or less realistic use of our library: solving arithmetic equations using term rewriting. In Section 10, we discuss some results obtained from benchmarking.

In the case study in this section, we use some of the more advanced features of our library, such as heterogeneously typed metavariables and guarded rewrite rules.

Consider the problem of solving the equation

$$1 + \frac{8}{(x-3)^2} = 3.$$

To solve such an equation with a single variable, we use the so-called *cover-up method*, which is based on covering up the part of the equation that contains the variable. We can define cover-up rewrite rules for addition, subtraction, multiplication, division, and exponentiation operations; with these rules we solve the example equation in the following sequence of steps:

$$\begin{aligned} & 1 + \frac{8}{(x-3)^2} = 3 \\ \Leftrightarrow & \frac{8}{(x-3)^2} = 2 \\ \Leftrightarrow & (x-3)^2 = 4 \\ \Leftrightarrow & x-3 = 2 \quad \vee \quad x-3 = -2 \\ \Leftrightarrow & x = 5 \quad \vee \quad x = 1. \end{aligned}$$

The domain of interest is represented by a variation of the datatype *Prop* from Section 1, that allows for formulae to be expressed over atoms of different types,

```
data Prop  $\alpha$  = Var  $\alpha$  | T | F | Not (Prop  $\alpha$ )
           | Prop  $\alpha$  : $\wedge$ : Prop  $\alpha$  | Prop  $\alpha$  : $\vee$ : Prop  $\alpha$ ,
```

a type of equations,

```
data Equation  $\alpha$  =  $\alpha$  : $\equiv$ :  $\alpha$ ,
```

and a type *Expr* of various arithmetic expressions,

```
data Expr = Const Rational | Varia String | Expr :+ Expr | Expr :- Expr
           | Expr :* Expr | Expr :/ Expr | Expr :^ Expr.
```

For each of these datatypes we need instances of the class *Representable* (as described in Section 3), the class *Typeable* (can be derived by the GHC), and *Rewritable* (one line).

Using the datatypes, the equation $1 + \frac{8}{(x-3)^2} = 3$ is represented as

```
Var ((Const 1 :+ (Const 8 :/ (Varia "x" :- Const 3) :^ Const 2))) : $\equiv$ : Const 3).
```

The solution to this equation, $x = 5 \vee x = -1$, is represented as

```
Var (Varia "x" : $\equiv$ : Const 5) : $\vee$ : Var (Varia "x" : $\equiv$ : Const (-1)).
```

Our rewrite system consists of simple rules for simplifying propositions, such as

```
orTrueLeft :: Rewritable  $\alpha$   $\Rightarrow$  Rule (Prop  $\alpha$ )
orTrueLeft = synthesise ( $\lambda p \rightarrow T$  : $\vee$ : p)
```

and some rules for rewriting additions, which require preconditions,

```
coverPlusLeft :: Rule (Equation Expr)
coverPlusLeft = synthesise ( $\lambda x y z \rightarrow$ 
  x :+ y : $\equiv$ : z  $\mapsto$  x : $\equiv$ : z :- y  $\S$  hasVaria x  $\wedge$  noVaria y).
```

In the rule *coverPlusLeft*, all metavariables range over expressions. We only want to apply this rule if there are variables in the expression x and no variables in the expression y , so as to guarantee the isolation of those variables on the left-hand side of the equation. The helper functions *hasVaria* and *noVaria* test the presence (or absence) of variables in an expression.

Dealing with exponentiation requires a more complex rule:

```
coverPowerEven :: Rule (Prop (Equation Expr))
coverPowerEven = synthesise ( $\lambda x n y \rightarrow$ 
  let z = y :^ Const (1 / n)
  in Var (x :^ Const n : $\equiv$ : y)  $\mapsto$  Var (x : $\equiv$ : z) : $\vee$ : Var (x : $\equiv$ : Const 0 :- z)
   $\S$  hasVaria x  $\wedge$  n > 0  $\wedge$  isEven n).
```

As this definition illustrates, complex rewrite rules can become quite verbose, but we can freely use local definitions to keep rules more or less readable. Since our rewrite rules are observable, a pretty-printer would be able to format such rules nicely. (Note, however, that guards in rewrite rules are not observable since these are just boolean values, as described earlier in Section 8.)

10 Benchmarks

The biggest disadvantage of generic programming techniques is that they can be a source of inefficiency. The introduction of representation types and corresponding conversions to and from the original datatypes generally imposes a penalty on execution time. We have measured the performance of our generic rewriting library to assess how large this penalty is, compared to hand-written code for a specific datatype. We have performed two separate tests of different complexities. The first one deals with logical propositions and uses neither preconditions nor metavariables of different types. The second one deals with arithmetic equations, and uses the full power of our generic rewriting library. Both are bundled with the library for analysis and repeatability.

10.1 Turning propositions into disjunctive normal form

Our first benchmark uses the datatype *Prop* of propositional formulae from the Introduction, extended with constructors for implication and equivalence. We have defined 16 rewrite rules and used these rules to bring the logical 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). None of the rules has preconditions, and all metavariables are of type *Prop*.

Conversion to DNF has been tested with four different strategies: such a strategy controls which rewrite rule is tried, and where. The strategies range from naïve (i.e., apply some rule somewhere), to more involved strategy specifications that stage the rewriting and use all kinds of traversal combinators. We implemented these combinators in a type-specific fashion. They could also be implemented generically, and not necessarily with the library we present. However, this would add another source of inefficiency to our tests, one that we do not wish to benchmark; hence our choice for implementing the strategies in a type-specific fashion.

We use QuickCheck (Claessen and Hughes, 2000) to generate a sequence of random propositions. The random-number generator is initiated with a fixed seed so that the same sequence is used for all test runs. We carefully profiled our tests to assure that the computation time was being spent mostly on the rewriting functionality, and not on auxiliary infrastructure such as data generation.

Because the strategy highly influences how many rules are tried, we vary the number of terms that has to be brought to disjunctive normal form depending on the strategy that is used. Table 1 shows for each strategy the number of terms that are normalised, how many rules are successfully applied, and the total number of rules that have been

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

Strategy	Terms	Rules applied	Rules tried	Ratio
dnf-1	10,000	217,076	113,511,244	0.19%
dnf-2	50,000	492,114	22,224,222	2.21%
dnf-3	50,000	487,490	22,467,730	2.17%
dnf-4	100,000	872,494	18,327,913	4.76%

Table 1: The strategies benchmarked.

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

Pattern Matching (PM): The first implementation defines the 16 rewrite rules as functions that use pattern matching. This implementation suffers from all the drawbacks that were mentioned in Section 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.

Specialised Rewriting (SR): We have also written a specialised rewriting system that operates on propositions, very much like that described in Section 2.2. The most significant difference is that we have reused the *Var* constructor for representing metavariables too, thus mixing object variables with metavariables.

Generic Rewriting (GR): Here we implemented the rules using the generic functions for rewriting that are introduced in this paper. The instance of the *Representable* type class is similar to the declaration in Section 3, except that it also includes the constructors for equivalence and implication.

All test runs were executed on a machine running Windows XP Professional x64 Edition with SP2 on an Intel Core 2 Duo 3Ghz with 2GB of RAM. The programs were compiled with the GHC (version 6.10.4) with standard optimisation level (using the `-O1` compiler flag). We do not use optimisation level `-O2` because we noticed that it sometimes reduced performance. Execution times were measured as the difference of the value returned by the function `System.CPUTime.getCPUTime` from the base libraries that ship with theGHC, after and before the execution of the test, and averaged over 10 runs.

Table 2 shows the performance for each implementation of the strategies. The absolute figures are given in seconds, and we also show the figures relative to the pattern-matching approach (PM).

The table shows that PM is significantly faster than the other approaches. The specialised rewriting approach (SR) adds observability of the rewrite rules, at the cost of approximately doubling execution time. The generic approach (GR), when compared

Strategy	Absolute (s)			Relative		
	PM	SR	GR	PM	SR	GR
dnf-1	3.11	10.89	37.21	1.00	3.49	11.94
dnf-2	2.52	4.82	15.03	1.00	1.92	5.98
dnf-3	2.49	4.87	15.45	1.00	1.95	6.19
dnf-4	3.94	7.28	19.45	1.00	1.84	4.93

Table 2: Benchmark results for the *Prop* datatype with `-O1`.

to the SR approach, suffers from a slowdown of a factor of about 3. This is probably due to the conversions to and from the structure representation of propositions. We also observe a correlation between strategy ratio of rule application (Table 1) and performance (the higher the ratio, the better the performance). This confirms that the overhead of both the SR and GR approaches is caused by the rewriting infrastructure: the PM approach has little overhead from trying rules as it uses Haskell’s native support for pattern matching.

Inspired by Magalhães et al. (2010), we repeated our benchmark setting compilation flags `-funfolding-creation-threshold` to 450 and `-funfolding-use-threshold` to 60. These flags control, respectively, the keenness of the compiler to export function definitions into interface files and to inline them. This has been shown to increase the performance of certain generic functions, since inlining “large” functions such as *to* and *from* exposes opportunities for further optimisations. We show the new results in Table 3. Note that the relative figures are still in relation to PM compiled with `-O1`, as this is the “standard” approach at the “standard” optimisation level.

Strategy	Absolute (s)			Relative		
	PM	SR	GR	PM	SR	GR
dnf-1	3.02	10.78	22.57	0.97	3.46	7.24
dnf-2	2.12	4.00	7.36	0.84	1.59	2.93
dnf-3	2.12	4.07	7.63	0.85	1.63	3.06
dnf-4	2.51	4.49	7.70	0.64	1.14	1.95

Table 3: Benchmark results for the *Prop* datatype with increased inlining.

Increased inlining effectively improves the performance. All the approaches benefit from it, but the most pronounced gains are seen in the GR approach, where performance is improved to between 40% and 60% of the original levels. Strategy dnf-4, in particular, shows the highest improvement, now taking only twice as much as the original PM approach.

10.2 Solving arithmetic equations

Our second benchmark is performed on a family of datatypes representing arithmetic equations, as introduced in Section 9. We use 25 rules, some with preconditions and some using metavariables of different types, therefore testing the full potential of our library in a realistic setting. These rules are applied to isolate variables on the left-hand sides of equations.

Again, we have used QuickCheck for test data generation. We test a single strategy, and use type-specific traversals for its application. We compare our library against a pattern-matching approach (PM) only, and again include figures with standard -O1 optimisation and with increased inlining as described previously. The results, as an average over 10 runs, are summarised in Table 4. We can conclude that the introduction of preconditions and metavariables of different types does not significantly influence performance. Promoting inlining continues to prove useful to increase the performance of our library.

Optimisation	Absolute (s)		Relative	
	PM	GR	PM	GR
Standard	0.57	2.44	1.00	4.29
Increased inlining	0.60	1.87	1.06	3.30

Table 4: Benchmark results for solving arithmetic equations.

Our benchmarks 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 specialising generic definitions, and by applying partial-evaluation techniques, we expect to get code that is more competitive to the hand-written definitions for a specific datatype.

11 Related Work

Jansson and Jeuring (2000) implement a generic rewriting library in PolyP (Jansson and Jeuring, 1997), an extension of Haskell with a special construct for generic programming. Our library differs in a number of aspects. First, we use no extensions of Haskell specific to generic programming. 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 datatype 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 a new constructor, or one of the constructors of the original datatype

is to be reused for metavariables. Neither solution is very satisfying, since either functions unrelated to rewriting must now handle the new metavariable constructor, or we are forced to introduce a safety problem in the library since an object variable may accidentally be considered a metavariable.

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), Bringert’s “almost compositional” functions (Bringert and Ranta, 2006), and probably more, can be used to define extensionally represented rewrite rules. These suffer from the disadvantages described in Section 2, but typically perform better than intensionally represented rules (see Section 10).

Our generic pattern-matching function is a variation on the generic unification functions of Jansson and Jearing (1998) and Sheard (2001). A generalisation of our library to full unification is possible, but probably hard to keep user-friendly as unification results may contain metavariable occurrences that can then no longer be hidden from the user. Adapting our library to use mutable variables to improve performance, as in Sheard’s work, should be relatively straightforward.

Brown and Sampson (2008) implement generic rewriting using the *Scrap Your Boilerplate*-library. Patterns 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 ill-typed rules are only detected at runtime.

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), *ASF+SDF* (Van Deursen et al., 1996), and *Stratego* (Bravenboer et al., 2008). 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.

12 Conclusions and Further Work

We have presented a library for datatype-generic term rewriting. Our library overcomes problems in previous generic rewriting libraries: users do not have to adapt or manually extend the datatypes that are used to represent terms; they do not need knowledge of the internals of the library; and they can document, test, and analyse their rewrite rules. The performance of our library is not as good as that of hand-written, datatype-specific rewrite functions, but we think the loss of performance is acceptable for many applications.

In contrast to rewrite rules that are defined using an extensional representation, our library requires that rule synthesisers do not inspect their metavariable arguments. Concretely, we do not allow arbitrary function applications in the right-hand side of a rule template, but unfortunately this restriction cannot be enforced statically.

There is ongoing work on generating test data for rewrite rules generically. That is, the left-hand side of a rewrite rule can be used as a template for test-data generation to improve testing coverage. We plan to use this approach in a testing framework that is to be shipped with our library.

Acknowledgements

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), 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), and by the Portuguese Foundation for Science and Technology (FCT) via the SFRH/BD/35999/-2007 it was carried out while the second and third author were employed at Utrecht University.

The authors would like to thank Chris Eidhof and Sebastiaan Visser for their work on testing rewrite rules using generic test-data generation and Andres Löh for productive discussions on this work. Finally, the authors are indebted to Doaitse Swierstra and the anonymous reviewers of the 2008 Workshop on Generic Programming and the Journal of Functional Programming for their useful suggestions.

References

- R. C. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In S. D. Swierstra, P. Rangel Henriques, and J. N. Oliveira, editors, *Advanced Functional Programming, Third International School, Braga, Portugal, September 12–19, 1998, Revised Lectures*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115. Springer-Verlag, 1999.
- P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–95, 2001.
- M. Bravenboer, K. T. Kalleberg, and Visser E. Stratego/XT 0.17: A language and toolset for program transformation. *Science of Computer Programming*, 72(1–2): 52–70, 2008.
- B. Bringert and A. Ranta. A pattern for almost compositional functions. In J. H. Reppy and J. 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.
- N. C. C. Brown and A. T. Sampson. Matching and modifying with generics. In P. Achten, P. Koopman, and M. 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.

- N. G. de Bruijn. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the Church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.
- M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In O. Danvy and B. 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, 2005a.
- M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12–14, 2005*, pages 1–13. ACM Press, 2005b.
- K. Claessen and J. 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.
- A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore, 1996.
- J. Goguen and M. Grant. *Algebraic Semantics of Imperative Programs*. The MIT Press, Cambridge, Massachusetts, 1997.
- B. Heeren, J. Jeuring, A. van Leeuwen, and A. Gerdes. Specifying strategies for exercises. In S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki, and F. Wiedijk, editors, *Intelligent Computer Mathematics, 9th International Conference, AISC 2008, 15th Symposium, Calculemus 2008, 7th International Conference, MKM 2008, Birmingham, UK, July 28–August 1, 2008, Proceedings*, volume 5144 of *Lecture Notes in Computer Science*, pages 430–445. Springer-Verlag, 2008.
- R. Hinze. Generic programs and proofs, 2000. Habilitationsschrift, University of Bonn.
- R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. *Science of Computer Programming*, 51(2):117–151, 2004.
- S. Holdermans, J. Jeuring, A. Löh, and A. Rodriguez. Generic views on data types. In T. Uustalu, editor, *Mathematics 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.
- P. Jansson and J. Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In J. 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.

- P. Jansson and J. Jeuring. PolyP: A polytypic programming language. In *Conference Record of POPL'97: The 24 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15–17 January 1997*, pages 68–114. ACM Press, 1997.
- P. Jansson and J. Jeuring. Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, 1998.
- R. Lämmel and S. 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.
- R. Lämmel and J. Visser. Typed combinators for generic traversal. In S. 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.
- José Pedro Magalhães, Stefan Holdermans, Johan Jeuring, and Andres Löf. Optimizing generics is easy! In John P. Gallagher and Janis Voigtländer, editors, *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010, Madrid, Spain, January 18–19, 2010*, pages 33–42. ACM Press, 2010.
- N. Mitchell and C. Runciman. Uniform boilerplate and list processing. In G. Keller, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, pages 49–60. ACM Press, 2007.
- T. van Noort, A. Rodriguez, S. Holdermans, J. Jeuring, and B. Heeren. A lightweight approach to datatype-generic rewriting. In R. Hinze and D. Syme, editors, *Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP 2008, Victoria, BC, Canada, September 20, 2008*, pages 13–24. ACM Press, 2008.
- E. Pasálic and N. Linger. Meta-programming with typed object-language representations. In G. Karsai and E. Visser, editors, *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24–28, 2004, Proceedings*, volume 3286 of *Lecture Notes in Computer Science*, pages 136–167. Springer-Verlag, 2004.
- S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In J. H. Reppy and J. L. Lawall, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16–21, 2006*, pages 50–61. ACM Press, 2006.
- T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Type checking with open type functions. In J. Hook and P. Thiemann, editors, *Proceedings of the 13th*

- ACM SIGPLAN International Conference on Functional Programming, ICFP 2008, Victoria, BC, Canada, September 22–24, 2008*, pages 51–62. ACM Press, 2008.
- T. 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.
- T. Sheard and S. 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.
- H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisiana, January 15–17, 2003*, pages 224–235. ACM Press, 2003.