A lightweight approach to datatype-generic rewriting

THOMAS VAN NOORT

Institute for Computing and Information Sciences, Radboud University Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
(*e-mail:* `thomas@cs.ru.nl`)

ALEXEY RODRIGUEZ YAKUSHEV
and STEFAN HOLDERMANS

Vector Fabrics, Paradijslaan 28, 5611 KN Eindhoven, The Netherlands
(*e-mail:* `{alexey,stefan}@vectorfabrics.com`)

JOHAN JEURING

Department of Information and Computing Sciences, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
and
School of Computer Science, Open University of the Netherlands,
P.O. Box 2960, 6401 DL Heerlen, The Netherlands
(*e-mail:* `johanj@cs.uu.nl`)

BASTIAAN HEEREN

School of Computer Science, Open University of the Netherlands,
P.O. Box 2960, 6401 DL Heerlen, The Netherlands
(*e-mail:* `bastiaan.heeren@ou.nl`)

JOSÉ PEDRO MAGALHÃES

Department of Information and Computing Sciences, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
(*e-mail:* `jpm@cs.uu.nl`)

Abstract

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 analyze rewrite rules. In this article, 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*:

$$p \wedge \neg p \to \bot.$$

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. Second, 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 analyzed 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 for 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 article, 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 ↦, 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 article 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.* 2005a, 2005b; 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 article is based on a paper presented at the 2008 Workshop on Generic Programming (Van Noort *et al.* 2008). The present article 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 article 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

---

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

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.

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

$$deMorgan :: \mathsf{Prop} \rightarrow \mathsf{Prop}$$
$$deMorgan \ (Not \ (p :\wedge: q)) = Not \ p :\vee: Not \ q$$
$$deMorgan \ p \qquad\qquad = p.$$

Note that the last line ensures that prevents arguments that do not match the pattern $\neg(p \wedge q)$ from causing 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

$$excludedMiddle :: \mathsf{Prop} \rightarrow \mathsf{Prop}$$
$$excludedMiddle \ (p :\vee: Not \ q) \mid p \equiv q = T$$
$$excludedMiddle \ p \qquad\qquad = p$$

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.

$$excludedMiddleM \;:: \mathsf{Prop} \rightarrow \mathsf{Maybe\ Prop}$$
$$excludedMiddleM \;(p :\vee: Not\ q)\mid p \equiv q = Just\ T$$
$$excludedMiddleM \;p \qquad\qquad\quad = Nothing.$$

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 & Runciman 2007) provides, among others, the combinator *transform*

$$transform \;:: \mathsf{Uniplate}\ \alpha \Rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$$

which applies its argument function in a bottom-up fashion to 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:

$$removeTautologies \;:: \mathsf{Prop} \rightarrow \mathsf{Prop}$$
$$removeTautologies = transform\ excludedMiddle.$$

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:

$$\textbf{data } \mathsf{Rule} \; \alpha = \mathit{Rule} \; \{ \mathit{lhs} :: \alpha, \mathit{rhs} :: \alpha \}.$$

Values of type $\mathsf{Rule} \; \alpha$ 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 $\mathsf{Rule} \; \mathsf{EProp}$, 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:

$$\textbf{data } \mathsf{EProp} = \mathit{EVar} \; \mathsf{String} \mid \mathit{ET} \mid \mathit{EF} \mid \mathit{ENot} \; \mathsf{EProp} \mid \mathsf{EProp} :\triangle: \mathsf{EProp}$$
$$\mid \mathsf{EProp} :\triangledown: \mathsf{EProp} \mid \mathit{Metavar} \; \mathsf{String}.$$

With values of type $\mathsf{Rule} \; \mathsf{EProp}$ in place, we need to define rewrite functions that interpret these values as functions over propositions represented by Prop:

$$\mathit{rewritePropWith} :: \mathsf{Rule} \; \mathsf{EProp} \to \mathsf{Prop} \to \mathsf{Prop}.$$

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

$\mathit{removeTautologies} :: \mathsf{Prop} \to \mathsf{Prop}$
$\mathit{removeTautologies} = \mathit{transform} \; (\mathit{rewritePropWith} \; \mathit{excludedMiddle})$
   where
     $\mathit{excludedMiddle} = \mathit{Rule} \; \{ \mathit{lhs} = \mathit{Metavar} \; \texttt{"p"} :\triangledown: \mathit{ENot} \; (\mathit{Metavar} \; \texttt{"p"}), \mathit{rhs} = \mathit{ET} \}.$

An apparent inconvenience of this style of defining rules is that we cannot reuse the type Prop of terms being rewritten and its constructors *Not* and :∨: . Instead, to provision for metavariables, we have 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*:

$$
\begin{aligned}
&closedWorldTemplate \quad :: \text{Template Prop} \\
&contradictionTemplate :: \text{Prop} \to \text{Template Prop} \\
&deMorganTemplate \qquad :: \text{Prop} \to \text{Prop} \to \text{Template Prop} \\
\\
&closedWorldTemplate \qquad = Not\ T \mapsto F \\
&contradictionTemplate\ p = p :\wedge: Not\ p \mapsto F \\
&deMorganTemplate\ p\ q \ = Not\ (p :\wedge: q) \mapsto Not\ p :\vee: Not\ q.
\end{aligned}
$$

Templates are constructed by means of an operator $\mapsto$,

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

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:

$$
\begin{aligned}
&closedWorld, contradiction, deMorgan :: \text{Rule Prop} \\
\\
&closedWorld \ = synthesise\ closedWorldTemplate \\
&contradiction = synthesise\ contradictionTemplate \\
&deMorgan \quad\ = synthesise\ deMorganTemplate.
\end{aligned}
$$

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

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

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, we have that

> *rewriteWith closedWorld* (*Not T*)   yields   *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,

> *rewriteWith contradiction* (*Var* "x" $:\wedge:$ *Not* (*Var* "y"))
>
> yields   *Var* "x" $:\wedge:$ *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,

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

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

> *rewriteWithM deMorgan* (*T* $:\wedge:$ *F*)   yields   *Nothing*,

while

> *rewriteWithM deMorgan* (*Not* (*T* $:\wedge:$ *F*))   yields   *Just* (*Not T* $:\vee:$ *Not F*).

As with other lightweight approaches to generic rewriting, such as Scrap Your Boilerplate (Lämmel & Peyton Jones 2003) and Uniplate (Mitchell & 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 α **where**
>   **type** Rep α :: $\star$
>   *from* :: $\alpha \rightarrow$ Rep α
>   *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:

$$\textbf{infixr } 6 \text{ :+:}$$
$$\textbf{infixr } 5 \text{ :::}$$
$$\textbf{data } \text{Nil} \quad = Nil$$
$$\textbf{data } \alpha \text{ :+: } \beta = Inl\ \alpha \mid Inr\ \beta$$
$$\textbf{data } \alpha \text{ ::: } \beta = \alpha ::: \beta.$$

A given datatype's representation type follows immediately from its structure. Choice among 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 & 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

$$\textbf{data } \text{Maybe } \alpha = Nothing \mid Just\ \alpha$$

is represented by the type Nil :+: (α ::: Nil) and we can write

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

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 ∘ from = id* and *from ∘ to ⊑ id* (Hinze 2000).

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

$$\textbf{data } [\alpha] = [\ ] \mid \alpha : [\alpha]$$

is in our approach represented by Nil :+: (α ::: [α] ::: Nil), yielding the declaration

```
instance Representable [α] where
    type Rep [α] = Nil :+: (α ::: [α] ::: Nil)

    from []       = Inl Nil
    from (x : xs) = Inr (x ::: xs ::: Nil)

    to (Inl Nil)              = []
    to (Inr (x ::: xs ::: Nil)) = x : xs.
```

Note that the generic representation types of recursive datatypes are themselves nonrecursive: *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,

$$\textbf{data}\ \mathsf{Prop} = \mathit{Var}\ \mathsf{String} \mid T \mid F \mid \mathit{Not}\ \mathsf{Prop} \mid \mathsf{Prop} :\wedge: \mathsf{Prop} \mid \mathsf{Prop} :\vee: \mathsf{Prop}$$

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 & 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 α, Typeable α,
   Eq (Rep α), Extensible (Rep α), Matchable (Rep α), Substitutable (Rep α),
   Sampleable (Rep α), Diffable (Rep α)) ⇒
  Rewritable α.

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 & 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 GHC. 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 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:

$$\textbf{class } \mathsf{Eq}\ \alpha\ \textbf{where}$$
$$(\equiv), (\not\equiv) :: \alpha \to \alpha \to \mathsf{Bool}$$
$$x \equiv y = \neg\ (x \not\equiv y)$$
$$x \not\equiv y = \neg\ (x \equiv y).$$

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:

$$(\equiv') :: \mathsf{Rewritable}\ \alpha \Rightarrow \alpha \to \alpha \to \mathsf{Bool}$$
$$x \equiv' y = \textit{from } x \equiv \textit{from } y.$$

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:

$$\textbf{instance } \mathsf{Eq}\ \mathsf{Nil}\ \textbf{where}$$
$$\textit{Nil} \equiv \textit{Nil} = \textit{True}.$$

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*:

$$\textbf{instance } (\mathsf{Eq}\ \alpha, \mathsf{Eq}\ \beta) \Rightarrow \mathsf{Eq}\ (\alpha \mathbin{:+:} \beta)\ \textbf{where}$$
$$\textit{Inl } x \equiv \textit{Inl } y = x \equiv y$$
$$\textit{Inr } u \equiv \textit{Inr } v = u \equiv v$$
$$\_ \quad\ \equiv \_ \quad\ = \textit{False}.$$

In the case for ::: we make use of that, 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:

$$\textbf{instance } (\mathsf{Rewritable}\ \alpha, \mathsf{Eq}\ \beta) \Rightarrow \mathsf{Eq}\ (\alpha \mathbin{:::} \beta)\ \textbf{where}$$
$$(x \mathbin{:::} xs) \equiv (y \mathbin{:::} ys) = x \equiv' y \land xs \equiv ys.$$

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'*,

$$match' \quad :: (\mathsf{Rewritable}\ \alpha, \mathsf{Mappable}\ \Gamma, \mathsf{Monad}\ \mu) \Rightarrow$$
$$\mathsf{Pattern}\ \Gamma\ \alpha \to \alpha \to \mu\ (\mathsf{Substitution}\ \Gamma)$$
$$substitute' :: (\mathsf{Rewritable}\ \alpha, \mathsf{Monad}\ \mu) \Rightarrow$$
$$\mathsf{Substitution}\ \Gamma \to \mathsf{Pattern}\ \Gamma\ \alpha \to \mu\ \alpha.$$

The type Pattern Γ α (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 :: $\star \to \star$ **where**
    Rule :: Mappable $\Gamma \Rightarrow$ Pattern Γ $\alpha \to$ Pattern Γ $\alpha \to$ Rule α.

Here, the existential type Rule is defined using the syntax of a so-called *generalised algebraic datatypes* or GADTs (Xi *et al.* 2003; Peyton Jones *et al.* 2006).

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

rewriteWithM :: (Rewritable α, Monad μ) $\Rightarrow$ Rule $\alpha \to \alpha \to \mu\ \alpha$
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:

$$rewriteWith :: \mathsf{Rewritable}\ \alpha \Rightarrow \mathsf{Rule}\ \alpha \to \alpha \to \alpha$$
$$rewriteWith\ rule\ x = \mathbf{case}\ rewriteWithM\ rule\ x\ \mathbf{of}$$
$$\qquad Nothing \to x$$
$$\qquad Just\ y \quad \to y.$$

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 Pasălić & Linger (2004) and implement metavariables as values of the GADT Ref of typed references:

$$\mathbf{data}\ \mathsf{Ref} :: \star \to \star \to \star\ \mathbf{where}$$
$$RZero :: \mathsf{Ref}\ (\alpha ::: \Gamma)\ \alpha$$
$$RSucc :: \mathsf{Ref}\ \Gamma\ \alpha \to \mathsf{Ref}\ (\beta ::: \Gamma)\ \alpha.$$

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 Γ α 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:

$$deref :: \mathsf{Ref}\ \Gamma\ \alpha \to \Gamma \to \alpha$$
$$deref\ RZero\ (x ::: xs) \qquad = x$$
$$deref\ (RSucc\ r)\ (x ::: xs) = deref\ r\ xs.$$

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

$$\mathbf{data}\ \mathsf{PMap} :: \star \to \star\ \mathbf{where}$$
$$PNil \quad :: \mathsf{PMap}\ \mathsf{Nil}$$
$$PCons :: \mathsf{Rewritable}\ \alpha \Rightarrow \mathsf{Maybe}\ \alpha \to \mathsf{PMap}\ \Gamma \to \mathsf{PMap}\ (\alpha ::: \Gamma).$$

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*,

$$lookup :: \mathsf{Monad}\ \mu \Rightarrow \mathsf{Ref}\ \Gamma\ \alpha \to \mathsf{PMap}\ \Gamma \to \mu\ \alpha$$
$$lookup\ RZero\ (PCons\ Nothing\ s) = fail\ \texttt{"unbound variable"}$$
$$lookup\ RZero\ (PCons\ (Just\ x)\ s) = return\ x$$
$$lookup\ (RSucc\ r)\ (PCons\ mb\ s) \quad = lookup\ r\ s$$

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 Γ **where**
>   *empty* :: PMap Γ
>
> **instance** Mappable Nil **where**
>   *empty* = *PNil*
>
> **instance** (Rewritable α, Mappable Γ) $\Rightarrow$ Mappable (α ::: Γ) **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:

> *update* :: Ref Γ α $\rightarrow$ α $\rightarrow$ PMap Γ $\rightarrow$ PMap Γ
> *update RZero x* (*PCons mb s*)   = *PCons* (*Just x*) *s*
> *update* (*RSucc r*) *x* (*PCons mb s*) = *PCons mb* (*update r x s*).

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

> *singleton* :: (Rewritable α, Mappable Γ) $\Rightarrow$ Ref Γ α $\rightarrow$ α $\rightarrow$ PMap Γ
> *singleton r x* = *update r x empty*.

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

> ($\oplus$) :: Monad μ $\Rightarrow$ PMap Γ $\rightarrow$ PMap Γ $\rightarrow$ μ (PMap Γ)
> *PNil* $\oplus$ *PNil*   = *return PNil*
> *PCons Nothing s* $\oplus$ *PCons Nothing s'* = *liftM* (*PCons Nothing*) (*s* $\oplus$ *s'*)
> *PCons Nothing s* $\oplus$ *PCons* (*Just y*) *s'* = *liftM* (*PCons* (*Just y*)) (*s* $\oplus$ *s'*)
> *PCons* (*Just x*) *s* $\oplus$ *PCons Nothing s'* = *liftM* (*PCons* (*Just x*)) (*s* $\oplus$ *s'*)
> *PCons* (*Just x*) *s* $\oplus$ *PCons* (*Just y*) *s'*
> | *x* $\equiv'$ *y*   = *liftM* (*PCons* (*Just x*)) (*s* $\oplus$ *s'*)
> | *otherwise*   = *fail* "merging failed".

Here, *liftM*,

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

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 Pattern Γ α, where α is the type of terms to be rewritten and Γ is a metavariable environment. The idea is to derive the definition of Pattern Γ α 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,

$$\textbf{data family } \text{Extended } \alpha :: \star \rightarrow \star$$

with index α. A type Extended α Γ 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:

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

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

$$\textbf{newtype instance } \text{Extended Int } \Gamma = \textit{Int}' \text{ Int}.$$

The cases for Float and Char are analogous; in the sequel, we provide instance declarations for Int as representatives for all base types.

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:

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

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,

$$\textbf{class } \text{Extensible } \alpha \; \textbf{where}$$
$$\textit{extend} \; :: \alpha \rightarrow \text{Extended } \alpha \text{ Nil}$$

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

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

Note that a value of a type Pattern α Nil, due to the empty metavariable environment, is guaranteed to not contain any metavariables.

Lifting base types reduces to wrapping values in extension constructors:

$$\mathbf{instance}\ \mathsf{Extensible}\ \mathsf{Int}\ \mathbf{where}\ extend = Int'.$$

Extension of the empty list involves converting from *Nil* to *Nil′*

$$\mathbf{instance}\ \mathsf{Extensible}\ \mathsf{Nil}\ \mathbf{where}$$
$$extend\ Nil = Nil'$$

while sums are extended recursively:

$$\mathbf{instance}\ (\mathsf{Extensible}\ \alpha, \mathsf{Extensible}\ \beta) \Rightarrow \mathsf{Extensible}\ (\alpha\ \text{:+:}\ \beta)\ \mathbf{where}$$
$$extend\ (Inl\ x) = Inl'\ (extend\ x)$$
$$extend\ (Inr\ y) = Inr'\ (extend\ y).$$

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

$$\mathbf{instance}\ (\mathsf{Rewritable}\ \alpha, \mathsf{Extensible}\ \beta) \Rightarrow \mathsf{Extensible}\ (\alpha\ \text{:::}\ \beta)\ \mathbf{where}$$
$$extend\ (x ::: xs) = extend'\ x\ :::'\ extend\ xs.$$

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:

$$\mathbf{type}\ \mathsf{Substitution}\ \Gamma = \mathsf{PMap}\ \Gamma.$$

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:

$$\mathbf{class}\ \mathsf{Substitutable}\ \alpha\ \mathbf{where}$$
$$substitute :: \mathsf{Monad}\ \mu \Rightarrow \mathsf{Substitution}\ \Gamma \rightarrow \mathsf{Extended}\ \alpha\ \Gamma \rightarrow \mu\ \alpha.$$

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:

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

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

instance Substitutable Int **where** *substitute s* (*Int′ n*) = *return n*.

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 α, Substitutable β) $\Rightarrow$ Substitutable (α :+: β) **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 α, Substitutable β) $\Rightarrow$ Substitutable (α ::: β) **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*,

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

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 consists of a class Matchable of representation types, which can be matched against their recursively extended counterparts,

class Matchable α **where**
match :: (Mappable Γ, Monad μ) $\Rightarrow$ Extended α Γ $\rightarrow$ α $\rightarrow$ μ (Substitution Γ)

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

match′ :: (Rewritable α, Mappable Γ, Monad μ) $\Rightarrow$
Pattern Γ α $\rightarrow$ α $\rightarrow$ μ (Substitution Γ)
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* $(Int'\ n)\ n'$
> $|\ n \equiv n'$ $=\ return\ empty$
> $|\ otherwise = fail$ `"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) $\Rightarrow$ 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) $\Rightarrow$ Matchable (α ::: β) **where**
>   *match* $(pat\ :::'\ es)\ (x\ :::\ xs) = join\ (liftM2\ (\oplus)\ (match'\ pat\ x)\ (match\ es\ xs))$.

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*,

$$join :: \ \text{Monad}\ \mu \Rightarrow \mu\ (\mu\ \alpha) \Rightarrow \mu\ \alpha$$

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 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. She could write the rewrite rule derived from the principle of contradiction, for example, as

> *contradiction* :: Rule Prop
> *contradiction* = *Rule lhs rhs*
>   **where**
>   *lhs* = *Inl* (*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 tedious and, moreover, 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 allows for rewrite rules to be defined in terms of the real constructors of the type of terms that are to be rewritten. A library function *synthesise* then takes care of translating the terms in rewrite rules into generic representations. The rule above, for example, can conveniently and concisely be written as

> *contradiction* :: Rule Prop
> *contradiction* = *synthesise* (λ*p* → *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 – and compare the resulting values. This approach restricts us to function parameters of types that have at least two values; but note that this restriction is by no means essential as rules with metavariables that range over types that have only one value are not meaningful.

For the function λ*p* → *p* :∧: *Not p* ↦ *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* :∧: *Not T* and the right-hand *F*; the second instantiation yields *F* :∧: *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 :∧: 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,

$$\textbf{class } \textsf{Sampleable } \alpha \textbf{ where}$$
$$left \ :: \alpha$$
$$right :: \alpha.$$

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:

$$left', right' :: \textsf{Rewritable } \alpha \Rightarrow \alpha$$
$$left' \ = to \ left$$
$$right' = to \ right.$$

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

$$\textbf{instance } \textsf{Sampleable Int } \textbf{where } left = 0; right = 1.$$

For Nil, it is not possible to produce distinct *left* and *right* values. Still, we have to provide an instance declaration to allow for types that contain Nil-values to be in Sampleable:

$$\textbf{instance } \textsf{Sampleable Nil } \textbf{where}$$
$$left \ \ = Nil$$
$$right = Nil.$$

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 are easy: here we have 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:[2]

---

[2] There is a minor caveat associated with the given Sampleable-declaration for sum types. Our library requires the values for *left* and *right* 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. Note that this may require an implicit reordering of constructors when defining or generating generic representations and precludes types that have no finite values. Then, we can use a single *left*-produced value in the definitions of *left* and *right* for :+: as the top-level constructors *Inl* and *Inr* already distinguish the values.

instance (Sampleable α, Sampleable β) $\Rightarrow$ Sampleable (α :+: β) **where**
   left   = Inl left
   right = Inr left.

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 α, Sampleable β) $\Rightarrow$ Sampleable (α ::: β) **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*,

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

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 $\Gamma\ \alpha$ into values of type Pattern $\Gamma\ \beta$ with both α and β in Typeable.

We define a class Diffable of representation types for which a diff can be computed:

class Diffable α **where**
  diff :: Typeable β $\Rightarrow$
      Extended $\alpha\ \Gamma$ $\rightarrow$ Extended $\alpha\ \Gamma$ $\rightarrow$ Maybe (Extended α (β ::: Γ)).

For each generic representation type α, the overloaded function *diff* takes two values of type Extended $\alpha\ \Gamma$ 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*. Note that values generated by *left* and *right* for α can only ever differ at top level.

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

   diff′ :: (Rewritable α, Typeable β) $\Rightarrow$
      Pattern $\Gamma\ \alpha$ $\rightarrow$ Pattern $\Gamma\ \alpha$ $\rightarrow$ Maybe (Pattern (β ::: Γ) α)
   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'. 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 *gcast* 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 $\beta ::: \Gamma$. 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:

$$\textbf{instance } \textsf{Diffable Int } \textbf{where}$$
$$\textit{diff } (\textit{Int}' \, n) \, (\textit{Int}' \, n')$$
$$| \; n \equiv n' \quad = \textit{Just } (\textit{Int}' \, n)$$
$$| \; \textit{otherwise} = \textit{Nothing}.$$

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

$$\textbf{instance } \textsf{Diffable Nil } \textbf{where}$$
$$\textit{diff } \textit{Nil}' \, \textit{Nil}' = \textit{Just } \textit{Nil}'.$$

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

$$\textbf{instance } (\textsf{Diffable } \alpha, \textsf{Diffable } \beta) \Rightarrow \textsf{Diffable } (\alpha \; \text{:+:} \; \beta) \; \textbf{where}$$
$$\textit{diff } (\textit{Inl}' \, e) \, (\textit{Inl}' \, e') = \textbf{case } \textit{diff } e \, e' \textbf{ of } \textit{Nothing} \rightarrow \textit{Nothing} \; ; \; \textit{Just } e'' \rightarrow \textit{Just } (\textit{Inl}' \, e'')$$
$$\textit{diff } (\textit{Inr}' \, e) \, (\textit{Inr}' \, e') = \textbf{case } \textit{diff } e \, e' \textbf{ of } \textit{Nothing} \rightarrow \textit{Nothing} \; ; \; \textit{Just } e'' \rightarrow \textit{Just } (\textit{Inr}' \, e'')$$
$$\textit{diff } \_ \_ \qquad\qquad = \textit{Nothing}.$$

Similarly, for $:::$, the comparison of two values $\textit{pat} :::' \textit{es}$ and $\textit{pat}' :::' \textit{es}'$ continues recursively underneath the constructor $:::'$:

$$\textbf{instance } (\textsf{Rewritable } \alpha, \textsf{Diffable } \beta) \Rightarrow \textsf{Diffable } (\alpha ::: \beta) \; \textbf{where}$$
$$\textit{diff } (\textit{pat} :::' \textit{es}) \, (\textit{pat}' :::' \textit{es}') =$$
$$\textbf{case } (\textit{diff}' \, \textit{pat} \, \textit{pat}', \textit{diff } \textit{es} \, \textit{es}') \textbf{ of}$$
$$(\textit{Just } \textit{pat}'', \textit{Just } \textit{es}'') \rightarrow \textit{Just } (\textit{pat}'' :::' \textit{es}'')$$
$$\_ \qquad\qquad\qquad \rightarrow \textit{Nothing}.$$

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,

$$\textbf{data } \textsf{Template } \alpha = \textit{Template } \alpha \; \alpha$$

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

$$
\begin{aligned}
&\textbf{infix } 1 \mapsto \\
&(\mapsto) :: \alpha \to \alpha \to \mathsf{Template}\ \alpha \\
&\mathit{lhs} \mapsto \mathit{rhs} = \mathit{Template\ lhs\ rhs}.
\end{aligned}
$$

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

$$
\begin{aligned}
&\textbf{class } \mathsf{Rewritable\ (Term}\ \alpha) \Rightarrow \mathsf{Synthesiser}\ \alpha\ \textbf{where} \\
&\quad \textbf{type } \mathsf{Term}\ \alpha :: \star \\
&\quad \textbf{type } \mathsf{Env}\ \alpha \quad :: \star \\
&\quad \mathit{patterns} :: \alpha \to (\mathsf{Pattern\ (Env}\ \alpha)\ (\mathsf{Term}\ \alpha), \mathsf{Pattern\ (Env}\ \alpha)\ (\mathsf{Term}\ \alpha)).
\end{aligned}
$$

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 $\alpha \to \beta \to$ Template γ 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:

$$
\begin{aligned}
&\mathit{synthesise} :: (\mathsf{Synthesiser}\ \alpha, \mathsf{Mappable\ (Env}\ \alpha)) \Rightarrow \alpha \to \mathsf{Rule\ (Term}\ \alpha) \\
&\mathit{synthesise}\ x = \textbf{let}\ (\mathit{lhs}, \mathit{rhs}) = \mathit{patterns}\ x\ \textbf{in}\ \mathit{Rule\ lhs\ rhs}.
\end{aligned}
$$

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:

$$
\begin{aligned}
&\textbf{instance } \mathsf{Rewritable}\ \alpha \Rightarrow \mathsf{Synthesiser\ (Template}\ \alpha)\ \textbf{where} \\
&\quad \textbf{type } \mathsf{Term\ (Template}\ \alpha) = \alpha \\
&\quad \textbf{type } \mathsf{Env\ (Template}\ \alpha) \quad = \mathsf{Nil} \\
&\quad \mathit{patterns}\ (\mathit{Template\ lhs\ rhs}) = (\mathit{extend'\ lhs}, \mathit{extend'\ rhs}).
\end{aligned}
$$

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 Nil α 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 $\alpha \to \beta$ to be in the class Synthesiser, α to be a type of rewritable terms and β to be a type of

synthesisers:

```
instance (Rewritable α, Synthesiser β) ⇒ Synthesiser (α → β) where
    type Term (α → β) = Term β
    type Env (α → β)  = α ::: Env β

    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'') → (lhs'', rhs'')
            _                        → 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 to the value produced by *left'* and once to 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 domain-specific types. Due to this sugarcoating, additional verification of rewrite rules is required.

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

$$funny :: \text{Rule Prop}$$
$$funny = synthesise\ (\lambda n \to f\ n \mapsto T)$$

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

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

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

$$unbound :: \text{Rule Prop}$$
$$unbound = synthesise\ (\lambda p \to T \mapsto T :\vee: p)$$

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

$$superfluous :: \mathsf{Rule\ Prop}$$
$$superfluous = synthesise\ (\lambda p\ q \rightarrow p : \vee : p \mapsto p)$$

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:

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

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** $\mathsf{Record} :: \star \rightarrow \star$ **where**
>   *RNil* $:: \mathsf{Record\ Nil}$
>   *RCons* $:: \mathsf{Bool} \rightarrow \mathsf{Record}\ \Gamma \rightarrow \mathsf{Record}\ (\alpha ::: \Gamma).$

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

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

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

> **data** $\mathsf{Rule} :: \star \rightarrow \star$ **where**
>   *Rule* $:: (\cdots, \mathsf{Recordable}\ \Gamma) \Rightarrow \mathsf{Pattern}\ \Gamma\ \alpha \rightarrow \mathsf{Pattern}\ \Gamma\ \alpha \rightarrow \mathsf{Rule}\ \alpha.$

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

> **class** $\mathsf{Validateable}\ \alpha$ **where**
>   *record* $:: \mathsf{Extended}\ \alpha\ \Gamma \rightarrow \mathsf{Record}\ \Gamma \rightarrow \mathsf{Record}\ \Gamma.$

Recall from Section 5 that a $\mathsf{Pattern}$ 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 $\Gamma\ \alpha \rightarrow$ Record $\Gamma \rightarrow$ Record Γ
$record'\ (Inl\ e)\ rec$ $=\ record\ e\ rec$
$record'\ (Inr\ RZero)\ (RCons\ b\ rec)$ $=\ RCons\ True\ rec$
$record'\ (Inr\ (RSucc\ r))\ (RCons\ b\ rec) = RCons\ b\ (record'\ (Inr\ r)\ rec)$.

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

instance Validateable Int **where** $record\ (Int'\ n) = id$.

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 $(\alpha$:+: $\beta)$ **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 $(\alpha$ ::: $\beta)$ **where**
$record\ (pat ::: '\ es) = record'\ pat \circ record\ es$.

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 $(\cdots,$ Validateable (Rep $\alpha)) \Rightarrow$ 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
      $check\ RNil$ $=\ True$
      $check\ (RCons\ b\ rec) = b \wedge check\ rec$.

Starting with a blank record, $validate$ records all occurrences of metavariables on 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,

$$\textbf{data } \mathsf{Lam} = \textit{Var } \mathsf{String} \mid \textit{Abs } \mathsf{String} \mathsf{Lam} \mid \textit{App } \mathsf{Lam} \mathsf{Lam}$$

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

$$\textit{fv} :: \mathsf{Lam} \to [\mathsf{String}].$$

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

$$\lambda x. e \ x \to e, \qquad \text{if } x \text{ not free in } e.$$

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 be written as in

> *etaReduction* :: Rule Lam
> *etaReduction* = *synthesise* $(\lambda x \ e \to \textit{Abs } x \ (\textit{App } e \ (\textit{Var } x)) \mapsto e \ \wr \ x \notin \textit{fv } e)$.

Here, we synthesise a rule over lambda-expressions from a function that produces a template constructed with the operators $\mapsto$ and $\wr$. 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 :: $\star \to \star$ **where**
>   *Rule* :: (Mappable Γ, Recordable Γ, Testable Γ) $\Rightarrow$
>   Pattern Γ $\alpha \to$ Pattern Γ $\alpha \to$ Guard Γ $\to$ 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,

$$\textbf{type family } \mathsf{Guard} \ \Gamma \ :: \star,$$

with instances

> **type instance** Guard Nil   = Bool
> **type instance** Guard $(\alpha ::: \Gamma) = \alpha \to$ 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 $\to$ Substitution Γ $\to$ 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):

$$\textbf{instance } \textsf{Testable Nil } \textbf{where}$$
$$\textit{test b PNil} = b.$$

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:

$$\textbf{instance } \textsf{Testable } \Gamma \Rightarrow \textsf{Testable } (\alpha ::: \Gamma) \textbf{ where}$$
$$\textit{test f } (PCons \, (Just \, x) \, s) = \textit{test s } (f \, x)$$
$$\textit{test f } (PCons \, Nothing \, s) = \textit{error } \texttt{"test failure"}.$$

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:

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

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). First, we extend templates with a boolean component:

$$\textbf{data } \textsf{Template } \alpha = \textit{Template } \alpha \, \alpha \, \textsf{Bool}.$$

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

$$\textbf{infix } 1 \mapsto$$
$$\textbf{infix } 0 \; \S$$
$$(\mapsto) :: \alpha \rightarrow \alpha \rightarrow \textsf{Template } \alpha$$
$$\textit{lhs} \mapsto \textit{rhs} = \textit{Template lhs rhs True}$$
$$(\S) :: \textsf{Template } \alpha \rightarrow \textsf{Bool} \rightarrow \textsf{Template } \alpha$$
$$\textit{Template lhs rhs } \_ \; \S \; b = \textit{Template lhs rhs b}.$$

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

$$\textbf{class } \cdots \Rightarrow \textsf{Synthesiser } \alpha \textbf{ where}$$
$$\vdots$$
$$\textit{guard} :: \alpha \rightarrow \textsf{Guard } (\textsf{Env } \alpha)$$

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

$$\textbf{instance} \ \cdots \Rightarrow \mathsf{Synthesiser} \ (\mathsf{Template} \ \alpha) \ \textbf{where}$$

$$\vdots$$

$$guard \ (Template \ lhs \ rhs \ b) = b.$$

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

$$\textbf{instance} \ \cdots \Rightarrow \mathsf{Synthesiser} \ (\alpha \rightarrow \beta) \ \textbf{where}$$

$$\vdots$$

$$guard \ f = guard \circ f.$$

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

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

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 this case study, we use some of the more advanced features of our library, such as heterogeneously typed metavariables and guarded rewrite rules. In Section 10, we discuss some results obtained from benchmarking.

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:

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

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,

$$\textbf{data } \mathsf{Prop }\ \alpha = \mathit{Var}\ \alpha \mid T \mid F \mid \mathit{Not}\ (\mathsf{Prop}\ \alpha)$$
$$\mid \mathsf{Prop}\ \alpha :\wedge: \mathsf{Prop}\ \alpha \mid \mathsf{Prop}\ \alpha :\vee: \mathsf{Prop}\ \alpha,$$

a type of equations,

$$\textbf{data } \mathsf{Equation}\ \alpha = \alpha :\equiv: \alpha,$$

and a type Expr of various arithmetic expressions,

$$\textbf{data } \mathsf{Expr} = \mathit{Const}\ \mathsf{Rational} \mid \mathit{Varia}\ \mathsf{String} \mid \mathsf{Expr} :+: \mathsf{Expr} \mid \mathsf{Expr} :-: \mathsf{Expr}$$
$$\mid \mathsf{Expr} :*: \mathsf{Expr} \quad \mid \mathsf{Expr} :/: \mathsf{Expr} \mid \mathsf{Expr} :\hat{}: \mathsf{Expr}.$$

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

$$\mathit{Var}\ ((\mathit{Const}\ 1 :+: (\mathit{Const}\ 8 :/: ((\mathit{Varia}\ \texttt{"x"} :-: \mathit{Const}\ 3) :\hat{}: \mathit{Const}\ 2))) :\equiv: \mathit{Const}\ 3).$$

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

$$\mathit{Var}\ (\mathit{Varia}\ \texttt{"x"} :\equiv: \mathit{Const}\ 5) :\vee: \mathit{Var}\ (\mathit{Varia}\ \texttt{"x"} :\equiv: \mathit{Const}\ (-1)).$$

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

$$\mathit{orTrueLeft} :: \mathsf{Rewritable}\ \alpha \Rightarrow \mathsf{Rule}\ (\mathsf{Prop}\ \alpha)$$
$$\mathit{orTrueLeft} = \mathit{synthesise}\ (\lambda p \to T :\vee: p \mapsto p)$$

and some rules for rewriting additions, which require preconditions,

$$\mathit{coverPlusLeft} :: \mathsf{Rule}\ (\mathsf{Equation}\ \mathsf{Expr})$$
$$\mathit{coverPlusLeft} = \mathit{synthesise}\ (\lambda x\ y\ z \to$$
$$x :+: y :\equiv: z \mapsto x :\equiv: z :-: y \ \S\ \mathit{hasVaria}\ x \wedge \mathit{noVaria}\ y).$$

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 the 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 (λx n y →
  let z = y :ˆ: Const (1 / n)
  in  Var (x :ˆ: Const n :≡: y) ↦ Var (x :≡: z) :∨: Var (x :≡: Const 0 :−: z)
      § hasVaria x ∧ n > 0 ∧ isEven n).
```

As this definition illustrates, complex rewrite rules can be 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 as generic functions, 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 & Hughes 2000) to generate a sequence of random propositions. The random-number generator is initiated with a fixed seed so that

Table 1. *The strategies benchmarked*

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%

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.

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 and avoiding the need to introduce an additional, extended datatype of propositions.

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 optimization level (using the `-O1` compiler flag). We do not use optimization 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.

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

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

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

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 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 optimizations. 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" optimization level.

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

Table 4. *Benchmark results for solving arithmetic equations*

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

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 optimization 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.

Our benchmarks confirm that observability of rules comes at the expense of loss in runtime efficiency. Furthermore, generic definitions introduce some additional overhead. The tradeoff between efficiency and genericity depends on the application at hand. For instance, the library would be suitable for the online exercise assistant, because runtime performance is less important in such a context.

We believe that improving the efficiency of generic library code is an interesting area for future research. By inlining and specializing generic definitions, and by applying 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 & Jeuring (2000) implement a generic rewriting library in PolyP (Jansson & 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 & Visser 2002), Scrap Your Boilerplate (Lämmel & Peyton Jones 2003), Uniplate

(Mitchell & Runciman 2007), Bringert's "almost compositional" functions (Bringert & 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 & Jeuring (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 & 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 & 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 analyze 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 "cheat" by inspecting 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.

References

Backhouse, R. C., Jansson, P., Jeuring, J. & Meertens, L. (1999) Generic programming: an introduction. In *Advanced Functional Programming, Third International School, Braga, Portugal, September 12–19, 1998, Revised Lectures*, Swierstra, S. D., Henriques, P. R. & Oliveira, J. N. (eds), *Lecture Notes in Computer Science*, vol. 1608. Springer-Verlag, pp. 28–115.

Borovanský, P., Kirchner, C., Kirchner, H. & Ringeissen, C. (2001) Rewriting with strategies in ELAN: a functional semantics, *Int. J. Found. Comput. Sci.*, 12 (1), 69–95.

Bravenboer, M., Kalleberg, K. T. & Visser, E. (2008) Stratego/XT 0.17: a language and toolset for program transformation. *Sci. Comput. Program.*, 72 (1–2), 52–70.

Bringert, B. & Ranta, A. (2006) A pattern for almost compositional functions. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16–21, 2006*, Reppy, J. H. & Lawall, J. L. (eds), ACM Press, pp. 216–226.

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

Bruijn, N. G. de. (1972) Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to the Church-rosser theorem, *Indagaciones Mathematische*, 34, 381–392.

Chakravarty, M. M. T., Keller, G. & Peyton Jones, S. (2005a) Associated type synonyms. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26–28, 2005*, In Danvy, O. & Pierce, B. C. (eds), ACM Press, pp. 241–253.

Chakravarty, M. M. T., Keller, G., Peyton Jones, S. & Marlow, S. (2005b) Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12–14, 2005*, Palsberg, J. & Abadi, M. (eds), ACM Press, pp. 1–13.

Claessen, K. & Hughes, J. (2000) 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*. ACM Press, pp. 268–279.

Deursen, A. van, Heering, J. & Klint, P. (eds). (1996) *Language Prototyping. an Algebraic Specification Approach*. AMAST Series in Computing, vol. 5. Singapore: World Scientific.

Goguen, J. & Grant, M. (1997) *Algebraic Semantics of Imperative Programs*. Cambridge, Massachusetts: The MIT Press.

Heeren, B., Jeuring, J., Leeuwen, A. van & Gerdes, A. (2008) Specifying strategies for exercises. In *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*, Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M. & Wiedijk, F. (eds), Lecture Notes in Computer Science, vol. 5144. Springer-Verlag, pp. 430–445.

Hinze, R. (2000) *Generic Programs and Proofs*. Habilitationsschrift: University of Bonn.

Hinze, R., Jeuring, J. & Löh, A. (2004) Type-indexed data types, *Sci. Comput. Program.*, 51 (2), 117–151.

Holdermans, S., Jeuring, J., Löh, A. & Rodriguez Yakushev, A. (2006) Generic views on data types. In *Mathemathics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3–5, 2006, Proceedings*, Uustalu, T. (ed), Lecture Notes in Computer Science, vol. 4014. Springer-Verlag, pp. 209–234.

Jansson, P. & Jeuring, J. (1997) 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*. ACM Press, pp. 68–114.

Jansson, P. & Jeuring, J. (1998) Polytypic unification, *J. Funct. Program.*, 8 (5), 527–536.

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

Lämmel, R. & Peyton Jones, S. (2003) 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*. ACM Press, pp. 26–37.

Lämmel, R. & Visser, J. (2002) Typed combinators for generic traversal. In *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19–20, 2002*, Krishnamurthi, S. & Ramakrishnan, C. R. (eds), Lecture Notes in Computer Science, vol. 2257. Springer-Verlag, pp. 137–154.

Magalhães, J. P., Holdermans, S., Jeuring, J. & Löh, A. (2010) Optimizing generics is easy! In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010, Madrid, Spain, January 18–19, 2010*, Gallagher, J. P. & Voigtländer, J. (eds), ACM Press, pp. 33–42.

Mitchell, N. & Runciman, C. (2007) Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, Keller, G. (ed), ACM Press, pp. 49–60.

Noort, T. van, Rodriguez Yakushev, A., Holdermans, S., Jeuring, J. & Heeren, B. (2008) A lightweight approach to datatype-generic rewriting. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP 2008, Victoria, BC, Canada, September 20, 2008*, Hinze, R. & Syme, D. (eds), ACM Press, pp. 13–24.

Pasălić, E. & Linger, N. (2004) Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24–28, 2004, Proceedings*, Karsai, G. & Visser, E. (eds), Lecture Notes in Computer Science, vol. 3286. Springer-Verlag, pp. 136–167.

Peyton Jones, S., Vytiniotis, D., Weirich, S. & Washburn, G. (2006) Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16–21, 2006*, Reppy, J. H. & Lawall, J. L. (eds), ACM Press, pp. 50–61.

Schrijvers, T., Peyton Jones, S., Sulzmann, M. & Vytiniotis, D. (2008) Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, ICFP 2008, Victoria, BC, Canada, September 22–24, 2008*, Hook, J. & Thiemann, P. (eds), ACM Press, pp. 51–62.

Sheard, T. (2001). 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*. ACM Press, pp. 86–97.

Sheard, T. & Peyton Jones, S. (2002) Template meta-programming for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Pittsburgh, Pennsylvania, 2002*. ACM Press, pp. 1–16.

Xi, H., Chen, C. & Chen, G. (2003) 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*. ACM Press, pp. 224–235.