A framework for polytypic programming on terms, with an application to rewriting

Patrik Jansson
Computing Science, Chalmers University of Technology, Sweden
patrikj@cs.chalmers.se, http://www.cs.chalmers.se/~patrikj/

Johan Jeuring
Computer Science, Utrecht University, the Netherlands
johanj@cs.uu.nl, http://www.cs.uu.nl/~johanj/

Abstract

Given any value of a datatype (an algebra of terms), and rules to rewrite values of that datatype, we want a function that rewrites the value to normal form if the value is normalizable. This paper develops a polytypic rewriting function that uses the parallel innermost rewriting strategy. It improves upon our earlier work on polytypic rewriting in two fundamental ways. Firstly, the rewriting function uses a term interface that hides the polytypic part from the rest of the program. The term interface is a framework for polytypic programming on terms. This implies that the rewriting function is independent of the particular implementation of polytypism. We give several functions and laws on terms, which simplify calculating with programs. Secondly, the rewriting function is developed together with a correctness proof. We just present the result of the correctness proof, the proof itself is published elsewhere.

1 Introduction

A term rewriting system is an algebra (a datatype of terms) together with a set of rewrite rules. The rewrite rules describe how to rewrite the terms of the algebra.

A rewrite rule is a pair (lhs, rhs) of terms containing variables with the interpretation that any term that matches the left hand side (lhs) may be rewritten to the right hand side (rhs) with the variables replaced by the matches from the left hand side. The code in this paper is expressed in the functional programming language Haskell 98 [10].

1.1 An example rewriting system

An example of a term datatype is the type *Expr*:

```
data Expr      =  EVar Int | Z | S Expr | Expr :+: Expr | Expr :*: Expr
type Rule t    =  (t, t)
plusZero       ::  Rule Expr
plusZero       =  (x :+: Z, x)
  where x      =  EVar 0
```

For example, with the rule *plusZero* the left hand side x :+: Z matches the expression $S\ Z$:+: Z with the substitution $\{x \mapsto S\ Z\}$. Thus the rewritten term is the right hand side x after the

substitution is applied: $S\ Z$. To introduce the notation we express this in Haskell syntax: the following expression evaluates to *True*.

```
let (lhs, rhs)    =  plusZero
    Just s        =  match lhs (S Z :+: Z)
in appSubst s rhs == S Z
```

The functions involved are the following: (for $t = Expr$)

```
appSubst   ::  Term t ⇒ Sub t → (t → t)
match      ::  Term t ⇒ t → t → Maybe (Sub t)
(==)        ::  Term t ⇒ t → t → Bool
```

Function *appSubst* takes a substitution and a term, and applies the substitution to the term. The type *Expr* is an instance of a type class for *Term*s defined in Section 2.1. The definitions of *appSubst* and the type constructor for *Sub*stitutions are given in Section 3.1. Function *match* (defined in Section 3.2) takes a term containing variables, and a term without variables, and returns *Just* a substitution s if the terms can be matched by means of s, and *Nothing* otherwise. The term equality test, (==), is defined in Section 2.2.

A rule set is a collection of rules, and a rule set matches a term if at least one of the rules matches that term. To keep the system deterministic, even when more than one rule matches, we order the rules and always use the first match. In practice this means that our rule set is a rule list.

```
type Rules t          =  [Rule t]
exprrules             ::  Rules Expr
exprrules             =  [plusZero, plusSucc, timesZero, timesSucc]
  where plusZero      =  (x :+: Z, x)
        plusSucc      =  (x :+: S y, S (x :+: y))
        timesZero     =  (x :*: Z, Z)
        timesSucc     =  (x :*: S y, (x :*: y) :+: x)
        (x, y)        =  (EVar 0, EVar 1)
```

Function *rewrite* (defined in Section 4.2) rewrites a term to normal form by repeatedly applying rules from a rule list:

```
rewrite :: Term t ⇒ Rules t → t → t
```

Because the rule list *exprrules* is normalizing, function *rewrite* will rewrite any term of type *Expr* to normal form. In general, *rewrite rs t* terminates if and only if the term t is normalizing with respect to the rule list rs.

1.2 Polytypic rewriting and the term interface

The types in the previous subsection already suggest that functions like *rewrite* and *match* are defined on an abstract type (class) of terms. Any regular datatype such as *Expr* can be made an instance of the *Term* class, and it follows that functions *rewrite* and *match* are *polytypic*.

This paper develops a polytypic rewriting function that uses the parallel innermost rewriting strategy. We have chosen the parallel innermost rewriting strategy because this lets us transform the rewriting function into an asymptotically optimal solution. The results in this paper improve upon our earlier work on polytypic rewriting [8] in two fundamental ways.

Firstly, the program uses a term interface that hides the polytypic part from the rest of the program. The term interface, which appeared in the types in the previous subsection as class

Term, is a framework for polytypic programming on terms. Several functions are generated for a type that is an instance of the *Term* class, such as a function that determines whether or not a term is a variable and a function that returns the children of a term. The rewriting function (including functions for matching and for applying a substitution) uses just these functions on terms. This idea was also present in our previous work [1, 6], but it was only applied to unification. It turns out that the same interface for terms can be used for matching and term rewriting.

Secondly, the program is developed together with a correctness proof, which says that our rewriting function rewrites any normalizable term to normal form. We just present the result of the correctness proof, the proof itself is published elsewhere [5, 7].

The rewriting functions and all other polytypic functions expressed using the term interface, are independent of the particular implementation of polytypism. Thus all such functions can be used in future polytypic programming languages such as Generic Haskell [2] too. This implies that the term interface is interesting in its own right. We describe several useful combinators and laws for programming against this interface. Examples of combinators on terms are *mapTerm*, which maps a function over all variables in a term, and *bottomup* which applies a term transformer bottom up to all levels of a term. Examples of laws that facilitate calculating with programs that manipulate terms, are *bottomup*-characterization, which says when a function can be expresses using *bottomup*, and *bottomup*-*mapTerm*-fusion, which fuses the composition of a *bottomup* and a *mapTerm* into one function (a *foldTerm*).

This paper is organized as follows. Section 2 introduces terms, combinators on terms, and laws for these combinators. Section 3 gives two applications of terms: substitutions and matching. These concepts are used in rewriting. Section 4 specifies and implements polytypic functions for rewriting and states the theorems they satisfy. Section 5 concludes.

2 A term interface

This section introduces an interface for terms, defines a few combinators that work on terms, and states some laws that relate these combinators. The proofs of these laws are presented in Jansson [5]. Appendix A shows that every regular datatype supports the term interface.

2.1 Terms

This subsection defines a Haskell class for types whose elements can be used as terms for matching, unification and in a term rewriting system. A careful analysis of the properties terms must satisfy reveals that

- a term has (updatable) children,
- two terms can be tested for top level equality,
- and one can check if a term is a variable.

Each of these requirements is captured in a class and the class of terms is the intersection of these classes.

$$\textbf{class}\ (\mathit{Children}\ t, \mathit{TopEq}\ t, \mathit{VarCheck}\ t) \Rightarrow \mathit{Term}\ t$$

In the following subsections we define the three classes *Children*, *TopEq* and *VarCheck* together with the laws we require from the instances to make the rewriting proofs go through later. The methods of these classes form an interface for terms that is sufficiently expressive to specify and implement rewriting, and the laws we require are sufficient to prove the correctness of the rewriting functions.

2.1.1 Children

The children (immediate subterms) of a term can be extracted and mapped over.

$$\begin{array}{lll} \textbf{class } \mathit{Children}\ t \textbf{ where } \mathit{children} & :: & t \to [t] \\ \qquad\qquad\qquad\qquad\quad \mathit{mapChildren} & :: & (t \to t) \to t \to t \end{array}$$

The functions *children* and *mapChildren* should be related by the following law:

$$\mathit{children} \circ \mathit{mapChildren}\ f \;=\; \mathit{map}\ f \circ \mathit{children}$$

Function *mapChildren* should preserve identities and composition:

$$\begin{array}{lcl} \mathit{mapChildren}\ \mathit{id} & = & \mathit{id} \\ \mathit{mapChildren}\ (f \circ g) & = & \mathit{mapChildren}\ f \circ \mathit{mapChildren}\ g \end{array}$$

2.1.2 Top level equality

Function *topEq* is a shallow equality test. A typical *topEq* checks if two terms have the same outermost constructor.

$$\textbf{class } \mathit{TopEq}\ t \textbf{ where } \mathit{topEq} :: t \to t \to \mathit{Bool}$$

We require *topEq* to be almost an equivalence relation:

$$\begin{array}{l} x \neq \bot \Rightarrow \mathit{topEq}\ x\ x \\ \mathit{topEq}\ x\ y \Rightarrow \mathit{topEq}\ y\ x \\ \mathit{topEq}\ x\ y \Rightarrow (\mathit{topEq}\ y\ z \Rightarrow \mathit{topEq}\ x\ z) \end{array}$$

It should not depend on the children:

$$\mathit{topEq}\ x\ y \iff \mathit{topEq}\ x\ (\mathit{mapChildren}\ f\ y)$$

And the number of children should be part of the top level:

$$\mathit{topEq}\ x\ y \Rightarrow (\mathit{length}\ (\mathit{children}\ x) == \mathit{length}\ (\mathit{children}\ y))$$

2.1.3 Checking for variables

It should be possible to check whether or not a term is a variable, and if it is, which variable. We model variables with the type *Var* (any type with equality would do).

$$\begin{array}{l} \textbf{newtype } \mathit{Var} \;=\; \mathit{MkVar}\ \mathit{Int} \textbf{ deriving } \mathit{Eq} \\ \textbf{class } \mathit{VarCheck}\ t \textbf{ where } \mathit{varCheck} :: t \to \mathit{Maybe}\ \mathit{Var} \end{array}$$

If a term is a variable, then it must not have children.

$$(\mathit{varCheck}\ t == \mathit{Just}\ v) \Rightarrow (\mathit{children}\ t == [\,])$$

2.2 Combinators on terms

In this subsection we define several general purpose combinators on terms. A first example is the function *size* that calculates the number of nodes in a term.

$$
\begin{aligned}
&size :: Term\ t \Rightarrow t \to Int \\
&size\ t\ =\ 1 + sum\ (map\ size\ (children\ t))
\end{aligned}
$$

A note on type contexts: Function *size* uses only one method from the term interface: function *children* from the class *Children*. Thus the most general type for *size* is $Children\ t \Rightarrow t \to Int$. Similarly, many of the other term combinators also use subsets of the methods in the term interface, but to avoid confusion we always use the type context $Term\ t \Rightarrow$.

Using *children* we extend the top level equality to deep equality:

$$
\begin{aligned}
&(==) :: Term\ t \Rightarrow t \to t \to Bool \\
&x == y\ =\ topEq\ x\ y\ \wedge\ and\ (zipWith\ (==)\ (children\ x)\ (children\ y))
\end{aligned}
$$

If *topEq* is almost an equivalence relation (as defined in section 2.1), then $(==)$ is an equivalence relation for all finite terms.

A simple application of the equality check is to define a predicate $fixedBy\ f$ that is true for elements which are fixed points of f:

$$
\begin{aligned}
&fixedBy :: Term\ t \Rightarrow (t \to t) \to t \to Bool \\
&fixedBy\ f\ x\ =\ x == f\ x
\end{aligned}
$$

Function $bottomup\ f$ applies the term transformer f at all levels of a term, bottom up. It is as close we can get to a generic catamorphism for types in the *Term* class. Function *bottomup* is more restricted than a normal catamorphism as the output is always of the same type as the input, but it is sufficient to specify and implement rewriting.

$$
\begin{aligned}
&bottomup :: Term\ t \Rightarrow (t \to t) \to (t \to t) \\
&bottomup\ f\ =\ f \circ mapChildren\ (bottomup\ f)
\end{aligned}
$$

Function *mapTerm* is one possible generic map function for *Term*s with variables. The application $mapTerm\ s$ maps s over all variables in a term, leaving the rest of the structure unchanged. It is implemented in terms of the more general function $foldTerm\ p\ s$ that also applies the function p to post-process the results from the children. Function *foldTerm* can be seen as the combination of a *bottomup* (a catamorphism) and a map.

$$
\begin{aligned}
&mapTerm && :: && Term\ t \Rightarrow (Var \to Maybe\ t) \to (t \to t) \\
&mapTerm\ s && = && foldTerm\ id\ s \\
&foldTerm && :: && Term\ t \Rightarrow (t \to t) \to (Var \to Maybe\ t) \to t \to t \\
&foldTerm\ p\ s\ t && = && \textbf{case}\ varCheck\ t\ \textbf{of} \\
&\quad Nothing && \to && p\ (mapChildren\ (foldTerm\ p\ s)\ t) \\
&\quad Just\ v && \to && \textbf{case}\ s\ v\ \textbf{of} \\
&\quad\quad Nothing && \to && p\ t \\
&\quad\quad Just\ t' && \to && t'
\end{aligned}
$$

Function *foldTerm* traverses all nodes in a term containing variables bottom up. If a node is a variable, then it is replaced by the term to which that variable is bound in the finite map s, or transformed by p if it is not bound by s. If a node is not a variable, then *foldTerm* is applied recursively to the children (if any) and the result is transformed by p.

2.3 Laws for term combinators

Using the properties required for the functions from the *Term* class we can derive a number of laws for the term combinators. The proofs of these laws are published elsewhere [5, 7]. The theorems for *bottomup* are restricted to finite terms as captured by the predicate *finite*:

Definition 1 *Finite terms:*

$$\begin{array}{lll} finite & :: & Term\ t \Rightarrow t \to Bool \\ finite & = & all\ finite \circ children \end{array}$$

To express that two functions are equal when restricted to a subset of their domains we use the following definition:

Definition 2 *Function equality on a subset:*

$$(\ \dot{=\!=}\ ) :: Eq\ a \Rightarrow (b \to Bool) \to (b \to a) \to (b \to a) \to (b \to Bool)$$
$$f \overset{p}{=\!=} g\ =\ \lambda x \to (p\ x) \Rightarrow (f\ x == g\ x)$$

The following theorem states when a function can be expressed using *bottomup*. It corresponds to *cata*-characterization in polytypic programming.

Theorem 3 *bottomup-characterization:*

$$(f \overset{finite}{=\!=\!=} g \circ mapChildren\ f) \iff (f \overset{finite}{=\!=\!=} bottomup\ g)$$

If we let $f\ =\ g\ =\ id$ in the *bottomup*-characterization theorem, then the premise $id \overset{finite}{=\!=\!=} id \circ mapChildren\ id$ follows trivially from the requirement $mapChildren\ id\ =\ id$ of the class *Children*. Thus we get the following corollary to *bottomup*-characterization:

Corollary 4 *bottomup-identity:*

$$id \overset{finite}{=\!=\!=} bottomup\ id$$

An easy consequence of *bottomup*-characterization is the following law for proving equality of functions defined using *bottomup*:

Theorem 5 *bottomup-equality:*

$$(g \circ mapChildren\ f \overset{finite}{=\!=\!=} h \circ mapChildren\ f) \iff (bottomup\ g \overset{finite}{=\!=\!=} bottomup\ h)$$
$$\textbf{where}\ f\ =\ bottomup\ g$$

Function *bottomup* is closely related to *foldTerm*; both traverse the term bottom up, but *bottomup* does not distinguish variables from other (sub)terms. The behavior of *bottomup* can be simulated by *foldTerm* if the substitution argument does *Nothing* for all variables:

Theorem 6 *bottomup is a foldTerm:*

$$foldTerm\ f\ (const\ Nothing) \overset{finite}{=\!=\!=} bottomup\ f$$

We could use this theorem to redefine *bottomup* in terms of *foldTerm*, but the current definition of *bottomup* is more efficient, and slightly more general (as it also works for term-like types without variables).

The final theorem of this section says that we can fuse the composition of a bottom-up traversal with a $mapTerm\ s$, where s is a function that maps variables to *Maybe* some value, into a *foldTerm*, provided that the bottom-up traversal is the identity on the result of s. The definitions of function *mapM* and some other utilities for calculating with *Maybe* values are presented in Figure 1.

Theorem 7 *bottomup-mapTerm-fusion:*

$$(mapM\ (bottomup\ f) \circ s\ =\ s) \Rightarrow (foldTerm\ f\ s \overset{finite}{=\!=\!=} bottomup\ f \circ mapTerm\ s)$$

$$
\begin{array}{l}
maybe :: b \to (a \to b) \to Maybe\ a \to b \\
maybe\ n\ j\ Nothing \quad = \quad n \\
maybe\ n\ j\ (Just\ x) \quad = \quad j\ x \\
mapM :: (a \to b) \to (Maybe\ a \to Maybe\ b) \\
mapM\ f\ =\ maybe\ Nothing\ (Just \circ f)
\end{array}
$$

Figure 1: Utility functions for *Maybe*-types.

3 Substitutions and matching

This section introduces two applications of terms: substitutions and matching. Both substitutions and matching are used in the following section on rewriting.

3.1 Substitutions

A substitution is a mapping from variables to terms that changes only a finite number of variables. As the concrete representation of substitutions is irrelevant for the definition of rewriting, we use an abstract datatype $Sub\ t$ for finite maps from variables to terms.

$$
\begin{array}{lcl}
idSubst & :: & Sub\ t \\
modBind & :: & (Var, t) \to Sub\ t \to Sub\ t \\
lookupIn & :: & Sub\ t \to (Var \to Maybe\ t)
\end{array}
$$

This could be implemented as a constructor class in Haskell, but we avoid that because we don't want to clutter up the types with an extra type context. The value *idSubst* represents the identity substitution, the call $modBind\ (v, t)\ s$ modifies the substitution s to bind v to t (leaving the bindings for other variables unchanged) and $lookupIn\ s\ v$ looks up the variable v in the substitution s, giving *Nothing* if the variable is not bound in s.

Using *lookupIn* a substitution can be viewed as a function from *variables* to terms. To use substitutions as functions from *terms* to terms we define *appSubst*:

$$
\begin{array}{l}
appSubst :: Term\ t \Rightarrow Sub\ t \to (t \to t) \\
appSubst\ s\ =\ mapTerm\ (lookupIn\ s)
\end{array}
$$

We can also define a variant of *appSubst* that does the equivalent of a bottom-up traversal with f after the substitution has been applied. A straightforward implementation would be the following:

$$
\begin{array}{l}
fromVarsUpAfterSubst :: Term\ t \Rightarrow (t \to t) \to (Sub\ t, t) \to t \\
fromVarsUpAfterSubst\ f\ (s, t)\ =\ bottomup\ f\ (appSubst\ s\ t)
\end{array}
$$

Instead, we use a simple corollary of *bottomup-mapTerm*-fusion (Theorem 7) to obtain a more efficient definition (for some combinations of f and s).

$$
\begin{array}{l}
fromVarsUpAfterSubst :: Term\ t \Rightarrow (t \to t) \to (Sub\ t, t) \to t \\
fromVarsUpAfterSubst\ f\ (s, t)\ =\ foldTerm\ f\ (lookupIn\ s)\ t
\end{array}
$$

Corollary 8 *bottomup-appSubst-fusion:*

$$
\begin{array}{l}
(mapM\ (bottomup\ f) \circ lookupIn\ s\ =\ lookupIn\ s) \\
\Rightarrow (fromVarsUpAfterSubst\ f\ (s, t) == bottomup\ f\ (appSubst\ s\ t))
\end{array}
$$

For example, if $bottomup\ f$ is an implementation of rewriting to normal form and the substitution binds all variables to terms in normal form, then the condition is satisfied.

3.2 Matching

Matching a pattern p with a term t yields *Just* a substitution s such that $appSubst\ s\ p == t$, or fails with *Nothing* if no such substitution exists. Both the pattern and the term may contain variables, but the matching only allows variables in the pattern to be instantiated — any variable in the term is treated as a term constant. Function *match* is defined in terms of $match'$ that carries around a current substitution, starting with the identity substitution.

```
match :: Term t ⇒ t → t → Maybe (Sub t)
match' :: Term t ⇒ t → t → Sub t → Maybe (Sub t)

match p t  =  match' p t idSubst
match' p t s  =  maybe no yes (varCheck p)
  where no  =  if topEq p t then
                 threadList (zipWith match' (children p) (children t)) s
               else
                 Nothing
        yes v  =  Just (modBind (v, t) s)
```

We assume that the patterns are linear - that is, no variable occurs twice in the same pattern. It is easy to extend this definition to work in the presence of nonlinear patterns; we do not, however, include the details here.

The utility functions *threadList* and (@@) compose monadic functions in sequence. Function *match* uses these for the *Maybe*-monad.

```
threadList  ::  Monad m ⇒ [a → m a] → (a → m a)
threadList  =   foldr (@@) return
(@@)        ::  Monad m ⇒ (a → m b) → (c → m a) → (c → m b)
f @@ g      =   λx → g x >>= f
```

4 Rewriting

This section specifies polytypic rewriting and presents an efficient implementation. We start in Section 4.1 with defining a function *rewrite_step*, which performs a single rewrite step on a term. Function *rewrite_step* is then used in a specification of (a clearly correct, but inefficient) function *rewrite* in Section 4.2. In a few steps, function *rewrite* can be transformed into an efficient rewriting function. We will just give the final result of this transformation: function $rewrite^E$ in Section 4.3. The full calculation can be found in Jansson [5, 7]. We end this section with a discussion on the asymptotic complexity of the rewriting functions in Section 4.4.

4.1 One step rewriting

Given a rule list *rs* and a term t to match we can select the first matching rule with *firstmatch rs t*:

```
firstmatch :: Term t ⇒ Rules t → t   →   Maybe (Sub t, t)
firstmatch [] t                      =   Nothing
firstmatch ((lhs, rhs) : rules) t    =   case match lhs t of
                          Just s     →   Just (s, rhs)
                          Nothing    →   firstmatch rules t
```

If a rule matches, then *firstmatch rs t* returns *Just* a pair (s, rhs) of the substitution and the right hand side of the matching rule.

Many of the functions defined in sequel are parametrized on the rule list (representing the rewriting system), but only *firstmatch* actively uses the rules. As the rule list argument is fixed during the other rewriting calculations, we write this argument as a subscript to improve readability. For example, we write $rewrite_{rs}\ t$ for the application of function *rewrite* to the rule list rs and the term t.

Using *firstmatch* and *appSubst* we can transform a rule list to a top level reduction function *reduceM* that gives *Just* the rewritten term or *Nothing*. An immediate variant is *reduce* that returns the term unchanged if no rule matches.

$$
\begin{array}{lcl}
reduceM & :: & Term\ t \Rightarrow Rules\ t \to (t \to Maybe\ t) \\
reduceM_{rs} & = & mapM\ (uncurry\ appSubst) \circ firstmatch\ rs \\
reduce & :: & Term\ t \Rightarrow Rules\ t \to t \to t \\
reduce_{rs}\ t & = & maybe\ t\ id\ (reduceM_{rs}\ t)
\end{array}
$$

The reduce functions only apply the rewrite rules on the top level of the term, but we want to apply the rules at any level. In a relational treatment of rewriting this corresponds to extending the top level reduction relation to a congruence. To retain the deterministic functional view we have to choose a rewriting strategy. We have chosen the parallel innermost rewriting strategy as this lets us transform the rewriting function into an asymptotically optimal solution. Innermost means that we order the subterms by their depth and apply the reduction function bottom up until the first match, and parallel means that all subterms at the same depth are reduced at the same time. Function *parallelInnermost* takes any top level term transformer to a global one step transformer, using the parallel innermost rewriting strategy. (The corresponding function for the parallel outermost rewriting strategy, *parallelOutermost*, is included here for comparison, but is not used in the sequel.)

$$
\begin{array}{lcl}
parallelInnermost & :: & Term\ t \Rightarrow (t \to t) \to (t \to t) \\
parallelInnermost\ f & = & contIfFixedBy\ (mapChildren\ (parallelInnermost\ f))\ f \\
parallelOutermost & :: & Term\ t \Rightarrow (t \to t) \to (t \to t) \\
parallelOutermost\ f & = & contIfFixedBy\ f\ (mapChildren\ (parallelOutermost\ f)) \\
contIfFixedBy & :: & Term\ t \Rightarrow (t \to t) \to (t \to t) \to (t \to t) \\
contIfFixedBy\ f\ r & = & \textbf{iff}\ fixedBy\ f\ \textbf{then}\ r\ \textbf{else}\ f
\end{array}
$$

where **iff** is lifted **if**. Combining *parallelInnermost* with *reduce* we arrive at one-step rewriting:

$$
\begin{array}{l}
rewrite\_step :: Term\ t \Rightarrow Rules\ t \to (t \to t) \\
rewrite\_step_{rs} = parallelInnermost\ reduce_{rs}
\end{array}
$$

4.2 Rewriting to normal form

The final step needed to obtain rewriting to normal form is, in relational terminology, the transitive closure. As a functional counterpart we use a fixed point operator *fp* that takes a one step reduction function r to a normalizer by applying r until the input term doesn't change:

$$
\begin{array}{l}
fp :: Term\ t \Rightarrow (t \to t) \to (t \to t) \\
fp\ f\ = \textbf{iff}\ fixedBy\ f\ \textbf{then}\ id\ \textbf{else}\ fp\ f \circ f
\end{array}
$$

The result $res == fp\ f\ x$, when *fp* terminates, is a fixed point in the sense that $res == f\ res$, that is, $fixedBy\ f\ res$ holds. Now we are ready to define rewriting to normal form:

$$
\begin{array}{l}
rewrite :: Term\ t \Rightarrow Rules\ t \to (t \to t) \\
rewrite_{rs} = fp\ rewrite\_step_{rs}
\end{array}
$$

Function $rewrite_{rs}$ rewrites a term until no rule applies anymore, that is, it rewrites a term to normal form. A term is in normal form for a rule list rs if it is unchanged by $rewrite\_step_{rs}$:

$$\begin{aligned} &normal :: Term\ t \Rightarrow Rules\ t \to (t \to Bool) \\ &normal_{rs} = fixedBy\ rewrite\_step_{rs} \end{aligned}$$

For rule lists rs corresponding to strongly normalizing rewrite systems, $rewrite_{rs}$ will take any term to its normal form, but $rewrite_{rs}$ also works for the subset of normalizing terms of any other rewriting system. If a term has multiple normal forms, then $rewrite_{rs}$ calculates only the one (if any) reachable by the parallel innermost rewriting strategy. If this strategy does not terminate for a certain term, then neither does $rewrite_{rs}$. More formally, we define normalizing terms and the first theorem for *rewrite*: (The operator $(\vee)$ is logical "or" lifted to predicates.)

Definition 9 *Normalizing terms:*

$$\begin{aligned} &normalizing :: Term\ t \Rightarrow Rules\ t \to t \to Bool \\ &normalizing_{rs} = normal_{rs} \vee normalizing_{rs} \circ rewrite\_step_{rs} \end{aligned}$$

Theorem 10 *Rewriting gives a normal form:*

$$normalizing_{rs} \Rightarrow (normal_{rs} \circ rewrite_{rs})$$

The proof of this theorem by fixed point induction is not difficult, but omitted here because of its length.

Function $rewrite_{rs}$ can be seen as an executable specification of rewriting to normal form for a given rule list and a given term. It can be useful for experimenting with different rule lists but for larger terms it is unacceptably inefficient. We define the norm of a term (with respect to a specific rule list) to be the number of (parallel innermost) reduction steps that it takes to reach normal form:

$$\begin{aligned} norm &:: Term\ t \Rightarrow Rules\ t \to t \to Int \\ norm_{rs}\ t &= \textbf{if}\ normal_{rs}\ t\ \textbf{then}\ 0\ \textbf{else}\ 1 + norm_{rs}\ (rewrite\_step_{rs}\ t) \end{aligned}$$

The time it takes to execute $rewrite_{rs}$ is linear in the norm, n, of the input term but quadratic in the (average) size, s, of the term being rewritten. Clearly it should be possible to do better than that - optimally we hope to obtain a running time of $O(n + s)$.

4.3 Efficient rewriting

In a number of steps we can transform the specification of rewriting, $rewrite_{rs}$, into an optimal function. Here, we only sketch the transformation, but it is presented in detail elsewhere [5, 7].

The first few transformation steps are used to remove the expensive equality tests in *fp* and *contIfFixedBy*. The essential change is that the rewriting function is transformed into an intermediate rewrite function that uses *reduceM* instead of *reduce*. These transformations are only valid if we restrict ourselves to *productive* rule lists. A rule list rs is productive if when a rule matches a term, applying that rule changes the term. This is a reasonable restriction. Otherwise, if a rule matches but leaves the term unchanged, then the intermediate rewriting function loops even though *rewrite* would have terminated with the unchanged term. The intermediate rewriting function has complexity $O(ns)$ but it is still not optimal.

After a few more transformations we end up with a less readable, but much more efficient rewriting function:

$$\begin{aligned} rewrite^E_{rs} &= bottomup\ frewrite^E_{rs} \\ frewrite^E &:: Term\ t \Rightarrow Rules\ t \to (t \to t) \\ frewrite^E_{rs} &= ffpM\ (firstmatch\ rs)\ (fromVarsUpAfterSubst\ frewrite^E_{rs}) \\ ffpM &:: (a \to Maybe\ b) \to (b \to a) \to (a \to a) \\ ffpM\ fM\ r &= \lambda x \to maybe\ x\ r\ (fM\ x) \end{aligned}$$

Theorem 11 *Efficient rewriting is correct with respect to its specification:*

For all productive rule lists rs:

$$rewrite_{rs} \overset{normalizing_{rs}}{===} rewrite^{E}_{rs}$$

This version of *rewrite* is the fifth (and most *E*fficient) version in the calculation, hence the superscript E on $rewrite^E$. Function $rewrite^E$ is linear in the number of steps needed to rewrite a term, and independent of the size of the intermediate terms. This big improvement is obtained by avoiding repeated traversals of already normal children. The improved version instead only traverses the right hand sides from the matching rules.

4.4 Efficiency comparison

A very simple measure of the running time for the different rewriting functions (two of which are given in this paper) is the number of Hugs-reduction steps (not to be confused with rewriting steps in the rewriting system) required to run the functions on some examples. The following table shows some measurements of the number of Hugs reductions required by the different versions to rewrite the expression 2^n for $n = 6, 7, 8$. The number 2 abbreviates $S\ (S\ Z) :: Expr$ and the exponentiation notation (2^n), is a shorthand for repeated multiplications (uses of $(:*:)$). The expression is normalized using the rewrite rules *exprrules* defined in Section 1.

expression	2^6	2^7	2^8	e
rewrite steps	107	179	323	n
Hugs-reductions $_A$	$7.4M$	$47M$	$344M$	$O(e^2n)$
Hugs reductions $_E$	$72k$	$122k$	$218k$	$O(n)$

The last column in the table gives the asymptotic complexity for the different versions in terms of the size of the answer e and the number of parallel innermost rewrite steps n. The number of rewrite steps n increases more slowly than the size of the answer e as the parallel rewriting strategy performs more and more inner reductions in parallel as the terms grow. Hence, n is not quite proportional to e, and we can analyze the complexity in terms of both variables. As we can see from the table both versions are linear in n but the dependence on e differs. As n is the number of rewrite steps, we can think of the dependence on e as the complexity per rewrite step.

For version $_A$ the complexity can be explained by the test for equality at every node in the term. As the equality check and the number of nodes are both linear in e, we get a quadratic dependency in total. An equality test that reports *False* is often quick, but determining that two terms are completely equal is of course linear. As the equality checks are performed to see if a term is in normal form, we can confirm the suspicion that this version does a lot of work on already normal (sub)terms.

Version $_E$ completely removes the unnecessary traversals of normal subterms, and thus reduces the cost of each rewrite step to a constant (determined by the rule list).

5 Conclusions

We have presented a framework for polytypic programming on terms, with which polytypic programs for matching, unification, rewriting, etc. can be constructed. The framework is an interface consisting of four functions. Using these four basic functions we have defined a set of combinators on terms, and we have stated several laws for these combinators. The framework has been used to calculate an efficient rewriting program from an inefficient, clearly correct specification.

Because the only polytypic components of the functions for rewriting, matching and unification are the functions in the term interface, our functions are independent of the particular implementation of polytypism. This is an important advantage. Other, less domain specific, frameworks for polytypic programming are the monadic traversal library of Moggi, Bellè and Jay [9] and the basic combinator library PolyLib [3, 5]. Very likely there are other domain specific polytypic libraries, but they can only be determined by developing many example polytypic programs.

References

[1] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1999.

[2] Ralf Hinze. A generic programming extension for Haskell. In Erik Meijer, editor, *Proceedings of the Third Haskell Workshop*, Technical report of Utrecht University, UU-CS-1999-28, 1999.

[3] P. Jansson and J. Jeuring. PolyLib – a polytypic function library. Workshop on Generic Programming, Marstrand, June 1998. Available from the Polytypic programming WWW page [4].

[4] Patrik Jansson. Polytypic programming. The WWW home page for polytypism: `http://www.cs.chalmers.se/~patrikj/poly/`.

[5] Patrik Jansson. *Functional Polytypic Programming.* PhD thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden, May 2000.

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

[7] Patrik Jansson and Johan Jeuring. Rewriting as a polytypic application. Work in progress, 2000.

[8] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming '96*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, 1996.

[9] E. Moggi, G. Bellè, and C.B. Jay. Monads, shapely functors and traversals. In *Category Theory and Computer Science, CTCS'99*, volume 29 of *ENTCS*, pages 265–286. Elsevier, 1999.

[10] Simon Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Simon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 — A non-strict, purely functional language. Available from `http://www.haskell.org/definition/`, February 1999.

A Polytypic term instances

Using PolyP we can show that all *Regular* datatypes are *Term*s by the instances in Figure 2. These definitions are included for readers familiar with polytypic programming. In this paper we do not explain how to define polytypic functions, for an introduction to polytypic programming, see Backhouse *et al.* [1]. The definitions of *fmap2*, *fflatten* and *fequal* are omitted.

```
instance Regular d ⇒ Children (d a) where
  children = fflatten ∘ fmap2 (const []) (:[]) ∘ out
  mapC f   = inn ∘ fmap2 id f ∘ out

instance (Regular d, Eq a) ⇒ TopEq (d a) where
  topEq t t' = fequal (==) (λ_ _ → True) (out t) (out t')

instance Regular d ⇒ VarCheck (d a) where
  varCheck = fvarCheck ∘ out

polytypic fvarCheck     ::  f a b → Maybe Var
  = case f of
      g + h             →   fvarCheck ▽ const Nothing
      Const Var         →   Just
      g                 →   const Nothing

polytypic fmap2     ::  (a → c) → (b → d) → f a b → f c d
polytypic fflatten  ::  f [a] [a] → [a]
polytypic fequal    ::  (a → b → Bool) → (c → d → Bool) →
                        f a c → f b d → Bool
```

Figure 2: A polytypic instantiation of *Term* using PolyP.