A Haskell Library for Term Rewriting[*]

Bertram Felgenhauer[1], Martin Avanzini[1] and Christian Sternagel[2]

[1] University of Innsbruck, Austria
{bertram.felgenhauer,martin.avanzini}@uibk.ac.at
[2] Japan Advanced Institute of Science and Technology, Japan
c-sterna@jaist.ac.jp

Abstract

We present a Haskell library for first-order term rewriting covering basic operations on positions, terms, contexts, substitutions and rewrite rules. This effort is motivated by the increasing number of term rewriting tools that are written in Haskell.

1 Introduction

First-order term rewrite systems (TRSs) [2] are a simple, yet Turing-complete model of computation. Consequently, many interesting properties like termination, complexity, unique normalization and confluence are undecidable for TRSs in general. Nevertheless, many techniques for establishing these properties have been developed and implemented in various termination, complexity and confluence tools. A number of these tools are written in Haskell, for example Matchbox [7], μTerm [3] and T$_C$T [1]. The certification tool CeTA [5] is developed using the Isabelle proof assistent [4], and uses code generation to produce a Haskell program.

Each of these tools has its own implementation of basic term rewriting functionality. We would like to change this situation by providing a common foundation for first-order term rewriting. To this end we have started developing a Haskell library called `term-rewriting`, aiming to be useful and easy to use. Our hope is to turn this into a community effort that benefits many people. Further information is available at our website:

`http://cl-informatik.uibk.ac.at/software/haskell-rewriting/`

The focus of our effort is the manipulation of first-order term rewrite systems on their own. There are different applications for first-order rewriting in Haskell programs. For example, in [6] generic programming techniques are used to allow Haskell programmers to specify transformations on algebraic data types in the form of rewrite rules.

In Section 2, we discuss guiding principles and lessons learned from previous work. Then we give an overview of the existing library in Section 3, and finally conclude in Section 4.

2 Design

In this section, we take a brief look at the term rewriting libraries used in Matchbox and T$_C$T, and then formulate some design principles for the `term-rewriting` library.

[*]Supported by the Austrian Science Fund (FWF): I603, J3202, P22467.

2.1 Prior art

The Matchbox termination tool[1] is based on the `haskell-tpdb` library[2], which provides a comprehensive parser for the Termination Problem Database (TPDB) XML format. At the time we discussed the library design, however, we were not aware of this development.

Previous versions of Matchbox used the `autolib-rewriting` library, which is part of the `auto/*` software collection. The library is mature, and has been in use for a long time. It certainly covers the basic functionality that we are looking for. The main problem is that it pulls in a lot of dependencies from the other parts of the `auto/lib`, which occasionally leads to type signatures that are hard to understand:

$$\mathit{is\_linear\_term} :: \mathsf{TRSC}\ v\ c \Rightarrow \mathsf{Term}\ v\ c \rightarrow \mathsf{Reporter}\ ()$$

(In this case it turns out that *is_linear_term* is actually an assertion that the given term is linear, and produces an informative message in the Reporter monad if the check fails.)

The complexity tool $\mathsf{T_CT}$ is developed in conjunction with its own term library `termlib`.[3] Again, basic functionality is covered. Its main drawback is that the term type is monomorphic, with variables and function symbols being represented—essentially—as integers. In practice, this means that virtually all code has to carry a signature holding additional information on function symbols and variables.

Neither library is very attractive for re-use, due to their complexity and seemingly ad-hoc design decisions. We tried to avoid this situation by following a few basic principles to be explained in the next subsection.

2.2 Principles

Our main goal is to have a library that is easy to use, and useful. We have established the following guidelines.

Minimal interface. This means foremost that each concept is represented by a single type, when possible, and that we avoid cluttering the interface with many variants of the same functionality without good reason.

Remark 1. One reviewer asked why we did not use type classes. There is ample of room for discussion here. Maybe the most convincing reason is that this would double the number of entities (a type class and a type for each concept) for—in our perception—little gain.

Simple interface. The library interface is plain Haskell98, without relying on advanced types or libraries. Data types should be as simple as possible.

Consensus. The implemented features should be generally useful. Put differently, the library should not force non-obvious design decisions on the user. This is best explained by an example. A noteworthy omission in the current library is a type for TRSs. We do provide operations on list of rules (Section 3), but a TRS comes with an associated signature, and we could not agree on what a signature should be.

Of course these guidelines are sometimes contradictory. Consider the example of critical pairs. By simplicity, they should just be a pair of terms. However, additional information like the rules that were involved in the overlap is often required. To represent this information, another type

[1]`https://github.com/jwaldmann/matchbox`
[2]`https://github.com/jwaldmann/haskell-tpdb`
[3]`http://cl-informatik.uibk.ac.at/software/tct/`

would be needed, violating minimality. We chose to implement a more complex type instead (Section 3).

3 Implementation

In this section, we give an overview of the `term-rewriting` library implementation. The modules inhabit the Data.Rewriting (abbreviated D.R) namespace. As a rule, we provide one module per concept, implemented using separate submodules for type, common operations and specialised functionality. We rely on qualified imports to disambiguate between operations that can be used on several types. For example, linearity is defined for terms, rules and TRSs, so we provide functions D.R.Term.*isLinear*, D.R.Rule.*isLinear* and D.R.Rules.*isLinear*. The key concepts are as follows:

Positions. A position is a list of natural numbers, each denoting an argument position. In accordance with accessing lists in Haskell, the first argument position has index 0.

$$\textbf{type}\ \mathsf{Pos} = [\mathsf{Int}]$$

Positions can be compared in various ways (e.g. above, below, parallel to).

Terms. Terms are polymorphic over function symbols f and variables v.

$$\textbf{data}\ \mathsf{Term}\ f\ v = \mathsf{Var}\ v \mid \mathsf{Fun}\ f\ [\mathsf{Term}\ f\ v]$$
$$\mathit{fold} :: (v \to a) \to (f \to [a] \to a) \to \mathsf{Term}\ f\ v \to a$$
$$\mathit{map} :: (f \to f') \to (v \to v') \to \mathsf{Term}\ f\ v \to \mathsf{Term}\ f'\ v'$$
$$\mathit{vars} :: \mathsf{Term}\ f\ v \to [v]$$
$$\mathit{funs} :: \mathsf{Term}\ f\ v \to [f]$$
$$\mathit{subtermAt} :: \mathsf{Term}\ f\ v \to \mathsf{Pos} \to \mathsf{Maybe}\ (\mathsf{Term}\ f\ v)$$

In addition to the displayed operations, we can check for ground terms, linear terms, and whether a term is a variant or an instance of another.

Remark 2. Interestingly, `haskell-tpdb` defines terms with swapped arguments in the Term type constructor: Our motivation was the convention of writing $\mathcal{T}(\mathcal{F},\mathcal{V})$ in the term rewriting literature, but there may be practical reasons

$$\textbf{data}\ \mathsf{Term}\ v\ s = \mathsf{Var}\ v \mid \mathsf{Node}\ s\ [\mathsf{Term}\ v\ s]$$

Our motivation was the convention of writing $\mathcal{T}(\mathcal{F},\mathcal{V})$ in the term rewriting literature, but there may be practical reasons for swapping the order.

Substitutions. In the term rewriting theory, substitutions are partial functions, usually with finite domain, from variables to terms. When applying a substitution to a term, variables for which the substitution is undefined are left untouched. In order to allow substitutions that change the type of variables (which arise naturally from matching terms), we distinguish between *generalized* substitutions and the standard kind. Attempting to apply a generalized substitution to a term that contains a variable with undefined replacement fails; the function yields no result, i.e., Nothing.

$$\textbf{newtype}\ \mathsf{GSubst}\ v\ f\ v' = \mathsf{GS}\ \{\mathit{unGS} :: \mathsf{Map}\ v\ (\mathsf{Term}\ f\ v')\}$$
$$\textbf{type}\ \mathsf{Subst}\ f\ v = \mathsf{GSubst}\ v\ f\ v$$

```
gApply :: Ord v ⇒ GSubst v f v′ → Term f v → Maybe (Term f v′)
apply :: Ord v ⇒ Subst f v → Term f v → Term f v
compose :: Ord v ⇒ Subst f v → Subst f v → Subst f v
```

Substitutions can be obtained by matching or unifying terms.

```
match :: (Eq f, Ord v, Eq v′) ⇒ Term f v → Term f v′ → Maybe (GSubst v f v′)
unify :: (Eq f, Ord v) ⇒ Term f v → Term f v → Maybe (Subst f v)
```

Rules. Rules are directed equations with terms on the left-hand and right-hand sides. They are valid rewrite rules if the left-hand side is not a variable and each variable that occurs in the right-hand side also occurs in the left-hand side.

```
data Rule f v = Rule {lhs :: Term f v, rhs :: Term f v}
isValid :: Ord v ⇒ Rule f v → Bool
```

The library supports checking of syntactical properties of rewrite rules, e.g., left- and right-linearity. One can also determine whether a rule is a variant or an instance of another one.

Lists of rules. A list of rules (which is a TRS without an associated signature) defines a rewrite relation on terms. The Reduct type records the position, used rule and substitution of a rewrite step in addition to the resulting term. We can compute the reducts of a term using a list of valid rules with respect to various strategies.

```
data Reduct f v v′ = Reduct {
  result :: Term f v,
  pos :: Pos, rule :: Rule f v′, subst :: GSubst v′ f v
}
type Strategy f v v′ = Term f v → [Reduct f v v′]
fullRewrite, innerRewrite, outerRewrite ::
  (Ord v′, Eq v, Eq f) ⇒ [Rule f v′] → Strategy f v v′
```

Critical Pairs. Critical pairs are the reducts arising from a critical overlap of rules. In the `term-rewriting` library, we annotate critical pairs by the source of the two rewrite steps (sometimes called a critical peak), the rules and the position of the left rewrite step.

```
data CP f v v′ = CP {
  left :: Term f (Either v v′), top :: Term f (Either v v′), right :: Term f (Either v v′),
  leftRule :: Rule f v, leftPos :: Pos, rightRule :: Rule f v′, subst :: Subst f (Either v v′)
}
cps :: (Ord v, Ord v′, Eq f) ⇒ [Rule f v] → [Rule f v′] → [CP f v v′]
```

We also support computation of inner and outer (i.e., root) critical pairs.

Contexts. A context is a term with a single hole. Very few operations are implemented for contexts.

```
data Ctxt f v = Hole | Ctxt f [Term f v] (Ctxt f v) [Term f v]
ofTerm :: Term f v → Pos → Maybe (Ctxt f v)
apply :: Ctxt f v → Term f v → Term f v
```

In addition, the library provides a data type (Problem) and parser for the WST (old TPDB) file format. There is also pretty printing support for terms, rules, substitutions and problems.

3.1 Example

As an example, we show an implementation of the local confluence check in the Knuth-Bendix criterion. The implementation is straightforward, using an auxiliary function *nf* that reduces a term to normal form with respect to the given TRS.

```
import qualified Data.Rewriting.Rules as Rules
import qualified Data.Rewriting.CriticalPair as CP
import Data.Rewriting.Rule (Rule)

  -- check confluence of a given terminating TRS
checkKnuthBendix :: (Eq f, Eq v, Ord v) => [Rule f v] -> Bool
checkKnuthBendix trs = all joinableCP (CP.cps' trs) where
    -- check joinability by comparing normal forms
  joinableCP c = nf (CP.left c) == nf (CP.right c)
    -- compute normal form of a term
  nf t = case Rules.fullRewrite trs t of
    [] -> t
    (r : _) -> nf (Rules.result r)
```

4 Future

We have described the current state of the `term-rewriting` Haskell library. In our view it covers enough functionality to be useful, even though most of it is trivial (notable exceptions are unification and the WST parser). However, the library has very few users: There is a converter from resource aware ML to term rewriting systems[4] and the beginnings of a confluence tool,[5] but nothing more. Thus, there are likely to be omissions in the interface.

For the future, we hope that the library becomes adopted more widely. The source code is hosted at github, so it is easy to submit bug reports and feature requests. We also have a mailing list for users, and the packages are available on Hackage.[6]

There are plenty of missing features, even leaving the contentious field of signatures aside. One interesting area is term graphs, where terms are represented with explicit sharing. Term graphs allow for efficient term rewriting and arise naturally as the result of unification. Unification can also gainfully take sharing information into account.

References

[1] M. Avanzini and G. Moser. Tyrolean Complexity Tool: Features and usage. In *Proc. 24th RTA*, LIPIcs, 2013. To appear.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, August 1998.

[4] `http://cl-informatik.uibk.ac.at/users/georg/cbr/tools/RaML/`

[5] `https://github.com/haskell-rewriting/confluence-tool`

[6] `http://hackage.haskell.org/`

[3] S. Lucas. MU-TERM: A tool for proving termination of context-sensitive rewriting. In *Proc. 16th RTA*, volume 3091 of *LNCS*, pages 200–209, 2004.

[4] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[5] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. 22nd TPHOLs*, volume 5674 of *LNCS*, pages 452–468, 2009.

[6] T. van Noort, A. Rodriguez, S. Holdermans, J. Jeuring, and B. Heeren. A lightweight approach to datatype-generic rewriting. In *Proc. ACM SIGPLAN WGP '08*, pages 13–24, 2008.

[7] J. Waldmann. Matchbox: A tool for match-bounded string rewriting. In *Proc. 16th RTA*, volume 3091 of *LNCS*, pages 85–94, 2004.