Rewrite Combinators

Stephen Diehl (@smdiehl)
Term Rewriting

- “Term rewriting is a system that consist of a set of objects, plus relations on how to transform those objects.”
- Very general idea, well studied subject.
- Shows up in numerical computing, program transformation, SMT solvers, logic programming, automated theorem proving, etc.
- Can encode arbitrary computation.

- Admits a nice family of combinator approaches to composing term rewrite systems.
Term Rewriting

You could have invented this machinery.

You probably already have if you’ve worked on problem domains involving manipulation of tree data structures.
Motivation

- Many years ago I was a student TAing a class on General Relativity when I independently reinvented a lot of this machinery when working on automating my grading obligations.
- Tensor calculus is an ugly subject that involves lots of terms and transformations and contractions over indices that easy to mess up by hand.

\[ R_{\mu\nu} - \frac{1}{2} R g_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu} \]

Ricci Curvature Tensor  Ricci Scalar  Metric Tensor  Stress-Energy Tensor
Motivation

- Problems that involve gnarly manipulations of lots of symbolic quantities according to rules, with intent to find some **normal form** or new set of **composite transformations** out of rules.

\[
R_{mnij} = g_{nk} \left( \frac{\partial \Gamma_{ni}^k}{\partial x^j} - \frac{\partial \Gamma_{nj}^k}{\partial x^i} + \Gamma_{nj}^a \Gamma_{ai}^k - \Gamma_{ni}^a \Gamma_{aj}^k \right)
\]

\[
= g_{nk} \left\{ \frac{\partial}{\partial x^i} \left[ \frac{1}{2} g^{kl} \left( \frac{\partial g_{jl}}{\partial x^n} + \frac{\partial g_{ln}}{\partial x^j} - \frac{\partial g_{nj}}{\partial x^l} \right) \right] - \frac{\partial}{\partial x^j} \left[ \frac{1}{2} g^{kl} \left( \frac{\partial g_{il}}{\partial x^n} + \frac{\partial g_{ln}}{\partial x^i} - \frac{\partial g_{ni}}{\partial x^l} \right) \right] \right\}
\]

\[
= \frac{1}{2} g_{nk} g^{kl} \left[ \frac{\partial}{\partial x^i} \left( \frac{\partial g_{jl}}{\partial x^n} - \frac{\partial g_{nj}}{\partial x^l} \right) - \frac{\partial}{\partial x^j} \left( \frac{\partial g_{il}}{\partial x^n} - \frac{\partial g_{ni}}{\partial x^l} \right) \right]
\]

\[
= \frac{1}{2} \left[ \frac{\partial^2 g_{jm}}{\partial x^i \partial x^n} - \frac{\partial^2 g_{nj}}{\partial x^i \partial x^m} - \frac{\partial^2 g_{im}}{\partial x^j \partial x^n} + \frac{\partial^2 g_{ni}}{\partial x^j \partial x^m} \right] \quad g_{nk} g^{kl} = \delta^m_i \rightarrow m = l
\]
Motivation

Can attach properties to terms such as symbol $W$ is an anti-commuting spinor in the left-handed Weyl representation of a Lorentz group. It’s coordinates transform under a set of rewrite rules.

There are many such rules involved in tensor manipulations.
Graph Reduction

Evaluation in GHC Haskell proceeds under a graph reduction model.

Program is is mapped to a directed graph data structure and program execution then consists of rewriting parts of this graph (i.e. "reducing") so as to move towards useful results.

\[(g, \ 2+2)\]
Term Rewriting

Similar to graph reduction but the rules and terms are separated into two sets. Rules are composed via strategies and exhaustively applying rules to subterms until no more rules apply (i.e. “normal form”).

Admits many degrees of freedom in terms of how evaluation and rule application can apply.

\[
\begin{align*}
\text{Impl}(x, y) & \rightarrow \text{Or(Not}(x), y) \\
\text{Eq}(x, y) & \rightarrow \text{And(Impl}(x, y), \text{Impl}(y, x)) \\
\text{Not(Not}(x)) & \rightarrow x \\
\text{Not(And}(x, y)) & \rightarrow \text{Or(Not}(x), \text{Not}(y)) \\
\text{Not(Or}(x, y)) & \rightarrow \text{And(Not}(x), \text{Not}(y))
\end{align*}
\]
Terms
Patterns
Rules
Strategies
(Theories)
Reduction & Traversal Order Dependence

Topdown
Innermost

\[
\begin{align*}
((2 + 2) + (2 + 2)) + (3 + 3) &= ((2 + 2) + (2 + 2)) + (3 + 3) \\
= ((2 + 2) + (2 + 2)) + 6 &= ((2 + 2) + 4) + (3 + 3) \\
= ((2 + 2) + 4) + 6 &= (4 + 4) + (3 + 3) \\
= (4 + 4) + 6 &= (4 + 4) + 6 \\
= 8 + 6 &= 8 + 6 \\
= 14 &= 14
\end{align*}
\]

Topdown Outermost
Terms

Vars

x, y, z

Functions

f(x), g(a, b), h(f(x, y), z)

Symbol Expressions

Nested Subterms
Terms

-- Term parameterised by function head type, and variable type

data Term a b where
  Var :: b -> Term a b
  Fun :: a -> [Term a b] -> Term a b

deriving (Show, Eq, Ord)
Bifunctor / Bitraversable

Can implement all expression manipulation in terms of Bifunctor / Bitraversable / Bifoldable over Term a b:

\[
\text{ttraverse} :: (v \to a) \to (f \to [a] \to a) \to \text{Term} f v \to a
\]

\[
\text{ttraverse} \text{ var } \text{ fun } (\text{Var } v) = \text{ var } v
\]

\[
\text{ttraverse} \text{ var } \text{ fun } (\text{Fun } f \text{ ts}) = \text{ fun } f (\text{fmap } (\text{ttraverse} \text{ var } \text{ fun}) \text{ ts})
\]

\[
\text{tmap} :: (f \to f') \to (v \to v') \to \text{Term} f v \to \text{Term} f' v'
\]

\[
\text{tmap} \text{ fun } \text{ var } = \text{ttraverse } (\text{Var } . \text{ var}) (\text{Fun } . \text{ fun})
\]
Terms

\[ f(g(x, y), z) \]

Function, arity=2

Term, arity=0
Terms

\[ a :: \text{Term Text Text} \]
\[ a = \text{Var } "a" \]

\[ b :: \text{Term Text Text} \]
\[ b = \text{Var } "b" \]

\[ f :: \text{Term Text Text} \]
\[ f = \text{Fun } "f" \ [a,b] \]
Positions

```
[0, 1]
[1]
[0, 0]
[0, 1]
```

2nd child of the 1st child of the root
Patterns

Patterns are terms parameterised over variables that can *match* on other terms.

A term either matches a pattern or it does not.

If it does match a pattern then it induces a matching *context* mapping pattern variables to term variables.

\[
\begin{array}{ccc}
\text{Pattern} & f(x, y) & \{ x \mapsto a, y \mapsto b \} & \text{Context} \\
\text{Scrutinee} & f(a, b) & \\
\end{array}
\]
Pattern Matching

```haskell
data Subst v f v' = Subst (M.Map v (Term f v'))

match :: (Eq f, Ord v, Eq v') => Term f v -> Term f v' -> Maybe (Subst v f v')
match t u = Subst <$> go t u (M.empty)
  where
    go (Var v) t subst = case M.lookup v subst of
      Nothing -> Just (M.insert v t subst)
      Just t' | t == t' -> Just subst
                _ -> Nothing

    go (Fun f ts) (Fun f' ts') subst
      | f /= f' || length ts /= length ts' = Nothing
      | otherwise = iterM (zipWith go ts ts') subst

    go _ _ _ = Nothing

iterM = foldr (=>>) pure
```
Rules

Rules combine patterns with expressions over the context can be substituted.

\[ l(x) \mapsto r(y) \]

1. **Expanding**: A rule is expanding if the right hand side is a function.
2. **Collapsing**: A rule is collapsing if the right hand side is a variable.
3. **Duplicating**: A rule is called duplicating if a variable occurs more on the right hand side than left.
4. **Erasing**: A rule is called erasing if a variable occurs less on the right hand side than left.

```haskell
data Rule f p v = Rule { lhs :: Term f p, rhs :: Term f v }
deriving (Show, Eq, Ord)
```
Rules

tapply :: (Ord v) => GSubst v f v' -> Term f v -> Maybe (Term f v')
tapply (Subst s) = ttraverse var fun
  where
    var v = M.lookup v s
    fun f ts = Fun f <$> sequence ts

apply :: (Ord v) => Subst f v -> Term f v -> Term f v
apply (Subst s) = ttraverse var fun
  where
    var v = M.findWithDefault (Var v) v s
    fun = Fun
Special Rules

Identity rule maps a term to itself without changing it.

\[ \text{id}(x) \]

Failure rule maps a term to nothing.

\[ \text{fail}(x) \]

\[ \text{rapply} \quad :: \quad (\text{Ord} \ v, \ \text{Eq} \ p, \ \text{Eq} \ f) \implies \text{Rule} \ f \ p \ v \implies \text{Term} \ f \ v \implies \text{Redux} \ (\text{Term} \ f \ v) \]

\[ \text{rapply} \ f \ x = \ldots \]
Strategies

Methods of combining rules into higher order rule systems which dispatch on intermediate redux terms.

A rule application results in one of two states. Failure, Success, or Identity.

```haskell
data Redux a
    = Failure
    | Success a
    | Identity a
    deriving (Eq, Ord, Show)
```
Strategy Primitives

Strategies combine rules into composite rules:

- `bottomup(s)`
- `innermost(s)`
- `all(s)`
- `try(s)`
- `fixpoint(s)`
- `repeat(s)`
- `seq(f,g)`
Higher Order Strategies

\[
\text{bottomup}(s) \quad = \quad \text{seq}(\text{all}(\text{bottomup}(s)),s)
\]
\[
\text{repeat}(s) \quad = \quad \text{try}(\text{seq}(s,\text{repeat}(s)))
\]
\[
\text{topdown}(s) \quad = \quad \text{seq}(s,\text{all}(\text{topdown}(s)))
\]
\[
\text{bottomup}(s) \quad = \quad \text{seq}(\text{all}(\text{bottomup}(s)),s)
\]

Can represent strategies as terms themselves and build rewrite systems over strategies.
-- Apply \( f \) and then \( g \) only if both succeed.
\[ \text{seq} :: \text{Rule } f \ p \ v \rightarrow \text{Rule } f \ p \ v \rightarrow \text{Term } f \ v \rightarrow \text{Redux } (\text{Term } f \ v) \]

-- Apply \( f \), if it succeeds then return the redex otherwise return identity.
\[ \text{try} :: \text{Rule } f \ p \ v \rightarrow \text{Term } f \ v \rightarrow \text{Redux } (\text{Term } f \ v) \]
\[ \text{try } f \ x = \text{case } \text{rapply } f \ x \ \text{of} \]
\[ \begin{align*}
\text{Failure} & \rightarrow \text{Identity } x \\
\text{Success } a & \rightarrow \text{Success } a \\
\text{Identity } a & \rightarrow \text{Identity } a
\end{align*} \]

-- Apply \( f \) to fixpoint.
\[ \text{fixpoint} :: \text{Rule } f \ p \ v \rightarrow \text{Term } f \ v \rightarrow \text{Redux } (\text{Term } f \ v) \]
\[ \text{fixpoint } f \ x = \text{case } \text{rapply } f \ x \ \text{of} \]
\[ \begin{align*}
\text{Failure} & \rightarrow \text{Failure} \\
\text{Success } a & \rightarrow \text{fixpoint } f \ x \\
\text{Identity } a & \rightarrow \text{Identity } a
\end{align*} \]

-- Repeatedly apply \( f \) until it fails, then yield last result.
\[ \text{repeat} :: \text{Rule } f \ p \ v \rightarrow \text{Term } f \ v \rightarrow \text{Redux } (\text{Term } f \ v) \]
\[ \text{repeat } f \ x = \text{case } \text{rapply } f \ x \ \text{of} \]
\[ \begin{align*}
\text{Failure} & \rightarrow \text{Identity } x \\
\text{Success } a & \rightarrow \text{repeat } f \ x \\
\text{Identity } a & \rightarrow \text{Identity } a
\end{align*} \]
Termination

1. Normalizing
2. Terminating
3. Confluence
4. Church-Rosser Property

Undecidable in the general case. Although many rewrite systems admit tractable termination analysis.

*Tyrolean Termination Tool 2*

http://colo6-c703.uibk.ac.at/ttt2/#descr

*Example:* Any composition of collapsing and erasing rules has termination closure.
Big Ideas

We can construct very general transformation languages with composition properties.

They can be parameterized over arbitrary Haskell types that implement Eq, Ord.

We get a general rule system for composing transformations.

Can abstract over groups of strategies, term languages, and rule sets to generate very powerful abstractions.
Example: Type Unification

Demo here.
Example: Theory of Groups

The theory of groups is given by three operations $m$ of arity two (the multiplication), $e$ of arity zero (the neutral element) and $i$ of arity one (the inverse), subject to the five expected relations

\[m(e, x_1) = x_1\]
\[m(x_1, e) = x\]
\[m(i(x_1), x_1) = e\]
\[m(x_1, i(x_1)) = e\]
\[m(m(x_1, x_2), x_3) = m(x_1, m(x_2, x_3))\]

\[d(x_1, d(d(d(x_1, x_1), x_2), x_3), d(d(d(x_1, x_1), x_1), x_3))) = x_2\]
Things I don’t have time for...

... but are terribly interesting.

1. Knuth-Bendix completion algorithm
2. Bounded term rewriting
3. Abstract rewrite machines
4. Maude / Pure
5. One-based theories
Further Resources

F. Baader, T. Nipkow, *Term Rewriting and All That*.
H. Cirstea, C. Kirchner, *Introduction to the rewriting calculus*.