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.

$$
\begin{aligned}
R_{mnij} &= g_{mk}\left(\frac{\partial \Gamma^k{}_{nj}}{\partial x^i} - \frac{\partial \Gamma^k{}_{ni}}{\partial x^j} + \Gamma^a{}_{nj}\Gamma^k{}_{ai} - \Gamma^a{}_{ni}\Gamma^k{}_{aj}\right) \\
&= g_{mk}\left\{\frac{\partial}{\partial x^i}\left[\frac{1}{2}g^{kl}\left(\frac{\partial g_{jl}}{\partial x^n} + \frac{\partial g_{ln}}{\partial x^j} - \frac{\partial g_{nj}}{\partial x^l}\right)\right] - \frac{\partial}{\partial x^j}\left[\frac{1}{2}g^{kl}\left(\frac{\partial g_{il}}{\partial x^n} + \frac{\partial g_{ln}}{\partial x^i} - \frac{\partial g_{ni}}{\partial x^l}\right)\right]\right\} \\
&= \frac{1}{2}g_{mk}g^{kl}\left[\frac{\partial}{\partial x^i}\left(\frac{\partial g_{jl}}{\partial x^n} - \frac{\partial g_{nj}}{\partial x^l}\right) - \frac{\partial}{\partial x^j}\left(\frac{\partial g_{il}}{\partial x^n} - \frac{\partial g_{ni}}{\partial x^l}\right)\right] \\
&= \frac{1}{2}\left[\frac{\partial^2 g_{jm}}{\partial x^i \partial x^n} - \frac{\partial^2 g_{nj}}{\partial x^i \partial x^m} - \frac{\partial^2 g_{im}}{\partial x^j \partial x^n} + \frac{\partial^2 g_{ni}}{\partial x^j \partial x^m}\right] \qquad g_{mk}g^{kl} = \delta^m_l \quad \rightarrow \quad m = l
\end{aligned}
$$

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.

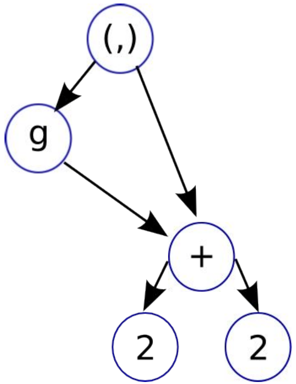
There are many such rules involved in tensor manipulations.

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.

```
Impl(x, y)      → Or(Not(x), y)
Eq(x, y)        → And(Impl(x, y), Impl(y, x))
Not(Not(x))     → x
Not(And(x, y))  → Or(Not(x), Not(y))
Not(Or(x, y))   → And(Not(x), Not(y))
```

Terms

Patterns

Rules

Strategies

(Theories)

Reduction & Traversal Order Dependence

Topdown Innermost

$$
\begin{aligned}
&((2+2)+(2+2))+(3+3)\\
=&((2+2)+(2+2))+6\\
=&((2+2)+4)+6\\
=&(4+4)+6\\
=&8+6\\
=&14
\end{aligned}
$$

$$
\begin{aligned}
&((2+2)+(2+2))+(3+3)\\
=&((2+2)+4)+(3+3)\\
=&(4+4)+(3+3)\\
=&(4+4)+6\\
=&8+6\\
=&14
\end{aligned}
$$

Topdown Outermost

Terms

Symbol Expressions

Vars

```
x, y, z
```

Functions

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

Nested Subterms

Terms

```haskell
-- 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)
```

Function Head

Argument List

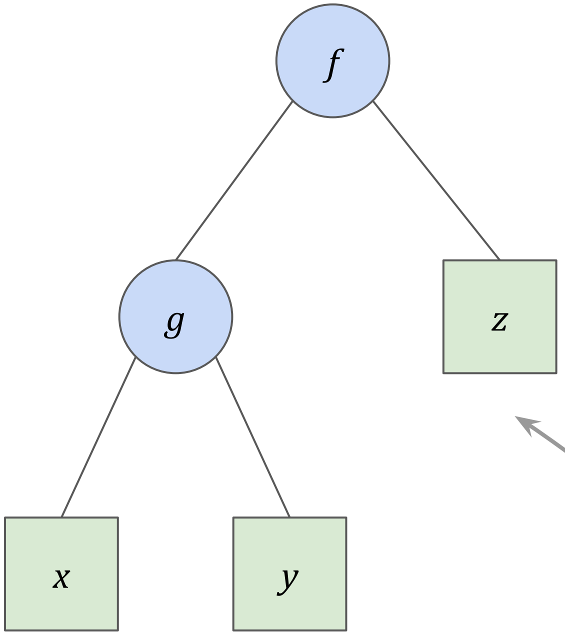
Bifunctor / Bitraversable

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

```
ttraverse :: (v -> a) -> (f -> [a] -> a) -> Term f v -> a
ttraverse var fun (Var v) = var v
ttraverse var fun (Fun f ts) = fun f (fmap (ttraverse var fun) ts)

tmap :: (f -> f') -> (v -> v') -> Term f v -> Term f' v'
tmap fun var = ttraverse (Var . var) (Fun . fun)
```

Terms



Function, arity=2

```
f(g(x, y), z)
```

Term, arity=0

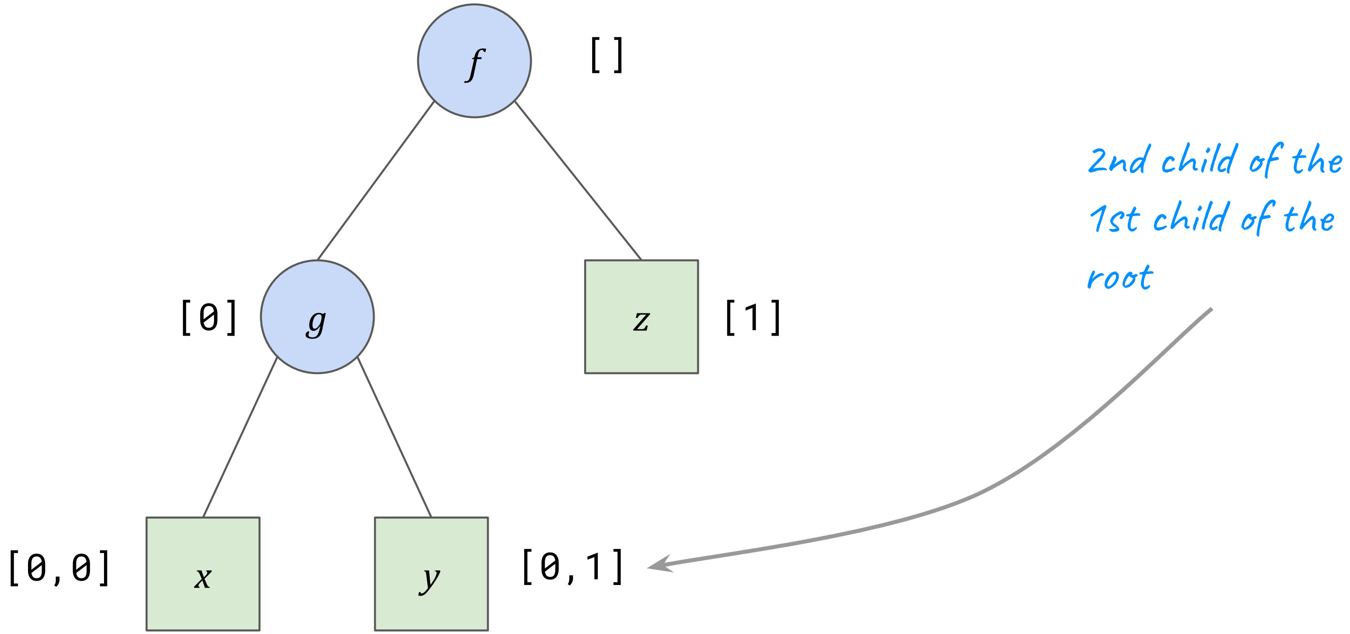
Terms

```haskell
a :: Term Text Text
a = Var "a"

b :: Term Text Text
b = Var "b"

f :: Term Text Text
f = Fun "f" [a,b]
```

Positions



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.

Pattern	`f(x,y)`	$\{ x \mapsto a, y \mapsto b \}$	Context
Scrutinee	`f(a,b)`		

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) ↦ 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

```haskell
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.

```
id(x)
```

Failure rule maps a term to nothing.

```
fail(x)
```

```
rapply :: (Ord v, Eq p, Eq f) => Rule f p v -> Term f v -> Redux (Term f v)
rapply f x = ...
```

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.

```
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)
```

Traversal Strategies

Control Flow Strategies

Higher Order Strategies

```
bottomup(s)   = seq(all(bottomup(s)),s)
  repeat(s)   = try(seq(s,repeat(s)))
  topdown(s)  = seq(s,all(topdown(s)))
  bottomup(s) = seq(all(bottomup(s)),s)
```

Can represent strategies *as terms themselves* and build rewrite systems over strategies.

```haskell
-- Apply f and then g only if both succeed.
seq :: Rule f p v -> Rule f p v -> Term f v -> Redux (Term f v)

-- Apply f, if it succeeds then return the redex otherwise return identity.
try :: Rule f p v -> Term f v -> Redux (Term f v)
try f x = case rapply f x of
  Failure    -> Identity x
  Success a  -> Success a
  Identity a -> Identity a

-- Apply f to fixpoint.
fixpoint :: Rule f p v -> Term f v -> Redux (Term f v)
fixpoint f x = case rapply f x of
  Failure    -> Failure
  Success a  -> fixpoint f x
  Identity a -> Identity a

-- Repeatedly apply f until it fails, then yield last result.
repeat :: Rule f p v -> Term f v -> Redux (Term f v)
repeat f x = case rapply f x of
  Failure    -> Identity x
  Success a  -> repeat f x
  Identity a -> Identity a
```

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

$m(e, x_1) = x_1$
$m(x_1, e) = x$
$m(i(x_1), x_1) = e$
$m(x_1, i(x_1)) = e$
$m(m(x_1, x_2), x_3) = m(x_1, m(x_2, x_3))$

$$\boxed{d(x_1, d(d(d(d(x_1, x_1), x_2), x_3), d(d(d(x_1, x_1), x_1), x_3))) = x_2}$$

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

D. E. Knuth, P. Bendix, *Simple Word Problems in Universal Algebras*.
F. Baader, T. Nipkow, *Term Rewriting and All That*.
T. Nipkow, C. Prehofer, *Higher-Order Rewriting and Equational Reasoning*.
H. Cirstea, C. Kirchner, *Introduction to the rewriting calculus*.
M. Fernandez, M. J. Gabbay, *Nominal Rewriting*.