Just for Show

A Purely Symbolic Effort in Mathematics

Gregory Wright

Alcatel-Lucent Bell Labs
Crawford Hill Laboratory
Holmdel, New Jersey

27 February 2013

Outline

A Bit of History

Symbolic algebra programs are among the oldest non-numeric programs, predating the introduction of Lisp in 1958.

Some of the earliest examples:

- Symbolic differentiation (folklore, ca. 1952)
- SAINT (Symbolic Automatic INTegrator (Slagle, 1961))
- SIN (Symbolic INtegrator (Moses, 1967))

A Bit of History

Symbolic algebra programs are among the oldest non-numeric programs, predating the introduction of Lisp in 1958.

Some of the earliest examples:

- Symbolic differentiation (folklore, ca. 1952)
- SAINT (Symbolic Automatic INTegrator (Slagle, 1961))
- SIN (Symbolic INtegrator (Moses, 1967))

A Bit of History

Symbolic algebra programs are among the oldest non-numeric programs, predating the introduction of Lisp in 1958.

Some of the earliest examples:

- Symbolic differentiation (folklore, ca. 1952)
- SAINT (Symbolic Automatic INTegrator (Slagle, 1961))
- SIN (Symbolic INtegrator (Moses, 1967))

A Bit of History

Symbolic algebra programs are among the oldest non-numeric programs, predating the introduction of Lisp in 1958.

Some of the earliest examples:

- Symbolic differentiation (folklore, ca. 1952)
- SAINT (Symbolic Automatic INTegrator (Slagle, 1961))
- SIN (Symbolic INtegrator (Moses, 1967))

A Bit of History

Attempts at general purpose symbolic algebra also began in the same era: For example,

- Schoonschip (1963 – 1967)
- MATHLAB (1964)
- Macsyma (1968 – 1995)
- Scratchpad/Axiom (1971 – present)
- Maple (1980 – present)
- SMP (1979)
- Mathematica (1988 – present)

A Bit of History

Attempts at general purpose symbolic algebra also began in the same era: For example,

- Schoonschip (1963 – 1967)
- MATHLAB (1964)
- Macsyma (1968 – 1995)
- Scratchpad/Axiom (1971 – present)
- Maple (1980 – present)
- SMP (1979)
- Mathematica (1988 – present)

A Bit of History

Attempts at general purpose symbolic algebra also began in the same era: For example,

- Schoonschip (1963 – 1967)
- MATHLAB (1964)
- Macsyma (1968 – 1995)
- Scratchpad/Axiom (1971 – present)
- Maple (1980 – present)
- SMP (1979)
- Mathematica (1988 – present)

A Bit of History

Attempts at general purpose symbolic algebra also began in the same era: For example,

- Schoonschip (1963 – 1967)
- MATHLAB (1964)
- Macsyma (1968 – 1995)
- Scratchpad/Axiom (1971 – present)
- Maple (1980 – present)
- SMP (1979)
- Mathematica (1988 – present)

A Bit of History

Attempts at general purpose symbolic algebra also began in the same era: For example,

- Schoonschip (1963 – 1967)
- MATHLAB (1964)
- Macsyma (1968 – 1995)
- Scratchpad/Axiom (1971 – present)
- Maple (1980 – present)
- SMP (1979)
- Mathematica (1988 – present)

A Bit of History

Attempts at general purpose symbolic algebra also began in the same era: For example,

- Schoonschip (1963 – 1967)
- MATHLAB (1964)
- Macsyma (1968 – 1995)
- Scratchpad/Axiom (1971 – present)
- Maple (1980 – present)
- SMP (1979)
- Mathematica (1988 – present)

A Bit of History

Attempts at general purpose symbolic algebra also began in the same era: For example,

- Schoonschip (1963 – 1967)
- MATHLAB (1964)
- Macsyma (1968 – 1995)
- Scratchpad/Axiom (1971 – present)
- Maple (1980 – present)
- SMP (1979)
- Mathematica (1988 – present)

A Bit of History

The development of Scratchpad/Axiom is important (for us) because it represents the first attempt to improve a symbolic algebra system by incorporating *types*.

Symbolic Mathematics v. Computer Algebra

They are different.

Computer algebra has evolved toward construction and enumeration of algebraic objects. Symbolic mathematics is usually the interactive manipulation of mathematical formulae in science and engineering.

Why Haskell?

Haskell has libraries that enable it to handle variety of algebraic objects (the Numeric Prelude and DoCon, the algebraic domain constructor). But in general it lacks the ability to manipulate symbolic expressions of these values.

Why Haskell?

Also, the programming language interfaces for the existing mainstream symbolic math programs (Maxima, Maple, Mathematica) are atrocious.

From a purely aesthetic standpoint, it would be nice to have a language for manipulating mathematical expressions that is a nice as Haskell.

Minimum Requirements

A symbolic mathematics system has two minimum requirements:

- It needs to automatically perform noncontroversial simplifications. This helps avoid intermediate expression bloat, as well as making final answers understandable.
- A pattern matching and replacement facility.

Minimum Requirements

A symbolic mathematics system has two minimum requirements:

- It needs to automatically perform noncontroversial simplifications. This helps avoid intermediate expression bloat, as well as making final answers understandable.
- A pattern matching and replacement facility.

Minimum Requirements

A symbolic mathematics system has two minimum requirements:

- It needs to automatically perform noncontroversial simplifications. This helps avoid intermediate expression bloat, as well as making final answers understandable.
- A pattern matching and replacement facility.

Bees in My Bonnet

I have a particular interest in calculations in quantum field theory. For these, I need

- Symbolic tensor expressions
- The ability to work with noncommuting objects

There are software packages that address some of my requirements (e.g., *Cadabra* and the *xTensor* package for Mathematica), but they don't fully solve my problem.

Bees in My Bonnet

I have a particular interest in calculations in quantum field theory. For these, I need

- Symbolic tensor expressions
- The ability to work with noncommuting objects

There are software packages that address some of my requirements (e.g., *Cadabra* and the *xTensor* package for Mathematica), but they don't fully solve my problem.

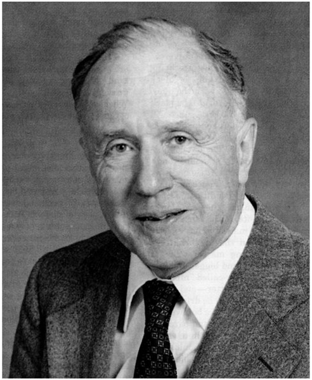
Bees in My Bonnet

I have a particular interest in calculations in quantum field theory. For these, I need

- Symbolic tensor expressions
- The ability to work with noncommuting objects

There are software packages that address some of my requirements (e.g., *Cadabra* and the *xTensor* package for Mathematica), but they don't fully solve my problem.

Why is it called the “Wheeler” library?



John Archibald Wheeler (1911 – 2008)

Desiderata

My goals are:

- Keep close to natural Haskell syntax.
- The user is not a compiler...
- ...which means use the natural operators $+$ and $*$ for addition and multiplication.
- No explicit simplification.
- Properly treat noncommuting objects.
- Automatic handling of tensor indices.

Desiderata

My goals are:

- Keep close to natural Haskell syntax.
- The user is not a compiler...
- ...which means use the natural operators $+$ and $*$ for addition and multiplication.
- No explicit simplification.
- Properly treat noncommuting objects.
- Automatic handling of tensor indices.

Desiderata

My goals are:

- Keep close to natural Haskell syntax.
- The user is not a compiler...
- ...which means use the natural operators $+$ and $*$ for addition and multiplication.
- No explicit simplification.
- Properly treat noncommuting objects.
- Automatic handling of tensor indices.

Desiderata

My goals are:

- Keep close to natural Haskell syntax.
- The user is not a compiler...
- ...which means use the natural operators $+$ and $*$ for addition and multiplication.
- No explicit simplification.
- Properly treat noncommuting objects.
- Automatic handling of tensor indices.

Desiderata

My goals are:

- Keep close to natural Haskell syntax.
- The user is not a compiler...
- ...which means use the natural operators $+$ and $*$ for addition and multiplication.
- No explicit simplification.
- Properly treat noncommuting objects.
- Automatic handling of tensor indices.

Desiderata

My goals are:

- Keep close to natural Haskell syntax.
- The user is not a compiler...
- ...which means use the natural operators $+$ and $*$ for addition and multiplication.
- No explicit simplification.
- Properly treat noncommuting objects.
- Automatic handling of tensor indices.

Desiderata

My goals are:

- Keep close to natural Haskell syntax.
- The user is not a compiler...
- ...which means use the natural operators $+$ and $*$ for addition and multiplication.
- No explicit simplification.
- Properly treat noncommuting objects.
- Automatic handling of tensor indices.

Vectors and Tensors

Vectors are objects with some definite properties under coordinate transformations (e.g., rotations). They are written

$$v^{\mu}$$

Tensors are objects with a bunch indices, each of which transforms like a vector

$$t^{\mu\nu\rho\sigma}$$

A special tensor, the *metric* computes the length of a vector

$$|v|^2 = \sum_{\mu,\nu=0}^{3} g_{\mu\nu} v^{\mu} v^{\nu}$$

Vectors and Tensors

But we never write the summation signs, repeated indices are *implicitly* summed over:

$$|v|^2 = g_{\mu\nu} v^\mu v^\nu$$

Repeated indices are also called “dummy indices”. A challenge is managing dummy indices so we can write things like

$$(a^\mu + b^\mu)(c_\mu + d_\mu)$$

and properly expand or factor them.

Expressions

```
-- The Expr data type:
--
data Expr where
    Const     :: Numeric -> Expr
    Applic    :: Function -> Expr -> Expr
    Symbol    :: Symbol -> Expr
    Sum       :: [ Expr ] -> Expr
    Product   :: [ Expr ] -> Expr
    Power     :: Expr -> Expr -> Expr
    Undefined :: Expr
```

Expressions

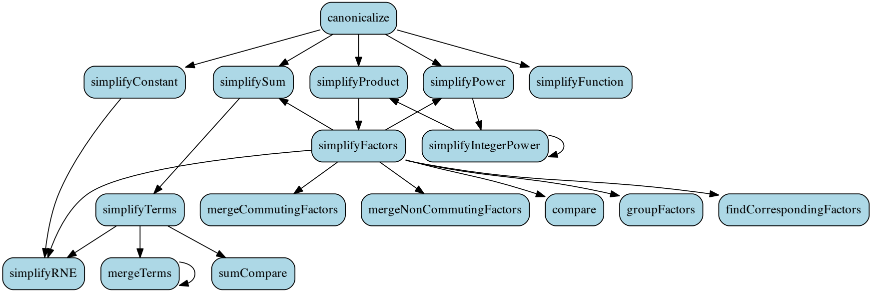
```
instance Num Expr where
          (+) f g       = canonicalize (Sum [f, g])
          (-) f g       = canonicalize (Sum [f, negate g])
          (*) f g       = canonicalize (Product [f, g])
          negate f      = canonicalize (Product [Const (-1), f
              ])
          abs f         = canonicalize (Applic Abs f)
          signum f      = canonicalize (Applic Signum f)
          fromInteger n = Const (I n)
```

Expressions

```
instance Ord (Expr) where
    compare (Const x) (Const y)             = compare x y
    compare (Const _) _                     = LT

    compare (Product _) (Const _)           = GT
    compare (Product x) (Product y)         = compareList x y
    compare p@(Product _) y                 = compare p (Product [ y ])

    compare (Power _ _) (Const _)           = GT
    compare p@(Power _ _) (Product y)       = compareList [ p ] y
    compare p@(Power _ _) p'@(Power _ _)    = comparePower p p'
    compare p@(Power _ _) y                 = comparePower p (Power y (Const 1))

    compare (Sum _) (Const _)               = GT
    compare s@(Sum _) p@(Product _)         = compare (Product [ s ]) p
    compare s@(Sum _) p@(Power _ _)         = compare (Power s (Const 1)) p
    compare (Sum x) (Sum y)                 = compareList x y
    compare s@(Sum _) y                     = compare s (Sum [ y ])

    compare (Applic _ _) (Const _)          = GT
    compare a@(Applic _ _) p@(Product _)    = compare (Product [ a ]) p
    compare a@(Applic _ _) p@(Power _ _)    = compare (Power a (Const 1)) p
```

⋮

Canonicalization



Representing Tensors

A clean syntax for representing tensors is to make the "kernel symbol" a *function*, which is applied to the indices. The result of applying the kernel symbol to the indices is the tensor object itself:

```
let
    g = minkowskiMetric "g"
in
    g alpha sigma * g beta rho   * g mu  nu    -
    g alpha nu    * g beta rho   * g mu  sigma -
    g alpha sigma * g beta mu    * g nu  rho   +
    g alpha beta  * g mu   sigma * g nu  rho   +
    g alpha rho   * g beta sigma * g mu  nu    -
    g alpha rho   * g beta mu    * g nu  sigma -
    g alpha nu    * g beta sigma * g mu  rho   +
    g alpha beta  * g mu   rho   * g nu  sigma
```

A Dirty Trick

```
-- A simple operator to toggle the variance.
-- It is an ugly hack, but letting "-" toggle the
-- variance is the least ugly option, given that we
-- don't have unary operators in Haskell.
--
instance Num VarIndex where
    negate (Covariant i)     = Contravariant i
    negate (Contravariant i) = Covariant i
    (+) _ _          = error "can't add slots"
    (*) _ _          = error "can't multiply slots"
    abs _            = error "can't take abs of a slot"
    signum _         = error "can't take signum of a slot"
    fromInteger _    = error "can't convert Integer to slot"
```

A Dirty Trick

This lets us write things like

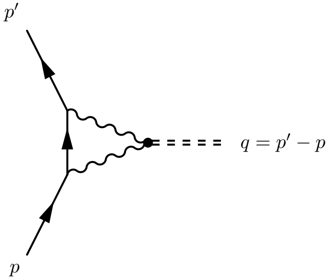
$$\delta^{\mu}{}_{\nu}$$

as

```
delta = mkKroneckerDelta minkowskiManifold "delta"
mu = minkowskiIndex_ "mu" "\\mu"
nu = minkowskiIndex_ "nu" "\\nu"

let d = delta mu (-nu)
```

Test Drive!

The question



What needs to be calculated

$$I_{\rho\sigma} = \int_0^1 dz \int_0^{1-z} dy \int d^4k \frac{N_{\rho\sigma}(k, y, z)}{(k^2 - M(y, z)^2 + i\epsilon)^3}$$

The question, in Wheeler

```
p  = momentum "p"
p' = momentum "p'"
k  = momentum "k"

m = scalar "m"
y = scalar "y"
z = scalar "z"

diracSpinor = diracSpinor_ (RepSpace "s")
diracGamma  = diracGamma_  (RepSpace "s")
diracSlash  = diracSlash_  (RepSpace "s")

inSpinor  = diracSpinor "u"
outSpinor = diracConjugate . diracSpinor "u"

triangle = outSpinor p' * (diracGamma mu * (diracSlash k + y * diracSlash p + z * diracSlash p' + m) *
                           diracGamma nu * ((1 - y) * p alpha - z * p' alpha - k alpha) *
                           vertexBelinfante (-alpha) (-beta) (-mu) (-nu) (-rho) (-sigma) *
                           ((1 - z) * p' beta - y * p beta - k beta ) ) *
           inSpinor p
```

Expanded

```
*Main> triangle_e
y * g (-d7) (-rho) * g (-d8) (-d9) * g (-d10) (-sigma) * k d7 * k d10 * p (-d11) *
(diracConjugate u) * gamma d8 * gamma d11 * gamma d9 * u + z * g (-d12) (-rho) * g (-
d13) (-d14) * g (-d15) (-sigma) * k d12 * k d15 * p' (-d16) * (diracConjugate u) * gamma
d13 * gamma d16 * gamma d14 * u + g (-d17) (-rho) * g (-d18) (-d19) * g (-d20) (-sigma)
* k (-d21) * k d17 * k d20 * (diracConjugate u) * gamma d18 * gamma d21 * gamma d19 * u
+ m * g (-d22) (-rho) * g (-d23) (-d24) * g (-d25) (-sigma) * k d22 * k d25 *
(diracConjugate u) * gamma d23 * gamma d24 * u - y * g (-d26) (-rho) * g (-d27) (-d28) *
g (-d29) (-sigma) * k d26 * p (-d30) * p d29 * (diracConjugate u) * gamma d27 * gamma
d30 * gamma d28 * u - z * g (-d31) (-rho) * g (-d32) (-d33) * g (-d34) (-sigma) * k d31
* p d34 * p' (-d35) * (diracConjugate u) * gamma d32 * gamma d35 * gamma d33 * u - g (-
d36) (-rho) * g (-d37) (-d38) * g (-d39) (-sigma) * k (-d40) * k d36 * p d39 *
(diracConjugate u) * gamma d37 * gamma d40 * gamma d38 * u - m * g (-d41) (-rho) * g (-
d42) (-d43) * g (-d44) (-sigma) * k d41 * p d44 * (diracConjugate u) * gamma d42 * gamma
d43 * u + y**2 * g (-d45) (-rho) * g (-d46) (-d47) * g (-d48) (-sigma) * k d45 * p (-
d49) * p d48 * (diracConjugate u) * gamma d46 * gamma d49 * gamma d47 * u + y * z * g (-
d50) (-rho) * g (-d51) (-d52) * g (-d53) (-sigma) * k d50 * p d53 * p' (-d54) *
(diracConjugate u) * gamma d51 * gamma d54 * gamma d52 * u + y * g (-d55) (-rho) * g (-
d56) (-d57) * g (-d58) (-sigma) * k (-d59) * k d55 * p d58 * (diracConjugate u) * gamma
d56 * gamma d59 * gamma d57 * u + m * y * g (-d60) (-rho) * g (-d61) (-d62) * g (-d63)
(-sigma) * k d60 * p d63 * (diracConjugate u) * gamma d61 * gamma d62 * u + y**2 * g (-
d64) (-rho) * g (-d65) (-d66) * g (-d67) (-sigma) * k d67 * p (-d68) * p d64 *
(diracConjugate u) * gamma d65 * gamma d68 * gamma d66 * u - y * g (-d69) (-rho) * g (-
d70) (-sigma) * g (-d71) (-d72) * k d69 * k d71 * p (-d73) * (diracConjugate u) * gamma
d70 * gamma d73 * gamma d72 * u + y * z * g (-d74) (-rho) * g (-d75) (-d76) * g (-d77)
(-sigma) * k d74 * p (-d78) * p' d77 * (diracConjugate u) * gamma d75 * gamma d78 *
gamma d76 * u + z**2 * g (-d79) (-rho) * g (-d80) (-d81) * g (-d82) (-sigma) * k d79 *
p' (-d83) * p' d82 * (diracConjugate u) * gamma d80 * gamma d83 * gamma d81 * u + z * g
(-d84) (-rho) * g (-d85) (-d86) * g (-d87) (-sigma) * k (-d88) * k d84 * p' d87 *
(diracConjugate u) * gamma d85 * gamma d88 * gamma d86 * u + m * z * g (-d89) (-rho) * g
```

...and on for another 38 pages.

Processing

```
-- Apply the Dirac equation wherever we can:
--
diracEquation     = (p  (-(mkPatternIndex "k")) * diracGamma   (mkPatternIndex "k")  * inSpinor p,   m
       * inSpinor p)
diracEquation'    = (p    (mkPatternIndex "k")  * diracGamma (-(mkPatternIndex "k")) * inSpinor p,   m
       * inSpinor p)
diracEquation''   = (p' (-(mkPatternIndex "k")) * outSpinor p' * diracGamma   (mkPatternIndex "k"),  m
       * outSpinor p')
diracEquation'''  = (p'   (mkPatternIndex "k")  * outSpinor p' * diracGamma (-(mkPatternIndex "k")), m
       * outSpinor p')

diracEquationIdentities = [ diracEquation
                          , diracEquation'
                          , diracEquation''
                          , diracEquation'''
                          ]

applyDiracEquation = applyUntilStable $ multiMatchAndReplace diracEquationIdentities

-- sp'' the the scalar part of the numerator, after applying simple
-- gamma matrix identities, then the Dirac equation for on-shell spinors.
--
sp'' = applyDiracEquation sp'
```

The answer

$$
\begin{aligned}
&[-m^3+mq^2+3m^3y-mq^2y-3m^3y^2+m^3y^3+3m^3z-mq^2z-6m^3yz+3mq^2yz \\
&+3m^3y^2z-mq^2y^2z-3m^3z^2+3m^3yz^2-mq^2yz^2+m^3z^3]\, g_{\rho\sigma}\bar{u}(p')u(p)+ \\
&[2m-5my+4my^2-my^3-5mz+8myz \\
&-3my^2z+4mz^2-3myz^2-mz^3]\, l_\sigma l_\rho \bar{u}(p')u(p)+ \\
&[-2m+my+2my^2-my^3+mz-4myz \\
&+my^2z+2mz^2+myz^2-mz^3]\, q_\sigma q_\rho \bar{u}(p')u(p)+ \\
&[-m^2+2m^2y-(1/2)q^2y-m^2y^2 \\
&+1/2q^2y^2+2m^2z-(1/2)q^2z-2m^2yz-m^2z^2+1/2q^2z^2]\,(l_\sigma\bar{u}(p')\gamma_\rho u(p)+l_\rho\bar{u}(p')\gamma_\sigma u(p))+ \\
&[2my-3my^2+my^3-2mz+my^2z+3mz^2-myz^2-mz^3]\,(l_\sigma q_\rho \bar{u}(p')u+l_\rho q_\sigma \bar{u}(p')u(p))+ \\
&[-2m^2y+1/2q^2y+2m^2y^2-(1/2)q^2y^2+2m^2z-(1/2)q^2z-2m^2z^2+1/2q^2z^2] \\
&(q_\sigma\bar{u}(p')\gamma_\rho u+q_\rho\bar{u}(p')\gamma_\sigma u(p))+ \\
&m[-4+2y+2z]g_{\rho\sigma}\bar{u}(p')u(p)+ \\
&[4-2y-2z]l_\rho\bar{u}(p')\gamma_\sigma u(p)+[4-2y-2z]\,l_\sigma\bar{u}(p')\gamma_\rho u(p)+ \\
&[2y-2z]q_\rho\bar{u}(p')\gamma_\sigma u(p)+[2y-2z]q_\sigma\bar{u}(p')\gamma_\rho u(p)
\end{aligned}
$$

Profiling

```
-- Given an index and a breadcrumb trail, replace the corresponding
-- index in the tree with the supplied index.  The argument order
-- is compatible with using replaceIndex in foldr.
--
replaceIndex :: VarIndexInContext -> Expr -> Expr
replaceIndex v e = snd $ ri (index v) (context v) ([], e)
    where
        ri :: VarIndex -> Breadcrumbs -> (Breadcrumbs, Expr) -> (Breadcrumbs, Expr)
        ri i' b' (b, t@(Symbol (Tensor _))) = {-# SCC "ri_1" #-} if (b == tail b')
                                               then (b, repIndex (head b') i' t)
                                               else (b, t)
        ri i' b' (b, Product ps)    = {-# SCC "ri_2" #-} (b, Product (zipWith (\n x -> snd (ri i' b' ((
               Pcxt n) : b, x))) [1..] ps))
        ri i' b' (b, Sum ts)        = {-# SCC "ri_3" #-} (b, Sum     (zipWith (\n x -> snd (ri i' b' ((
               Scxt n) : b, x))) [1..] ts))
        ri _  _  u@(_, _)           = {-# SCC "ri_4" #-} u

        repIndex                                :: Cxt -> VarIndex -> Expr -> Expr
        repIndex (Tcxt n) ind (Symbol (Tensor t)) = {-# SCC "repIndex" #-} Symbol (Tensor $ t {slots =
                (replace n ind (slots t))})
            where
                replace j x l = map (\(k, y) -> if j == k then x else y) $ zip [1..] l
        repIndex _ _ _                          = error "Can't happen: error replacing index"
```

Profiling

```
$ cabal clean
cleaning...
$ cabal configure --user \
--enable-library-profiling \
--ghc-option=-auto-all
Resolving dependencies...
Configuring Wheeler-0.3...
$ cabal install --enable-library-profiling \
--ghc-option=-auto-all
```

Profiling

```
$ rm *.hi *.o ScalarTriangle
$ ghc --make -prof -auto-all -O0 -rtsopts -o ScalarTriangle ScalarTriangle.hs
[1 of 4] Compiling Minkowski        ( Minkowski.hs, Minkowski.o )
[2 of 4] Compiling Gravity          ( Gravity.hs, Gravity.o )
[3 of 4] Compiling Utility          ( Utility.hs, Utility.o )
[4 of 4] Compiling Main             ( ScalarTriangle.hs, ScalarTriangle.o )
Linking ScalarTriangle ...
$ ./ScalarTriangle +RTS -p
(- m**3 + m * q2 + 3 * m**3 * y - m * q2 * y - 3 * m**3 * y**2 + m**3 * y**3
+ 3 * m**3 * z - m * q2 * z - 6 * m**3 * y * z + 3 * m * q2 * y * z +3 * m**3 *
y**2 * z - m * q2 * y**2 * z - 3 * m**3 * z**2 + 3 * m**3 * y * z**2 - m * q2
* y * z**2 + m**3 * z**3) * g (-rho) (-sigma) * (diracConjugate u) * u ...
```

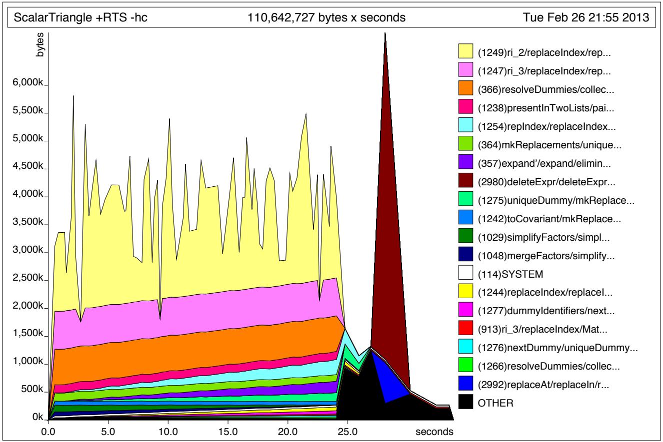
A Profile

```
    Wed Feb 13 16:53 2013 Time and Allocation Profiling Report  (Final)

           ScalarTriangle +RTS -p -RTS

        total time  =        5.80 secs   (290 ticks @ 20 ms)
        total alloc = 16,173,278,128 bytes  (excludes profiling overheads)

COST CENTRE  MODULE                                %time %alloc

ri_1         Math.Symbolic.Wheeler.DummyIndices  40.0    0.0
ri_2         Math.Symbolic.Wheeler.DummyIndices  37.2   70.9
replaceIndex Math.Symbolic.Wheeler.DummyIndices  11.4    0.0
ri_3         Math.Symbolic.Wheeler.DummyIndices   4.1    5.7
deleteExpr   Math.Symbolic.Wheeler.Replacer       0.7    6.4
repSpaces    Math.Symbolic.Wheeler.Expr           0.7    5.7
replaceAt    Math.Symbolic.Wheeler.Replacer       0.3    2.1
groupExprs   Math.Symbolic.Wheeler.Canonicalize   0.3    1.4
subExprs     Math.Symbolic.Wheeler.Matcher2       0.0    1.2
productMatch Math.Symbolic.Wheeler.Matcher2       0.0    1.1
```

Another Profile

```
$ ./ScalarTriangle +RTS -hc
(- m**3 + m * q2 + 3 * m**3 * y - m * q2 * y - ...
$ hp2ps -c ScalarTriangle.hp > ScalarTriangle.ps
```

Another Profile



Status

So what's the status of the library?

- It is being actively developed.
- Able to tackle real problems in a limited domain.
- Still has performance issues.
- How to make it easily extensible is still an open question.

Status

So what's the status of the library?

- It is being actively developed.
- Able to tackle real problems in a limited domain.
- Still has performance issues.
- How to make it easily extensible is still an open question.

Status

So what's the status of the library?

- It is being actively developed.
- Able to tackle real problems in a limited domain.
- Still has performance issues.
- How to make it easily extensible is still an open question.

Status

So what's the status of the library?

- It is being actively developed.
- Able to tackle real problems in a limited domain.
- Still has performance issues.
- How to make it easily extensible is still an open question.

Status

So what's the status of the library?

- It is being actively developed.
- Able to tackle real problems in a limited domain.
- Still has performance issues.
- How to make it easily extensible is still an open question.

Break!

The Expression Problem

The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts).

– Philip Wadler, 1998

The Expression Problem

The expression problem is important for symbolic mathematics because in a perfect world, we could write a small library core and extend it smoothly, adding new mathematical objects. For example, we may want to extend the operations of addition and multiplication to vectors and matrices.

Expressions, again

```
-- The Expr data type:
--
data Expr where
    Const     :: Numeric -> Expr
    Applic    :: Function -> Expr -> Expr
    Symbol    :: Symbol -> Expr
    Sum       :: [ Expr ] -> Expr
    Product   :: [ Expr ] -> Expr
    Power     :: Expr -> Expr -> Expr
    Undefined :: Expr
```

Symbols

```haskell
data Symbol = Simple S
            | Indexed I
            | Tensor T
            | DiracSpinor D
```

To extend the `Symbol` type requires editing the source and recompiling. Can we avoid this?

Three Approaches to the Expression Problem

- The universal type
- Final interpreter representation
- Coproducts, or “Data Types à la Carte”

Three Approaches to the Expression Problem

- The universal type
- Final interpreter representation
- Coproducts, or “Data Types à la Carte”

Three Approaches to the Expression Problem

- The universal type
- Final interpreter representation
- Coproducts, or “Data Types à la Carte”

Three Approaches to the Expression Problem

- The universal type
- Final interpreter representation
- Coproducts, or “Data Types à la Carte”

Final interpreters

A final interpreter replaces a data constructor with a function.

The idea is the that instead of encoding the syntax of an expression (the "initial" form) we represent it by application of semantic functions that carry out the intended operations.

Final interpreters

Instead of

```
-- A simplified Expr data type:
--
data Expr where
    Literal :: Int  -> Expr
    Add     :: Expr -> Expr -> Expr
```

Use

```
class Expr repr where
    literal :: Int  -> repr
    add     :: repr -> repr -> repr
```

Final interpreters

We need instances that carry out the operations:

```
instance Expr Int where
    literal n = n
    add x y   = x + y
```

Final interpreters

We can have more than one interpretation:

```
instance Expr String where
    literal n = show n
    add x y   = "(" ++ show x ++ " + " ++ show y ++ ")"
```

Final interpreters

The interpreter can be extended:

```
class MulExpr repr where
    mul     :: repr -> repr -> repr
```

With associated instances

```
instance MulExpr Int where
    mul x y   = x * y
instance MulExpr String where
    mul x y   = "(" ++ show x ++ " * " ++ show y ++ ")"
```

Final interpreters, Good and Bad News

The good news is that this all works, so far. Another piece of good news is that this approach preserves type inference: no additional type annotations are needed.

The bad news is that non-fold style processing is awkward (though the simplest examples are possible; see Oleg Kiselyov's article); it's not known how to automatically translate complex operations like the `canonicalize` function to a final interpreter.

Coproducts

Another way to solve the expression problem was proposed by Wouter Swierstra in his article, *Data Types à la Carte*. He keeps the data constructors but uses a coproduct of types signatures to build an extensible data type.

Coproducts

An example:

```
data Expr f = In (f (Expr f))

data Val e   = Val Int
data Add e   = Add e e

instance Functor Val where
  fmap f (Val x) = Val x

instance Functor Add where
  fmap f (Add e1 e2) = Add (f e1) (f e2)
```

Coproducts

The problem is to make the `f` in `Expr f` contain multiple types.

```
data (f :+: g) e = Inl (f e)
                 | Inr (g e)

instance (Functor f, Functor g) => Functor (f :+: g) where
  fmap f (Inl e1) = Inl (fmap f e1)
  fmap f (Inr e2) = Inr (fmap f e2)

foldExpr :: Functor f => (f a -> a) -> Expr f -> a
foldExpr f (In t) = f (fmap (foldExpr f) t)
```

Coproducts

Now it is possible to evaluate expressions:

```
class Functor f => Eval f where
  evalAlgebra :: f Int -> Int

instance Eval Val where
  evalAlgebra (Val x) = x
instance Eval Add where
  evalAlgebra (Add x y) = x + y
instance (Eval f, Eval g) => Eval (f :+: g) where
  evalAlgebra (Inl x) = evalAlgebra x
  evalAlgebra (Inr y) = evalAlgebra y

eval :: Eval f => Expr f -> Int
eval ex = foldExpr evalAlgebra ex

addExample :: Expr (Val :+: Add)
addExample = In (Inr (Add (In (Inl (Val 338))) (In (Inl (Val 1219)))))
```

Coproducts

Smart constructors can avoid some of the pain:

```
class (Functor sub, Functor sup) => sub :<: sup where
  inj :: sub a -> sup a

instance Functor f => f :<: f where
  inj = id
instance (Functor f, Functor g) => f :<: (f :+: g) where
  inj = Inl
instance (Functor f, Functor g, Functor h, f :<: g) => f :<: (h :+: g) where
  inj = Inr . inj

inject :: (g :<: f) => g (Expr f) -> Expr f
inject = In . inj

infix 6 &+
(&+) :: (Add :<: f) => Expr f -> Expr f -> Expr f
(&+) x y = inject (Add x y)

val  :: (Val :<: f) => Int -> Expr f
val x = inject (Val x)
```

Coproducts

The real payoff: the `inj` function has a partial inverse.

```
class (Functor sub, Functor sup) => sub :<: sup where
  inj :: sub a -> sup a
  prj :: sup a -> Maybe (sub a)
```

Coproducts

Now we can tackle something closer to our real problem:

```
match :: (g :<: f) => Expr f -> Maybe (g (Expr f))
match (In t) = prj t

distrib :: (Add :<: f, Mul :<: f) => Expr f -> Maybe (Expr f)
distrib t = do
    Mul a b <- match t
    Add c d <- match b
    return (a &* c &+ a &* d)
```

The Expression Problem in Symbolic Mathematics

Getting closer, but still not there yet.

Perhaps scaling back our ambitions is in order: could we live with, say, the latest extensible record techniques – just introducing new mathematical objects – and give up on introducing additional operations?

Further Reading

J. S. Cohen.
Computer Algebra and Symbolic Computation: Elementary Algorithms.
A.K. Peters, 2002.

J. S. Cohen.
Computer Algebra and Symbolic Computation: Mathematical Methods.
A.K. Peters, 2003.

O. Kiselyov.
Typed Tagless Final Interpreters.
Lecture Notes in Computer Science 7470, pp. 130–174, 2012, and
http://okmij.org/ftp/tagless-final/course/index.html

W. Swierstra.
Data Types à la Carte.
Journal of Functional Programming, 18(4):423–436, 2008.