From structured theories to efficient code in 6 easy steps

Jacques Carette

contains work (past and present) done with William M. Farmer, Russell O'Connor, Spencer Smith, Mustafa Elsheik, Oleg Kiselyov, Ken Shan and Tom Andersen.

McMaster University

JLM – Friday March 9, 2012

Context and goals

MathScheme: a Mechanized Mathematics System

- Join Symbolic Computation, Computer Algebra and Theorem Proving

Context and goals

MathScheme: a Mechanized Mathematics System

- Join Symbolic Computation, Computer Algebra and Theorem Proving

One of our goals:

- Produce efficient, generic, provably correct code

Context and goals

MathScheme: a Mechanized Mathematics System

- Join Symbolic Computation, Computer Algebra and Theorem Proving

One of our goals:

- Produce efficient, generic, provably correct code

Method:

- Generative

Context and goals

MathScheme: a Mechanized Mathematics System

- Join Symbolic Computation, Computer Algebra and Theorem Proving

One of our goals:

- Produce efficient, generic, provably correct code

Method:

- Generative
- Leverage known information

The 6 steps

$$
\begin{array}{c}
\text{Colimit of diagrams, fibered functors} \\
\Downarrow \\
\text{Concepts and theory combinators} \\
\Downarrow \\
\text{Algebraic theories} \\
\Downarrow \\
\text{Biform theories} \\
\Downarrow \\
\text{Generic algorithm} \\
\Downarrow \\
\text{Library} \\
\Downarrow \\
\text{Efficient Program}
\end{array}
$$

The 6 steps

Colimit of diagrams, fibered functors

$\Downarrow$

Concepts and theory combinators

$\Downarrow$

Algebraic theories

$\Downarrow$

Biform theories

$\Downarrow$

Generic algorithm

$\Downarrow$

Library

$\Downarrow$

Efficient Program

Generic efficiency

$$\text{Generic algorithms} \xRightarrow{\text{instantiate}} \text{Library} \xRightarrow{\textit{use}} \text{Efficient Program}$$

ex: LAPACK, NAG. SPIRAL.

Generic efficiency

$$\text{Generic algorithms} \overset{\text{instantiate}}{\Longrightarrow} \text{Library} \overset{\textit{use}}{\Longrightarrow} \text{Efficient Program}$$

ex: LAPACK, NAG. SPIRAL.

$$\text{Library of generic algorithms} \overset{\text{instantiate by use}}{\Longrightarrow} \text{Efficient Program}$$

ex: `C++` Templates. BOOST++, Linbox.

Generic efficiency

$$\text{Generic algorithms} \overset{\text{instantiate}}{\Longrightarrow} \text{Library} \overset{\textit{use}}{\Longrightarrow} \text{Efficient Program}$$

ex: LAPACK, NAG. SPIRAL.

$$\text{Library of generic algorithms} \overset{\text{instantiate by use}}{\Longrightarrow} \text{Efficient Program}$$

ex: `C++` Templates. BOOST++, Linbox.

$$\text{Library of generic algorithm interfaces} \overset{\text{dispatch by use}}{\Longrightarrow} \text{Efficient? Program}$$

ex: Maple's Linear Algebra.

Generic efficiency

$$\text{Generic algorithms} \overset{\text{instantiate}}{\Longrightarrow} \text{Library} \overset{\textit{use}}{\Longrightarrow} \text{Efficient Program}$$

ex: LAPACK, NAG. SPIRAL.

$$\text{Library of generic algorithms} \overset{\text{instantiate by use}}{\Longrightarrow} \text{Efficient Program}$$

ex: `C++` Templates. BOOST++, Linbox.

$$\text{Library of generic algorithm interfaces} \overset{\text{dispatch by use}}{\Longrightarrow} \text{Efficient? Program}$$

ex: Maple's Linear Algebra.

$$\text{Library of generic algorithm} \overset{\text{dynamic dispatch}}{\Longrightarrow} \text{Slow Program}$$

ex: Maple's Domains, most OO languages.

Generating extensions

$$\text{Library of generic algorithms} \xrightarrow{\text{resolve}} \text{Program}$$

Resolve at compile time: instantiate. Resolve at run time: dispatch.

Build a typed generating extension:

Generating extensions

$$\text{Library of generic algorithms} \xRightarrow{\text{resolve}} \text{Program}$$

Resolve at compile time: instantiate. Resolve at run time: dispatch.

Build a typed generating extension:

1. Make your algorithms generic

Generating extensions

$$\text{Library of generic algorithms} \xrightarrow{\text{resolve}} \text{Program}$$

Resolve at compile time: instantiate. Resolve at run time: dispatch.

Build a typed generating extension:

1. Make your algorithms generic
2. Virtualize your core language (can do in ML, Haskell, Scala)

Generating extensions

$$\text{Library of generic algorithms} \overset{\text{resolve}}{\Longrightarrow} \text{Program}$$

Resolve at compile time: instantiate. Resolve at run time: dispatch.

Build a typed generating extension:

1. Make your algorithms generic
2. Virtualize your core language (can do in ML, Haskell, Scala)
3. Instantiate with interpreter, compiler and partial evaluator,...

Generating extensions

$$\text{Library of generic algorithms} \xrightarrow{\text{resolve}} \text{Program}$$

Resolve at compile time: instantiate. Resolve at run time: dispatch.

Build a typed generating extension:

1. Make your algorithms generic
2. Virtualize your core language (can do in ML, Haskell, Scala)
3. Instantiate with interpreter, compiler and partial evaluator,...
4. Carefully track static vs dynamic information (staging)

Generating extensions

$$\text{Library of generic algorithms} \overset{\text{resolve}}{\Longrightarrow} \text{Program}$$

Resolve at compile time: instantiate. Resolve at run time: dispatch.

Build a typed generating extension:

1. Make your algorithms generic
2. Virtualize your core language (can do in ML, Haskell, Scala)
3. Instantiate with interpreter, compiler and partial evaluator,...
4. Carefully track static vs dynamic information (staging)

Theory for 2–3:

J. Carette, O. Kiselyov, C.-c. Shan, *Finally tagless, partially evaluated: tagless staged interperters for simpler typed languages*, Journal of Functional Programming, 19 (5):509–543, 2009.

Examples:

J. Carette, M. Elsheik, S. Smith. *Generative Geometric Kernel*, PEPM 2011.

J. Carette, O. Kiselyov, *Multi-stage programming with functors and monads: eliminating abstraction overhead from generic code*, Science of Computer Programming, 76(5):53–62, 2011.

Values, code and syntax

Domain	3	is really	write as
value	3	$\{\{\},\{\{\}\},\{\{\},\{\{\}\}\}\}$	
code	3	00000011	
syntax	3	trois	

Values, code and syntax

Domain	3	is really	write as
value	3	$\{\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}\}$	3
code	3	00000011	$.\!< 3 >\!.$
syntax	3	trois	$\ulcorner 3 \urcorner$

Values, code and syntax

Domain	3	is really	write as
value	3	$\{\{\},\{\{\}\},\{\{\},\{\{\}\}\}\}$	3
code	3	00000011	$.\!< 3 >\!.$
syntax	3	trois	$\ulcorner 3 \urcorner$

Equations: $1+2=3$, $.\!< 1{+}2 >\!. \neq$ `.< 3 >.`, $\ulcorner 1+2 \urcorner \neq \ulcorner 3 \urcorner$,

Values, code and syntax

Domain	3	is really	write as
value	3	$\{\{\},\{\{\}\},\{\{\},\{\{\}\}\}\}$	3
code	3	00000011	$.\!<3>\!.$
syntax	3	trois	$\ulcorner 3 \urcorner$

Equations: $1+2=3$, $.\!<\text{1+2}>\!. \neq$ `.< 3 >.`, $\ulcorner 1+2 \urcorner \neq \ulcorner 3 \urcorner$,

but: `run .< 1+2 >.` $=$ `run .< 3 >.`, $[\![\ulcorner 1+2 \urcorner]\!] = [\![\ulcorner 3 \urcorner]\!]$
Furthermore: code is **opaque**, syntax is **not**.

A Maple analogy: $2*3$ vs $() -> 2*3$ vs $2 \mathbin{\&*} 3$

Values, code and syntax

Domain	3	is really	write as
value	3	$\{\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}\}$	3
code	3	00000011	$.< 3 >.$
syntax	3	trois	$\ulcorner 3 \urcorner$

Equations: $1+2=3$, $.< 1{+}2 >. \neq .< 3 >.$, $\ulcorner 1+2 \urcorner \neq \ulcorner 3 \urcorner$,

but: `run .< 1+2 >.` $=$ `run .< 3 >.`, $[\![\ulcorner 1+2 \urcorner]\!] = [\![\ulcorner 3 \urcorner]\!]$
Furthermore: code is **opaque**, syntax is **not**.

A Maple analogy: $2 * 3$ vs $() -> 2 * 3$ vs $2 \mathbin{\&*} 3$

Code generation: syntax $\Rightarrow$ code.
Staged program: manipulates both values and code.

```
type ('a,'b) staged = Now of 'b | Later of ('a,'b) code
```

Going generic, then generative

```
let rec norm = function
    | [] -> 0.
    | x::xs -> abs_float x +. norm xs
> val norm : float list -> float = <fun>
```

Going generic, then generative

```
let rec norm = function
  | [] -> 0.
  | x::xs -> abs_float x +. norm xs
```

Generalize the type of “numbers” to a “normed set” NS over an arbitrary commutative monoid CM.

```
let rec norm = function
  | [] -> NS.CM.zero
  | x::xs -> NS.CM.plus (NS.norm x) (norm xs)
> val norm : NS.n list -> NS.CM.n = <fun>
```

Going generic, then generative

```
let rec norm = function
   | [] -> 0.
   | x::xs -> abs_float x +. norm xs
```

Lift from the normed set type to a staged version

```
let rec norm = function
   | [] -> Staged.of_immediate NS.CM.zero
   | x::xs -> NS.CM.plus_s (NS.norm_s x) (norm xs)
> val norm : ('a,NS.n) staged list -> ('a,NS.CM.n) staged
```

Going generic, then generative

```
let rec norm = function
  | [] -> 0.
  | x::xs -> abs_float x +. norm xs
```

`norm` is parametric in `NS`:

```
module GenericNorm (NS : NORMED_SET) = struct
  let rec norm = function
   | [] -> Staged.of_immediate NS.CM.zero
   | x::xs -> NS.CM.plus_s (NS.norm_s x) (norm xs)
end
```

Going generic, then generative

```
let rec norm = function
    | [] -> 0.
    | x::xs -> abs_float x +. norm xs
```

Abstracting the container (recursion pattern is a mapfold, also known as MapReduce; repeat previous 3 steps):

```
module GenericNorm(NS : NORMED_SET)
          (C: FOLDABLE with t = CM.n) = struct
  let norm = C.mapfold(NS.norm_s)(NS.CM.plus_s)
                  (Staged.of_immediate NS.CM.zero)
end
```

Now generic over: container, normed set and stage.

Going generic, then generative

```
let rec norm = function
  | [] -> 0.
  | x::xs -> abs_float x +. norm xs
```

Collecting variabilities:

```
module type NORM_VAR = sig
  module NS : NORMED_SET
  module C : FOLDABLE with t = NS.n
end
```

and build a generator:

```
module GenNorm (NV : NORM_VAR) = struct
 let gen_norm () =
  let module GP = GenericNorm(NV.NS)(NV.C) in
   .< fun x -> .~(Staged.to_code
       (GP.norm (Staged.of_atom .<x>.))) >.
end
```

Going generic, then generative

```
let rec norm = function
    | [] -> 0.
    | x::xs -> abs_float x +. norm xs
```

or run it now:

```
module Norm (NV : NORM_VAR) = struct
 let norm x =
  let module GP = GenericNorm(NV.NS)(NV.C) in
  GP.norm (Staged.of_immediate x)
end
```

Going generic, then generative

```
let rec norm = function
  | [] -> 0.
  | x::xs -> abs_float x +. norm xs
```

Generated code example with a generic 3-tuple container:

.< **fun** x −> **let** (a,b,c) = x in abs a + abs b + abs c >.

Going generic, then generative

```
let rec norm = function
    | [] -> 0.
    | x::xs -> abs_float x +. norm xs
```

Generated code example with a generic 3-tuple container:

```
.< fun x -> let (a,b,c) = x in abs a + abs b + abs c >.
```

Or on the explicit list [1.; 0.; 3.; .<y>.; .<z>.]

```
.<4. + abs_float y + abs_float z>.
```

(which is "just" symbolic computation!)

Going generic, then generative

```
let rec norm = function
  | [] -> 0.
  | x::xs -> abs_float x +. norm xs
```

The key “simplification” rules ($(*, 1)$ monoid, $(+, -, 0)$ group):

```
let monoid one bnow blater x y = match x, y with
  | (Now a), b when a = one -> b
  | a, (Now b) when b = one -> a
  | _                       -> binary bnow blater x y

let inverse zero uinv bnow blater x y = match x, y with
  | (Now a), b when a = zero -> uinv b
  | a, (Now b) when b = zero -> a
  | _                        -> binary bnow blater x y
```

Example: Generative Geometric Kernel (GGK)

High-level picture:

Geometric Objects
Affine Space
Linear Algebra
Number Types / Abstract Algebra
Code

Example: Generative Geometric Kernel (GGK)

Drill-down:

	Orientation	Inside	Simplex
Geometric	Vertex	Sidedness	Hypersphere
Objects	Insphere	Hyperplane Operations	Hyperplane

Affine Space	Vector	Affine Transforms	Point

Linear Algebra	Tuple	Matrix	Determinant

Algebra	RealField	Order	Field
	Set	Ring	Monoid

Code	Staged Types
	Base Types

Example: Generative Geometric Kernel (GGK)

Architecture:

Generic Algorithms
Symbolic Computation
Virtualized Base Language

Example: Generative Geometric Kernel (GGK)

Architecture:

Generic Algorithms
Polymorphic over data rep., domain, stage, ...
Symbolic Computation
Symbol shuffling with a purpose
Virtualized Base Language
staged, overloaded language w/ simplification

Colimit of diagrams, fibered functors

$\Downarrow$

Concepts and theory combinators

$\Downarrow$

Algebraic theories

$\Downarrow$

Biform theories

$\Downarrow$

Generic algorithm

$\Downarrow$

Library

$\Downarrow$

Efficient Program

Algebraic Theories

```
Monoid := Theory {
  U : type;
  * : (U, U) -> U;
  e : U;
  axiom rightIdentity_*_e : forall x:U. x*e = x;
  axiom leftIdentity_*_e : forall x:U. e*x = x;
  axiom associative_* : forall x,y,z:U. (x*y)*z=x*(y*z)}
```

Algebraic Theories

```
Monoid := Theory {
  U : type;
  * : (U, U) -> U;
  e : U;
  axiom rightIdentity_*_e : forall x:U. x*e = x;
  axiom leftIdentity_*_e : forall x:U. e*x = x;
  axiom associative_* : forall x,y,z:U. (x*y)*z=x*(y*z)}

CommutativeMonoid := Theory {
  U : type;
  * : (U, U) -> U;
  e : U;
  axiom rightIdentity_*_e : forall x:U. x*e = x;
  axiom leftIdentity_*_e : forall x:U. e*x = x;
  axiom associative_* : forall x,y,z:U. (x*y)*z=x*(y*z)
  axiom commutative_* : forall x,y,z:U. x*y=y*x}
```

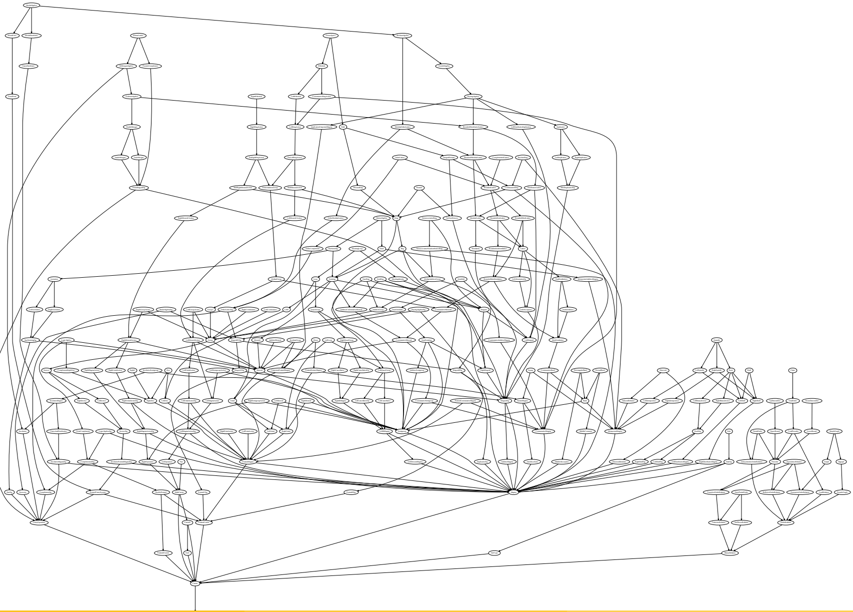
Algebraic Theories

```
Monoid := Theory {
  U : type;
  * : (U, U) -> U;
  e : U;
  axiom rightIdentity_*_e : forall x:U. x*e = x;
  axiom leftIdentity_*_e : forall x:U. e*x = x;
  axiom associative_* : forall x,y,z:U. (x*y)*z=x*(y*z)}

AdditiveMonoid := Theory {
  U : type;
  + : (U, U) -> U;
  0 : U;
  axiom rightIdentity_+_0 : forall x:U. x+0 = x;
  axiom leftIdentity_+_0 : forall x:U. 0+x = x;
  axiom associative_+ : forall x,y,z:U. (x+y)+z=x+(y+z) }
```

Algebraic Theories

```
Monoid := Theory {
  U : type;
  * : (U, U) -> U;
  e : U;
  axiom rightIdentity_*_e : forall x:U. x*e = x;
  axiom leftIdentity_*_e : forall x:U. e*x = x;
  axiom associative_* : forall x,y,z:U. (x*y)*z=x*(y*z)}

AdditiveCommutativeMonoid := Theory {
  U : type;
  + : (U, U) -> U;
  0 : U;
  axiom rightIdentity_+_0 : forall x:U. x+0 = x;
  axiom leftIdentity_+_0 : forall x:U. 0+x = x;
  axiom associative_+ : forall x,y,z:U. (x+y)+z=x+(y+z)
  axiom commutative_+ : forall x,y,z:U. x+y=y+x}
```

A fraction of the Algebraic Zoo



Combinators for theories

Extension:

```
CommutativeMonoid := Monoid extended by {
  axiom commutative_* : forall x,y,z:U. x*y=y*x}
```

Combinators for theories

Extension:

```
CommutativeMonoid := Monoid extended by {
  axiom commutative_* : forall x,y,z:U. x*y=y*x}
```

Renaming:

```
AdditiveMonoid := Monoid[ * |-> +, e |-> 0 ]
```

Combinators for theories

Extension:

```
CommutativeMonoid := Monoid extended by {
  axiom commutative_* : forall x,y,z:U. x*y=y*x}
```

Renaming:

```
AdditiveMonoid := Monoid[ * |-> +, e |-> 0 ]
```

Combination:

```
AdditiveCommutativeMonoid :=
  combine AdditiveMonoid, CommutativeMonoid over Monoid
```

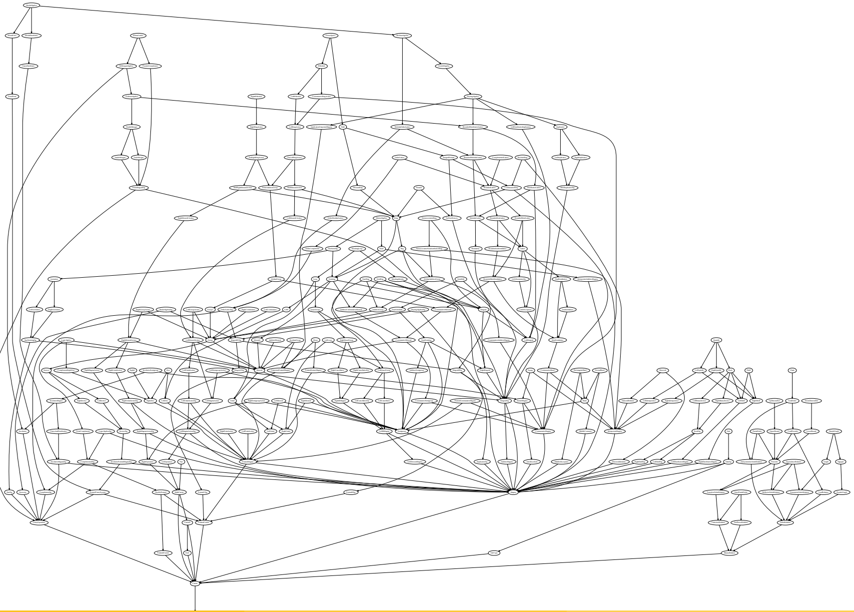
Library fragment 1

```
MoufangLoop := combine Loop, MoufangIdentity over Magma
LeftShelfSig := Magma[ * |-> |> ]
LeftShelf := LeftDistributiveMagma [ * |-> |> ]
RightShelfSig := Magma[ * |-> <| ]
RightShelf := RightDistributiveMagma[ * |-> <| ]
RackSig := combine LeftShelfSig, RightShelfSig over Carrier
Shelf := combine LeftShelf, RightShelf over RackSig
LeftBinaryInverse := RackSig extended by {
    axiom leftInverse_|>_<| : forall x,y:U. (x |> y) <| x = y }
RightBinaryInverse := RackSig extended by {
    axiom rightInverse_|>_<| : forall x,y:U. x |> (y <| x) = y }
Rack := combine RightShelf, LeftShelf, LeftBinaryInverse,
    RightBinaryInverse over RackSig
LeftIdempotence := IdempotentMagma[ * |-> |> ]
RightIdempotence := IdempotentMagma[ * |-> <| ]
LeftSpindle := combine LeftShelf, LeftIdempotence over LeftShelfSig
RightSpindle := combine RightShelf, RightIdempotence over RightShelfSig
Quandle := combine Rack, LeftSpindle, RightSpindle over Shelf
```

Library fragment 2

```
NearSemiring := combine AdditiveSemigroup, Semigroup, RightRingoid ove
NearSemifield := combine NearSemiring, Group over Semigroup
Semifield := combine NearSemifield, LeftRingoid over RingoidSig
NearRing := combine AdditiveGroup, Semigroup, RightRingoid over Ringoi
Rng := combine AbelianAdditiveGroup, Semigroup, Ringoid over RingoidSi
Semiring := combine AdditiveCommutativeMonoid, Monoid1, Ringoid, Left0
SemiRng := combine AdditiveCommutativeMonoid, Semigroup, Ringoid over
Dioid := combine Semiring, IdempotentAdditiveMagma over AdditiveMagma
Ring := combine Rng, Semiring over SemiRng
CommutativeRing := combine Ring, CommutativeMagma over Magma
BooleanRing := combine CommutativeRing, IdempotentMagma over Magma
NoZeroDivisors := Ringoid0Sig extended by {
  axiom onlyZeroDivisor_*_0: forall x:U.
    ((exists b:U. x*b = 0) and (exists b:U. b*x = 0)) implies (x = 0) 
Domain := combine Ring, NoZeroDivisors over Ringoid0Sig
IntegralDomain := combine CommutativeRing, NoZeroDivisors over Ringoid
DivisionRing := Ring extended by {
  axiom divisible : forall x:U. not (x=0) implies
    ((exists! y:U. y*x = 1) and (exists! y:U. x*y = 1)) }
Field := combine DivisionRing, IntegralDomain over Ring
```

The Algebraic Zoo again



Colimit of diagrams, fibered functors

$\Downarrow$

Concepts and theory combinators

$\Downarrow$

Algebraic theories

$\Downarrow$

Biform theories

$\Downarrow$

Generic algorithm

$\Downarrow$

Library

$\Downarrow$

Efficient Program

A little theory

Given some dependent type theory, its category of contexts $\mathbb{C}$ has objects

$$\Gamma := \langle x_0 : \sigma_0; \ldots; x_{n-1} : \sigma_{n-1} \rangle ,$$

such that for each $i < n$ the judgement

$$\langle x_0 : \sigma_0; \ldots; x_{i-1} : \sigma_{i-1} \rangle \vdash \sigma_i : \mathrm{Type}\ (\text{or } : \mathrm{Prop})$$

holds. A morphism $\Gamma \to \Delta (= \langle y : \sigma \rangle_0^{m-1})$ is an assignment (substitution) $[y_0 \mapsto t_0, \ldots, y_m \mapsto t_{m-1}]$ such that

$$\Gamma \vdash t_0 : \tau_0 \qquad \ldots \qquad \Gamma \vdash t_{m-1} : \tau_{m-1}\,[y \mapsto t]_0^{m-2}$$

$$\begin{tikzcd}
\Gamma^+ \arrow[r, "f^+"] \arrow[d, "A"'] & \Delta^+ \arrow[d, "B"] \\
\Gamma \arrow[r, "f^-"'] & \Delta
\end{tikzcd}$$

Definition

The category of general extensions $\mathbb{E}$ has all general extensions from $\mathbb{B}$ as objects, and given two general extensions $A : \Gamma^+ \to \Gamma$ and $B : \Delta^+ \to \Delta$, an arrow $f : A \to B$ is a commutative square from $\mathbb{B}$.

and just a bit more theory

Theorem

The functor $\mathsf{cod} : \mathbb{E} \to \mathbb{B}$ *is a fibration.*

and just a bit more theory

Theorem

The functor $\mathrm{cod} : \mathbb{E} \to \mathbb{B}$ *is a fibration.*

$$
\begin{array}{rlrlrl}
a, b, c & \in \text{labels} & \tau & \in \text{Type} & \text{tpc} ::= & \text{extend } A \text{ by } \{l\} \\
A, B, C & \in \text{names} & k & \in \text{Kind} & \mid & \text{combine } A\ r_1,\ B\ r_2 \\
l & \in \text{judgments}^* & t & \in \text{Term} & \mid & A\ ;\ B \\
r & \in (a_i \mapsto b_i)^* & \theta & \in \text{Prop} & \mid & A\ r \\
 & & & & \mid & \text{Empty} \\
 & & & & \mid & \text{Theory } \{l\}
\end{array}
$$

and just a bit more theory

Theorem

The functor $\mathrm{cod} : \mathbb{E} \to \mathbb{B}$ *is a fibration.*

$$\begin{aligned}
[\![-]\!]_{\mathbb{B}} &: \mathsf{tpc} \rightharpoonup |\mathbb{B}| \\
[\![\mathsf{Empty}]\!]_{\mathbb{B}} &= \langle\,\rangle \\
[\![\mathsf{Theory}\ \{l\}]\!]_{\mathbb{B}} &\cong \langle l \rangle \\
[\![A\ r]\!]_{\mathbb{B}} &= [\![r]\!]_{\pi} \cdot [\![A]\!]_{\mathbb{B}} \\
[\![A; B]\!]_{\mathbb{B}} &= [\![B]\!]_{\mathbb{B}} \\
[\![\mathsf{extend}\ A\ \mathsf{by}\ \{l\}]\!]_{\mathbb{B}} &\cong [\![A]\!]_{\mathbb{B}} \mathbin{;} \langle l \rangle \\
[\![\mathsf{combine}\ A_1 r_1, A_2 r_2]\!]_{\mathbb{B}} &\cong D
\end{aligned}$$

$$\begin{tikzcd}
D \arrow[r, "{[\![r_1]\!]_\pi \circ \delta_{A_1}}"] \arrow[d] \arrow[dr, phantom, "{[\![r_2]\!]_\pi \circ \delta_{A_2}}"] & A_1 \arrow[d, "\delta_A"] \\
A_2 \arrow[r, "\delta_A"'] & A
\end{tikzcd}$$

Theorem

The functor $\mathrm{cod} : \mathbb{E} \to \mathbb{B}$ *is a fibration.*

$$
\begin{aligned}
[\![-]\!]_{\mathbb{E}} &: \ \mathsf{tpc} \rightharpoonup |\mathbb{E}| \\
[\![\mathsf{Empty}]\!]_{\mathbb{E}} &= \ \mathsf{id}_{\langle\,\rangle} \\
[\![\mathsf{Theory}\ \{l\}]\!]_{\mathbb{E}} &\cong \ !_{\langle l \rangle} \\
[\![A\ r]\!]_{\mathbb{E}} &= \ [\![r]\!]_{\pi} \cdot [\![A]\!]_{\mathbb{E}} \\
[\![A; B]\!]_{\mathbb{E}} &= \ [\![A]\!]_{\mathbb{E}} \circ [\![B]\!]_{\mathbb{E}} \\
[\![\mathsf{extend}\ A\ \mathsf{by}\ \{l\}]\!]_{\mathbb{E}} &\cong \ \delta_A \\
[\![\mathsf{combine}\ A_1 r_1, A_2 r_2]\!]_{\mathbb{E}} &\cong \ [\![r_1]\!]_{\pi} \circ \delta_{T_1} \circ [\![A_1]\!]_{\mathbb{E}} \\
&\cong \ [\![r_2]\!]_{\pi} \circ \delta_{T_2} \circ [\![A_2]\!]_{\mathbb{E}}
\end{aligned}
$$

$$
\begin{tikzcd}
D \arrow[r, "{[\![r_1]\!]_{\pi} \circ \delta_{T_1}}"] \arrow[d, "{[\![r_2]\!]_{\pi} \circ \delta_{T_2}}"'] & T_1 \arrow[d, "A_1"] \\
T_2 \arrow[r, "A_2"'] & T
\end{tikzcd}
$$

and just a bit more theory

Theorem

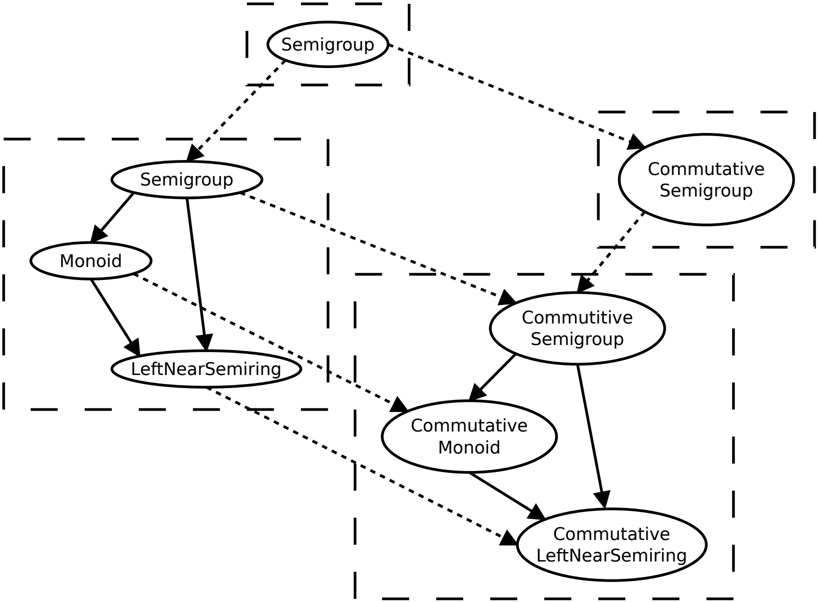
The functor cod : $\mathbb{E} \to \mathbb{B}$ *is a fibration.*

$$
\begin{aligned}
[\![-]\!]_{\mathbb{E}} &: \text{ tpc} \rightharpoonup |\mathbb{E}| \\
[\![\text{Empty}]\!]_{\mathbb{E}} &= \text{ id}_{\langle\,\rangle} \\
[\![\text{Theory } \{l\}]\!]_{\mathbb{E}} &\cong \,!_{\langle l \rangle} \\
[\![A\ r]\!]_{\mathbb{E}} &= [\![r]\!]_{\pi} \cdot [\![A]\!]_{\mathbb{E}} \\
[\![A; B]\!]_{\mathbb{E}} &= [\![A]\!]_{\mathbb{E}} \circ [\![B]\!]_{\mathbb{E}} \\
[\![\text{extend } A \text{ by } \{l\}]\!]_{\mathbb{E}} &\cong \delta_A \\
[\![\text{combine } A_1 r_1, A_2 r_2]\!]_{\mathbb{E}} &\cong [\![r_1]\!]_{\pi} \circ \delta_{T_1} \circ [\![A_1]\!]_{\mathbb{E}} \\
&\cong [\![r_2]\!]_{\pi} \circ \delta_{T_2} \circ [\![A_2]\!]_{\mathbb{E}}
\end{aligned}
$$

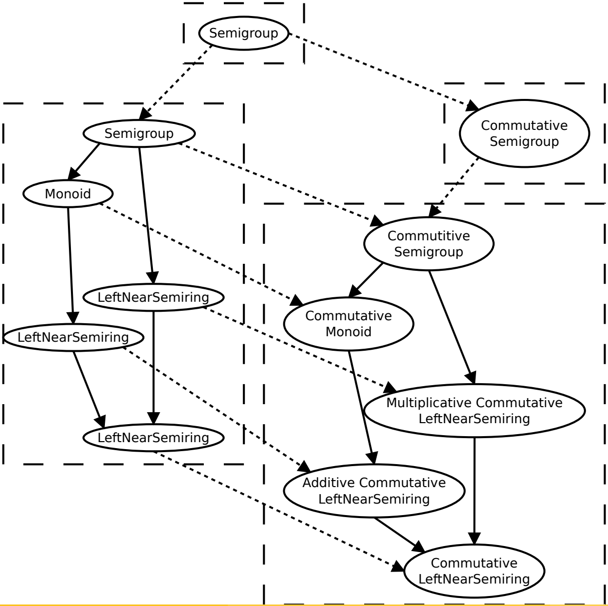
$$
\begin{tikzcd}
D \arrow[r, "{[\![r_1]\!]_{\pi} \circ \delta_{T_1}}"] \arrow[d] \arrow[dr, phantom, "{[\![r_2]\!]_{\pi} \circ \delta_{T_2}}"] & T_1 \arrow[d, "A_1"] \\
T_2 \arrow[r, "A_2"'] & T
\end{tikzcd}
$$

J. Carette, R. O'Connor *Theory Presentation Combinators*, submitted to Calculemus 2012.

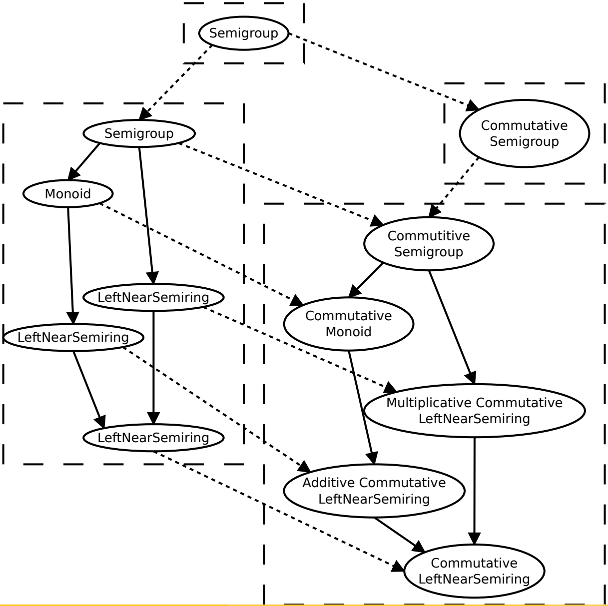
More structure



More structure

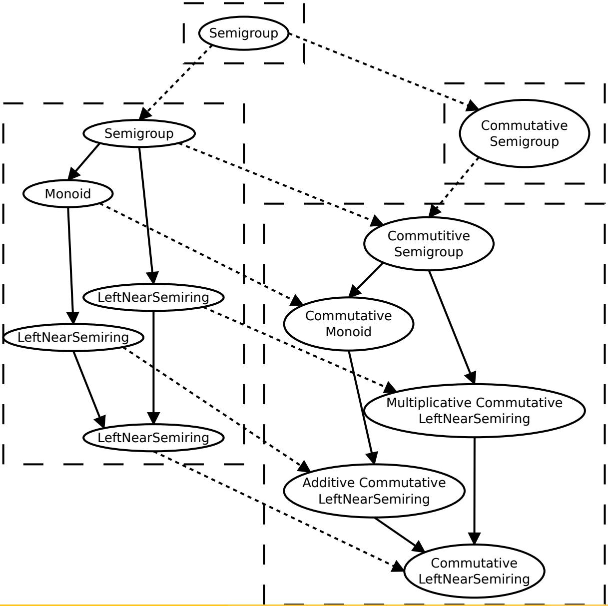


More structure



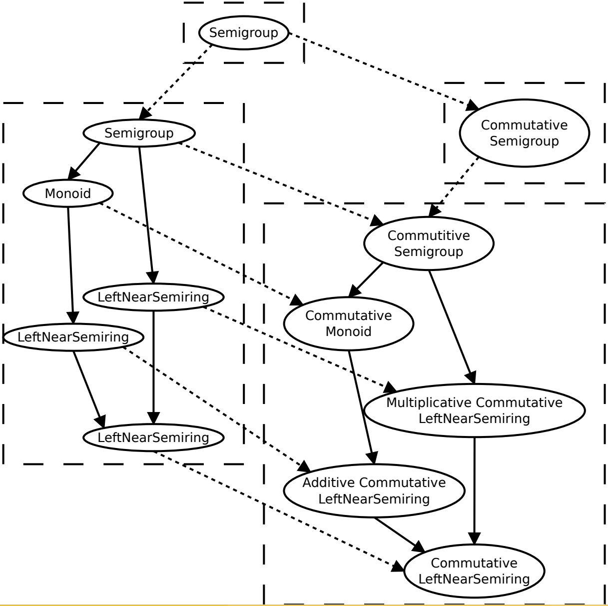
- Order

More structure



- Order
- Partial operations

More structure



- Order
- Partial operations
- Constructive equality

Colimit of diagrams, fibered functors

$\Downarrow$

Concepts and theory combinators

$\Downarrow$

Algebraic theories

$\Downarrow$

Biform theories

$\Downarrow$

Generic algorithm

$\Downarrow$

Library

$\Downarrow$

Efficient Program

Biform monoids

```
Monoid := Theory {
  U : type;
  e : U;
  * : (U, U) -> U;
  ax: forall x:U. e*x = x;
  ax: forall x:U. x*e = x;
  ax: forall x,y,z:U.(x*y)*z=x*(y*z)}
```

Syntax (remember ⌜3⌝ ?)

```
MonoidTerm := Theory {
  type MTerm = (data X .
    #e : X |
    #* : (X, X) -> X)
}
```

Biform monoids

```
Monoid := Theory {
  U : type;
  e : U;
  * : (U, U) -> U;
  ax: forall x:U. e*x = x;
  ax: forall x:U. x*e = x;
  ax: forall x,y,z:U.(x*y)*z=x*(y*z)}
```

Syntax (remember ⌜3⌝ ?)

```
MonoidTerm := Theory {
  type MTerm = (data X .
    #e : X |
    #* : (X, X) -> X)
}
```

Biform Theory: axiomatic + syntax + transformers.

```
length :: MonoidTerm -> Nat
length trm = gfold (+) 1 trm
```

Biform monoids

```
Monoid := Theory {
  U : type;
  e : U;
  * : (U, U) -> U;
  ax: forall x:U. e*x = x;
  ax: forall x:U. x*e = x;
  ax: forall x,y,z:U.(x*y)*z=x*(y*z)}
```

Syntax (remember ⌜3⌝ ?)

```
MonoidTerm := Theory {
  type MTerm = (data X .
    #e : X |
    #* : (X, X) -> X)
}
```

Biform Theory: axiomatic + syntax + transformers.

```
length :: MonoidTerm -> Nat
length trm = gfold (+) 1 trm

leftSimp  :: MonoidTerm -> MonoidTerm
leftSimp  = fun (#*(a,b)) when a = #e -> b
rightSimp :: MonoidTerm -> MonoidTerm
rightSimp = fun (#*(a,b)) when b = #e -> b
```

Biform monoids

```
Monoid := Theory {
  U : type;
  e : U;
  * : (U, U) -> U;
  ax: forall x:U. e*x = x;
  ax: forall x:U. x*e = x;
  ax: forall x,y,z:U.(x*y)*z=x*(y*z)}
```

Syntax (remember ⌜3⌝ ?)

```
MonoidTerm := Theory {
  type MTerm = (data X .
    #e : X |
    #* : (X, X) -> X)
}
```

Biform Theory: axiomatic + syntax + transformers.

```
length :: MonoidTerm -> Nat
length trm = gfold (+) 1 trm

simp :: MonoidTerm -> MonoidTerm
simp t = match t with
  | (#* (a,b)) when a = #e -> b
  | (#* (a,b)) when b = #e -> b
  | _                      -> t
```

Biform monoids

```
Monoid := Theory {
  U : type;
  e : U;
  * : (U, U) -> U;
  ax: forall x:U. e*x = x;
  ax: forall x:U. x*e = x;
  ax: forall x,y,z:U.(x*y)*z=x*(y*z)}
```

Syntax (remember ⌜3⌝ ?)

```
MonoidTerm := Theory {
  type MTerm = (data X .
    #e : X |
    #* : (X, X) -> X)
}
```

Biform Theory: axiomatic + syntax + transformers.

```
length :: MonoidTerm -> Nat
length trm = gfold (+) 1 trm
```

Generic

```
simp :: MonoidTerm -> MonoidTerm
simp t = match t with
 | (#* (a,b)) when a = #e -> b
 | (#* (a,b)) when b = #e -> b
 | _                      -> t
```

Derived from length reducing axioms

Colimit of diagrams, fibered functors

$\Downarrow$

Concepts and theory combinators

$\Downarrow$

Algebraic theories

$\Downarrow$

Biform theories

$\Downarrow$

Generic algorithm

$\Downarrow$

Library

$\Downarrow$

Efficient Program

Monoids generate types [1]

```
Monoid := Theory {
  U : type;
  e : U;
  * : (U, U) -> U;
  ax: forall x:U. e*x = x;
  ax: forall x:U. x*e = x;
  ax: forall x,y,z:U.(x*y)*z=x*(y*z)}
```

Monoid type, as values

```
module type MONOID = sig
  type n
  val plus : n -> n -> n
  val zero : n
end
```

[1] *simplified metaocaml for clarity*

Monoids generate types [1]

```
Monoid := Theory {
  U : type;
  e : U;
  * : (U, U) -> U;
  ax: forall x:U. e*x = x;
  ax: forall x:U. x*e = x;
  ax: forall x,y,z:U.(x*y)*z=x*(y*z)}
```

Monoid type, as code

```
module type MONOIDCODE = sig
  type n
  type nc = n code
  val plus : nc -> nc -> nc
  val zero : nc
end
```

Monoid type, as values

```
module type MONOID = sig
  type n
  val plus : n -> n -> n
  val zero : n
end
```

[1] *simplified metaocaml for clarity*

Monoids generate types [1]

```
Monoid := Theory {
  U : type;
  e : U;
  * : (U, U) -> U;
  ax: forall x:U. e*x = x;
  ax: forall x:U. x*e = x;
  ax: forall x,y,z:U.(x*y)*z=x*(y*z)}
```

Monoid type, as values

```
module type MONOID = sig
  type n
  val plus : n -> n -> n
  val zero : n
end
```

Monoid type, as code

```
module type MONOIDCODE = sig
  type n
  type nc = n code
  val plus : nc -> nc -> nc
  val zero : nc
end
```

Monoid type, staged

```
type x staged = Now of x
              | Later of x code
module type MONOIDSTAGED = sig
  type n
  type ns = n staged
  val plus : ns -> ns -> ns
  val zero : ns
end
```

if you see a pushout, you're right!

[1] *simplified metaocaml for clarity*

Monoids: from syntax to code

```
MonoidTerm := Theory {
  type MTerm = (data X .
    #e : X |
    #* : (X, X) -> X)
}
```

```
module type MONOIDSTAGED = sig
  type n
  type ns = n staged
  val plus : ns -> ns -> ns
  val zero : ns
end
```

Monoids: from syntax to code

```
MonoidTerm := Theory {
  type MTerm = (data X .
    #e : X |
    #* : (X, X) -> X)
}
```

```
module type MONOIDSTAGED = sig
  type n
  type ns = n staged
  val plus : ns -> ns -> ns
  val zero : ns
end
```

Equality is “free”

```
simp :: MonoidTerm -> MonoidTerm
simp t = match t with
 | (#* (a,b)) when a = #e -> b
 | (#* (a,b)) when b = #e -> b
 | _                      -> t
```

Equality is “now”

```
let monoid one bnow blater x y =
  match x, y with
  | (Now a), b when a = one -> b
  | a, (Now b) when b = one -> a
  | _ -> binary bnow blater x y
```

Concrete Monoids[2]

```
module IntM = struct
  type n = Int
  let plus = (+)
  let zero = 0
end
```

```
module IntMC = struct
  type n = Int
  type 'a nc = ('a,n) code
  let plus = .<fun x y -> .~x + .~y>.
  let zero = .< 0 >.
end
```

```
module IntMS = struct
  type n = Int
  type 'a ns = ('a,n) staged
  let plus = monoid IntM.bzero IntM.plus IntMC.plus
  let zero = of_immediate IntM.zero
end
```

[2] Yes, more category theory hidden here

$$
\begin{array}{c}
\text{Colimit of diagrams, fibered functors} \\
\Downarrow \\
\text{Concepts and theory combinators} \\
\Downarrow \\
\text{Algebraic theories} \\
\Downarrow \\
\text{Biform theories} \\
\Downarrow \\
\text{Generic algorithm} \\
\Downarrow \\
\text{Library} \\
\Downarrow \\
\text{Efficient Program}
\end{array}
$$