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
MathScheme: a Mechanized Mathematics System

- Join Symbolic Computation, Computer Algebra and Theorem Proving
MathScheme: a Mechanized Mathematics System

Join Symbolic Computation, Computer Algebra and Theorem Proving

One of our goals:

Produce efficient, generic, provably correct code
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
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

Colimit of diagrams, fibered functors
⇓
Concepts and theory combinators
⇓
Algebraic theories
⇓
Biform theories
⇓
Generic algorithm
⇓
Library
⇓
Efficient Program
The 6 steps

Colimit of diagrams, fibered functors

Concepts and theory combinators

Algebraic theories

Biform theories

Generic algorithm

Library

Efficient Program
Generic efficiency

Generic algorithms $\rightarrow$ Library $\rightarrow$ Efficient Program

ex: LAPACK, NAG. SPIRAL.
Generic efficiency

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

ex: LAPACK, NAG. SPIRAL.

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

ex: C++ Templates. BOOST++, Linbox.
Generic efficiency

Generic algorithms $\Rightarrow$ Library $\Rightarrow$ Efficient Program

ex: LAPACK, NAG. SPIRAL.

Library of generic algorithms $\Rightarrow$ Efficient Program

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

Library of generic algorithm interfaces $\Rightarrow$ Efficient? Program

ex: Maple’s Linear Algebra.
Generic efficiency

Generic algorithms \(\Rightarrow\) Library \(\Rightarrow\) Efficient Program

ex: LAPACK, NAG, SPIRAL.

Library of generic algorithms \(\Rightarrow\) Efficient Program

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

Library of generic algorithm interfaces \(\Rightarrow\) Efficient? Program

ex: Maple’s Linear Algebra.

Library of generic algorithm \(\Rightarrow\) Slow Program

ex: Maple’s Domains, most OO languages.
Generating extensions

Library of generic algorithms resolve → Program

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

Build a typed generating extension:
Generating extensions

Library of generic algorithms ▶️ Program

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

Build a typed generating extension:

1. Make your algorithms generic

Examples:
Generating extensions

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

Library of generic algorithms \( \xrightarrow{\text{resolve}} \) 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

Library of generic algorithms \(\xrightarrow{\text{resolve}}\) 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

Library of generic algorithms \( \xrightarrow{\text{resolve}} \) 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:


Examples:


### Values, code and syntax

<table>
<thead>
<tr>
<th>Domain</th>
<th>3</th>
<th>is really</th>
<th>write as</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td><code>{}</code>, <code>{}</code>, <code>{}</code>, <code>{}</code></td>
<td></td>
</tr>
<tr>
<td>code</td>
<td>3</td>
<td><code>00000011</code></td>
<td></td>
</tr>
<tr>
<td>syntax</td>
<td>3</td>
<td><code>trois</code></td>
<td></td>
</tr>
</tbody>
</table>

Furthermore: code is opaque, syntax is not.

A Maple analogy:

- Code generation: syntax ⇒ code.
- Staged program: manipulates both values and code.

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

J.Carette (McMaster)
## Values, code and syntax

<table>
<thead>
<tr>
<th>Domain</th>
<th>3</th>
<th>is really</th>
<th>write as</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>{{}, {}, {}, {{}}}</td>
<td>3</td>
</tr>
<tr>
<td>code</td>
<td>3</td>
<td>00000011</td>
<td>.&lt; 3 &gt;.</td>
</tr>
<tr>
<td>syntax</td>
<td>3</td>
<td>trois</td>
<td>「3」</td>
</tr>
</tbody>
</table>

**Equations:**

\[
\begin{align*}
1 + 2 &= 3, \\
\{1 + 2\} \neq \{3\}
\end{align*}
\]

**Furthermore:**

Code is opaque, syntax is not.

**A Maple analogy:**

\[
2 \ast 3 \text{ vs } (2 - < > 3 \text{ vs } 2 \& 3)
\]

**Code generation:**

Syntax \(\Rightarrow\) code.

A staged program manipulates both values and code.

\[
type\ ('a,' b)\ staged = \begin{cases} 'a,' b & \text{if now} \\ ('a,' b) & \text{if later} \end{cases}
\]

J.Carette (McMaster)
Values, code and syntax

<table>
<thead>
<tr>
<th>Domain</th>
<th>3</th>
<th>is really</th>
<th>write as</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>{}, {}, {}, {}</td>
<td>3</td>
</tr>
<tr>
<td>code</td>
<td>3</td>
<td>00000011</td>
<td>.&lt; 3 &gt;.</td>
</tr>
<tr>
<td>syntax</td>
<td>3</td>
<td>trois</td>
<td>⌜3⌝</td>
</tr>
</tbody>
</table>

Equations: $1 + 2 = 3$, .< 1+2 >. $\neq$. .< 3 >., ⌜1 + 2⌝ $\neq$ ⌜3⌝,
Equations: \( 1 + 2 = 3 \), \(<1+2>\neq <3>\), \([1+2] \neq [3]\),

but: \texttt{run } \(<1+2>\texttt{=} \texttt{run } \(<3>\texttt{, [}1+2\texttt{]} = [}3\texttt{]}

Furthermore: code is \textbf{opaque}, syntax is \textbf{not}.

A Maple analogy: \(2*3\) vs \((\) \(\rightarrow\) \(2*3\) vs \(2 \&* 3\)
Values, code and syntax

<table>
<thead>
<tr>
<th>Domain</th>
<th>3</th>
<th>is really</th>
<th>write as</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>{{}, {}, {}, {}, {}}}</td>
<td>3</td>
</tr>
<tr>
<td>code</td>
<td>3</td>
<td>00000011</td>
<td>.&lt; 3 &gt;.</td>
</tr>
<tr>
<td>syntax</td>
<td>3</td>
<td>trois</td>
<td>3</td>
</tr>
</tbody>
</table>

Equations: $1 + 2 = 3$, .< 1+2 >. $\neq$ .< 3 >., $\lceil 1 + 2 \rceil \neq \lceil 3 \rceil$, but: run .< 1+2 >. = run .< 3 >., $\lceil 1 + 2 \rceil = \lceil 3 \rceil$

Furthermore: code is **opaque**, syntax is **not**.

A Maple analogy: $2 * 3$ vs () $\rightarrow 2 * 3$ vs $2 \&* 3$

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

```ocaml
let rec norm = function
  | [] -> 0.
  | x::xs -> abs_float x +. norm xs
>
val norm : float list -> float = <fun>
```
Going generic, then generative

```ml
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.

```ml
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

```ocaml
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
```

J.Carette (McMaster)  Theories to code
Going generic, then generative

```haskell
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

```plaintext
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):

```plaintext
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

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

Collecting variabilities:

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

and build a generator:

```ocaml
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

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

or run it now:

```ocaml
module Norm (NV : NORM_VAR) = struct
  let norm x =
    let module GP = GenericNorm(NV.NS)(NV.C) in
    GP.norm (Staged.ofImmediate x)
end
```
Going generic, then generative

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

Generated code example with a generic 3-tuple container:

```ocaml
< fun x -> let (a,b,c) = x in abs a + abs b + abs c >.
```
Going generic, then generative

```ocaml
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

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

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

```ocaml
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
```

```ocaml
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:

<table>
<thead>
<tr>
<th>Geometric Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Affine Space</td>
</tr>
<tr>
<td>Linear Algebra</td>
</tr>
<tr>
<td>Number Types / Abstract Algebra</td>
</tr>
<tr>
<td>Code</td>
</tr>
</tbody>
</table>
Example: Generative Geometric Kernel (GGK)

Drill-down:

<table>
<thead>
<tr>
<th>Geometric</th>
<th>Orientation</th>
<th>Inside</th>
<th>Simplex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometric Vertex</td>
<td>Vertex</td>
<td>Sidedness</td>
<td>Hypersphere</td>
</tr>
<tr>
<td>Objects</td>
<td>Insphere</td>
<td>Hyperplane Operations</td>
<td>Hyperplane</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Affine Space</th>
<th>Vector</th>
<th>Affine Transforms</th>
<th>Point</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Linear Algebra</th>
<th>Tuple</th>
<th>Matrix</th>
<th>Determinant</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Algebra</th>
<th>RealField</th>
<th>Order</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Set</td>
<td>Ring</td>
<td>Monoid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Staged Types</th>
<th>Base Types</th>
</tr>
</thead>
</table>
Example: Generative Geometric Kernel (GGK)

Architecture:

<table>
<thead>
<tr>
<th>Generic Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic Computation</td>
</tr>
<tr>
<td>Virtualized Base Language</td>
</tr>
</tbody>
</table>
Example: Generative Geometric Kernel (GGK)

Architecture:

<table>
<thead>
<tr>
<th>Generic Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polymorphic over data rep., domain, stage, ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbolic Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol shuffling with a purpose</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtualized Base Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>staged, overloaded language w/ simplification</td>
</tr>
</tbody>
</table>
Colimit of diagrams, fibered functors

Concepts and theory combinators

Algebraic theories

Biform theories

Generic algorithm

Library

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) }
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
Extension:

\[
\text{CommutativeMonoid} := \text{Monoid extended by } \{ \\
\text{axiom commutative}_\ast : \text{forall } x, y, z:U. \ x\ast y = y\ast x \}\n\]
Combinators for theories

Extension:

\[ \text{CommutativeMonoid} := \text{Monoid extended by } \{ \text{axiom commutative}_\ast : \forall x, y, z : U. x \ast y = y \ast x \} \]

Renaming:

\[ \text{AdditiveMonoid} := \text{Monoid} \left[ \ast \rightarrow +, \ e \rightarrow 0 \right] \]
Combinators for theories

Extension:

\[
\text{CommutativeMonoid} := \text{Monoid extended by } \{ \\
\quad \text{axiom commutative}_\ast : \text{for all } x, y, z : U. \ x \ast y = y \ast x \} \\
\]

Renaming:

\[
\text{AdditiveMonoid} := \text{Monoid}[^\ast \rightarrow + , \ e \rightarrow 0] \\
\]

Combination:

\[
\text{AdditiveCommutativeMonoid} := \text{combine AdditiveMonoid , CommutativeMonoid over Monoid} \\
\]
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
NearSemiring := combine AdditiveSemigroup, Semigroup, RightRingoid over

NearSemifield := combine NearSemiring, Group over Semigroup

Semifield := combine NearSemifield, LeftRingoid over RingoidSig

NearRing := combine AdditiveGroup, Semigroup, RightRingoid over RingoidSig

Rng := combine AbelianAdditiveGroup, Semigroup, Ringoid over RingoidSig

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 Ringoid0Sig

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

Concepts and theory combinators

Algebraic theories

Biform theories

Generic algorithm

Library

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

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

holds. A morphism $\Gamma \rightarrow \Delta (= \langle y : \sigma \rangle_{0}^{m-1})$ is an assignment (substitution) $[y_0 \mapsto t_0, \ldots, y_m \mapsto t_{m-1}]$ such that

$$\Gamma \vdash t_0 : \tau_0 \ \ldots \ \Gamma \vdash t_{m-1} : \tau_{m-1} \ [y \mapsto t]_{0}^{m-2}$$

Definition

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

**Theorem**

*The functor $\text{cod} : \mathcal{E} \to \mathcal{B}$ is a fibration.*
and just a bit more theory

Theorem

The functor $\text{cod} : \mathcal{E} \to \mathcal{B}$ is a fibration.

\begin{align*}
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} & | & \text{combine } A \ r_1, \ B \ r_2 \\
l &\in \text{judgments}^* & t &\in \text{Term} & | & A \ ; \ B \\
r &\in (a_i \mapsto b_i)^* & \theta &\in \text{Prop} & | & A \ r \\
\end{align*}
Theorem

The functor \( \text{cod} : \mathcal{E} \rightarrow \mathcal{B} \) is a fibration.

\[
\begin{align*}
[-]_\mathcal{B} : \text{tpc} & \rightarrow |\mathcal{B}| \\
\text{Empty}_\mathcal{B} & = \langle \rangle \\
\text{Theory} \{l\}_\mathcal{B} & \cong \langle l \rangle \\
[A \ r]_\mathcal{B} & = [r]_\pi \cdot [A]_\mathcal{B} \\
[A ; B]_\mathcal{B} & = [B]_\mathcal{B} \\
\text{extend } A \text{ by } \{l\}_\mathcal{B} & \cong [A]_\mathcal{B} \bowtie \langle l \rangle \\
\text{combine } A_1 r_1, A_2 r_2]_\mathcal{B} & \cong D
\end{align*}
\]
Theorem

*The functor* \( \text{cod} : \mathbb{E} \rightarrow \mathbb{B} \) *is a fibration.*

\[
\begin{align*}
\left[ - \right]_E & : \text{tpc} \rightarrow |\mathbb{E}| \\
\left[ \text{Empty} \right]_E &= \text{id} \langle \rangle \\
\left[ \text{Theory} \{l\} \right]_E &\cong !\langle l \rangle \\
\left[ A \; r \right]_E &= \left[ r \right]_\pi \cdot \left[ A \right]_E \\
\left[ A; B \right]_E &= \left[ A \right]_E \circ \left[ B \right]_E \\
\left[ \text{extend } A \text{ by } \{l\} \right]_E &\cong \delta_A \\
\left[ \text{combine } A_1 r_1, A_2 r_2 \right]_E &\cong \left[ r_1 \right]_\pi \circ \delta_{T_1} \circ \left[ A_1 \right]_E \\
&\cong \left[ r_2 \right]_\pi \circ \delta_{T_2} \circ \left[ A_2 \right]_E
\end{align*}
\]
and just a bit more theory

**Theorem**

The functor \( \text{cod} : \mathbb{E} \to \mathbb{B} \) is a fibration.

\[
\llbracket \cdot \rrbracket_\mathbb{E} : \text{tpc} \to |\mathbb{E}|
\]

\[
\llbracket \text{Empty} \rrbracket_\mathbb{E} = \text{id} \langle \rangle
\]

\[
\llbracket \text{Theory } \{l\} \rrbracket_\mathbb{E} \cong ! \langle l \rangle
\]

\[
\llbracket A \cdot r \rrbracket_\mathbb{E} = \llbracket r \rrbracket_\mathbb{E} \cdot \llbracket A \rrbracket_\mathbb{E}
\]

\[
\llbracket A; B \rrbracket_\mathbb{E} = \llbracket A \rrbracket_\mathbb{E} \circ \llbracket B \rrbracket_\mathbb{E}
\]

\[
\llbracket \text{extend } A \text{ by } \{l\} \rrbracket_\mathbb{E} \cong \delta_A
\]

\[
\llbracket \text{combine } A_1 r_1, A_2 r_2 \rrbracket_\mathbb{E} \cong \llbracket r_1 \rrbracket_\mathbb{E} \circ \delta_{T_1} \circ \llbracket A_1 \rrbracket_\mathbb{E}
\]

\[\cong \llbracket r_2 \rrbracket_\mathbb{E} \circ \delta_{T_2} \circ \llbracket A_2 \rrbracket_\mathbb{E}\]

More structure
More structure

- Semigroup
- Monoid
- LeftNearSemiring
- Commutative Semigroup
- Commutative Monoid
- Additive Commutative LeftNearSemiring
- Multiplicative Commutative LeftNearSemiring
- Commutative LeftNearSemiring
More structure

- Semigroup
- Commutative Semigroup
- Monoid
- Commutative Monoid
- LeftNearSemiring
- Additive Commutative LeftNearSemiring
- Multiplicative Commutative LeftNearSemiring
- Commutative LeftNearSemiring

Order
More structure

- Order
- Partial operations
More structure

- Order
- Partial operations
- Constructive equality
Colimit of diagrams, fibered functors

Concepts and theory combinators

Algebraic theories

Biform theories

Generic algorithm

Library

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

Biform Theory: axiomatic + syntax + transformers.

J.Carette (McMaster)  
Theories to code  
28 / 36
Biform monoids

Monoid := Theory { 
  U : type;
  e : U;
  ∗ : (U, U) −→ U;
ax: for all x : U. e ∗ x = x;
ax: for all x : U. x ∗ e = x;
ax: for all x, y, z : U. (x ∗ y) ∗ z = x ∗ (y ∗ z) }

Syntax (remember \(\exists?\))

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: for all x:U. e*x = x; 
ax: for all x:U. x*e = x; 
ax: for all 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: for all x : U. e * x = x; 
ax: for all x : U. x * e = x; 
ax: for all x, y, z : U. (x * y) * z = x * (y * z)}

Syntax (remember \( \exists 3 \)?)

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 : for all x:U. e*x = x;
  ax : for all x:U. x*e = x;
  ax : for all 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

Generic

Derived from length reducing axioms
Colimit of diagrams, fibered functors

Concepts and theory combinators

Algebraic theories

Biform theories

Generic algorithm

Library

Efficient Program
Monoids generate types\(^1\)

Monoid := Theory {  
U : type;  
e : U;  
∗ : (U, U) \rightarrow U;  
ax : for all x : U. e ∗ x = x;  
ax : for all x : U. x ∗ e = x;  
ax : for all x, y, z : U. (x ∗ y) ∗ z = x ∗ (y ∗ z)}

Monoid type, as values

module type MONOID = sig  
type n  
val plus : n \rightarrow n \rightarrow n  
val zero : n  
end

\(^1\) simplified metaocaml for clarity
Monoids generate types

Monoid := Theory {
    U : type;
    e : U;
    * : (U, U) -> U;
    ax: for all x:U. e*x = x;
    ax: for all x:U. x*e = x;
    ax: for all x, y, z:U. (x*y)*z = x*(y*z)
}
Monoids generate types

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

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

if you see a pushout, you're right!

\textsuperscript{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
MonoidTerm := Theory {
  type MTerm = (data X .
    #e : X |
    #* : (X, X) \rightarrow X)
}

module type MONOIDSTAGED = sig
  type n
  type ns = n staged
  val plus : ns \rightarrow ns \rightarrow ns
  val zero : ns
end

Equality is “free”

simp :: MonoidTerm \rightarrow MonoidTerm
simp t = match t with
  | (#* (a,b)) when a = #e \rightarrow b
  | (#* (a,b)) when b = #e \rightarrow b
  | _ \rightarrow t

Equality is “now”

let monoid one bnow blater x y =
  match x, y with
  | (Now a), b when a = one \rightarrow b
  | a, (Now b) when b = one \rightarrow a
  | _ \rightarrow binary bnow blater x y
Concrete Monoids

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 = ofImmediate IntM.zero
end

\[Yes, \text{ more category theory hidden here}\]