Deriving on Steroids - for proof assistants

Jacques Carette

McMaster University

August 20, 2018
The Big Picture: MetaProgramming on Theories

(Co)Limits of diagrams, fibered functors

Concepts and theory combinators

(Generalized) Algebraic theories

Biform theories

Generic algorithm

Library

Efficient Program
The Big Picture: MetaProgramming on Theories

(Co)Limits of diagrams, fibered functors

Concepts and theory combinators

(Generalized) Algebraic theories

Biform theories

Generic algorithm

Library

Efficient Program
Theory graphs
Theory graphs

- **Magma**
  - Binary Operation
  - Closure

- **Semigroup**
  - Associativity
  - **Monoid**
    - Identity Element
  - **Group**
    - Inverse

- **Semiring**
  - Additive monoid & multiplicative monoid

- **Ring**
  - **Integral Domain**
    - Commutative ring with unity and no zero divisors
  - **Field**
    - Commutative ring with unity & cancellation property (in which every non-zero element is a unit)

- **Semimodule**
  - Additive Monoid(T) & Commutative Semiring(S)

- **Vector Space**
  - Vectors over a field
  - vector & field (scalars)
  - addition with scalar multiplication
  - vector addition:
    - $a(v+u) = a\ v + a\ u$
    - $(a+b)\ v = a\ v + b\ v$
    - $a\ (b\ v) = (a\ b)\ v$
    - $1\ v = v$
Theory graphs

J.Carette

Deriving
Theory graphs

-0p (Trivial)

8p 6p 2r (S 5) 3q (K 2)

10p 12r (S 3.5) 10r 4s (S 4.2) 4q (K 1)

12p 14r

18r

12q

16r

20s (S 3)

4q (K 1)

6s (S 4)

8q
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)
}
(Presentations of) 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 
}
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) }
(Presentations of) 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 }
Pseudo-Combinators for (presentations of) theories

Following Burstall & Goguen (OBJ); Kapur, Musser, Stepanov (Tecton)

Extension:

CommutativeMonoid := Monoid extended by 
{axiom commutative * : for all x, y, z:U. x*y = y*x}
Pseudo-Combinators for (presentations of) theories

Following Burstall & Goguen (OBJ); Kapur, Musser, Stepanov (Tecton)

Extension:

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

Renaming:

\[
\text{AdditiveMonoid} := \text{Monoid}[ \ast \mapsto +, e \mapsto 0 ]
\]
Pseudo-Combinators for (presentations of) theories

Following Burstall & Goguen (OBJ); Kapur, Musser, Stepanov (Tecton)

Extension:

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

Renaming:

$\text{AdditiveMonoid} := \text{Monoid}[* \mapsto +, e \mapsto 0]$  

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 Ringoid
NearSemifield := combine NearSemiring, Group over Semigroup
Semifield := combine NearSemifield, LeftRingoid over RingoidSig
NearRing := combine AdditiveGroup, Semigroup, RightRingoid over Ringoid
Rng := combine AbelianAdditiveGroup, Semigroup, Ringoid over RingoidSig
Semiring := combine AdditiveCommutativeMonoid, Monoid1, Ringoid, Left0
SemiRng := combine AdditiveCommutativeMonoid, Semigroup, Ringoid over Ringoid
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
A fraction of the Algebraic Zoo
Breaks down
Thy1 := Empty extended by \{ U : type \}
Thy2 := Empty extended by \{ U : type \}
Thy3 := combine Thy1, Thy2 over Empty
Breaks down

\[\begin{align*}
\text{Thy1} & := \text{Empty} \text{ extended by } \{ U : \text{type} \} \\
\text{Thy2} & := \text{Empty} \text{ extended by } \{ U : \text{type} \} \\
\text{Thy3} & := \text{combine} \text{ Thy1, Thy2 over Empty}
\end{align*}\]

Lesson from PL theory

Find a good denotational semantics, then come back to the syntax
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

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

\[ \begin{array}{c}
\Gamma^+ \xrightarrow{f^+} \Delta^+ \\
\downarrow \quad \quad \quad \quad \quad \quad \downarrow \\
A \quad \quad \quad \quad \quad \quad \quad \quad B \\
\Gamma \xrightarrow{f^-} \Delta
\end{array} \]

**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 $\mathbb{B}$. 
Theorem

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

**Theorem**

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

\[
\begin{align*}
a, b, c & \in \text{labels} \\
A, B, C & \in \text{names} \\
l & \in \text{declarations}^* \\
r & \in (a_i \mapsto b_i)^*
\end{align*}
\]

\[
\text{tpc ::= extend } A \text{ by } \{ l \}
\]

\[
\begin{align*}
| & \text{ combine } A \ r_1, \ B \ r_2 \\
| & A ; \ B \\
| & A \ r \\
| & \text{Empty} \\
| & \text{Theory } \{ l \}
\end{align*}
\]
Theorem

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

\[
\begin{align*}
[-]_B : \text{tpc} & \to |\mathbb{B}| \\
[\text{Empty}]_B & = \langle \rangle \\
[\text{Theory} \{l\}]_B & \cong \langle l \rangle \\
[A \ r]_B & = [r]_\pi \cdot [A]_B \\
[A; B]_B & = [B]_B \\
[\text{extend } A \text{ by } \{l\}]_B & \cong [A]_B \cdot \langle l \rangle \\
[\text{combine } A_1 r_1, A_2 r_2]_B & \cong D
\end{align*}
\]
and just a bit more theory

**Theorem**

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

\[
\begin{align*}
\langle - \rangle_E &: \text{tpc} \rightarrow |E| \\
\langle \text{Empty} \rangle_E &= \text{id}\langle \rangle \\
\langle \text{Theory } \{ l \} \rangle_E &\equiv !\langle l \rangle \\
\langle A \ r \rangle_E &= \langle r \rangle_\pi \cdot \langle A \rangle_E \\
\langle A; B \rangle_E &= \langle A \rangle_E \circ \langle B \rangle_E \\
\langle \text{extend } A \text{ by } \{ l \} \rangle_E &\equiv \delta_A \\
\langle \text{combine } A_1 \ r_1, A_2 \ r_2 \rangle_E &\equiv \langle r_1 \rangle_\pi \circ \delta_{T_1} \circ \langle A_1 \rangle_E \\
&\quad \equiv \langle r_2 \rangle_\pi \circ \delta_{T_2} \circ \langle A_2 \rangle_E
\end{align*}
\]
That does scale
That does scale

- 1046 theories
- 2429 lines of code (including comments and many theories defined over 3 lines for human readability)
- expanded theories: 13751 lines
- type checked by export to Matita
More structure

- Order
- Partial operations
- Equality
- Multivalued operations
More structure

- Order
- Partial operations
- Equality
- Multivalued operations
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 (term language)

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

MonoidTerm := Theory {
    type MTerm = (data X .
        #e : X |
        #∗ : (X, X) → X)
}

Biform Theory: axiomatic + syntactic theory + transformers.

length :: MTerm → 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 (term language)

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

Biform Theory: axiomatic + syntactic theory + transformers.

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

leftSimp :: MTerm -> MTerm
leftSimp = fun (#*(a,b)) when a = #e -> b
rightSimp :: MTerm -> MTerm
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 (term language)

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

Biform Theory: axiomatic + syntactic theory + transformers.

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

simp :: MTerm -> MTerm
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 (term language)

MonoidTerm := Theory { 
    type MTerm = (data X . 
        #e : X | 
        #∗ : (X, X) → X) 
} 

Biform Theory: axiomatic + syntactic theory + transformers.

length :: MTerm → Nat 
length trm = gfold (+) 1 trm 

simp :: MTerm → MTerm 
simp t = match t with 
    | (#∗ (a, b)) when a = #e → b 
    | (#∗ (a, b)) when b = #e → b 
    | ± → t 

Generic

Derived from length reducing axioms
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
Different interpretations of theories

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) }

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

1 simplified metaocaml for clarity
Different interpretations of theories \(^1\)

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) \}\n
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

\(^1\) simplified metaocaml for clarity
MonoidTerm := Theory {
  type MTerm = (data X .
    #e : X |
    #* : (X, X) -> X)
}

module type MONOIDSTAGED = sig
  type n
  type ns = n staged
  val zero : ns
  val plus : ns -> ns -> ns
end
MonoidTerm := Theory {
  type MTerm = (data X .
    #e : X |
    #∗ : (X, X) → X)
}

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

Equality is “free”

simp : : MTerm → MTerm
simp t = match t with
  | (#∗ (a, b)) when a = #e → b
  | (#∗ (a, b)) when b = #e → b
  | _ → t

Equality is “now”

let monoid zero plusN plusL x y =
  match x, y with
    | (Now a), b when a = zero → b
    | a, (Now b) when b = one → a
    | _ → lift2 plusN plusL 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.zero IntM.plus IntMC.plus
  let zero = Now IntM.zero
end

Machinery for free

Given a structured graph of theories, one can get a (naïve) optimizing compiler.

J.Carette

Deriving
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.zero IntM.plus IntMC.plus
  let zero = Now IntM.zero
end

Machinery for free

Given a structured graph of theories, one can get a (naïve) optimizing compiler.
### MSL

Monoid := Theory {
  U : type;
  * : (U,U) -> U;
  e : U;
  axiom right_identity_*_e :
    forall x : U . (x * e) = x
  axiom left_identity_*_e :
    forall x : U . (e * x) = x;
  axiom associativity_* :
    forall x,y,z : U . ((x * y) * z) = (x * (y * z ));
}

### Coq

Class Monoid {A : type}
  (dot : A -> A -> A)
  (one : A) : Prop := {
    dot_assoc :
      forall x y z : A,
      (dot x (dot y z))
      = dot (dot x y) z
  unit_left :
    forall x, dot one x = x
  unit_right :
    forall x, dot x one = x
};

### Alternative Definition:

Record monoid := {
  dom : Type;
  op : dom -> dom -> dom
  where "x * y" := (op x y);
  id : dom where "1" := id ;
  assoc : forall x y z, x * (y * z) = (x * y) * z;
  left_neutral : forall x, 1 * x = x;
  right_neutral : forall x, x * 1 = x
}.

### Haskell

```haskell
class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mappend = (<>)
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

### Agda

```agda
data Monoid (A : Set)
  (Eq : Equivalence A) : Set
  where
    monoid :
      (z : A)
      (_+_ : A -> A -> A)
      (left_Id : LeftIdentity Eq z _+_)
      (right_Id : RightIdentity Eq z _+_)
      (assoc : Associative Eq _+_ ) ->
        Monoid A Eq
```

### Alternative Definition:

```agda
record Monoid c ℓ :
  Set ( suc (c ⊔ ℓ)) where
    infixl 7 _•_
    infix 4 _≈_
    field
      Carrier : Set c
      _≈_ : Rel Carrier
      _•_ : Op 2 Carrier
      isMonoid :
        IsMonoid _≈_ _•_ "
where
  record IsMonoid (• : Op2) (ε : A)
    : Set (a ⊔ ℓ) where
    field
      isSemigroup : IsSemigroup •
      identity : Identity •
      identity : LeftIdentity •
      identity = proj1 identity
      identity : RightIdentity •
      identity = proj2 identity
```

### Isabelle

```isabelle
class semigroup =
  fixes mult :: α => α => α
  (infixl ⊗ 70)
  assumes assoc :: (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)
class monoidl = semigroup +
  fixes neutral :: α
  (1)
  assumes neutl : 1 ⊗ x = x
class monoid = monoidl +
  assumes x ⊗ 1 = x
```

### Lean

```lean
universe u
variables{ α : Type u }
class monoid (α : Type u) extends
  semigroup α, has_one α :=
  (one_mul : ∀ a : α, 1 * a = a)
  (mul_one : ∀ a : α, a * 1 = a)
```

### J.Carette

Deriving
Universal Algebra...

Most of these work for Generalized Algebraic Theories (à la Cartmell):

- Signature
- Term Algebra
  - “generic functions” (à la *Scrap your Boilerplate*)
  - Structural induction
- Term Algebra parametrized by a “theory” of variables
  - predicate for ground terms
  - “simplifier” for open terms (correct but usually incomplete)
- **Homomorphism**; homomorphism composition; isomorphism
- kernel of homomorphism
- Theory of congruence relations over a theory
- Induced congruence of a homomorphism
- Interpreter from Term Algebra to any instance of a theory
- Partial evaluator
- Sub-theory, Product Theory, Co-product Theory
- Internalization (making a record that represents a theory)
ack down! Given:

- The theory presentation of 2-categories, can you specialize to category?
- Category to monoid?
- Monoidal Category to... monoid?
- Braided Category to...?

How do you (as a library builder) not repeat yourself,
while giving end-users a huge, rich, as-they-expect it to look library?
Computer Science?

Axiomatic presentations of the theories of:

- Data Structures
Axiomatic presentations of the theories of:

- **Data Structures**

![Diagram of Data Structures](image)

- **Algorithms**

- **Models of Computation**
  - FSM
  - DPDA
  - TM
  - 23 Registers Machines
  - SECD

- Ongoing work with Ph.D. student Lijun Zhu
Computer Science?

Axiomatic presentations of the theories of:

- **Data Structures**


- **Algorithms**

- **Douglas Smith**’s SpecWare

- **Ralf Hinze**’s (et al.)’s recursion schemes extracted from categorical adjustment and/or Kan extensions.

- **Models of Computation**

- FSM, DPDA, TM, 23 Registers Machines, SECD

- Ongoing work with Ph.D. student Lijun Zhu
Axiomatic presentations of the theories of:

- **Data Structures**
- **Algorithms**
  - Douglas Smith’s *SpecWare*
  - Ralf Hinze’s (et al.)’s *recursion schemes* extracted from categorical adjustment and/or Kan extensions.
- **Models of Computation**
  - FSM, DPDA, TM, 23 Registers Machines, SECD
  - ongoing work with Ph.D. student Lijun Zhu