MTC

Meta-Theory à la Carte

Tom Schrijvers

Universiteit Gent

with Ben Delaware and Bruno Oliveira

accepted for POPL’13
Teaser
Modular Mechanized Meta-Theory
<table>
<thead>
<tr>
<th>Features</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Evaluator</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>▪</td>
</tr>
<tr>
<td>Booleans</td>
<td>▪</td>
</tr>
<tr>
<td>Case Exps</td>
<td>▪</td>
</tr>
<tr>
<td>Lambdas</td>
<td>▪</td>
</tr>
<tr>
<td>Fix</td>
<td>▪</td>
</tr>
</tbody>
</table>

**Programming Language** ➤ **Proof Assistant**
Fortunately, regular algebras are compatible with Mendler-based Church proof algebra. The corresponding proof is simply a fold over a provided with the necessary proof cases, e.g., for the algebra is a dependent product refer to the original term, it can simultaneously build a copy expressive with the help of tuples. The dependent products in Coq 3.2 Type Dependency with Dependent Products This restriction is a showstopper for the semantics setting of this type soundness destructive and their result type cannot depend on the term being destructed. For example, it is impossible to express the proof for structive and their result type cannot depend on the term being induction, which endangered strong normalization of the calculus such as 

\[ \forall A \quad \Gamma \vdash e : t \rightarrow \Gamma \vdash [e] : t \]  

\[ \forall A \]

\[ \exists \]

\[ \Gamma \]

\[ [e] \]

\[ : t \]
3 Main Problems

1. How to adapt EP solution to Coq
2. How to add induction principles
3. How to make induction principles modular
Solution Involves

- Data Types à la Carte
- Church encodings
- Mendler algebras
- the universal property of folds
- axiom-free induction for Church encodings
Step 1

The Expression Problem in a Proof Assistant
<table>
<thead>
<tr>
<th>Features</th>
<th>Evaluator</th>
<th>Compiler</th>
<th>Pretty Printer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Booleans</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Case Exps</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Lambdas</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Fix</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Programming Language ➤ Proof Assistant
Starting Point: Data Types à la Carte

- Expression Problem solution in Haskell
- no theorems and proofs
- not suitable for Coq due to general recursion

Data Types à la Carte
Wouter Swierstra (JFP’08)
Running Example

```
data Arith = Lit Int | Add Arith Arith
data Value = I Int | B Bool
[] :: Arith → Maybe Value
[(Lit n)] = Just (I x)
[(Add x y)] = case ([x], [y]) of
    (Just (I x), Just (I y)) → Just (I (x + y))
    _                      → Nothing
```
DTC
Type Modularization

\[
\text{data } \text{Arith}_F a = \text{Lit } \text{Nat} \mid \text{Add } a a
\]

\[
\text{data } \text{Fix}_{\text{DTC}} f = \text{In } (f (\text{Fix}_{\text{DTC}} f))
\]

\[
\text{type } \text{Arith} = \text{Fix}_{\text{DTC}} \text{Arith}_F
\]

modular definitions

general recursion
DTC
Type Extension

data \text{Arith}_F a = \text{Lit} \text{ Nat} \mid \text{Add} a a

data \text{Logic}_F a = \text{If} a a a \\
\quad \mid \text{BLit} \text{ Bool}

data (\oplus) f g a = \text{Inl} (f a) \mid \text{Inr} (g a)

Fix_{DTC} (\text{Arith}_F \oplus \text{Logic}_F)
DTC
Modular Functions

type Algebra f a = f a → a
fold\_\_DTC :: Functor f ⇒ Algebra f a → Fix\_\_DTC f → a
fold\_\_DTC\_\_alg\_\((\text{In } fa)\) = alg (fmap (fold\_\_DTC\_\_alg) fa)

eval\_\_Arith :: Algebra\_\_Arith\_F\_\((\text{Maybe Value})\)
eval\_\_Arith\_\((\text{Lit } n)\) = Just (I n)
eval\_\_Arith\_\((\text{Add } v_1\_\_v_2)\) =
case (v_1\_\_v_2) of
(Just (I x), Just (I y)) → Just (I (x + y))

\[\square :: Fix\_\_DTC\_\_Arith\_F → \text{Maybe Value}\]
\[\square = \text{fold}\_\_DTC\_\_eval\_\_Arith\]

...
DTC

Function Extension

\[(\boxplus) :: Algebra f a \to Algebra g a \to Algebra (f \oplus g) a\]

\[\text{alg}_1 \boxplus \text{alg}_2 = \lambda e \to \text{case } e \text{ of}\]
\[\text{Inl } e_1 \to \text{alg}_1 e_1\]
\[\text{Inr } e_2 \to \text{alg}_2 e_2\]

\[\text{eval}_{\text{Logic}} :: Algebra \text{ Logic}_F (\text{Maybe Value})\]
\[\text{eval}_{\text{Logic}} (\text{If } v_1 v_2 v_3) = \text{if } (v_1 \equiv \text{Just } (B \text{ True}))\]
\[\quad \text{then } v_2 \text{ else } v_3\]

\[\text{eval}_{\text{Logic}} (\text{BLit } b) = \text{Just } (B \text{ } b)\]

\[\llbracket \rrbracket :: \text{Fix}_{\text{DTC}} (\text{Arith}_F \oplus \text{Logic}_F) \to \text{Maybe Value}\]
\[\llbracket \rrbracket = \text{fold}_{\text{DTC}} (\text{eval}_{\text{Arith}} \oplus \text{eval}_{\text{Logic}})\]
Mechanizing DTC

\[
data \ Fix_{DTC} f = \text{In} (f \ (\Fix_{DTC} f))
\]

type \ \text{Algebra} f \ a = f \ a \rightarrow a

\[
fold_{DTC} :: \text{Functor} f \Rightarrow \text{Algebra} f \ a \rightarrow \Fix_{DTC} f \rightarrow a
\]

\[
fold_{DTC} \ \text{alg} \ (\text{In} \ fa) = \text{alg} \ (\text{fmap} \ (\text{fold}_{DTC} \ \text{alg}) \ fa)
\]
CHALLENGE 1

Mechanizing DTC without general recursion
Church Encoding

values encoded as their folds

type \( \text{Fix } f = \forall a. \text{Algebra } f \ a \rightarrow a \)
fold :: Algebra f a \rightarrow Fix f \rightarrow a
fold alg fa = fa alg

lit :: Nat \rightarrow \text{Fix } \text{Arith}_F
lit n = \lambda \text{alg} \rightarrow \text{alg } (\text{Lit } n)

add :: \text{Fix } \text{Arith}_F \rightarrow \text{Fix } \text{Arith}_F \rightarrow \text{Fix } \text{Arith}_F
add e_1 e_2 = \lambda \text{alg} \rightarrow \text{alg } (\text{Add } (\text{fold } \text{alg } e_1) (\text{fold } \text{alg } e_2))

non-recursive definitions
Lack of Control

\[
\begin{align*}
\text{eval}_{\text{Logic}} :: \text{Algebra Logic}_F & \ (\text{Maybe Value}) \\
\text{eval}_{\text{Logic}} (\text{If } v_1 v_2 v_3) &= \text{if } (v_1 \equiv \text{Just } (B \ True)) \\
& \quad \text{then } v_2 \text{ else } v_3 \\
\text{eval}_{\text{Logic}} (\text{BLit } b) &= \text{Just } (B \ b)
\end{align*}
\]

branches are always evaluated
We want Control

\[
\begin{align*}
[e_1] & \leadsto \text{true} & [e_2] & \leadsto v_2 \\
\iff [\text{if } e_1 e_2 e_3] & \leadsto v_2 \\
[e_1] & \leadsto \text{false} & [e_3] & \leadsto v_3 \\
\iff [\text{if } e_1 e_2 e_3] & \leadsto v_3
\end{align*}
\]
Control with Mendler Algebras

type $\text{Algebra}_M f a = \forall r (r \rightarrow a) \rightarrow f r \rightarrow a$

recursive call

structural induction

eval_{Logic} :: \text{Algebra}_M \text{Logic}_F (\text{Maybe Value})
eval_{Logic} \llbracket \cdot \rrbracket (B\text{Lit } b) = \text{Just } (B \ b)neval_{Logic} \llbracket \cdot \rrbracket (\text{If } e_1 \ e_2 \ e_3) = \text{if } (\llbracket e_1 \rrbracket \equiv \text{Just } (B \ \text{True})) \text{ then } \llbracket e_2 \rrbracket \text{ else } \llbracket e_3 \rrbracket
Mendler Encoding

values encoded as Mendler folds

\[
\text{type } Algebra_M f a = \forall r. (r \to a) \to f r \to a
\]

\[
\text{type } Fix_M f = \forall a. Algebra_M f a \to a
\]

\[
\text{fold}_M :: Algebra_M f a \to Fix_M f \to a
\]

\[
\text{fold}_M \text{ alg } fa = fa \text{ alg}
\]

\[
\text{lit :: Nat } \to \text{ Fix}_M \text{ Arith}_F
\]

\[
\text{lit } n = \lambda \text{alg } \to \text{alg } (\text{fold}_M \text{ alg}) (\text{Lit } n)
\]

\[
\text{add :: Fix Arith}_F \to \text{ Fix Arith}_F \to \text{ Fix Arith}_F
\]

\[
\text{add } e_1 e_2 = \lambda \text{alg } \to \text{alg } (\text{fold}_M \text{ alg}) (\text{Add } e_1 e_2)
\]
Mendler Algebra Composition

\[
data (\oplus) f g a = \text{Inl} (f a) \mid \text{Inr} (g a)
\]

\[
(\boxplus_M) :: \text{Algebra}_M f a \to \text{Algebra}_M g a \to \text{Algebra}_M (f \oplus g) a
\]

\[
\text{alg}_1 \boxplus_M \text{alg}_2 = \lambda \text{rec } e \to \text{case } e \text{ of } \\
\quad \text{Inl } e_1 \to \text{alg}_1 \text{ rec } e_1 \\
\quad \text{Inr } e_2 \to \text{alg}_2 \text{ rec } e_2
\]
Type-Directed Algebra Composition

class \( \text{Eval } f \) where
\[
\text{eval}_{\text{alg}} :: \text{Algebra}_M f \text{ Value}
\]

instance \((\text{Eval } f, \text{Eval } g) \Rightarrow \text{Eval } (f \oplus g)\) where
\[
\text{eval}_{\text{alg}} = \text{eval}_{\text{alg}} \oplus_M \text{eval}_{\text{alg}}
\]

\[
\text{eval} :: \text{Eval } f \Rightarrow \text{Fix}_M f \rightarrow \text{Value}
\]
\[
\text{eval} = \text{fold}_M \text{eval}_{\text{alg}}
\]

as shown by DTC
Clumsy Manual Injection

\[
\begin{align*}
\text{exp}_1 &: \text{Fix}_{DT} (\text{Arith}_F \oplus \text{Logic}_F) \\
\text{exp}_1 &= \text{In} (\text{Inl} (\text{Add} (\text{In} (\text{Inl} (\text{Lit} 1))) (\text{In} (\text{Inl} ((\text{Lit} 2))))))
\end{align*}
\]
Automatic Injection

class $f \prec g$ where

$\text{inj} :: f \ a \rightarrow g \ a$

instance $(f \prec g) \Rightarrow f \prec (g \oplus h)$ where

$\text{inj} \ fa = \text{Inl} \ (\text{inj} \ fa)$

instance $(f \prec h) \Rightarrow f \prec (g \oplus h)$ where

$\text{inj} \ fa = \text{Inr} \ (\text{inj} \ fa)$

instance $f \prec f$ where

$\text{inj} \ fa = fa$

DTC idea; more symmetric in Coq
Smart Constructors

\[\text{in}_f :: f \ (\text{Fix}_M f) \to \text{Fix}_M f\]
\[\text{in}_f \ f \text{exp} = \lambda \text{alg} \to \text{alg} \ (\text{fold}_M \text{alg}) \ f \text{exp}\]

\[\text{inject} :: (g \prec: f) \Rightarrow g \ (\text{Fix}_M f) \to \text{Fix}_M f\]
\[\text{inject} \ g \text{exp} = \text{in}_f \ (\text{inj} \ g \text{exp})\]

\[\text{lit} :: (\text{Arith}_F \prec: f) \Rightarrow \text{Nat} \to \text{Fix}_M f\]
\[\text{lit} \ n = \text{inject} \ (\text{Lit} \ n)\]

\[\text{blit} :: (\text{Logic}_F \prec: f) \Rightarrow \text{Bool} \to \text{Fix}_M f\]
\[\text{blit} \ b = \text{inject} \ (\text{BLit} \ b)\]

DTC, adapted to Mendler algebras
Summary So Far

- We have recovered DTC in Coq
- using Church/Mendler encodings

- but what about theorems & proofs?
Step 2

Reasoning with Church Encodings
<table>
<thead>
<tr>
<th>Features (Syntax)</th>
<th>Functions</th>
<th>Semantics</th>
<th>Proofs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>🟦</td>
<td>🟦</td>
<td>🟦</td>
</tr>
<tr>
<td>Booleans</td>
<td>🟦</td>
<td>🟦</td>
<td>🟦</td>
</tr>
<tr>
<td>Case Exps</td>
<td>🟦</td>
<td>🟦</td>
<td>🟦</td>
</tr>
<tr>
<td>Lambdas</td>
<td>🟦</td>
<td>🟦</td>
<td>🟦</td>
</tr>
<tr>
<td>Fix</td>
<td>🟦</td>
<td>🟦</td>
<td>🟦</td>
</tr>
</tbody>
</table>
Encodings and Induction Principles

- no free induction principles from Coq
- *calculus of constructions* used axioms
- *calculus of inductive constructions* avoids encodings
Can we do better?
CHALLENGE 2

Axiom-free Induction Principles for Church Encodings
Coq’s Induction Principles

\[ A_{\text{ind}} :: \forall P :: (\text{Arith} \rightarrow \text{Prop}). \]

\[ \forall H_l :: (\forall n. P \ (\text{Lit } n)). \]

\[ \forall H_a :: (\forall a \ b. P \ a \rightarrow P \ b \rightarrow P \ (\text{Add } a \ b)). \]

\[ \forall a. P \ a \]

\[ A_{\text{ind}} \ P \ H_l \ H_a \ e = \]

\[ \text{case } e \text{ of} \]

\[ \text{Lit } n \rightarrow H_l \ n \]

\[ \text{Add } x \ y \rightarrow H_a \ a \ b \ (A_{\text{ind}} \ P \ H_l \ H_a \ x) \]

\[ (A_{\text{ind}} \ P \ H_l \ H_a \ y) \]

Looks like a fold of a proof algebra
Encoded Folds with Proof Algebra?

- folds are destructive:
  
  algebra cannot refer to original term

- theorems are dependently typed
  
  algebra must refer to original term

\[ \forall e. \Gamma \vdash e : t \rightarrow \Gamma \vdash [e] : t \]
Hutton’s Tutorial

Hutton:

tuples (products) make folds more expressive

MTC:

dependent products make proof folds more expressive
Poor Man’s Induction Algebra

\[ A^2_{\text{ind}} :: \forall P :: (\text{Fix}_M \ \text{Arith}_F \rightarrow \text{Prop}). \]
\[ \forall H_l :: (\forall n. P (\text{lit } n)). \]
\[ \forall H_a :: (\forall a \ b. P a \rightarrow P b \rightarrow P (\text{add } a \ b)). \]

Algebra \text{Arith}_F (\sum e P e)

\[ A^2_{\text{ind}} P H_l H_a e = \]

\text{case } e \text{ of}

\begin{align*}
\text{Lit } n & \rightarrow \exists (\text{lit } n) (H_l n) \\
\text{Add } x \ y & \rightarrow \exists (\text{add } (\pi_1 x) (\pi_1 y)) (H_a (\pi_1 x) (\pi_1 y) \\
& \quad \quad (\pi_2 x) (\pi_2 y))
\end{align*}

given \textbf{one term}, this algebra returns
\textbf{another term} with the property P
Upgrading Poor Man’s Induction

input term == output term

- well-formedness of proof algebra: the algebra reconstructs the input term
- well-formedness of input term: the term folds the algebra properly
Well-Formed Proof Algebra

I-layer equality:

∀alg :: Algebra f (Σ e.P e).π₁ ∘ alg = inf f ∘ fmap π₁

easy to show on:

A²_{ind} P H_I H_a e =
case e of
    Lit n → ∃ (lit n) (H_I n)
    Add x y → ∃ (add (π₁ x) (π₁ y)) (H_a (π₁ x) (π₁ y)) (π₂ x) (π₂ y))
Well-Formed Terms

Hutton: universal property

\[ h = \text{fold}'_M \text{alg} \iff h \circ \text{in}_f = \text{alg} \ h \]

encoded in terms

\[ \text{type } UP \ f \ e = \]
\[ \forall a \ (\text{alg} :: \text{Algebra}_M f \ a) \ (h :: \text{Fix}_M f \rightarrow a). \]
\[ (\forall e'. h \ (\text{in}_f e') = \text{alg} \ h \ e') \rightarrow h \ e = \text{fold}'_M \text{alg} \ e \]
Proper Induction

**Theorem 3.1.** Given a functor \( f \), property \( P \), and a well-formed \( P \)-proof algebra \( \text{alg} \) for any Church-encoded \( f \)-term \( e \) with the universal property, we have that \( P e \) holds.

**Proof.** Given that \( \text{fold}'_M \text{alg} e \) has type \( \Sigma e'.P e' \), we have that \( \pi_2 (\text{fold}'_M \text{alg} e) \) is a proof for \( P (\pi_1 (\text{fold}'_M \text{alg} e)) \). From that the lemma is derived as follows:

\[
P (\pi_1 (\text{fold}'_M \text{alg} e)) \implies \{\text{-well-founded algebra and fusion law -}\} P (\text{fold}'_M \text{in}_f e)
\]

\[
\iff \{\text{-reflection law -}\} P e
\]

\( \square \)
Summary

• **axiom-free** induction principle for Mendler-encoded terms

• using **universal property** of folds

• but what about **modular proofs**?
Step 3

Extensible Reasoning with Church Encodings
<table>
<thead>
<tr>
<th>Features (Syntax)</th>
<th>Semantics</th>
<th>Proofs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Functions</td>
<td>Predicates</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Booleans</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Case Exps</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Lambdas</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Fix</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
CHALLENGE 3

Modular Proofs
Proof Delegation

proof algebra

\( f \oplus g \)

proof algebra

proof algebra

f

g
Extensible Proof Algebras

\[ A^2_{\text{ind}} :: \forall f. \text{Arith}_F <: f \Rightarrow \]
\[ \forall P :: (\text{Fix}_M f \rightarrow \text{Prop}). \]
\[ \forall H_l :: (\forall n. P (\text{lit } n)). \]
\[ \forall H_a :: (\forall a b. P a \rightarrow P b \rightarrow P (\text{add } a b)). \]

Algebra \text{Arith}_F (\Sigma e. P e)

generalized to compositions
Composition Properties

Consider Proving:

\[ \forall e. \text{typeof } e = \text{Just } \text{nat} \rightarrow \exists m :: \text{nat.eval } e = \text{vi } m \]

Case 1:

\[ \text{typeof} (\text{lit } n) = \text{Just } \text{nat} \rightarrow \exists m :: \text{nat.eval} (\text{lit } n) = \text{vi } m \]

abstract in f with
ArithF <: f
Well-Formed Delegation

class \((\text{Eval } f, \text{Eval } g, f \preceq g) \Rightarrow \text{WF}_{\text{Eval } f g}\) where
\[
\text{wf}_{\text{eval}_{\text{alg}}} :: \forall r \ (\text{rec} :: r \to \text{Nat}) \ (e :: f \ r).
\]
\[
\text{eval}_{\text{alg}} \text{rec} (\text{inj } e :: g \ r) = \text{eval}_{\text{alg}} \text{rec } e
\]

follows from \(\oplus\)
functor composition

but not apparent to Coq
<table>
<thead>
<tr>
<th>Features (Syntax)</th>
<th>Functions</th>
<th>Semantics</th>
<th>Proofs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Booleans</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Case Exps</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Lambdas</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Fix</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
Extensible Inductive Predicates

Some definitions about expressions are expressed as relations instead of functions.
Inductive Predicates

indexed inductive datatypes of kind Prop

data WTValue :: Value → Type → Prop where
  WTNat :: ∀n. WTValue (I n) TNat
  WTBool :: ∀b. WTValue (B b) TBool
Extensible Inductive Datatypes

• Adapt encoding to indexed functors
  \[ f :: (i \rightarrow \text{Prop}) \rightarrow (i \rightarrow \text{Prop}) \]

• for instance

```haskell
data WTNat_F :: v \rightarrow t \rightarrow (WTV :: (v, t) \rightarrow \text{Prop}) 
  \rightarrow (v, t) \rightarrow \text{Prop}

where WTNat :: \forall n. (NVal_F \triangleleft: v, \text{Functor } v 
  , NTyp_F \triangleleft: t, \text{Functor } t) 
  \Rightarrow WTNat_F v t WTV (v\| n, t\|nat)
```

Framework Summary
<table>
<thead>
<tr>
<th>Class Definition</th>
<th>Description</th>
</tr>
</thead>
</table>
| **class Functor f where**  
  fmap :: (a → b) → (f a → f b)  
  fmap_id :: fmap id = id  
  fmap_fusion :: ∀g h.  
    fmap h ⚪ fmap g = fmap (h ⚪ g)  | **Functors**  
  Supplied by the user |
| **class f ⚪: g where**  
  inj :: f a → g a  
  prj :: g a → Maybe (f a)  
  inj_prj :: prj ga = Just fa → ga = inj fa  
  prj_inj :: prj ⚪ inj = id  | **Functor Subtyping**  
  Inferred |
| **class (Functor f, Functor g, f ⚪: g) ⇒**  
  WF_Functor f g where  
  wf_functor :: ∀a b (h :: a → b).  
    fmap h ⚪ inj = inj ⚪ fmap h  | **Functor Delegation**  
  Inferred |
| **class (Functor h, f ⚪: h, g ⚪: h) ⇒**  
  DistinctSubFunctor f g h where  
  inj_discriminate :: ∀a (fe :: f a)  
    (ge :: g a). inj fe ≠ inj ge  | **Functor Discrimination**  
  Inferred |
As these indexed variants are meant to construct inductive predicates, their parameters range over Prop instead of Set. Fortunately, this shift obviates the need for universal properties for iFix-ed values: it does not matter how an inductive predicate is built, just whether it exists. Analogues to WF Functor, WF Algebra, and DistinctSubFunctor are similarly unnecessary.

5.3. Case Study: Soundness of an Arithmetic Language

Here we briefly illustrate modular reasoning with a case study proving soundness for the Arith F⊕ Logic language.
More in the Paper
Higher-Order Features

- PHOAS-based lambda-abstraction feature
- closure values
-Mixin algebras
- bounded fixpoints for general recursion
- reasoning
Algebra Adapters

Controlled Evaluation

Mendler Algebras

Parameterized Mendler Algebras

Parameterized Algebras

General Recursion

Mixin Algebras

Parameterized Mixin Algebras

Parameterized Mixin Algebras
Case Study: mini-ML

evaluation, typing, soundness, continuity

\[
e ::= \mathbb{N} \mid e + e
\]
\[
B \mid \text{if } e \text{ then } e \text{ else } e
\]
\[
\text{case } e \text{ of } \{ z \Rightarrow e ; S n \Rightarrow e \}
\]
\[
\text{lam } x : T . e \mid e e \mid x
\]
\[
\text{fix } x : T . e
\]

\[
V ::= \mathbb{N}
\]
\[
B \mid \text{bool}
\]
\[
\text{closure } e \bar{V}
\]

\[
T ::= \text{nat}
\]
\[
\text{bool}
\]
\[
T \rightarrow T
\]

\[
\sim 1100 \text{ LoC / feature}
\]
\[
\sim 100 \text{ LoC / language}
\]
\[
\sim 2500 \text{ LoC framework}
\]
Summary

- modular mechanized meta-theory
- Church encodings for extensible inductive definition
- Mendler and mixin algebras

Meta-Theory à la Carte
Ben Delaware, Bruno Oliveira, T.S. (POPL’13)
Related Work

- Tinkertype, Ott, Boite’04, Mulhern’06, Delaware’11
  manual or tool-based composition of text

- Schwaab & Siek ’12
  MTC-like development in Agda
Ongoing & Future Work

1. Side-Effects

2. Adequacy
The Side-Effect Problem

New features alter algebra signatures

Modular Monadic Reasoning, a (Co-)Routine
Steven Keuchel, T.S. (IFL’12)
The Adequacy Problem

MTC

↑

↓

non-modular definitions
Thank You!
Extra
Pattern Matching

class f <: g where
    inj :: f a → g a
    prj :: g a → Maybe (f a)

out_f :: Functor f ⇒ Fix_M f → f (Fix_M f)
out_f exp = fold_M (λrec fr → fmap (in_f o rec) fr) exp

project :: (g <: f, Functor f) ⇒
    Fix_M f → Maybe (g (Fix_M f))
project exp = prj (out_f exp)
Extensible Return Types

data $VNat_F a = I \text{Nat}$
data $VBool_F a = B \text{Bool}$
data $\text{Stuck}_F a = \text{Stuck}$

$vi :: (VNat_F \preceq r) \Rightarrow \text{Nat} \rightarrow \text{Fix}_M r$

$vi \ n = \text{inject} \ (I \ n)$

$vb :: (VBool_F \preceq r) \Rightarrow \text{Bool} \rightarrow \text{Fix}_M r$

$vb \ b = \text{inject} \ (B \ b)$

$\text{stuck} :: (\text{Stuck}_F \preceq r) \Rightarrow \text{Fix}_M r$

$\text{stuck} = \text{inject} \ \text{Stuck}$
Extensible Return Types

class Eval f[r] where
eval_alg :: Algebra_M f (Fix_M r)

instance (Stuck_F ≺: r, VNat_F ≺: r, Functor r)
⇒ Eval Arith_F r where
  eval_alg [·] (Lit n) = vi n
  eval_alg [·] (Add e₁ e₂) =
    case (project [e₁], project [e₂]) of
      (Just (I n₁), (Just (I n₂))) → vi (n₁ + n₂)
      _ → stuck