Computational Logic and Programming Languages at The University of Iowa

Aaron Stump
Computer Science
The University of Iowa
Iowa City, Iowa

The Computational Logic Group at U. Iowa

Led by myself and Cesare Tinelli.

Alumni at Coverity, Kestrel, Georgia Regents U. (tenure-track), Two Sigma, NASA Langley, and others.

Research interests: SMT, ATP, model checking, hybrid systems (Cesare); type theory, rewriting, functional programming (Aaron).

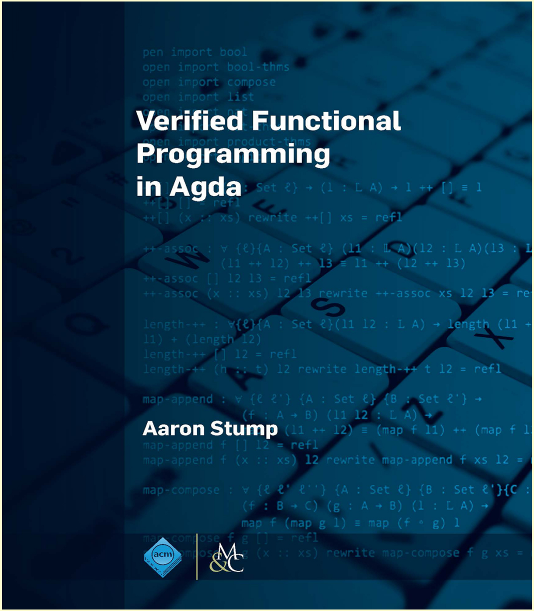
Software and systems: Kind2, StarExec.

Main research target: verification.

Advertisement

- Currently have four open postdoc positions:
  - importing proofs produced by SMT solvers into Coq (Cesare)
  - SMT-based model-checking with Kind2 (Cesare)
  - programming languages for quantum computing (Aaron)
  - type theory for lambda encodings (Aaron)
- Also, tenure-track faculty position:
  - PL/FM for security
  - Cesare and I will have a lot of input in the hiring decision

Please talk to me if you are interested in any of these!



Introduction to dependently typed FP in Agda.

Intended for undergrads without FP or type theory background; also an extended Agda tutorial.

Booleans, natural numbers, lists, Braun trees, binary search trees, well-founded recursion, type-level computation, normalization by evaluation.

Due out 2016 from ACM Books.

Lambda Encodings Reborn

Aaron Stump
Computational Logic Center
Computer Science
The University of Iowa

Behold the Mighty Coq

A glorious confluence of logic and engineering!

Rightly fêted, ardently adopted!

Potently expressive!

And yet...

Its flight lacks a certain *je ne sais quoi*.

(Agda is no better off)

Coq, the funny bits

- Type preservation does not hold with coinductive types
- Large eliminations disallowed with impredicative inductive types
- Datatypes must be not just positive, but *strictly positive*
- Higher-order encodings are prohibited
  - cannot have a constructor `lam` of type `(trm -> trm) -> trm`
  - leads to cottage industry of representing variables
  - many elegant idioms not allowed (cf. Twelf)

We have hobbled type theory by clipping its higher-order wings.

My dream: more elegant type theory with full support for higher-order encodings.

My dream: more elegant type theory with full support for higher-order encodings.

Starting point: lambda encodings

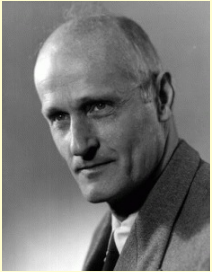
Encode all data (structures) as functions.

Example: Church encoding

Data are defined to be their own fold functions.

Numbers are defined to be iterators:

$$\ulcorner n \urcorner := \lambda s.\lambda z.\ \underbrace{s \cdots (s}_{n}\ z)$$



Accessors (like predecessor) are inefficient.

Kleene's predecessor:

$$(x, y) \mapsto (suc\ x, x)$$

The charges against lambda encodings in type theory

- Asymptotically inefficient accessors [Parigot 1989]
- Cannot prove disjoint-range property of constructors ($0 \neq 1$)
- Cannot derive induction principles [Geuvers 2001]
- Large eliminations not possible

Case closed!

Not so fast!

Parigot [1988] showed how to get efficient accessors.

Define data as recursors, not iterators.

$$\ulcorner n \urcorner := \lambda s.\lambda z.s \ulcorner n-1 \urcorner \cdots (s \ulcorner 1 \urcorner (s \ulcorner 0 \urcorner z))$$

For example, $\ulcorner 3 \urcorner$ is

$$\lambda s.\lambda z.s \ulcorner 2 \urcorner (s \ulcorner 1 \urcorner (s \ulcorner 0 \urcorner z))$$

Predecessor takes constant time.

Typable in System F + positive-recursive types.

$$\mathbb{N} := \mu\mathbb{N}.\ \forall X.\ (\mathbb{N} \to X \to X) \to X \to X$$

Exponential-space normal forms, but not with graph sharing.

New solutions

Induction:

New type construct for the limit of

$$\begin{array}{lcl} \mathbb{N}_0 & := & \mathcal{U} \\ \mathbb{N}_{k+1} & := & \iota n : \mathbb{N}_k . \forall P : \mathbb{N}_k \to \star . \\ & & (\forall n : \mathbb{N}_k . P\ n \to P\ (S\ n)) \to P\ Z \to P\ n \end{array}$$

$$\begin{array}{lcl} \mathbb{N} & := & \nu\, Nat : \star \mid S \in Nat \to Nat,\ Z \in Nat . \iota\, n : Nat . \\ & & \forall\, P : Nat \to \star . (\forall\, n : Nat . P\ n \to P\ (S\ n)) \to P\ Z \to P\ n \end{array}$$

Large eliminations:

Construct to lift *simply typed* terms to the type level.

$$\uparrow_{(\star \to \star) \to \star \to \star} (\lambda s . \lambda z . s\ z) \simeq \lambda S : \star \to \star . \lambda Z : \star . S\ Z$$

Lattice-theoretic semantics, consistency proof.

Prototype tool called Cedille.

Why do this?

We can drop the datatype subsystem completely.

~~Inductive nat : Set := ...~~

Much simpler definition for the type theory.
No more rules like:

Elimination (definition by cases):

$$\frac{\begin{array}{l}\Gamma' \vdash P\colon (\Delta)(\mathsf{T}\ \overline{x_\Delta}) \to \mathsf{Type} \\ \Gamma', \Theta_i \vdash e_i\colon (P\ \overline{p_i}\ (\mathsf{c}_i\ \overline{x_{\Theta_i}})) \quad 1 \le i \le n \\ \Gamma' \vdash \overline{d} :: \Delta \\ \Gamma' \vdash t\colon (\mathsf{T}\ \overline{d})\end{array}}{\Gamma' \vdash \left(\mathsf{Cases}\ t\ \mathsf{of} \left\{\begin{array}{l}(\mathsf{c}_1\ \overline{x_{\Theta_1}}) \mapsto e_1 \\ \vdots \\ (\mathsf{c}_n\ \overline{x_{\Theta_n}}) \mapsto e_n\end{array}\right.\right) : (P\ \overline{d}\ t)};$$

Crazy examples

Statically typed format, with local definitions

Augustsson [1998] proposed computing type of `format s` from `s`.

```
format "%s are %n - %n" : string → ℕ → ℕ → string
```

Let's add local definitions to the format string(!)

We will use a higher-order datatype.

Just print bit strings.

```
format (fapp farg (flit tt)) ==>
λ x → x :: tt :: []

format (flet farg (λ i → fapp i (fapp (flit tt) i))) ==>
λ x → x :: tt :: x :: []
```

In Agda with `-no-positivity-check`

Format specifier is indexed by argument specifier of type

```
data formatti : Set where
  iarg : formatti
  inone : formatti
  iapp : formatti → formatti → formatti
```

The datatype of format specifiers:

```
data formati : formatti → Set where
 farg : formati iarg
 fapp : {a b : formatti} → formati a → formati b →
        formati (iapp a b)
 flet : {a b : formatti} →
        formati a → (formati inone → formati b) →
        formati (iapp a b)
 fbitstr : bitstr → formati inone

format : {i : formatti} → formati i → format-t i
```

In Cedille

The crucial datatype definition:

```
formati =
 λ i : formatti .
  ∀ X : formatti → ⋆ .
   (X iarg) →
   (∀ a : formatti . ∀ b : formatti.
      X a → X b → X (iapp a b)) →
   (∀ a : formatti . ∀ b : formatti.
      X a → (X inone → X b) → X (iapp a b)) →
   (bitstr → X inone) →
   X i
```

Can type `format` without disabling anything in the type theory!

Where next?

Current theory based on realizability, Curry-style typing.

Need to move to Church style for practical use.

Vast new unexplored terrain: higher-order encodings in type theory.

Implementation: runtime code generation instead of closures?

Thanks!