Lambda Encodings Reborn

Aaron Stump

Computational Logic Center
Computer Science
The University of Iowa
A **Golden Age** for Theorem Provers

- Powerful software tools for computer-checked proofs
  - **Coq** (France)
  - **Agda** (Sweden)
  - **Isabelle** (Germany/UK)

- **Trustworthy** proofs for Math/CS

- Many amazing examples
  - **CS:** Quark verified web-browser kernel [Jang et al. 2012]
  - **CS:** Compcert optimizing C compiler [Leroy. 2006]
  - **Math:** Feit-Thompson theorem [Gonthier et al. 2013]
  - **Math:** Kepler conjecture (completed fall 2014)

- Starting to have an impact in USA
  - Key technology for some top assistant profs (MIT, UW, Cornell)
Trouble in Paradise
Trouble in Paradise
Bugs in the Theorem Prover

- **Soundness bug:** *False* can be proved
  - Martin-Löf Type Theory (1971) with Type : Type
  - Shown unsound by Girard (1972)

- **Type preservation bug:** \( t : T \) and \( t \leadsto t' \) but not \( t' : T \)
  - Type preservation bug discovered, Oury (2008)
  - Still present in Coq 8.4 (current version)!

- **Anomalies:**
  - Agda (2005), discovered to be anti-classical (2010)
  - Agda and Coq, discovered incompatible with isomorphism (2013)
    - Contradiction from \((False \rightarrow False) = True\)
    - Based on a subtle bug latent for 17 years!
    - Problem for homotopy type theory
These bugs all have one thing in common
These bugs all have one thing in common

They all depend on the **DATATYPE SUBSYSTEM**
Idea:
Idea:

Let’s get rid of datatypes!
Programs = Functions + Data
Programs = Functions + Data

+ Observations/IO
+ Concurrency
+ Mutable state
+ Exceptions/control
+ ...
Programs = Functions + Data
+ Observations/IO
+ Concurrency
+ Mutable state
+ Exceptions/control
+ ...
Programs  =  Functions
+ Observations/IO
+ Concurrency
+ Mutable state
+ Exceptions/control
+ ...
Programs = Functions

+ Observations/IO
+ Concurrency
+ Mutable state
+ Exceptions/control
+ ...

Lambda Encodings
Lambda Encodings

- Encode all data as functions in lambda ($\lambda$) calculus
- Several different encodings known, starting with Church 1941
- No need for datatypes (except primitive types)
- Simplify design for
  - programming languages
  - languages for computer-checked proofs
Lambda Encodings

- Encode all data as functions in lambda ($\lambda$) calculus
- Several different encodings known, starting with Church 1941
- No need for datatypes (except primitive types)
- Simplify design for
  - programming languages
  - languages for computer-checked proofs

Simpler language design => fewer bugs
Lambda Encodings

- Encode all data as functions in lambda (\( \lambda \) ) calculus
- Several different encodings known, starting with Church 1941
- No need for datatypes (except primitive types)
- Simplify design for
  - programming languages
  - languages for computer-checked proofs

Simpler language design => fewer bugs

Maybe even prove soundness in a theorem prover!
Lambda Encodings

- Encode all data as functions in lambda (\(\lambda\)) calculus
- Several different encodings known, starting with Church 1941
- No need for datatypes (except primitive types)
- Simplify design for
  - programming languages
  - languages for computer-checked proofs

*Simpler language design \(\Rightarrow\) fewer bugs*

*Maybe even prove soundness in a theorem prover!*

- Benchmark example datatype: natural numbers
What is a number?
What is a number?

Church (1941): a number is an *iterator*
The Church Encoding

- **Iterator**: a function that can apply $f$ repeatedly ($n$ times) to $a$.

\[
\text{Iterate } n \, f \, a = f \cdots (f \, a)
\]

- In the Church encoding, numbers are iterators

\[
\begin{align*}
0 & = \lambda f. \lambda a. a \\
1 & = \lambda f. \lambda a. f \, a \\
2 & = \lambda f. \lambda a. f \, (f \, a) \\
3 & = \lambda f. \lambda a. f \, (f \, (f \, a)) \\
& \vdots \\
suc & = \lambda n. \lambda f. \lambda a. f \, (n \, f \, a)
\end{align*}
\]
Church Encoding: Basic Operations

- For addition, iterate \texttt{suc}:

\[
\begin{align*}
    n + m &= 1 + \cdots + 1 + m \\
    \text{add} &= \lambda \ n. \ \lambda \ m. \ n \ \text{suc} \ m
\end{align*}
\]

- For multiplication by \textit{m}, iterate adding \textit{m}:

\[
\begin{align*}
    n \ast m &= m + \cdots m + 0 \\
    \text{mult} &= \lambda \ n. \ \lambda \ m. \ n \ (\text{add} \ m) \ 0
\end{align*}
\]

- Alternative clever versions due to Rosser

\[
\begin{align*}
    \text{exp} &= \lambda \ n. \ \lambda \ m. \ m \ n \\
    (4 \ 2) &= 16
\end{align*}
\]
Typing the Church Encoding

\[ Nat = \forall X : Type. (X \rightarrow X) \rightarrow X \rightarrow X \]

\[ 2 : Nat = \lambda X : Type. \lambda f : X \rightarrow X. \lambda a : X. f (f a) \]

- Typable in polymorphic lambda calculus (System F), Girard/Reynolds
- In System F, typable programs guaranteed to terminate!
- Sound basis for computer-checked proofs
  - Proofs = programs (Curry, Howard)
  - Induction = recursion
  - This requires all programs (= proofs) to terminate
  - Coq, Agda based on this idea
Everything looks good!
Everything looks good!

Church: “But how do you do predecessor?”
Everything looks good!

Church: “But how do you do predecessor?”

Kleene:

\[(x, y) \mapsto (suc \ x, x)\]

\[
\begin{align*}
(0, 0) & \mapsto (1, 0) \mapsto (2, 1) \mapsto (3, 2)
\end{align*}
\]
Everything looks good!

Church: “But how do you do predecessor?”

Kleene:

\((x, y) \mapsto (\text{suc } x, x)\)

\[
\begin{align*}
(0, 0) &\mapsto (1, 0) \\
(2, 1) &\mapsto (3, 2) \\
3 &\mapsto \vdash
\end{align*}
\]

Predecessor of \(n\) takes \(O(n)\) steps!
What is a number?
What is a number?

Parigot (1988): a number is a *recursor*
The Parigot Encoding

- **Recursor**: like an iterator, but given the predecessors!

\[ \text{Rec } n \ f \ a = f \ (n - 1) \ \cdots \ (f \ 1 \ (f \ 0 \ a)) \]

- In the Parigot encoding, numbers are recursors

\[ n \ f \ a = f \ (n - 1) \ \cdots \ (f \ 1 \ (f \ 0 \ a)) \]

0 = \lambda f. \lambda a. a
1 = \lambda f. \lambda a. f \ 0 \ a
2 = \lambda f. \lambda a. f \ 1 \ (f \ 0 \ a)
3 = \lambda f. \lambda a. f \ 2 \ (f \ 1 \ (f \ 0 \ a))
...

\text{suc} = \lambda n. \lambda f. \lambda a. f \ n \ (n \ f \ a)
\text{add} = \lambda n. \lambda m. n \ (\lambda p. \text{suc}) \ m
\text{mult} = \lambda n. \lambda m. n \ (\lambda p. \text{add} \ m) \ 0
\text{pred} = \lambda n. n \ (\lambda p. \lambda d. p) \ 0
Typing the Parigot Encoding

\[ \text{Nat} = \forall X : \text{Type}.(\text{Nat} \to (X \to X)) \to (X \to X) \]

- Typable in System F + positive-recursive types (Parigot, Mendler)
- Recursive use of \text{Nat} is positive:
  - occurs in the left part of an even number of arrows
  - for polarity, \( p \to q \) is like \( \neg p \lor q \)
- Typable programs still guaranteed to terminate!
- Suitable basis for computer proofs under Curry-Howard
Expected asymptotic time complexities!
Expected asymptotic time complexities! Awesome!
Expected asymptotic time complexities! Awesome!

Typable in a terminating type theory!
Expected asymptotic time complexities! Awesome!

Typable in a terminating type theory! Awesome!
Expected asymptotic time complexities! Awesome!

Typable in a terminating type theory! Awesome!

Numbers require exponential space!
Expected asymptotic time complexities! Awesome!

Typable in a terminating type theory! Awesome!

Numbers require exponential space! Oh dear.
What is a number?
What is a number?

Stump-Fu (2014): a number is the ordered collection of iterators for all its predecessors
Embedded-Iterators Encoding (Stump-Fu 2014)

- Same asymptotic time complexities as Parigot
- **But:** normal form of numeral $n$ is only $O(n^2)$
- Basic idea:
  \[ 3 = (c3,(c2,(c1,(c0,0)))) \]

  where $cN$ is the Church encoding of $N$

- Use embedded Church-encoded numbers for iteration

  \[
  0 = \lambda f. \lambda a. a \\
  1 = \lambda f. \lambda a. f \text{ c1 } 0 \\
  2 = \lambda f. \lambda a. f \text{ c2 } 1 \\
  3 = \lambda f. \lambda a. f \text{ c3 } 2 \\
  \ldots \\
  \text{succ} = \lambda n. n \ (\lambda c. \lambda p. \lambda f. \lambda a. f \ (\text{csucc c) n}) \ 1
  \]

- Put embedded iterators in binary to reduce space to $O(n \log_2 n)$
Typing the Embedded-Iterators Encoding

\[ Nat = \forall X : \text{Type}. (\text{CNat} \rightarrow (\text{Nat} \rightarrow X)) \rightarrow (X \rightarrow X) \]

- Like Parigot encoding, typable in System F + positive-rec. types
- Recursive use of \( Nat \) is positive
Implementation

- **fore** tool for $F_\omega$ + positive-recursive type definitions
- Compiles **fore** terms to Racket, Haskell
- For Racket, erase all type annotations
- For Haskell, encodings are actually typable with `newtype`

```haskell
newtype CNat =
  FoldCNat { unfoldCNat :: forall (x :: *) . (x -> x) -> x -> x}
```

- Observe computed answers by translating to native data
- Emitted programs optionally count reductions

```haskell
cadd :: CNat -> CNat -> CNat
cadd = (\ n -> (\ m -> (incr ((incr ((unfoldCNat n) csuc)) m))))
```
Experiments

- Based on the following example programs:
  - Compute $2^n$
  - Compute $x - x$, where $x = 2^n$
  - Mergesort a list of small Parigot-encoded numbers
    - Use Braun trees as intermediate data structure
    - Faster, more natural iteration

- For Racket (CBV), some adjustments needed:

  `Bool : * = ∀ X : *, X → X → X .`
  `true : Bool = λ X:* , λx:X, λy: X, x .`

  becomes

  `Bool : * = ∀ X : *, (unit → X) → (unit → X) → X .`
  `true : Bool = λ X:* , λx:unit → X, λy: unit → X, x triv.`
  `false : Bool = λ X:* , λx: unit → X, λy: unit → X, y triv .`
Sizes of Normal Forms

<table>
<thead>
<tr>
<th>Numeral</th>
<th>Church</th>
<th>Parigot</th>
<th>Stump Fu</th>
<th>Stump Fu (bnats)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exponentiation Test in Racket

![Bar chart showing the number of reductions for different powers of two using various reduction strategies.](chart.png)

- **Church**
- **Church R**
- **Parigot**
- **Cbv Parigot**
- **Stump Fu**
- **Stump Fu (bnats)**

**Power of two**

**Number of reductions**
Exponentiation Test in Haskell

- Church, Church R, Parigot exactly the same reductions
- Embedded iterators: slightly fewer reductions in Haskell

<table>
<thead>
<tr>
<th>power</th>
<th>SF Racket</th>
<th>SF Haskell</th>
<th>SF (bnats) Racket</th>
<th>SF (bnats) Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>19765</td>
<td>19709</td>
<td>279455</td>
<td>260818</td>
</tr>
<tr>
<td>12</td>
<td>78185</td>
<td>78129</td>
<td>1336475</td>
<td>1246109</td>
</tr>
<tr>
<td>14</td>
<td>311709</td>
<td>311653</td>
<td>6249007</td>
<td>5822720</td>
</tr>
<tr>
<td>16</td>
<td>1245649</td>
<td>1245593</td>
<td>28647524</td>
<td>26681058</td>
</tr>
</tbody>
</table>
Subtraction Test in Racket

![Graph showing number of reductions for different powers of two with Church, Parigot, Cbv Parigot, and Stump Fu.]
Subtraction Test in Haskell

- Church, Embedded iterators take slightly less time
- Parigot takes much less:

Each predecessor takes one step less with lazy evaluation

\[(x, y) \mapsto (\text{suc } x, x)\]
Sorting Test in Racket

- Mergesort list of small numbers
- Use Braun trees (balanced) as intermediate data structure

![Bar chart showing number of reductions for list sizes up to 1000 million for Cbv Church, Cbv Parigot, and Stump Fu.](chart.png)
14: embedded iterators 350 times fewer reductions
14: Parigot 2.8 times fewer
Comparison with Native Racket

For list of length 8 million (23):

Parigot almost **3x faster** than native Racket!
Summary

- New embedded-iterators encoding
  - Expected asymptotic time complexities (like Parigot)
  - Size of normal form of $n$ is $O(n^2)$, even $O(n \log_2 n)$
  - Best encoding if size of normal form matters
- Promising empirical results for lambda encodings
  - CBV Parigot beating native Racket sorting by 3x on large lists!
- Typable in total type theories (F or F + pos.-rec. types)
- Hope for using lambda encodings for practical data (structures)
Future Work

- Much still to do for computer-checked proofs
- To derive induction, need dependent types
  - “Induction Is Not Derivable in Second Order Dependent Type Theory” [Geuvers, 2001]
  - “Self Types for Dependently Typed Lambda Encodings” [Fu, Stump, 2014]
- Combining general-recursive programs, proofs
- Lifting lambda encodings from term to type level

$$\textit{arrows } A \; n = A \to \cdots A \to A$$
A Paradise of

\( \lambda \)