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 $t' : T$
  - Coq (1986) with coinductive types (1996)
  - 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 $(\mathit{False} \to \mathit{False}) = \mathit{True}$
    - 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

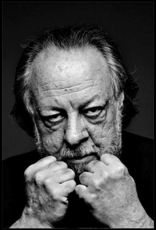
Programs = Functions + Data

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

Programs = Functions + Data

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


POOF!



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 (λ) 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 (λ) 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 (λ) 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 (λ) 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!

- 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.

$$Iterate\ n\ f\ a = \underbrace{f \cdots (f}_{n}\ a)$$

- In the Church encoding, numbers **are** iterators

$$n\ f\ a = \underbrace{f \cdots (f}_{n}\ a)$$

```
0 = λ f. λ a. a
1 = λ f. λ a. f a
2 = λ f. λ a. f (f a)
3 = λ f. λ a. f (f (f a))
...
suc = λ n. λ f. λ a. f (n f a)
```

Church Encoding: Basic Operations

- For addition, iterate `suc`:

$$n + m = \underbrace{1 + \cdots + 1 +}_{n} m$$

```
add = λ n. λ m. n suc m
```

- For multiplication by m, iterate adding m:

$$n * m = \underbrace{m + \cdots m +}_{n} 0$$

```
mult = λ n. λ m. n (add m) 0
```

- Alternative clever versions due to Rosser

```
exp = λ n. λ m. m n
```

```
(4 2) = 16
```

Typing the Church Encoding

$$Nat = \forall X : Type.(X \to X) \to X \to X$$

$$2 : Nat = \lambda X : Type.\lambda f : X \to X.\lambda a : X.\underbrace{f\ \underbrace{(f\ a)}_{:X}}_{:X}$$

- 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?"

$$(x, y) \mapsto (\mathit{suc}\ x, x)$$

Kleene:

$$\underbrace{(0, 0) \mapsto (1, 0) \mapsto (2, 1) \mapsto (3, 2)}_{3}$$

Everything looks good!

Church: “But how do you do predecessor?”

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

Kleene:

$$\underbrace{(0,0) \mapsto (1,0) \mapsto (2,1) \mapsto (3,2)}_{3}$$

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!

$$Rec\ n\ f\ a = f\ (n-1)\ \cdots (f\ 1\ (f\ 0\ a))$$

- In the Parigot encoding, numbers **are** recursors

$$n\ f\ a = f\ (n-1)\ \cdots (f\ 1\ (f\ 0\ a))$$

```
0 = λ f. λ a. a
1 = λ f. λ a. f 0 a
2 = λ f. λ a. f 1 (f 0 a)
3 = λ f. λ a. f 2 (f 1 (f 0 a))
...

suc = λ n. λ f. λ a. f n (n f a)
add = λ n. λ m. n (λ p. suc) m
mult = λ n. λ m. n (λ p. add m) 0
pred = λ n. n (λ p. λ d. p) 0
```

Typing the Parigot Encoding

$$Nat = \forall X : Type.(\underline{Nat} \rightarrow (X \rightarrow X)) \rightarrow (X \rightarrow X)$$

- Typable in System F + positive-recursive types (Parigot,Mendler)
- Recursive use of *Nat* is positive:
  - occurs in the left part of an even number of arrows
  - for polarity, $p \rightarrow q$ is like $\neg p \vee q$
- 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

```
0 = λ f. λ a. a
1 = λ f. λ a. f c1 0
2 = λ f. λ a. f c2 1
3 = λ f. λ a. f c3 2
...
suc = λ n. n (λ c. λ p. λ f. λ a. f (csuc c) n) 1
```

- Use embedded Church-encoded numbers for iteration

```
add = λ n . λ m . n (λ c . λ p . c suc m) m
```

- Put embedded iterators in binary to reduce space to $O(n \log_2 n)$

Typing the Embedded-Iterators Encoding

$$Nat = \forall X : Type.(CNat \to (\underline{Nat} \to X)) \to (X \to X)$$

- Like Parigot encoding, typable in System F + positive-rec. types
- Recursive use of *Nat* is positive

Implementation

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

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

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

```
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

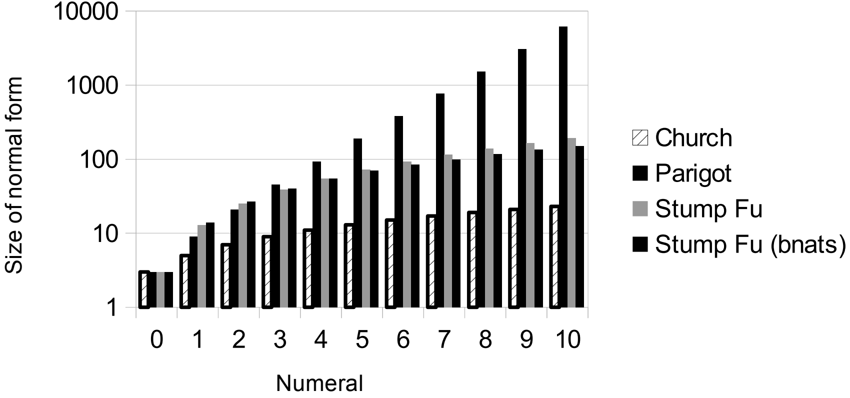
- For Racket (CBV), some adjustments needed:

```
Bool : ⋆ = ∀ X : ⋆ , X → X → X .
true : Bool = λ X:⋆, λx:X, λy: X, x.
false : Bool = λ X:⋆, λx: X, λy: X, y .
```

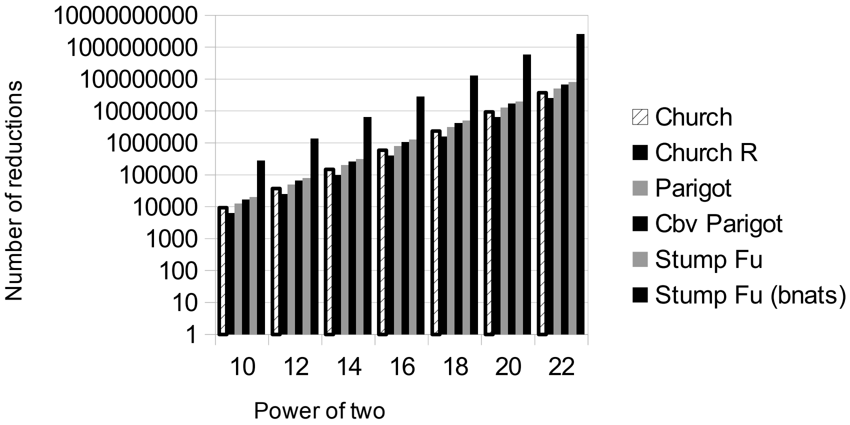
becomes

```
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



Exponentiation Test in Racket

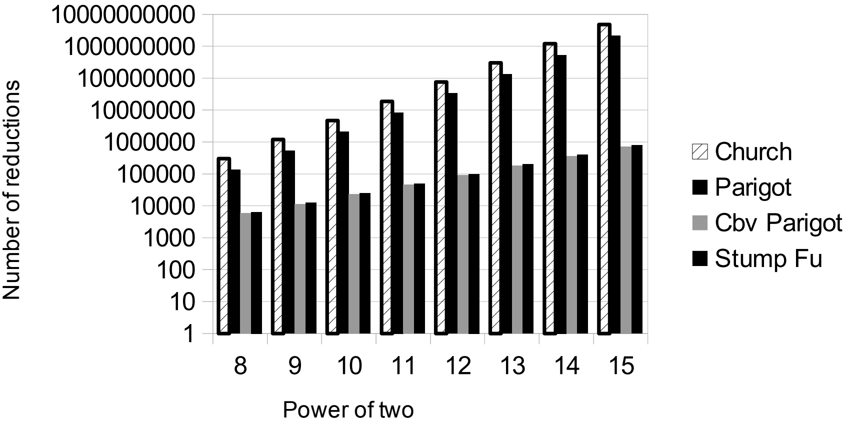


Exponentiation Test in Haskell

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

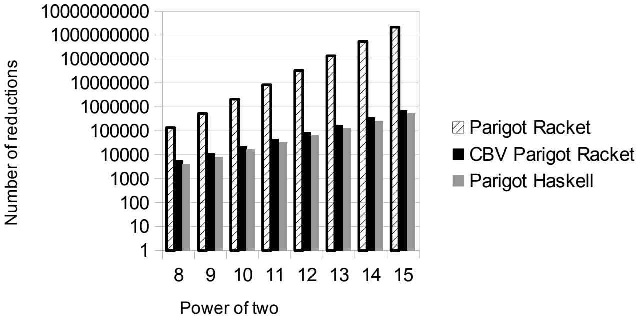
power	SF Racket	SF Haskell	SF (bnats) Racket	SF (bnats) Haskell
10	19765	19709	279455	260818
12	78185	78129	1336475	1246109
14	311709	311653	6249007	5822720
16	1245649	1245593	28647524	26681058

Subtraction Test in Racket



Subtraction Test in Haskell

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

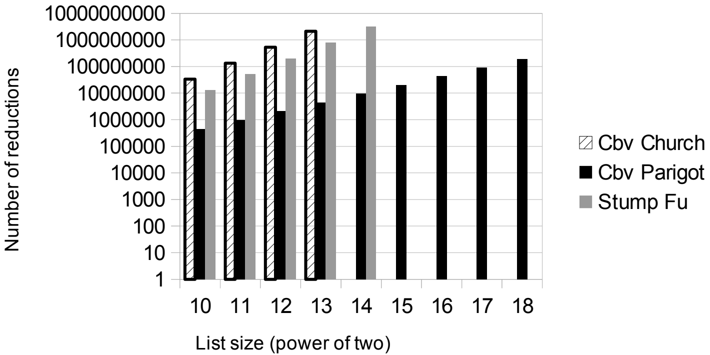


- Each predecessor takes one step less with lazy evaluation

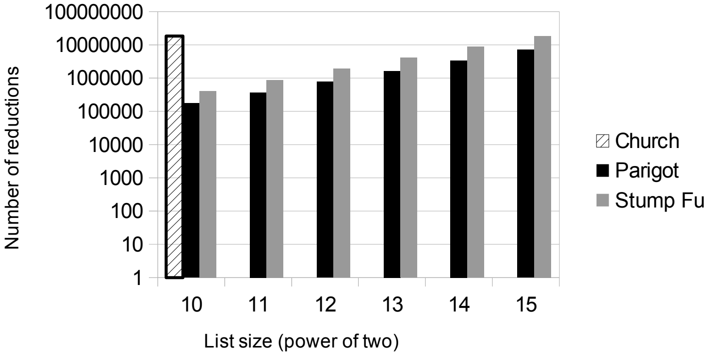
$$(x, y) \mapsto (\underline{suc\ x}, x)$$

Sorting Test in Racket

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

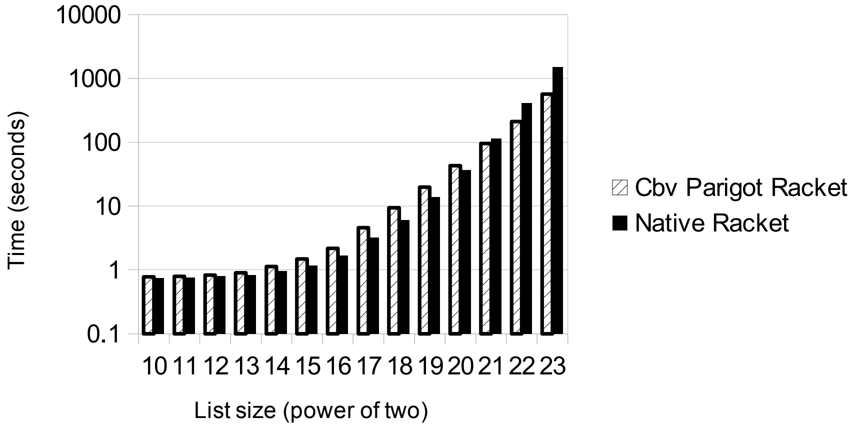


Sorting Test in Haskell



- 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

$$arrows\ A\ n = \underbrace{A \to \cdots A \to}_{n}\ A$$

A Paradise of

λ