Efficiency of Lambda Encodings in Total Type Theory

Aaron Stump   Peng Fu

Computational Logic Center
Computer Science
The University of Iowa

MVD '14

Programs

Programs =

$$\text{Programs} \quad = \quad \text{Functions} + \text{Data}$$

Programs = Functions + Data
+ Observations/IO

Programs = Functions + Data
+ Observations/IO
+ Concurrency

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
- Several different encodings known
- No need for datatypes (except *primitive types*)
- Simplify language design
  - Especially for type theories (*Coq*, *Agda*)
  - So need typed encodings
- Common benchmark example: unary numerals

$$0, \quad suc\, 0, \quad suc\,(suc\, 0), \quad \cdots$$

- How is performance?

The Church Encoding

- Data encoded as iterators (fold functions)

```
0 = λ s. λ z. z
1 = λ s. λ z. s z
2 = λ s. λ z. s (s z)
3 = λ s. λ z. s (s (s z))
...
```

- So *n s z* reduces to $s^n\ z$

```
suc = λ n. λ s. λ z. s (n s z)
```

- For addition, iterate `suc`:

```
add = λ n . λ m . n suc m
```

- Alternative clever versions due to Rosser
- Can be typed in System F
- But predecessor of *n* takes $O(n)$ steps!

The Parigot Encoding

- Data encoded as *recursors*

```
0 = λ s. λ z. z
1 = λ s. λ z. s 0 z
2 = λ s. λ z. s 1 (s 0 z)
3 = λ s. λ z. s 2 (s 1 (s 0 z))
...
suc = λ n. λ s. λ z. s n (n s z)
```

- Predecessor now takes $O(1)$ steps

```
pred = λ n. n (λ p . λ x . p) 0
```

- Can be typed in System F + positive-recursive type definitions
- But normal form of numeral n is size $O(2^n)$

New: Embedded-Iterators Encoding

- Same asymptotic time complexities as Parigot
- But: normal form of numeral n is only $O(n^2)$
- Basic idea: encode 2 as $(c2,(c1,(c0,0)))$, where $c2$, $c1$, and $c0$ are the Church-encodings of 2, 1, and 0 respectively

```
0 = λ s. λ z. z
1 = λ s. λ z. s c1 0
2 = λ s. λ z. s c2 1
3 = λ s. λ z. s c3 2
...
suc = λ n. n (λ c. λ p. λ s . λ z. s (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
```

- Typable in System F + positive-recursive type definitions
- Put embedded iterators in binary to reduce space to $O(n \log_2 n)$

Typing the Encodings

Church:

```
CNat : ⋆ = ∀ X : ⋆ , (X → X) → X → X .
Czero = λ X : ⋆ , λ s : X → X , λ z : X , z .
Cone = λ X : ⋆ , λ s : X → X , λ z : X , s z .
```

Parigot:

```
rec PNat : ⋆ = ∀ X : ⋆ , (PNat → X → X) → X → X .
Pzero = [ PNat ] λ X : ⋆ , λ s : PNat → X → X , λ z : X , z .
Pone = [ PNat ] λ X : ⋆ , λ s : PNat → X → X , λ z : X , s Pzero z .
```

Embedded iterators:

```
rec SFNat : ⋆ = ∀ X : ⋆, (CNat → SFNat → X) → X → X .
SFzero = [SFNat] λ X : ⋆, λ s : CNat → SFNat → X , λ z : X , z .
SFOne = [SFNat] λ X : ⋆, λ s : CNat → SFNat → X , λ z : X ,
                  s Cone SFzero .
```

Implementation

- `fore` tool for F_ω + positive-recursive type definitions
- Compiles `fore` terms to Racket, Haskell
- For Racket, erase all type annotations
- For Haskell, use `newtype`

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

- Translate computed answers by translating to native data

```
toInt :: CNat -> Int
toInt n = unfoldCNat n (\ x -> 1 + x) 0

instance Show CNat where
  show n = show (toInt n)
```

- 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

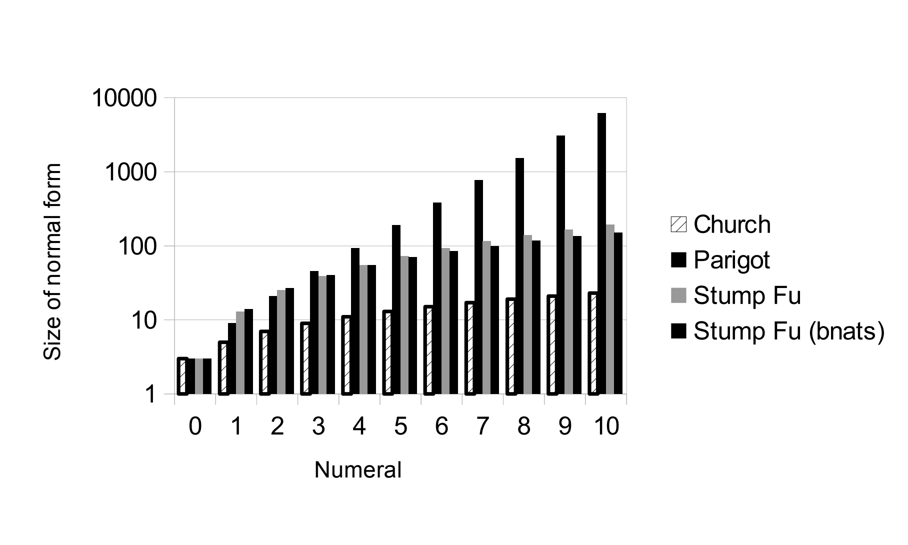
- For Racket (CBV), some adjustments needed:

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

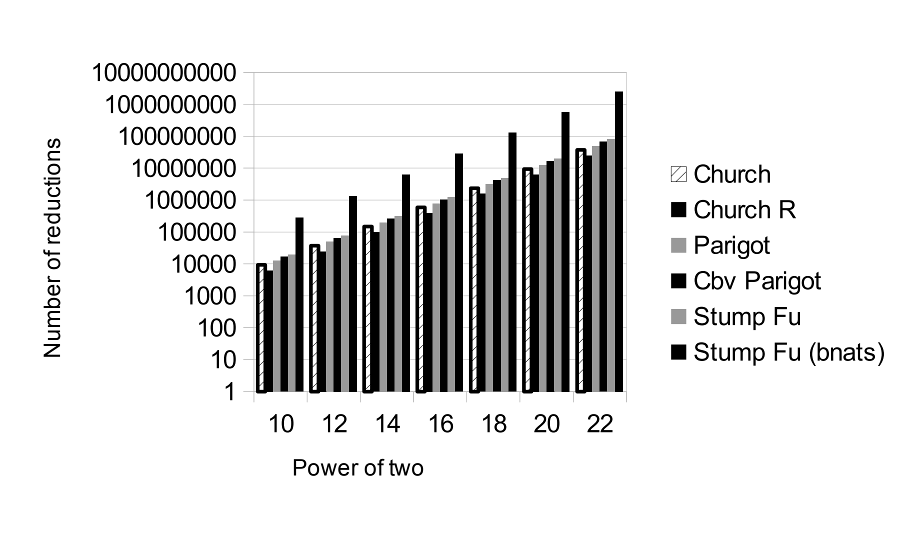
becomes

```
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

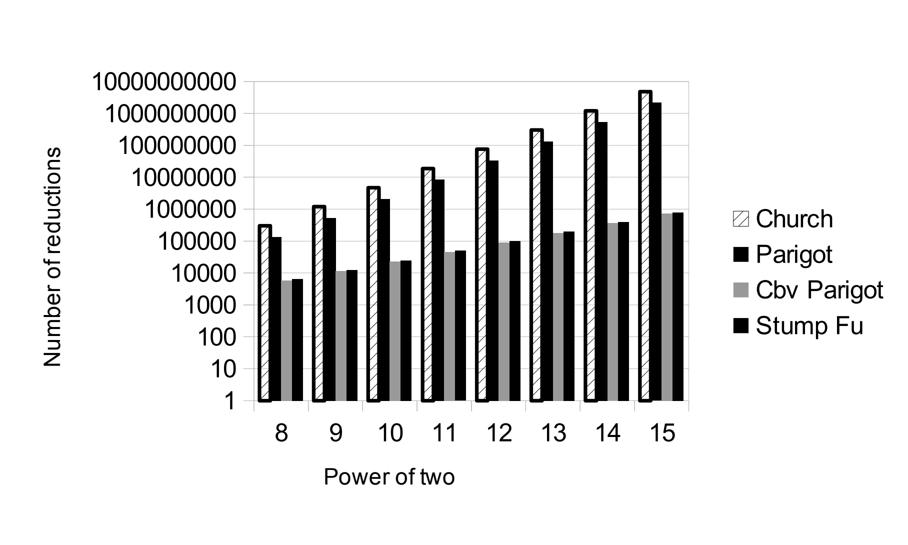


Exponentiation Test in Haskell

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

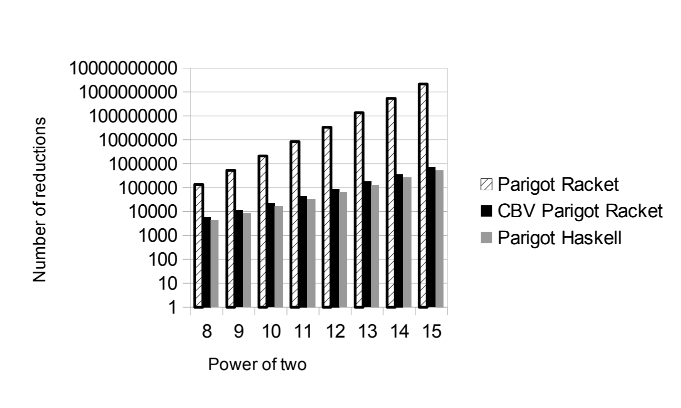
power	SF Racket	SF Haskell	SF (bnats) Racket	SF (bnats) Haskell
10	19765	19709	279455	260818
12	78185	78129	1336475	1246109
14	311709	311653	6249007	5822720
16	1245649	1245593	28647524	26681058

Subtraction Test in Racket



Subtraction Test in Haskell

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

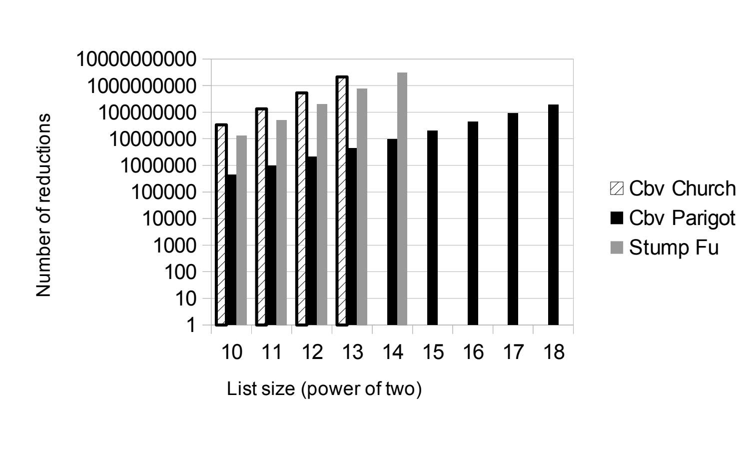


- Each predecessor takes one step less with lazy evaluation

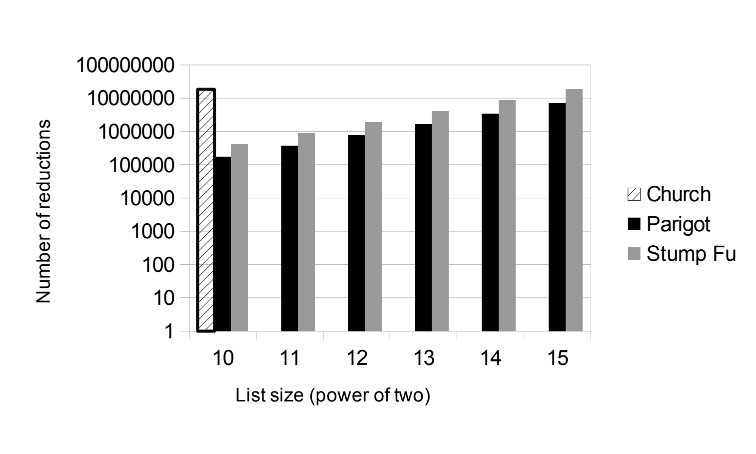
$$(x, y) \mapsto (\underline{suc\ x}, x)$$

Sorting Test in Racket

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

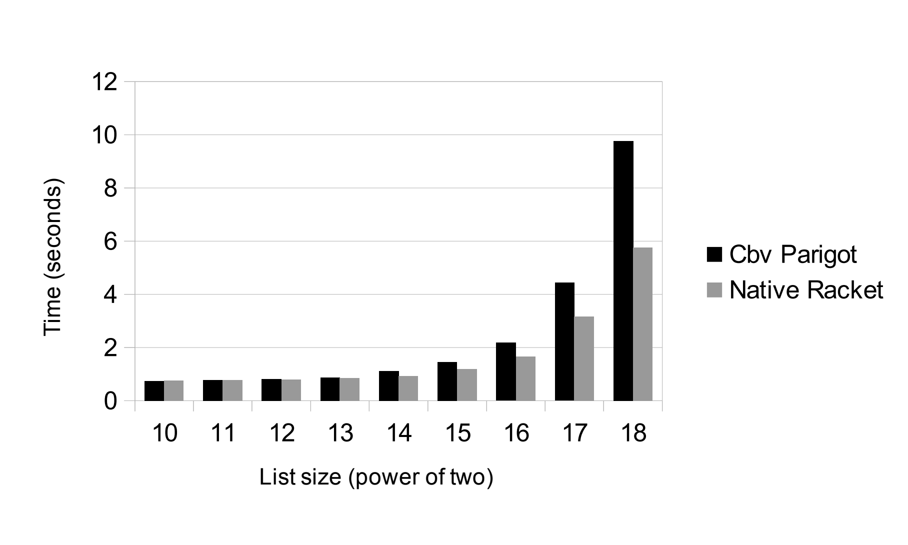


Sorting Test in Haskell



- 14: embedded iterators 350 times fewer reductions
- 14: Parigot 2.8 times fewer

Comparison with Native Racket



Conclusion

- New embedded iterators encoding
  - Expected asymptotic time complexities (like Parigot)
  - Size of normal form of n is $O(n^2)$
  - Best encoding if size of normal form matters
- Promising empirical results for embedded iterators, Parigot
  - CBV Parigot within a factor of 2 (wallclock) of native Racket sort!
- Hope for using lambda encodings for data (structures)

Conclusion

- New embedded iterators encoding
  - Expected asymptotic time complexities (like Parigot)
  - Size of normal form of n is $O(n^2)$
  - Best encoding if size of normal form matters
- Promising empirical results for embedded iterators, Parigot
  - CBV Parigot within a factor of 2 (wallclock) of native Racket sort!
- Hope for using lambda encodings for data (structures)

Programs = Functions

StarExec

A Web Service for Evaluating Logic Solvers

www.starexec.org

Aaron Stump, Geoff Sutcliffe, Cesare Tinelli

Cross-Community Web Service for Logic Solvers

- Cluster with 192 compute nodes
  - dual-processor, quad-core
  - most have 256GB physical memory
- Upload solvers, benchmarks
- Run jobs
- Many different communitites already there
  - SMT, TPTP, QBF, SyGuS, Termination, Confluence
  - Each ran a competition summer 2014 on StarExec
- Open to anyone to register or try as guest

`www.starexec.org`