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  =
Programs = Functions + 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
  
  \[ 0, \ suc\ 0, \ suc\ (suc\ 0), \ \ldots \]

- How is performance?
The Church Encoding

- Data encoded as iterators (fold functions)
  
  \[0 = \lambda s. \lambda z. \, z\]
  
  \[1 = \lambda s. \lambda z. \, s \, z\]
  
  \[2 = \lambda s. \lambda z. \, s \, (s \, z)\]
  
  \[3 = \lambda s. \lambda z. \, s \, (s \, (s \, z))\]
  
  ... 

- So \(n \, s \, z\) reduces to \(s^n \, z\)

\[\text{suc} = \lambda n. \lambda s. \lambda z. \, s \, (n \, s \, z)\]

- For addition, iterate \(\text{suc}\):

\[\text{add} = \lambda n. \lambda m. \, n \, \text{suc} \, m\]

- 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*

  \[ 0 = \lambda s. \lambda z. z \]
  \[ 1 = \lambda s. \lambda z. s\ 0\ z \]
  \[ 2 = \lambda s. \lambda z. s\ 1\ (s\ 0\ z) \]
  \[ 3 = \lambda s. \lambda z. s\ 2\ (s\ 1\ (s\ 0\ z)) \]
  ...

  \[ \text{suc} = \lambda n. \lambda s. \lambda z. s\ n\ (n\ s\ z) \]

- Predecessor now takes \(O(1)\) steps

  \[ \text{pred} = \lambda n. n\ \left(\lambda p. \lambda x. p\right)\ 0 \]

- 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 $(c_2, (c_1, (c_0, 0)))$, where $c_2$, $c_1$, and $c_0$ are the Church-encodings of 2, 1, and 0 respectively

$$
\begin{align*}
0 &= \lambda s. \lambda z. z \\
1 &= \lambda s. \lambda z. s\, c_1\, 0 \\
2 &= \lambda s. \lambda z. s\, c_2\, 1 \\
3 &= \lambda s. \lambda z. s\, c_3\, 2 \\
&\quad \ldots \\
\text{suc} &= \lambda n. n\, (\lambda c. \lambda p. \lambda s. \lambda z. s\, (c\, \text{suc}\, c)\, n)\, 1
\end{align*}
$$

- Use embedded Church-encoded numbers for iteration
- $\text{add} = \lambda n. \lambda m. n\, (\lambda c. \lambda p. c\, \text{suc}\, m)\, m$
- 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:

\[ \text{CNat} : \forall \ X : \ *, \ (X \to X) \to X \to X . \]
\[ \text{Czero} = \lambda \ X : \ *, \ \lambda \ s : X \to X , \ \lambda \ z : X , \ z . \]
\[ \text{Cone} = \lambda \ X : \ *, \ \lambda \ s : X \to X , \ \lambda \ z : X , \ s \ z . \]

Parigot:

\[ \text{rec PNat} : \forall \ X : \ *, \ (\text{PNat} \to X \to X) \to X \to X . \]
\[ \text{Pzero} = [\text{PNat}] \ \lambda \ X : \ *, \ \lambda \ s : \text{PNat} \to X \to X , \ \lambda \ z : X , \ z . \]
\[ \text{Pone} = [\text{PNat}] \ \lambda \ X : \ *, \ \lambda \ s : \text{PNat} \to X \to X , \ \lambda \ z : X , \ s \ \text{Pzero} \ z . \]

Embedded iterators:

\[ \text{rec SFNat} : \forall \ X : \ *, \ (\text{CNat} \to \text{SFNat} \to X) \to X \to X . \]
\[ \text{SFzero} = [\text{SFNat}] \ \lambda \ X : \ *, \ \lambda \ s : \text{CNat} \to \text{SFNat} \to X , \ \lambda \ z : X , \ z . \]
\[ \text{SFOne} = [\text{SFNat}] \ \lambda \ X : \ *, \ \lambda \ s : \text{CNat} \to \text{SFNat} \to X , \ \lambda \ z : X , \ s \ \text{Cone} \ \text{SFzero} . \]
Implementation

- **fore** tool for $F_\omega +$ positive-recursive type definitions
- Compiles **fore** terms to Racket, Haskell
- For Racket, erase all type annotations
- For Haskell, use **newtype**

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

- Translate computed answers by translating to native data

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

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

- 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:

  \[
  \text{Bool} : * = \forall X : *, X \rightarrow X \rightarrow X .
  \]

  \[
  \text{true} : \text{Bool} = \lambda X:* , \lambda x:X , \lambda y:X , x.
  \]

  \[
  \text{false} : \text{Bool} = \lambda X:* , \lambda x:unit \rightarrow X , \lambda y:unit \rightarrow X , y .
  \]

  becomes

  \[
  \text{Bool} : * = \forall X : *, (\text{unit} \rightarrow X) \rightarrow (\text{unit} \rightarrow X) \rightarrow X .
  \]

  \[
  \text{true} : \text{Bool} = \lambda X:* , \lambda x:\text{unit} \rightarrow X , \lambda y:\text{unit} \rightarrow X , x \text{ triv}.
  \]

  \[
  \text{false} : \text{Bool} = \lambda X:* , \lambda x:\text{unit} \rightarrow X , \lambda y:\text{unit} \rightarrow X , y \text{ triv} .
  \]
Sizes of Normal Forms

- Church
- Parigot
- Stump Fu
- Stump Fu (bnats)

Size of normal form vs. Numeral

- Church
- Parigot
- Stump Fu
- Stump Fu (bnats)
Exponentiation Test in Racket

![Bar chart showing the number of reductions for different lambda encodings across powers of two. The chart compares Church, Church R, Parigot, Cbv Parigot, Stump Fu, and Stump Fu (bnats).]
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

Number of reductions

Power of two

- Church
- Parigot
- Cbv Parigot
- Stump Fu

Stump, Fu  Efficiency of Lambda Encodings  MVD '14
Subtraction Test in Haskell

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

Each predecessor takes one step less with lazy evaluation

\[(x, y) \mapsto (\text{succ } x, x)\]
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

![Bar chart comparing Cbv Parigot and Native Racket for list size (power of two) vs time (seconds). The chart shows a significant difference in time for list sizes 10 and 18, with Native Racket being slower.]

Stump, Fu
Efficiency of Lambda Encodings
MVD ’14
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 communities 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