Under consideration for publication in J. Functional Programming

Efficiency of Lambda-Encodings in Total Type Theory

Aaron Stump
Computer Science, The University of Iowa, Iowa City, Iowa, USA
astump@acm.org
Peng Fu
Computer Science, The University of Dundee, Dundee, Scotland
pfu@dundee.ac.uk

Abstract

This paper proposes a new typed lambda-encoding for inductive types which, for Peano numerals, has the expected time complexities for basic operations like addition and multiplication, has a constant-time predecessor function, and requires only quadratic space to encode a numeral. This improves on the exponential space required by the Parigot encoding. Like the Parigot encoding, the new encoding is typable in System F-omega plus positive-recursive type definitions, a total type theory. The new encoding is compared with previous ones through a significant case study: mergesort using Braun trees. The practical runtime efficiency of the new encoding, and the Church and Parigot encodings, are compared by two translations, one to Racket and one to Haskell, on a small suite of benchmarks.

1 Introduction

The idea of encoding data as functional terms in typed lambda calculus has significant appeal, as it shows that primitive datatypes are, at least in principle, unnecessary. The traditional objection to lambda-encoding data in typed lambda calculi has been asymptotic inefficiencies for the encoded data or the operations on them. The well-known Church encoding of standard datatypes and natural operations on them can be typed in System F, a total type theory (all terms are guaranteed to terminate). But operations to extract subdata from data – like the tail from a list – take time linear in the size of the data. At least for Peano numerals, this is a provable lower bound (Parigot, 1989). For practical functional programming, destructors of inductive datatypes should be computed in constant time. Parigot improved on this situation with an encoding which intrinsically supports recursion, and has constant-time predecessor (Parigot, 1988). Parigot's encoding is typable in System F plus positive-recursive type definitions, which is also a total type theory. It has one major drawback, however: the size of a numeral n is $O(2^n)$.

In this paper, we introduce a new lambda-encoding, called **embedded iterators**, which has similar computational properties as Parigot's, but which requires only $O(n^2)$ space to represent the unary numeral n, while preserving all the expected time complexities for the basic arithmetic operations (Section 4). If one is willing to accept an increase in the time complexity for successor from $O(1)$ to $O(\log n)$, then this requirement can be reduced to

$O(n \log n)$ for numeral n (Section 4.3). Like the Parigot encoding, the embedded-iterators encoding is typable in System F extended with global positive-recursive type definitions. We will first review Church, Scott, and Parigot encodings (Section 3), based on System F_ω with global positive-recursive type definitions (Section 2). Then we will present the embedded-iterators encoding (Section 4). We will also discuss how to encode some container datatypes, which, unlike the situation for numeric data (for which using numeric representations native to the computing hardware is mandatory for performance), could actually be useful in practice (Section 5).

The paper's final contribution is an empirical assessment of these different encoding schemes. We give a performance comparison of these encodings, using Racket and Haskell as platforms for efficient execution (Section 6). One interesting finding is that the Parigot and embedded-iterators encodings perform well in practice, despite the theoretical asymptotically greater size of their normal forms. This suggests that typed lambda encodings may be more suitable for practical use than previously believed.

2 F_ω^{rec}: F_ω with Global Positive-Recursive Type Definitions

The F_ω^{rec} type theory we will use in this paper is a version of the well-known F_ω system, which extends the impredicative polymorphism of System F with type-level computation. F_ω^{rec} system extends F_ω with global recursive definitions of types at any kind, where the defined type symbol can appear only positively in the definition. We call these *positive-recursive* type definitions, and define an occurrence to be positive iff it is in the domain part of an even number of arrow types, and not in the argument part of a type-level application (see Definition 2.1 below). The latter restriction is to avoid misjudging the second occurrence of X in $(\lambda Y : *.Y \to X)\ X$, for example, as positive. Using global recursive definitions of types instead of recursive types ($\mu X.T$) we can more easily specify how to fold and unfold recursive types. The cost is that nested datatypes, like the type of finitely branching trees, cannot be defined. This limitation could be removed with polarized kinds, as used by Abel and Matthes for their system Fix$^\omega$ (Abel & Matthes, 2004). Fix$^\omega$ uses a type-level fixed-point operator instead of recursive type equations.

2.1 Syntax

The following definitions constitute a new formulation of a system in (Fu & Stump, 2014). The syntax for kinds, types, terms, and contexts is:

$$
\begin{array}{lll}
\textit{Term variables } x & & \\
\textit{Type variables } X & & \\
\textit{Kinds } \kappa & ::= & * \mid \kappa' \to \kappa \\
\textit{Types } T & ::= & X \mid \forall X : \kappa . T \mid T_1 \to T_2 \mid \lambda X : \kappa_1 . T \mid T_1\ T_2 \\
\textit{Terms } t & ::= & x \mid \lambda x : T . t \mid \lambda X : \kappa . t \mid t\ t' \mid t\ T \mid [X!] \mid [X] \\
\textit{Contexts } \Gamma & ::= & \cdot \mid \Gamma, x : T \mid \Gamma, X : \kappa \mid \Gamma, X : \kappa \mapsto T
\end{array}
$$

The term constructs $[X!]$ and $[X]$ are for the folding and unfolding, respectively, of recursive type-definitions. The entry $X : \kappa \mapsto T$ in contexts is for a recursive type definition: X of kind κ is recursively defined to be T, where T may mention X.

$$\frac{}{\mathrm{Pol}(X,X,\mathbf{0})} \qquad \frac{X \not\equiv Y}{\mathrm{Pol}(X,Y,b)} \qquad \frac{\mathrm{Pol}(X,T_1,\neg b) \quad \mathrm{Pol}(X,T_2,b)}{\mathrm{Pol}(X,T_1 \to T_2,b)}$$

$$\frac{\mathrm{Pol}(X,T_1,b) \quad X \notin \mathrm{FV}(T_2)}{\mathrm{Pol}(X,(T_1\ T_2),b)} \qquad \frac{\mathrm{Pol}(X,T,b)}{\mathrm{Pol}(X,(\forall Y:\kappa.T),b)} \qquad \frac{\mathrm{Pol}(X,T,b)}{\mathrm{Pol}(X,\lambda Y:\kappa.T,b)}$$

Fig. 1. Definition of polarity in F^{rec}_{ω}

$$\frac{}{\cdot\ \mathrm{ok}} \qquad \frac{\Gamma \vdash T : * \quad \Gamma\ \mathrm{ok}}{\Gamma, x:T\ \mathrm{ok}} \qquad \frac{\Gamma\ \mathrm{ok}}{\Gamma, X:\kappa\ \mathrm{ok}}$$

$$\frac{\Gamma, X:\kappa \vdash T:\kappa \quad \mathrm{Pol}(X,T,\mathbf{0}) \quad \Gamma\ \mathrm{ok}}{\Gamma, X:\kappa \mapsto T\ \mathrm{ok}} \qquad \frac{(X:\kappa) \in \Gamma \quad \Gamma\ \mathrm{ok}}{\Gamma \vdash X:\kappa} \qquad \frac{\Gamma, X:\kappa \vdash T:\kappa'}{\Gamma \vdash \lambda X:\kappa.T : \kappa \to \kappa'}$$

$$\frac{\Gamma, X:\kappa \vdash T : *}{\Gamma \vdash \forall X:\kappa.T : *} \qquad \frac{\Gamma \vdash T : * \quad \Gamma \vdash T' : *}{\Gamma \vdash T \to T' : *} \qquad \frac{\Gamma \vdash T : \kappa_1 \to \kappa_2 \quad \Gamma \vdash T' : \kappa_1}{\Gamma \vdash T\ T' : \kappa_2}$$

Fig. 2. Context formation and kinding for F^{rec}_{ω}

2.2 Classification

Figure 2 gives rules inductively defining the judgments Γ ok, for well-formedness of the context Γ, and $\Gamma \vdash T : \kappa$, stating that type T has kind κ in context Γ. We use the following adaptation of the notion of polarity to restrict type definitions to be positive-recursive.

Definition 2.1 (*Polarity*)
Let $b \in \{\mathbf{0},\mathbf{1}\}$, and define $\neg\mathbf{0} := \mathbf{1}, \neg\mathbf{1} := \mathbf{0}$. We define relation $\mathrm{Pol}(X,T,b)$ by the rules of Figure 1. Informally, this means all occurences of X in T have polarity b. We say X occurs only positively in T if $\mathrm{Pol}(X,T,\mathbf{0})$, and only negatively if $\mathrm{Pol}(X,T,\mathbf{1})$.

Figure 3 gives rules defining the judgment $\Gamma \vdash t : T$, stating that term t has type T in context Γ. One of the rules uses $\cong$ for the least congruence relation satisfying the standard equation for type-level β-equivalence:

$$(\lambda X:\kappa.T)\ T' \cong [T'/X]T$$

Figure 3 has specialized rules for folding and unfolding recursive type-definitions of symbols X to be types T of kind κ. Because such definitions can be at kinds higher than $*$, we need to handle arguments to the recursively defined type. We do this by quantifying over the types which are inputs to X, in the types for $[X!]$ and $[X]$. We use $\bar{\kappa}$ to denote a finite, possibly empty sequence of kinds $\kappa_1,\cdots,\kappa_n$. We then use the notation $\bar{\kappa} \to \kappa'$ for the right-nested function type $\kappa_1 \to \cdots \to \kappa_n \to \kappa'$, which is just κ' if $\bar{\kappa}$ is empty. Finally, we write $X\ \bar{X}$ for the left-nested application $((X\ X_1)\cdots)\ X_n$, which is just X if $\bar{X}$ is empty. Also, assuming $\bar{X}$ is a sequence of distinct type variables of the same length as $\bar{\kappa}$, $\forall\bar{X}:\bar{\kappa}.\ T$ denotes $\forall X_1:\kappa_1.\ \cdots \forall X_n:\kappa_n.\ T$ – again, just T if $\bar{X}$ is empty. Incorporating this sequence $\bar{\kappa}$ into the typing rules for $[X!]$ and $[X]$ generalizes the simple case where X has kind $*$, and $[X!]$ and $[X]$ simply witness the isomorphism between X and T.

$$\frac{(x:T)\in\Gamma \quad \Gamma \text{ ok}}{\Gamma\vdash x:T} \qquad \frac{\Gamma\vdash t:(T_1\to T_2) \quad \Gamma\vdash t':T_1}{\Gamma\vdash t\ t':T_2} \qquad \frac{\Gamma,x:T_1\vdash t:T_2}{\Gamma\vdash \lambda x:T_1.t:(T_1\to T_2)}$$

$$\frac{\Gamma\vdash t:T_1 \quad \Gamma\vdash T_1\cong T_2}{\Gamma\vdash t:T_2} \qquad \frac{\Gamma\vdash t:\forall X:\kappa.T \quad \Gamma\vdash T':\kappa}{\Gamma\vdash t\ T':[T'/X]T} \qquad \frac{\Gamma,X:\kappa\vdash t:T \quad X\notin \text{FVar}(\Gamma)}{\Gamma\vdash \lambda X:\kappa.t:\forall X:\kappa.T}$$

$$\frac{X:(\bar\kappa\to *)\mapsto T\in\Gamma \quad \Gamma \text{ ok} \quad \Gamma\vdash \bar T:\bar\kappa}{\Gamma\vdash [X]:\forall\bar X:\bar\kappa.\ T\ \bar X\to X\ \bar X} \qquad \frac{X:(\bar\kappa\to *)\mapsto T\in\Gamma \quad \Gamma \text{ ok}}{\Gamma\vdash [X!]:\forall\bar X:\bar\kappa.\ X\ \bar X\to T\ \bar X}$$

Fig. 3. Typing for F_ω^{rec}

$$\begin{array}{lcllcl} |\lambda x:T.t| & = & \lambda x.|t| & |\lambda X:\kappa.t| & = & |t| \\ |t\ t'| & = & |t|\ |t'| & |t\ T| & = & |t| \\ |[X!]| & = & \lambda x.x & |x| & = & x \\ |[X]| & = & \lambda x.x & & & \end{array}$$

Fig. 4. Erasure to untyped lambda calculus

2.3 Strong Normalization

In Appendix C of (Fu & Stump, 2014), the following theorem is proved for a more declarative formulation of F_ω^{rec}: folding and unfolding of recursive types take place as part of a nonalgorithmic definitional equality, and the system is a type-assignment system (so the only term constructs are those of pure untyped lambda calculus). Similarly to the proof in (Abel & Matthes, 2004), a complete lattice $(\llbracket\kappa\rrbracket,\subseteq_\kappa,\cap_\kappa)$ is defined, for each kind κ. For the base case, $(\llbracket *\rrbracket,\subseteq_*,\cap_*)$ is the set of reducibility candidates ordered by inclusion, with intersection for the meet operation. These lattices allow the interpretation of positive-recursive type definitions via fixed points. The erasure $|t|$ is defined in Figure 4. Erasure eliminates $[X!]$ and $[X]$, so F_ω^{rec} does not need reduction or equivalence rules relating these constructs.

Theorem 2.1
$\Gamma\vdash t:T$ implies that $|t|$ is strongly normalizing (no infinite reduction sequence from $|t|$).

2.4 Implementation

We have implemented F_ω^{rec} in a tool called `fore`, available from the Software section of the first author's web page. `fore` is written in the dependently typed functional programming language Agda, version 2.4.2.2 (Norrell & the Agda Development Team, 2014). Agda compiles to Haskell, which enables `fore` to execute reasonably efficiently. `fore` includes support for non-recursive term- and type definitions, with the latter unfolded automatically as needed during type checking. Types can be positively recursively defined with the keyword `rec`. `fore` also uses Unicode symbols for λ, $\forall$, and $\to$. The distinction between term and type variables is implemented by consulting the context to see whether the variable is declared to have a type or a kind. So a lexical distinction between term and type variables is not required.

`fore` includes a type checker for an algorithmic adaptation of F_ω^{rec}. In particular, one has to eliminate the otherwise nondeterministically applicable conversion rule, in favor of sometimes reducing computed types to head normal form. This is possible, thanks to

confluence and strong normalization of type-level reduction (which is just that of simply typed lambda calculus). The examples in the rest of the paper have all been type-checked with `fore`. The `fore` tool can translate input programs written in the `fore` input syntax to either Haskell or Racket.

Agda supports verification of pure functional programs via dependent types and the Curry-Howard isomorphism. While we have not attempted to verify deep properties like type preservation for our type checker with respect to the operational semantics of Racket or Haskell – which would be a major undertaking – we have used Agda's verification capabilities in the course of our development to improve the quality of our tool. One example theorem we have proved is correctness of an algorithm for inserting a minimal set of disambiguating parentheses into terms to be printed back to the user. We have also expressed some simple data structure invariants using dependent types.

3 Previous Lambda Encodings

In this section we recall the Church, Scott, and Parigot encodings of natural numbers in F_ω^{rec}. The Church encoding, of course, requires only System F (a subsystem of F_ω^{rec}), while the other two do require positive-recursive type definitions. Type-level λ-abstraction is not needed for these encodings, but is required for container types (see Section 5). This section will consider just unary (aka Peano) natural numbers, via their `fore` input sources. We elide the definitions of basic non-recursive datatypes for pair types, sum types, booleans, the unit type, and a maybe type. Because these are not recursive, the different lambda-encodings all agree for them.

3.1 The Church encoding

The Church encoding represents each natural number n as its own iterator $\lambda s.\lambda z.s^n\ z$, where $s^n\ z$ is meta-notation for $(s \cdots (s\ z))$, with n copies of s (Church, 1941). Böhm and Berarducci showed how to type these in System F (Böhm & Berarducci, 1985). In the notation of `fore`, the (non-recursive) definitions for the type `CNat` of Church numerals and the constructors `CZero` (for 0) and `CSuc` (for successor) are these, where each definition lists the defined symbol, its classifier (type or kind), and then the term or type it is defined to equal.

```
CNat : * =
  ∀ X : * , (X → X) → X → X .
Czero : CNat =
  λ X : * , λ s : X → X , λ z : X , z .
Csuc : CNat → CNat =
  λ n : CNat , λ X : * , λ s : X → X , λ z : X , s (n X s z) .
```

It is convenient also to define `Cone` for 1:

```
Cone : CNat =
  λ X : * , λ s : X → X , λ z : X , s z .
```

Iterative versions of addition, multiplication, and exponentiation are defined this way:

```
Cadd : CNat → CNat → CNat =
  λ n : CNat , λ m : CNat , n CNat Csuc m .
Cmult : CNat → CNat → CNat =
  λ n : CNat , λ m : CNat , n CNat (Cadd m) Czero .
Cexp : CNat → CNat → CNat =
  λ n : CNat , λ m : CNat , n CNat (Cmult m) Cone .
```

Alternative clever definitions of these are attributed to Rosser (Barendregt, 1985):

```
CaddR : CNat → CNat → CNat =
  λ n : CNat , λ m : CNat , λ X : * , λ s : X → X , λ z : X ,
    n X s (m X s z).
CmultR : CNat → CNat → CNat =
  λ n : CNat , λ m : CNat , λ X : * , λ s : X → X , λ z : X ,
    n X (m X s) z .

CexpR : CNat → CNat → CNat =
  λ n : CNat , λ m : CNat , λ X : * ,
    m (X → X) (n X) .
```

We will evaluate both sets of definitions in Section 6. See (Hinze, 2005) for a study of how the different definitions can be derived from alternative specifications of the operations.

The definition of predecessor in the Church encoding provably requires linear time (Parigot, 1989). The standard algorithm is due to Kleene. Informally, one iterates the function $(x,m) \mapsto (m,m+1)$ starting from $(0,0)$. After $n+1$ iterations, the result is $(n,n+1)$, and hence the predecessor is available as the first component of the pair (and after 0 iterations 0 is still the cut-off predecessor). Subtraction is then just iterated predecessor.

3.2 The Scott encoding

In a total type theory, the Scott encoding is of limited use, because unlike the Church encoding, Scott-encoded data do not intrinsically support iteration or recursion. So the operations on Scott-encoded data do not appear to be definable without a more general recursion operator, which is not available in a total type theory. Nevertheless, the Scott encoding is a stepping stone to the Parigot encoding, so we include the definitions here. See page 504 of (Curry *et al.*, 1972) for the attribution to Scott.

```
rec SNat : * =
  ∀ X : * , (SNat → X) → X → X .
Szero : SNat =
  [SNat] λ X : * , λ s : SNat → X , λ z : X , z .
Ssuc : SNat → SNat =
  λ n : SNat , [SNat] λ X : * , λ s : SNat → X , λ z : X , s n .
```

Note the use of the fold operator `[SNat]` to map terms of the following type to `SNat`:

```
∀ X : * , (SNat → X) → X → X
```

The essential difference with the Church encoding is that the abstracted successor `s` takes in not the type parameter X as its input, but rather `SNat`. This necessitates the use of a recursive type definition for `SNat`, since `SNat` appears on the right-hand side of the definition. That occurrence is positive, though, and so is allowed in F_{ω}^{rec}. We see from the definition of `Ssuc` (successor) that each non-zero numeral $n+1$ is represented in the encoding by applying the abstracted successor `s` to the predecessor n.

3.3 The Parigot encoding

Parigot introduced an encoding of numerals which in a sense combines the Church and Scott encodings (indeed, Parigot numerals are sometimes also called Church-Scott numerals) (Parigot, 1988). Where the Church encoding represents numerals as their own iterators, the Parigot encoding represents them as their own recursors. Here are the basic definitions, which as for the Scott encoding require a positive-recursive definition for the type for numerals:

```
rec PNat : * =
  ∀ X : * , (PNat → X → X) → X → X .
Pzero : PNat =
  [ PNat ] λ X : * , λ s : PNat → X → X , λ z : X , z .
Psuc : PNat → PNat =
  λ n : PNat , [ PNat ] λ X : * , λ s : PNat → X → X , λ z : X ,
    s n ([PNat !] n X s z) .
```

In the definition of `Psuc` (successor), we see that the abstracted `s` is applied both to the predecessor number `n` as in the Scott encoding, and also to an application of `n` to `X s z`, as in the Church encoding. For this application of `n`, we first unfold the type `PNat` of `n` using our unfold operator `[PNat !]`. The dual use of the predecessor is the strength of the Parigot encoding, because numerals both intrinsically support recursion and allow constant-time access to subdata. It is also the source of two serious weaknesses:

- Unlike in the Church and Scott encodings, the size of a numeral n in normal form is exponential in n.
- The representation is not adequate, in the sense that there are closed normal forms of type `PNat` which do not represent any unary number, because the Scott part of the encoding and the Church part are out of sync. An example is a term like

  ```
  λ X : * , λ s : PNat → X → X , λ z : X , s Pzero (s Pzero z)
  ```

Here are the definitions of one, addition, multiplication, and exponentation:

```
Pone : PNat =
  [ PNat ] λ X : * , λ s : PNat → X → X , λ z : X , s Pzero z .
Padd : PNat → PNat → PNat =
  λ n : PNat , λ m : PNat , [PNat!] n PNat (λ P : PNat , Psuc) m .
Pmult : PNat → PNat → PNat =
  λ n : PNat , λ m : PNat , [PNat!] n PNat (λ P : PNat , Padd m) Pzero .
Pexp : PNat → PNat → PNat =
  λ n : PNat , λ m : PNat , [PNat!] n PNat (λ P : PNat , Pmult m) Pone .
```

Constant-time predecessor and subtraction by iterated predecessor are easily defined (though see the next section for an important twist):

```
Ppred : PNat → PNat =
  λ n : PNat , [PNat!] n PNat (λ P : PNat , λ d : PNat , P) Pzero .
Psubtract : PNat → PNat → PNat =
  λ n : PNat , λ m : PNat , [PNat!] m PNat (λ P : PNat , Ppred) n .
```

3.4 The Parigot encoding for call-by-value reduction

Parigot designed the preceding encoding so that the predecessor operation would take constant time. And so indeed it does, if one is using call-by-name or normal-order reduction, for example. But if one uses call-by-value – the reduction strategy employed by widely used functional programming languages like OCaml and Racket – then predecessor is not a constant-time operation, because `Ppred` (defined just above) still works by iterating with the number `n` of which the predecessor is desired. The function that is being iterated, namely `λ P : PNat , λ d : PNat , P`, does not need to use the result `d` of iteration; it just immediately returns the predecessor `P` which the number `n` passes to it. But with call-by-value reduction, that result `d` of iteration will still be computed before it is discarded by the iterated function. So in call-by-value reduction, the Parigot encoding does not have constant-time predecessor. Note that here we are speaking of call-by-value reduction for pure untyped lambda (for the erasures of the terms), where the values are the λ-abstractions. For an example: let us write $\check{n}$ for the erasure (to untyped lambda calculus) of Parigot-encoded natural number n, and (ambiguously) `Ppred` for the erasure of `Ppred` as defined above. Then we have this example call-by-value reduction (writing $\rightsquigarrow^n$ for the n-fold composition of $\rightsquigarrow$):

$$\begin{array}{ll} \texttt{Ppred}\ \check{2} & \rightsquigarrow^3 \\ (\lambda P.\lambda d.P)\ \check{1}\ ((\lambda P.\lambda d.P)\ \check{0}\ \check{0}) & \rightsquigarrow \\ (\lambda d.\check{1})\ ((\lambda P.\lambda d.P)\ \check{0}\ \check{0}) & \rightsquigarrow^3 \\ \check{1} & \end{array}$$

The point is that the last shown step is reducing the argument $((\lambda P.\lambda d.P)\ \check{0}\ \check{0})$ to a value v before discarding it when reducing $(\lambda d.\check{1})\ v$. This leads to $O(n)$ call-by-value steps to reduce `Ppred` $\check{n}$, instead of $O(1)$ steps.

This problem can be easily remedied, however, using the following alternative definition for the type `Pnat` of Parigot-encoded numerals:

```
rec PNat : * = ∀ X : * , (PNat → (unit → X) → X) → X → X .
```

Compare this with the type from the previous section:

```
rec PNat : * = ∀ X : * , (PNat → X → X) → X → X .
```

The only difference is that we are specifying that the iterated function will be called not with the result (of type `X`) of iteration with the predecessor, but with a function of type `unit → X`. This is just a thunk which, when called with the trivial value `triv` of type `unit`, will return the result of iteration with the predecessor. So if one does not wish to use this result, as we do not in the case of `Ppred`, then it will not be computed in call-by-value

reduction. This change for more efficient call-by-value reduction is easily accommodated in the definitions of the basic arithmetic operations, which we omit for space reasons. It is also preferable to use the following similarly modified definition for booleans:

```
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 .
```

Note that this definition actually guards the cases, as one would if implementing an if-then-else construct in a call-by-value language. It would also be reasonable to do this for similar arguments in other datatypes, like the occurrence of `X` corresponding to the zero case in the above definition of `PNat`. We have not found this necessary for our examples.

3.5 Abstract Comparison

We can compare the three previous encodings abstractly, by assuming that F is a type-level function which uses its argument only in positive positions. Then as explained also by Wadler (Philip Wadler, 1990), the Church encoding of $\mu X.F\ X$ is the type $\mu_C\ F$ defined by

$$\mu_C\ F = \forall X : *, (F\ X \to X) \to X$$

The Scott encoding is the type $\mu_S\ F$ positive-recursively defined by

$$\mu_S\ F = \forall X : *, ((F\ (\mu_S\ F)) \to X) \to X$$

Finally, the Parigot encoding is the type $\mu_P\ F$ positive-recursively defined by

$$\mu_P\ F = \forall X : *, ((F\ ((\mu_P\ F) \times X)) \to X) \to X$$

The examples considered above are intuitionistically equivalent versions of these types. For example, according to the above abstract scheme, if we denote $\lambda X : *.(1+X)$ as F_{Nat}, then the Church encoding for the type $\mu X.1+X$ of the natural numbers is $\mu_C\ F_{Nat}$ defined as

$$\forall X : *, (1 + X \to X) \to X$$

This is intuitionistically equivalent to the type traditionally used, which we saw above:

$$\forall X : *, (X \to X) \to X \to X$$

This version is more convenient to use, because it does not contain an embedded sum type.

4 The Embedded Iterators Encoding for Peano Numerals

Our new encoding, which we show first for Peano numerals, relies on the Church encoding as defined in Section 3.1. Like the Parigot encoding, we combine Church and Scott aspects of numerals: numerals should intrinsically support iteration (Church), and also allow constant-time access to subdata (Scott). But we do this in a different way from the Parigot encoding. The crude starting point for the idea is to note that if we just had a pair of a Church numeral and a Scott numeral both representing n, then we could both iterate with

n and obtain the predecessor of n in constant time. But this simple idea would not allow us to get the predecessor of the predecessor of n in constant time.

We extend the above simplistic idea recursively as follows. Each encoded numeral n will either be represented by a trivial value for zero or else by a n-deep right-nesting of (Church-encoded) pairs, ending in the trivial value for zero. The first component of the first pair is Church-encoded n, and for a pair k levels deep, the first component will be Church-encoded $n-k$. So iterators for all the numbers from n down to 0 are embedded in the representation of n, and embedded-iterators numerals can be seen as lists of Church numerals (and this can be generalized to other datatypes). Informally, a number like 3 will be represented as follows, where here and subsequently, we use $\dot{n}$ as mathematical notation for Church-encoded n:

$$(\dot{3},(\dot{2},(\dot{1},0)))$$

The size of this term is quadratic in n, unlike the Parigot encoding, which is exponential in n. But similarly to the Parigot encoding, with this representation one always has access to an iterator $\dot{m}$ for a successor number $m = n+1$, as well as to the predecessor numeral $(\dot{n},(\ldots,0))$. Accessing the components of a Church-encoded pair takes constant time, so the predecessor can be obtained in constant time. Abstractly, if F uses its argument only positively, and writing (as above) $\mu_C\ F$ for the type $\forall X : *, (F\ X \to X) \to X$ of the Church encoding of $\mu X.F\ X$, then our encoding of $\mu X.F\ X$ is the type $\mu_{SF}\ F$ positive-recursively defined by:

$$\mu_{SF}\ F = F((\mu_C\ F) \times (\mu_{SF}\ F))$$

As above, we will make use of types which are intuitionistically equivalent to those prescribed by this abstract scheme. For the leading example of natural numbers, which we will see in more detail in the next section, we have $F(X) = 1 + X$, and so

$$Nat = 1 + (CNat \times Nat)$$

The embedded-iterators encoding bears a passing resemblance to a definition of numerals due to Barendregt, where the interpretation of $n+1$ is the pair $(False, n)$ (Barendregt, 1985). In the embedded-iterators encoding, this is $(\dot{n}, n)$ (although Barendregt's encoding cleverly avoids the need for a sum type).

It is important to note that like the Parigot encoding, the embedded-iterators encoding is not adequate: F_ω^{rec} can assign the type $\mu_{SF}\ F_{Nat}$ to terms like $(\dot{3},(\dot{3},(\dot{3},0)))$, which do not follow the intended structure for the encoding. This limitation could possibly be resolved with dependent typing, though this remains to future work.

4.1 The type and constructors for numerals

The type we will use for numerals in our encoding is:

```
rec SFNat : * = ∀ X : *, (CNat → SFNat → X) → X → X .
```

This is intuitionistically equivalent to the type which the abstract scheme above dictates:

```
rec D : * = sum unit (pair CNat D) .
```

Informally, `D` requires that every number is either an inject-left of a unit value or inject-right of a pair of a Church-encoded natural number and another `D`. Our definition of `SFNat`

accomplishes the same thing, since it requires one to provide an `X` value for the case where the number is zero, and a function taking in a `CNat` and a `SFNat` (equivalent to taking in a pair of the two) and returning an `X`. So `SFNat` is just an optimized version of `D`.

The above definition of `SFNat` resembles the definition for the Scott encoding:

```
rec SNat : * = ∀ X : * , (SNat → X) → X → X .
```

The difference is that in the successor case, an value of type `SFNat` has access to a `CNat` given the Church-encoding of the same number. This `CNat` can be used to do iteration, something that is not possible intrinsically with the Scott-encoded numeral.

The definition of zero erases to the same term as for the Church- and Parigot-encodings:

```
SFzero : SFNat =
  [SFNat] λ X : *, λ s : CNat → SFNat → X , λ z : X , z .
```

We can easily also define one:

```
SFone : SFNat =
  [SFNat] λ X : *, λ s : CNat → SFNat → X , λ z : X ,
            s Cone SFzero .
```

Note that we are applying `s` to `Cone` as well as to `SFZero`. This fits with our plan of having the encoding for nonzero n contain $\dot{n}$, as well as the encoding of the predecessor of n. Subsequent numerals $m = n+1$ have the form:

$$\lambda X : *, \lambda s : \texttt{CNat} \to \texttt{SFNat} \to X, \lambda z : X, s\ \dot{m}\ \hat{n}$$

where we are writing $\hat{n}$ to indicate the embedded-iterators encoding of n. Each numeral m is encoded by a term which contains the Church-encodings of m down to 0. Hence, each encoding needs only quadratic space. Here is the definition of successor:

```
SFsuc : SFNat → SFNat =
  λ n : SFNat ,
    [SFNat!] n SFNat
      (λ c : CNat, λ p : SFNat,
         [SFNat] λ X : *, λ s : CNat → SFNat → X, λ z : X ,
                   s (Csuc c) n)
      SFone .
```

This term unfolds the definition of `SFNat` so that it can apply `n` to the result type `SFNat` and a value for `s` (from the definition of the `SFNat` type) and a value for `z`. The value for `s` is a function which takes in the Church-encoded version `c` of `n`, and the predecessor numeral `p`. It returns a new numeral (starting from the fold `[SFnat]`) where the `s` function is applied to the successor of `c` (which yields the Church-encoding for the successor of the numeral represented by `n`) and also `n` itself, which, of course, is the predecessor of the new numeral.

Constant-time predecessor is very easy to define, since we must just return the predecessor number in the successor case, and zero in the zero case:

```
SFpred : SFNat → SFNat =
  λ n : SFNat ,
    [SFNat!] n SFNat (λ c : CNat, λ s : SFNat, s) SFzero.
```

Note that unlike with the Parigot encoding, this predecessor operation is constant-time regardless of whether reduction is call-by-name (or normal-order) or call-by-value. To see this, let us temporarily write `SFpred` for the erasure of `SFpred` as just defined, and $\hat{n}$ for the embedded-iterators encoding of natural number n. Then all β-reduction sequences for `SFpred` $\hat{m}$, where $m = n+1$, have length independent of m. Here is one:

$$\begin{array}{ll} \texttt{SFpred}\ \hat{m} & \rightsquigarrow \\ \hat{m}\ (\lambda c.\lambda s.s)\ \hat{0} & \rightsquigarrow \\ (\lambda z.(\lambda c.\lambda s.s)\ \dot{m}\ \hat{n})\ \hat{0} & \rightsquigarrow^3 \\ \hat{n} & \end{array}$$

$O(1)$ reduction steps for predecessor regardless of strategy is a further benefit of the embedded iterators encoding.

4.2 Basic arithmetic operations

Armed with zero, one, successor, and predecessor, the other basic arithmetic operations follow a simple pattern: there is a case for when the parameter of iteration is zero, and then in the other case, we extract the embedded Church numeral and use it to carry out the iteration. Here is the definition of addition:

```
SFadd : SFNat → SFNat → SFNat =
  λ n : SFNat , λ m : SFNat ,
    [SFNat!] n SFNat
             (λ c : CNat, λ p : SFNat , c SFNat SFsuc m)
             m.
```

The parameter of iteration is `n`. We must first unfold the type `SFNat`, and then apply the result to result type `SFNat`, the case for when n is a successor number, and the case for when it is zero. The case for the successor number takes in the Church-encoded version `c` of `n`, together with the predecessor `p`. The latter is ignored, since we just need to use `c` to iterate the successor function `SFsuc` starting from the second input numeral `m`. The terms for multiplication and exponentiation follow this pattern also, and using our constant-time predecessor function, subtraction is defined similarly, by iterating `SFpred` (code omitted).

4.3 Further reducing the space required

Instead of storing Church-encoded unary numerals throughout the `SFNat`, we can store Church-encoded binary numerals, for significant space savings. Binary numerals can be Church-encoded as proposed by Mogensen (Mogensen, 2001). We think of binary numerals as having three constructors: for the empty binary numeral, for prepending a 0 bit to a binary numeral, and for prepending a 1 bit to a binary numeral. So the type is:

```
BNat : * = ∀ X : *, (X → X) → (X → X) → X → X.
```

It is a routine if tedious exercise to implement the operation `Bsuc` which, given a binary numeral representing n, returns a new binary numeral representing $n+1$. To define iterative operations like addition, we also need a function `BtoCNat` which can convert a binary

number to a unary one, which can then be used for iteration. Armed with these functions (code omitted), we can modify the definition of `SFNat` from the previous section, so that each number n is represented as $(b(n),(b(n-1),\ldots,b(0)))$, where $b(x)$ is a binary number representing x. The required space becomes $O(\Sigma_{i=0}^{N}(\log_2 i))$, which is $O(n\log_2(n))$. Some crucial definitions are:

```
rec SFNat : * = ∀ X : *, (BNat → SFNat → X) → X → X .
SFsuc : SFNat → SFNat =
  λ n : SFNat ,
    [SFNat!] n SFNat
      (λ c : BNat, λ p : SFNat,
         [SFNat] λ X : *, λ s : BNat → SFNat → X, λ z : X ,
                   s (Bsuc c) n)
      SFone .
```

The cost of `SFsuc` is now $O(\log n)$ given an `SFNat` representing n, because we must call `Bsuc` (successor on binary numbers) on the embedded binary numeral c. Our previous iterative operations on `SFNat` are adapted to the version using binary numerals, by calling `BtoCNat` on the embedded numeral. For example, here is the code for addition:

```
SFadd : SFNat → SFNat → SFNat =
  λ n : SFNat , λ m : SFNat ,
    [SFNat!] n SFNat
             (λ c : BNat, λ s : SFNat ,
                BtoCNat c SFNat SFsuc m)
             m.
```

4.4 Comparing the sizes of normal forms

Figure 5 shows the sizes of normal forms for the first few numerals. These sizes were computed by normalizing the numerals and counting the number of subterms, in `fore`. We see the predicted exponential blow-up for the sizes of Parigot-encoded numerals, and the space savings obtained by using binary instead of unary embedded iterators ("Stump Fu" versus "Stump Fu (bnats)" in the figure).

5 Lambda-Encoding Container Datatypes

In this section, we consider lambda-encoding for polymorphic lists using the Church, Parigot, and embedded-iterators encodings. We have chosen to implement mergesort using a form of Braun tree (cf. (Okasaki, 1997)) as an intermediate data structure. Braun trees provide a very simple form of balanced binary tree, which is a convenient fit for the recursive subdivision of the input list which mergesort employs. Furthermore, they constitute a second, nonlinear, example of a container datatype.

In Section 6 we will evaluate the versions of mergesort on call-by-value Church- and Parigot-encoded data structures, but for simplicity we present here mergesort on the standard versions of these encodings, from Sections 3.1 and 3.4 (which do not augment types with `unit` $\rightarrow$ in various places). In all cases, we will be sorting lists of elements of type `A`

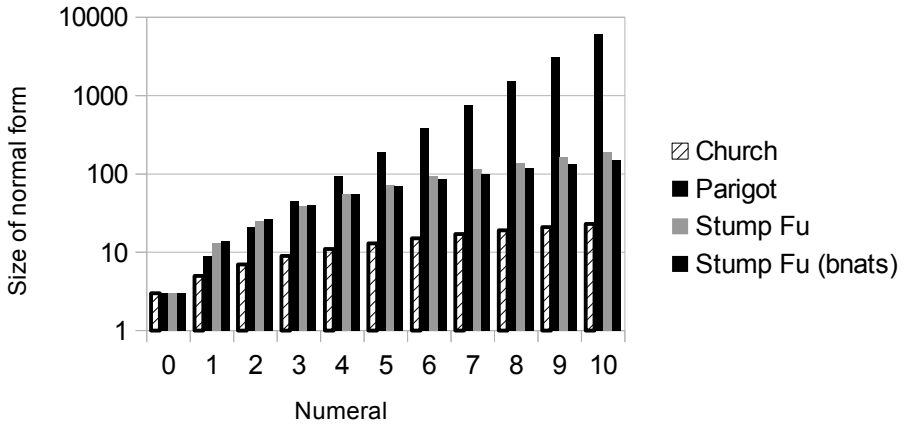

Fig. 5. Comparison of sizes of normal forms for numerals with different encodings

using a given comparator function of type `A → A → bool`, which returns `true` if the first element should be treated as less than or equal to the second, and hence closer to the front of the sorted list; and `false` otherwise.

Before we consider the different implementations based on lambda-encodings, let us look at an implementation of the same algorithm in Agda. This will help make the basic ideas of the algorithm clear, as the version in Agda is much more concise and readable, due to Agda's pattern matching and type inference. While the F_{ω}^{rec} code for these algorithms is much more complex, the reader is asked to bear in mind that F_{ω}^{rec} should not be directly compared to Agda: F_{ω}^{rec} could serve as the core language for a tool with similar features as Agda, where much of the information could likewise be inferred and elided. Like F_{ω}^{rec}, Agda statically enforces termination of all programs.

5.1 Mergesort in Agda

The following code is included as a file `mergeSort.agda` in the `fore` distribution (see the `README` file for instructions on how to check this with Agda). Here, we will look at a simply typed version of the code, where the only property being statically enforced by Agda is termination. For a version of Braun trees where the balancing property is statically enforced, see `lib/braun-tree.agda` in the `fore` distribution.

The module defined in `mergeSort.agda` is parametrized by a type `A`, and an ordering `_<A_` on that type. The definition of the `Braun` type, together with the `braunInsert` function for inserting a node, is given in Figure 6. Inserting a node uses the fundamental insight embodied in Braun trees, which is an elegant way to maintain the balancing property of the tree. This property says that for every node in the tree, the left and right subtrees have the same number of elements, or else the left subtree has exactly one more element than the

```
data Braun(A : Set) : Set where
  braunLeaf : A → Braun A
  braunNode : Braun A → Braun A → Braun A

braunInsert : {A : Set} → A → Braun A → Braun A
braunInsert a (braunLeaf a') = braunNode (braunLeaf a) (braunLeaf a')
braunInsert a (braunNode l r) = braunNode (braunInsert a r) l

listToBraunTree : {A : Set} → A → 𝕃 A → Braun A
listToBraunTree a [] = braunLeaf a
listToBraunTree a (a' :: as) = braunInsert a (listToBraunTree a' as)
```

Fig. 6. Agda code for Braun trees

```
merge : {A : Set} → (A → A → 𝔹) → 𝕃 A → 𝕃 A → 𝕃 A
merge _ [] ys = ys
merge _ xs [] = xs
merge cmp (x :: xs) (y :: ys) =
  if cmp x y then x :: (merge cmp xs (y :: ys))
             else y :: (merge cmp (x :: xs) ys)

mergeSorth : {A : Set} → (A → A → 𝔹) → Braun A → 𝕃 A
mergeSorth _ (braunLeaf a) = [ a ]
mergeSorth cmp (braunNode l r) =
  merge cmp (mergeSorth cmp l) (mergeSorth cmp r)

mergeSort : {A : Set} → (A → A → 𝔹) → 𝕃 A → 𝕃 A
mergeSort _ [] = []
mergeSort cmp (a :: as) = mergeSorth cmp (listToBraunTree a as)
```

Fig. 7. Agda code for merge sort using Braun trees

right. To maintain this property when inserting into a `braunNode`, we insert into the right subtree, and then swap right and left subtrees. If the sizes of the original left and right subtrees were equal, then the size of the new left subtree will be one greater than the size of the right. If the size of the original left was one greater than that of the original right, the sizes of the new left and right will be equal. Thus, the balancing property is maintained. (Again, in `lib/braun-tree.agda`, dependent types are used to enforce this statically, but here we just use a simply typed version.) Figure 6 also includes a function `listToBraunTree` for converting a list to a Braun tree by iterating `braunInsert`. `braunInsert` runs in $O(\log_2 n)$ time for a tree of size n, due to the balancing property of Braun trees. `listToBraunTree` then runs in $O(n \log_2 n)$ time. The code for `mergeSort` is then given in Figure 7. This function first constructs a Braun tree for the input list (using `listToBraunTree`), and then calls `mergeSorth`. This function recursively sorts the subtrees of an input `braunNode`, and then calls `merge`. As noted above, Agda is able to confirm statically that these functions are terminating on all inputs.

```
Braun : * → * =
  λ A:*, ∀ X:*, (A → X) → (X → X → X) → X.
braunLeaf : ∀ A:*, A → Braun A =
  λ A:*, λ a:A,
    λ X:*, λ l:A → X, λ n:X → X → X,
      l a.
braunNode : ∀ A:*, Braun A → Braun A → Braun A =
  λ A:*, λ L:Braun A, λ R:Braun A,
    λ X:*, λ l:A → X,
    λ n:X → X → X,
      n (L X l n) (R X l n).
braunInsert : ∀ A:*, A → Braun A → Braun A =
  λ A:*, λ a:A, λ b:Braun A,
    b (pair (Braun A) (Braun A))
      (λ aa : A, mkpair (Braun A) (Braun A)
                  (braunLeaf A aa)
                  (braunNode A (braunLeaf A a) (braunLeaf A aa)))
      (λ pL : pair (Braun A) (Braun A), λ pR : pair (Braun A) (Braun A),
         pL (pair (Braun A) (Braun A))
           (λ L : Braun A, λ iL : Braun A,
             pR (pair (Braun A) (Braun A))
              (λ R : Braun A, λ iR : Braun A,
                mkpair (Braun A) (Braun A)
                  (braunNode A L R)
                  (braunNode A iR L))))
      (Braun A)
      (λ b : Braun A, λ ib : Braun A , ib).
```

Fig. 8. Church-encoded Braun trees

5.2 Mergesort for Church-encoded lists

The `fore` code for Braun trees is shown in Figure 8. Recall from the previous section that when inserting a value a into a Braun tree which is a node, we insert a into the right subtree, but then reverse left and right subtrees. This means that we need access to the unmodified left subtree of Braun tree b, during the course of inserting an element into b. Since we cannot obtain subtrees in constant-time with the Church encoding, the implementation in Figure 8 iteratively computes a pair consisting of the modified and the unmodified versions of the input Braun tree. The modified version has the element a inserted, while the unmodified one does not. This approach takes $O(n)$ time to insert an element into a Braun tree of size n, instead of the $O(\log_2(n))$ required with constant-time access to the subtrees. So we can certainly expect a performance penalty from this step, compared to the Parigot and embedded-iterators encodings.

The type for lists, the definitions of the constructors, and the definition of a function to build a singleton list containing a given element `a` are as expected for the Church encoding, so we elide the definitions. Additionally, we need a function to convert a list to a Braun tree. Since Braun trees as we have defined them above are non-empty, `listToBraunTree` must be given an element `a` of type `A`, as well as a list of `A`s. We then iteratively call `braunInsert`, starting with a call to `braunLeaf` as the base case:

```
listToBraunTree : ∀ A:*, A → List A → Braun A =
```

```
λ A:*, λ a:A, λ l:List A,
  l (Braun A)
    (λ a : A, λ r :  Braun A, braunInsert A a r)
    (braunLeaf A a).
```

If `braunInsert` had time complexity $O(\log_2(n))$ in the size n of the Braun tree, then `listToBraunTree` would have complexity $O(n\log_2(n))$. But since `braunInsert` takes $O(n)$ with the Church encoding, this `listToBraunTree` function requires quadratic time.

We need functions `head` and `tail` for obtaining the head and tail, respectively, of a non-empty list. We elide these definitions for space reasons. As for predecessor and indeed all functions returning immediate subdata, `tail` must be computed by iteration with the Church encoding. For `head`, we use a `maybe` type to return just the head of the list, if the list is not empty, and nothing otherwise.

Using `tail`, we can now define the crucial helper function `merge`, in Figure 9. `merge` takes in the comparator function and two lists assumed to be sorted, and returns the merged sorted result. In some cases `merge` will make recursive calls with the first list decreased, when the head of the first list is less than or equal to the head of the second; and in some cases, the first list will not decrease, but the second will (when the head of the second list is less than the head of the first). To do this in a terminating way, we follow the approach used in the Coq standard library (`Sorting/Mergesort.v`) (The Coq development team, 2014), and use a nested iteration. We have an outer iteration on the first list, for the cases where the first list will be decreased; and an inner iteration on the second list for the cases where the first list is unchanged but the second decreases. The code is rendered a little more verbose by the need to analyze the `maybe` value returned by `head`. But let us look at this code at the heart of the function:

```
cmp a b (List A)
  (Cons A a (outer pa lb))
  (Cons A b (inner pb))))
```

In the surrounding context, we have:

- `outer : List A → List A → List A`, for making the outer recursive call when the first list will decrease,
- `inner : List A → List A`, for making the inner recursive call when the first list will stay the same and the second will decrease,
- `a : A`, the head of the first list,
- `b : A`, the head of the second list,
- `pa : List A`, the tail of the first list,
- `lb : List A`, the second list,
- `pb : List A`, the tail of the second list

So the central code of `merge`, displayed above, is returning `a` as the head of the result list with a call to `outer` as the tail, in case `a` is less than or equal to `b`; and `b` as the head and a call to `inner` as the tail otherwise. Due to the repeated use of `tail` in each iteration, `merge` takes quadratic time in the sum of the sizes of the input lists. Thankfully, the code for `mergeSort` is then straightforward; see Figure 10.

```
merge : ∀ A:*, (A → A → Bool) → List A → List A → List A =
  λ A:*, λ cmp : A → A → Bool, λ la : List A,
    la (List A → List A → List A)
      (λ a : A, λ outer : List A → List A → List A,
         λ la : List A, λ lb : List A,
           head A la (List A)
             (λ ha : A,
                lb (List A → List A)
                  (λ b : A, λ inner : List A → List A,
                     λ lb : List A,
                        head A lb (List A)
                          (λ hb : A,
                            cmp ha hb (List A)
                             (Cons A ha (outer (tail A la) lb))
                             (Cons A hb (inner (tail A lb))))
                          la)
                  (λ lb:List A, la)
                  lb)
            lb)
       (λ la : List A, λ lb : List A, lb)
       la.
```

Fig. 9. Merge function for Church-encoded lists

```
mergeSort : ∀ A:*, (A → A → Bool) → List A → List A =
  λ A:*, λ cmp : A → A → Bool, λ la : List A,
    head A la (List A)
      (λ a : A,
        listToBraunTree A a (tail A la)
           (List A)
           (singleton A)
           (λ la : List A, λ lb : List A,
              merge A cmp la lb))
      (Nil A) .
```

Fig. 10. Mergesort function for Church-encoded lists

5.3 Mergesort for Parigot-encoded lists

We now redo the work of the previous section, using the Parigot encoding. Braun trees and the `braunInsert` function are defined in Figure 11. The type constructor `Braun` is defined recursively, as always for Parigot-encoded recursive datatypes, so that the subtrees (of type `Braun A`) can be input arguments to the second function required by the definition of `Braun`. Fold and unfold operations `[Braun] A` and `[Braun!] A`, respectively, are used in the constructors. The code for `braunInsert` is much simpler than the version for Church-encoded Braun trees (Figure 8 above), because we no longer need to compute the unmodified Braun tree along with the modified one. The unmodified subtrees `L` and `R` are already available with the Parigot encoding, and the modified subtrees `iL` and `iR` are available as the results of iteration.

The definitions for lists, their constructors, and the `listToBraunTree` function are then completely as expected for the Parigot encoding, so we elide the definitions. The code in

```
rec Braun : * → * =
  λ A:*, ∀ X:*, (A → X) → (Braun A → Braun A → X → X → X) → X.
braunLeaf : ∀ A:*, A → Braun A =
  λ A:*, λ a:A,
    [Braun] A λ X:*, λ l:A → X,
    λ n:Braun A → Braun A → X → X → X,
      l a.
braunNode : ∀ A:*, Braun A → Braun A → Braun A =
  λ A:*, λ L:Braun A, λ R:Braun A,
    [Braun] A λ X:*, λ l:A → X,
    λ n:Braun A → Braun A → X → X → X,
      n L R ([Braun!] A L X l n) ([Braun!] A R X l n).
braunInsert : ∀ A:*, A → Braun A → Braun A =
  λ A:*, λ a:A, λ b:Braun A,
    [Braun!] A b (Braun A)
      (λ aa : A, braunNode A (braunLeaf A a) (braunLeaf A aa))
      (λ L : Braun A, λ R : Braun A, λ iL :  Braun A, λ iR :  Braun A,
         braunNode A iR L) .
```

Fig. 11. Parigot-encoded Braun trees

```
merge : ∀ A:*, (A → A → Bool) → List A → List A → List A =
  λ A:*, λ cmp : A → A → Bool, λ la : List A,
    [List!] A la (List A → List A)
      (λ a : A, λ laa : List A, λ outer : List A → List A,
         λ lb : List A,
            [List!] A lb (List A)
               (λ b : A, λ lbb : List A, λ inner : List A,
                  cmp a b (List A)
                    (Cons A a (outer (Cons A b lbb)))
                    (Cons A b inner))
               (Cons A a laa))
       (λ lb : List A, lb).
```

Fig. 12. Mergesort with the Parigot encoding

Figure 12 for `merge` is much more straightforward than for Church-encoded lists. `merge` is greatly simplified by the fact that the Parigot encoding makes the tails of the lists available while iterating. This allows us to simplify the types of `outer` and `inner`, so that now `outer` just takes in the second list (instead of the first and the second), and `inner` does not require any lists at all, but instead is just the iterative result of merging the tail of the second list with the first list unchanged. The code for `mergeSort` reveals no new matters of interest, so it is elided.

5.4 Mergesort for lists with the embedded-iterators encoding

Finally, let us consider mergesort for the embedded-iterators encoding. Our Braun trees will embed Church numerals for the height of the node, at each node. This is sufficient for iterating through both subtrees of the tree separately. Similarly, our lists will embed Church numerals for the length of the list.

Braun trees are defined in Figure 13. We define a helper function `getBraunCNat` to get the Church-encoded numeral for the height of the given Braun tree. The `braunInsert` function requires some explanation. To insert an element into a Braun tree, we will need to make a number of recursive calls, corresponding to the height of the tree. So we use `getBraunCNat` to get this height, and then use it to iteratively construct a function of type `Braun A` $\rightarrow$ `Braun A`. In each case of that iterative construction, we must analyze the input Braun tree `b`. When we are in the successor case of our iteration, `b` must be a proper node (not a leaf). Unfortunately, as we do not have any dependent typing in F_{ω}^{rec}, the type system cannot discern this invariant, and so we must include code for the impossible off-case (labeled "`% should not happen`") where `b` is a leaf. Something similar happens in the zero case of our iteration on the height of the Braun tree. In the successor case, where we have analyzed `b` and found it to be a node, we can call the function `r` which we iteratively computed for the predecessor of the depth of the node. We call this on the right subtree, which, as with the Scott and Parigot encodings, is available at this point with our embedded-iterators encoding.

The definition of the `List` datatype using the embedded-iterators encoding is:

```
rec List : * → * =
  λ A:*, ∀ X:*, (CNat → A → List A → X) → X → X.
```

The first argument of a list as described by this definition is a function which will be given the head and tail of the list (of types `A` and `List A`), but also a Church-encoded natural number for the length of the list (i.e., one plus the length of the tail), which can be obtained with an elided helper function `getLen`. We also elide definitions of the list constructors and the `listToBraunTree` function, as these reveal no new issues.

Finally, we can implement the `merge` and `mergesort` functions, shown in Figure 14. For `merge`, we get the embedded lengths of the input lists `la` and `lb`, and sum them using the Rosser addition function `CaddR` on `CNat`s. This will allow us to recurse as deeply into the two input lists as might be necessary. Using the sum of the lengths, we iteratively construct a function taking in two input lists `la` and `lb`, analyzing them both, and performing the comparison with `cmp` on their heads as in the previous implementations. We use `r` to make recursive calls with the tail `pa` of `la` and `lb`, or else `la` and the tail `pb` of `lb`. For `mergeSort`, we use a beta-redex to introduce a name `b` for the Braun tree we get from `listToBraunTree A a laa`. We call `getBraunCNat` to get the embedded depth of this Braun tree, from which we iteratively construct a function from `Braun A` to `List A`. In both the successor and the step cases, the terms we are using to construct this function analyze the input Braun tree `x`. Again, we have some off cases which cannot happen, but cannot be statically ruled out. The code uses `merge` to combine the results `r L` and `r R` of recursively sorting the left and right subtrees of `x`; and in the case where Braun tree `x` has depth 0 and is hence a leaf, just returning a singleton list.

5.5 Discussion

Of the three implementations above, the one using Church-encoded data structures (Braun trees and lists) is the least satisfactory. We are forced into several inefficient computations, which are also complicated to implement. The function for inserting data into a Braun tree

```
rec Braun : * → * =
  λ A:*, ∀ X:*, (A → X) → (CNat → Braun A → Braun A → X) → X.
getBraunCNat : ∀ A:*, Braun A → CNat =
  λ A:*, λ b:Braun A,
    [Braun!] A b CNat
      (λ a : A, Czero)
      (λ c : CNat, λ l : Braun A, λ r : Braun A, c).
braunLeaf : ∀ A:*, A → Braun A =
  λ A:*, λ a:A,
    [Braun] A λ X:*, λ l:A → X,
    λ n:CNat → Braun A → Braun A → X, l a.
braunNode : ∀ A:*, Braun A → Braun A → Braun A =
  λ A:*, λ L:Braun A, λ R:Braun A,
    [Braun] A λ X:*, λ l:A → X,
    λ n:CNat → Braun A → Braun A → X,
      (n (Csuc (getBraunCNat A L)) L R).
braunPair : ∀ A:*, A → A → Braun A =
  λ A:*, λ a:A, λ aa : A, braunNode A (braunLeaf A a) (braunLeaf A aa).
braunInsert : ∀ A:*, A → Braun A → Braun A =
  λ A:*, λ a:A, λ b:Braun A,
    getBraunCNat A b (Braun A → Braun A)
      (λ r :  Braun A → Braun A,
         λ b : Braun A,
           [Braun !] A b (Braun A)
             (λ aa : A , braunPair A a aa) % should not happen
             (λ q : CNat, λ L : Braun A, λ R : Braun A,
                braunNode A (r R) L))
      (λ b : Braun A,
         [Braun !] A b (Braun A)
           (λ aa : A , braunPair A a aa)
           (λ q : CNat , λ L : Braun A, λ R : Braun A ,
              L)) % should not happen
      b .
```

Fig. 13. Braun trees with the embedded-iterators encoding

simultaneously computes the modifed (data inserted) and unmodified Braun tree, so that it can rebuild the entire tree, including modified and unmodified subtrees. This increases the asymptotic time complexity of this function from logarithmic to linear – a serious performance penalty. Also, we have to use an iterative `tail` function, which increases the asymptotic time-complexity of the `merge` function from linear to quadratic.

The embedded-iterators encoding does not incur penalties in asymptotic time-complexity. But it suffers from the fact that iteration and analysis of data are separate, and so in several situations we find we are in an analysis case (e.g., the input list is empty) which cannot happen due to the iteration case we are in (e.g., the length of the input list is non-zero). This requires us to include dummy code for those off cases. Dependent types (not available in F_ω^{rec}) might allow us to drop them.

The implementation with Parigot-encoded data structures is superior. Since analysis and iteration happen simultaneously with the Parigot encoding, we do not have off cases as in the embedded-iterators encoding. The code is the simplest of the three implementations,

```
merge : ∀ A:*, (A → A → Bool) → List A → List A → List A =
  λ A:*, λ cmp : A → A → Bool, λ la : List A, λ lb : List A,
    CaddR (getLen A la) (getLen A lb) (List A → List A → List A)
      (λ r :  List A → List A → List A,
         λ la : List A, λ lb : List A,
           [List!] A la (List A)
              (λ ca : CNat, λ a : A, λ pa : List A,
                 [List!] A lb (List A)
                   (λ cb : CNat, λ b : A, λ pb : List A,
                     cmp a b (List A)
                       (Cons A a (r pa lb))
                       (Cons A b (r la pb)))
                 la)
              lb)
      (λ la : List A, λ lb : List A, Nil A) % la and lb are Nil
      la lb.
mergeSort : ∀ A:*, (A → A → Bool) → List A → List A =
  λ A:*, λ cmp : A → A → Bool, λ la : List A,
    [List!] A la (List A)
      (λ i : CNat, λ a : A, λ laa : List A,
        (λ b : Braun A,
         getBraunCNat A b (Braun A → List A)
           (λ r :  Braun A → List A,
            λ x : Braun A,
              [Braun!] A x (List A)
                (singleton A) % should not happen
                (λ q : CNat, λ L : Braun A, λ R : Braun A,
                   merge A cmp (r L) (r R)))
           (λ x : Braun A,
              [Braun !] A x (List A)
                (singleton A)
                (λ q : CNat , λ L : Braun A, λ R : Braun A ,
                   Nil A)) % should not happen
           b)
        (listToBraunTree A a laa))
      (Nil A).
```

Fig. 14. Merge sort with the embedded-iterators encoding

and has the expected asymptotic time-complexities. The only concern generally is the exponential size of normal forms for the Parigot encoding. But as we will see next, this actually does not occur in practice with efficient implementations of lambda calculus.

6 Performance Comparison

In this section, we present empirical data obtained with Racket Version 6.0.1 (Flatt & PLT, 2010) and the ghc implementation, version 7.6.3, of Haskell, to compare the runtime performance of the above lambda encodings. We compare wallclock times for the encodings on three families of benchmarks:

- computing 2^n

- computing $x - x$ where x is defined to be 2^n (in Racket this means 2^n will be computed once, not twice); since subtraction is defined as iterated predecessor, this tests the cost of computing predecessor for an encoding
- running merge sort on a list of length 2^n, obtained by concatenating the list of the first eight Parigot-encoded numerals, 2^{n-3} times.

All tests were run on a standard laptop computer with a 1.60GHz Intel Core 2 Duo processor with 128Kb L1 cache, 3072Kb L2 cache, and 5GB main memory, running Ubuntu Linux version 12.04.

6.1 Experiments using Racket

Our `fore` implementation can emit erased F_{ω}^{rec} terms in Racket syntax. The erasure is a slightly optimized version of the function given earlier (Figure 4): the fold and unfold operations are eliminated completely where they are applied. For one example, the Racket definition emitted by `fore` for the basic version of addition on Church-encoded numerals is:

```
(define Cadd (lambda (n) (lambda (m) ((n Csuc) m))))
```

To run the tests, we invoke Racket on the Racket source files generated by `fore`.

Figure 15 shows the times in seconds required for computing 2^n, for even values of n from 10 to 22, for all different encodings we considered above. In this and in all subsequent figures, we have shaded the first encoding in each group with diagonal hatching, to help make the starting point of the group visually distinct. Our benchmark families all increase exponentially in the difficulty required for a standard implementation, as a function of a parameter n plotted on the x-axis. So we will use a log scale for the y-axis.

In Figure 15, we see that the Rosser definition ("Church R" in the Figure) of exponentiation on Church-encoded numerals is superior to the others by a notable margin. There is some benefit to using the call-by-value version of the Parigot encoding. Notice that as predicted above, Racket does not actually require exponential space in practice to store Parigot-encoded numbers. If it did, the memory required to store 2^{22} would vastly exceed the physical memory of the test computer (indeed, of all computers in existence), and the benchmarks would not complete execution. We will consider this point further below (Section 6.4). The embedded-iterators encoding is slightly slower, and embedded-iterators with compressed iterators is much slower. We do not consider this last alternative in subsequent tests, due to its poor performance here.

Figure 16 shows the results for the subtraction test ($x - x$ where x is 2^n, and we perform that exponentiation only once). As expected, the Church and unmodified Parigot encoding are very slow with respect to the call-by-value Parigot encoding and to the embedded iterators (Stump-Fu) encoding.

Figure 17 shows the results for the mergesort test in Racket. We use a version of the Church encoding modified for call-by-value reduction, similarly to the way the Parigot encoding is modified as described in Section 3.4. Without a call-by-value version of the booleans, for example, algorithms like merging two sorted lists take exponential time, because using an unmodified Church boolean (the result of comparing the two heads of the lists) to select between two alternatives will evaluate both, with call-by-value reduction.

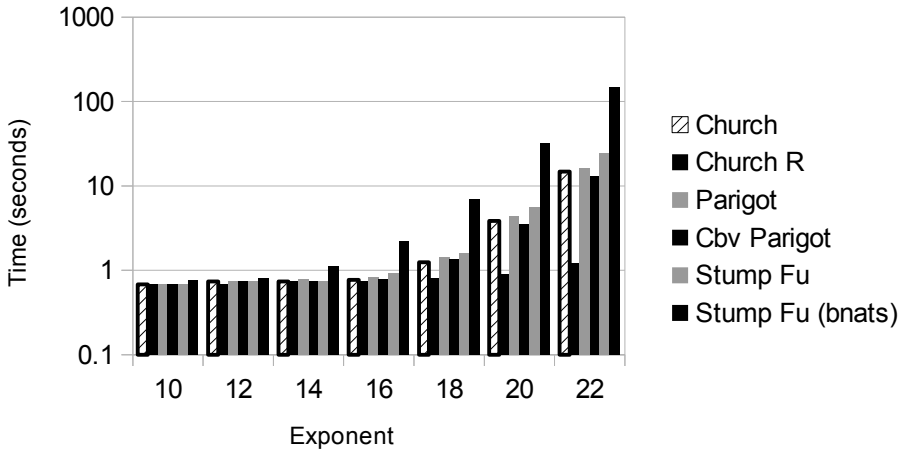

Fig. 15. Comparison of different encodings for the exponentiation test, using Racket

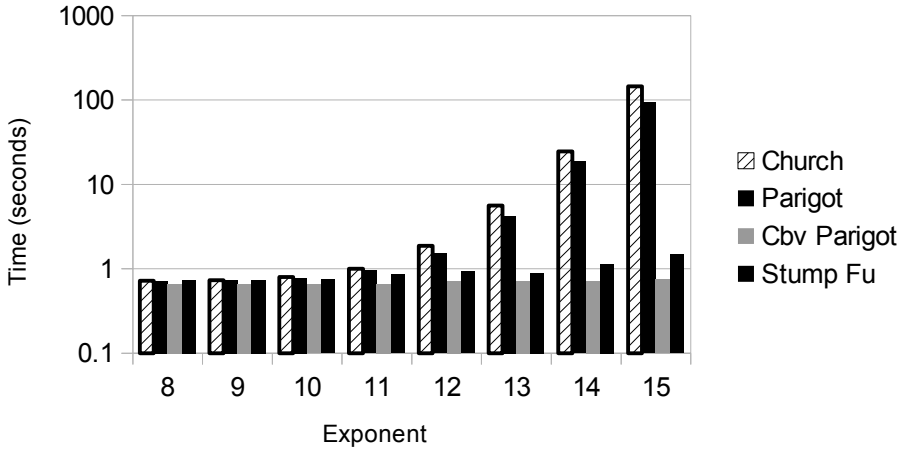

Fig. 16. Comparison of different encodings for the subtraction test ($x - x$ where x is 2^n), using Racket

Even so, the performance is much worse than for call-by-value Parigot, which is much better than the embedded-iterators encoding on this benchmark. We will see a different situation when we repeat this test in Haskell.

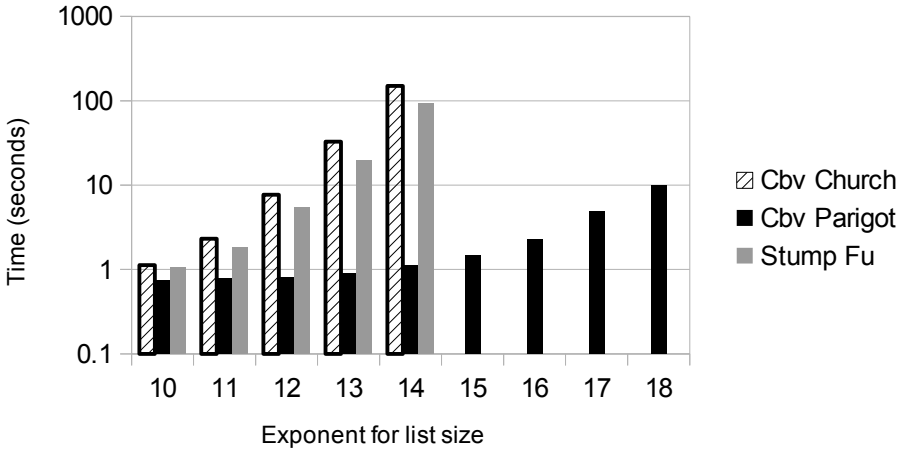

Fig. 17. Comparison of different encodings for the test with mergesort, using Racket

6.2 Experiments using Haskell

Using Haskell's higher-rank polymorphism and its `newtype` mechanism for introducing recursive types with a sole constructor, we can type-check all the above lambda-encodings in Haskell. Because Haskell's directly supports only predicative polymorphism, `newtype` must be used to wrap higher-rank types for instantiation of universally quantified type variables. A similar mechanism in Clean was used in a recent similar study (Koopman *et al.*, 2014). We must use `newtype` even for the translations of non-recursive types like `CNat`, because of the restriction to predicative polymorphism. So we use alternative versions of the `fore` source files for Church-encoded data, where all types are recursively defined. We declare `CNat` as a recursive type in `fore`:

```
rec CNat : * = ∀ x : * , (x → x) → x → x .
```

Then `fore` compiles this to the following Haskell `newtype` declaration:

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

Uses of fold and unfold operations in the `fore` code are then translated to calls to `FoldCNat` and `unfoldCNat`, respectively. For example, the `fore` definition of successor

```
csuc : CNat → CNat =
  λ n : CNat ,
    [CNat] λ x : * , λ s : x → x , λ z : x ,
             s ([CNat!] n x s z) .
```

is translated to the following Haskell code:

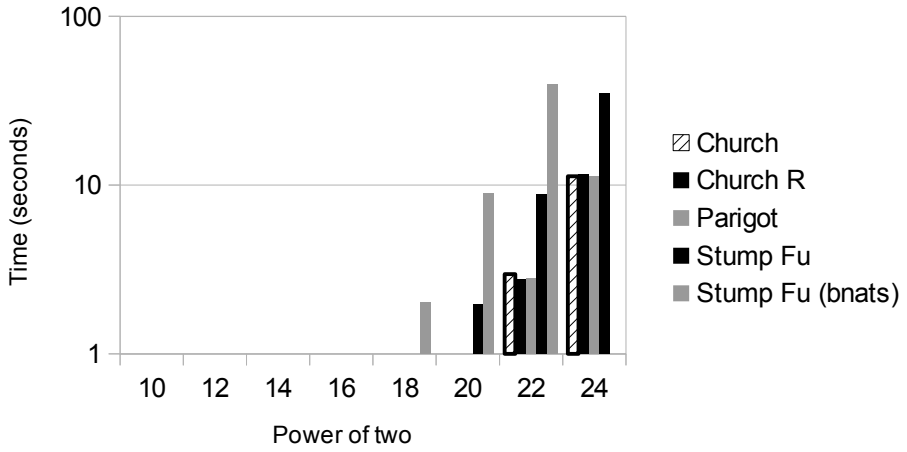

Fig. 18. Comparison of different encodings for the exponentiation test, using Haskell

```
csuc :: CNat -> CNat
csuc =
  (\ n -> (FoldCNat (\ s -> (\ z -> (s ((((unfoldCNat n) s) z)))))))
```

We first use `ghc` to compile the Haskell source files generated by `fore`, and then run the generated executables. Note that care must be taken to force Haskell's lazy evaluation to execute the test. We do this by printing a value which ensures computation of the result.

Figure 18 compares the wallclock time for the different encodings, for the exponentiation test. The times for computing small powers, like 2^{10}, are measured at 0 seconds. For larger powers, we see Church, Church with Rosser definitions, and Parigot at about the same times. Embedded iterators and embedded iterators with compressed iterators are slower and much slower, respectively. The results are consistent with those from Racket.

Figure 19 shows the rest of the subtraction test. As expected, Parigot and embedded iterators are much faster than the asymptotically less efficient Church version. Similar results appear for the sorting test, in Figure 20. Interestingly, here embedded iterators does not lag behind Parigot, as we saw it did with Racket. This suggests embedded iterators may be more performant with lazy evaluation than with eager evaluation.

6.3 Comparing with native implementations

Having compared the different encodings using two different efficient implementations of lambda calculus, we cannot help but be curious: how do the lambda encodings compare against native sorting functions in Racket and Haskell? For Racket, we will compare with the native `sort` function provided by Racket, running (of course) on native Racket lists. For Haskell, we will use `Data.List.sort`, from the `base` package that ships with `ghc`, again operating on native Haskell lists. The data are still Parigot-encoded natural numbers,

exponent	Church	Parigot	Stump Fu
8	0	0	0
9	0.01	0	0
10	0.04	0	0
11	0.18	0	0
12	0.94	0	0
13	5.13	0	0.01
14	32.9	0.1	0.02
15	183.39	0.3	0.05

Fig. 19. Comparison of different encodings for the subtraction test, using Haskell

exponent for list size	Church	Parigot	Stump Fu
7	0	0	0
8	0.01	0	0
9	0.04	0	0
10	0.16	0	0.01
11	0.66	0	0.02
12	3.03	0.02	0.08
13	15.05	0.04	0.24
14	75.93	0.1	0.5
15	399.67	0.23	1.09

Fig. 20. Comparison of different encodings for the sorting test, using Haskell

and the comparator function is the same as above, except wrapped to produce results of the type expected by the native sorting function. The rather surprising results is shown in Figure 21. For the larger list sizes (e.g., $2^{22} = 4194304$), the sorting function using CBV Parigot-encoded lists is significantly faster, by two or three times, compared to the native Racket implementation. Parigot lags native Haskell by roughly an order of magnitude.

It has been observed in practice that for lists with many repeated elements, the widely used quicksort algorithm can suffer performance degradation.[1] Figure 22 shows the wall-clock times for a second sorting test, for Racket only, where the lists to be sorted consist of pseudo-randomly generated native numbers, and the maximum number requested from the pseudo-random generator is twice the length of the list. Here we see Racket's sorting function on native Racket lists pulling far ahead of mergesort with Braun trees on lists encoded with CBV Parigot (e.g., 20 times faster for the largest test, 22, that could be completed by the lambda-encodings implementation, without exceeding 4GB memory). Thus, the positive results of Figure 21 compared to Racket may be an artifact of the particular form of lists to sort, or sorting algorithm.

6.4 Discussion of performance of Parigot encoding

It may be surprising that both Racket and Haskell have no problem computing with Parigot-encoded data whose normal forms would be, if computed out in full in pure lambda calculus, of beyond astronomical size. But neither Racket nor Haskell implements β-reduction

[1] Thanks to Algorithms colleague Kasturi Varadarajan for pointing this out.

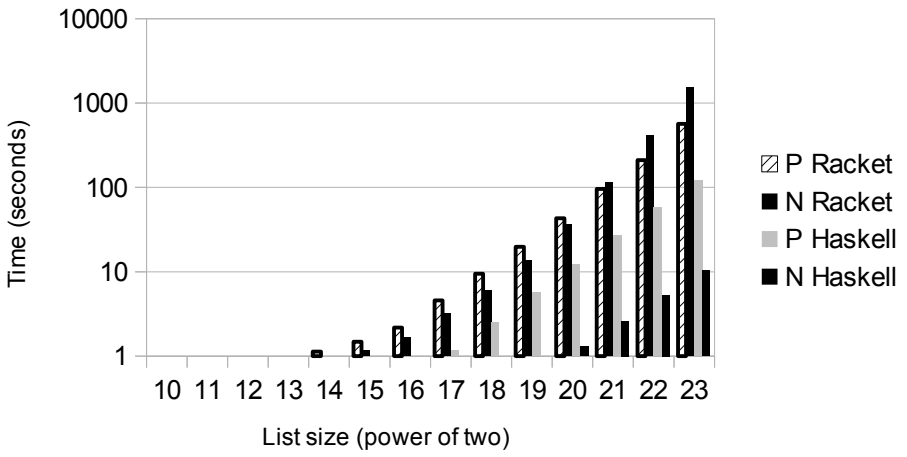

Fig. 21. Comparison of call-by-value Parigot in Racket (**P Racket**), native Racket (**N Racket**), Parigot in Haskell (**P Haskell**), and native Haskell (**N Haskell**), on the first sorting test (long list of small numbers)

in a strict sense. They implement optimized versions of β-reduction, in both cases using sharing of common subterms (Felleisen *et al.*, 2009; Jones, 1987). It is interesting to contrast these results with normal-order (i.e., leftmost) reduction, for example for terms of the form `Padd` N `Pzero`, where N is a Parigot-encoded natural number. Consider the case where N is Parigot-encoded two (let us call it `Ptwo`). Then we have this normal-order reduction sequence for the erased terms, where we introduce definitions for certain terms as we go, with equality steps:

$$
\begin{array}{ll}
\texttt{Padd Ptwo Pzero} & \rightsquigarrow^2 \\
\texttt{Ptwo } (\lambda P.\ \texttt{Psuc}) \texttt{ Pzero} & = \\
\texttt{Ptwo } S \texttt{ Pzero} & \rightsquigarrow^2 \\
S \texttt{ Pone } (S \texttt{ Pzero Pzero}) & \rightsquigarrow \\
\texttt{Psuc } (S \texttt{ Pzero Pzero}) & = \\
\texttt{Psuc } Q & \rightsquigarrow \\
\lambda s.\lambda z.s\ Q\ (Q\ s\ z) & \rightsquigarrow^4 \\
\lambda s.\lambda z.s \texttt{ Pone } (Q\ s\ z) & \rightsquigarrow^4 \\
\lambda s.\lambda z.s \texttt{ Pone } (\texttt{Pone } s\ z) & \rightsquigarrow^2 \\
\lambda s.\lambda z.s \texttt{ Pone } (s \texttt{ Pzero } z) &
\end{array}
$$

This is a total of sixteen steps, and indeed, with the normal-order evaluator include in `fore`, we have observed that the number of steps to normalize `Padd N Pzero` using normal-order reduction is exactly 2^{n+2}, where `N` is the Parigot representation of n. The offending step is the one shown above where redex Q is duplicated. This will not happen in either Racket or Haskell. In Racket, call-by-value reduction will first reduce Q to a value. In Haskell, call-by-need reduction will share Q, and only compute its normal form once.

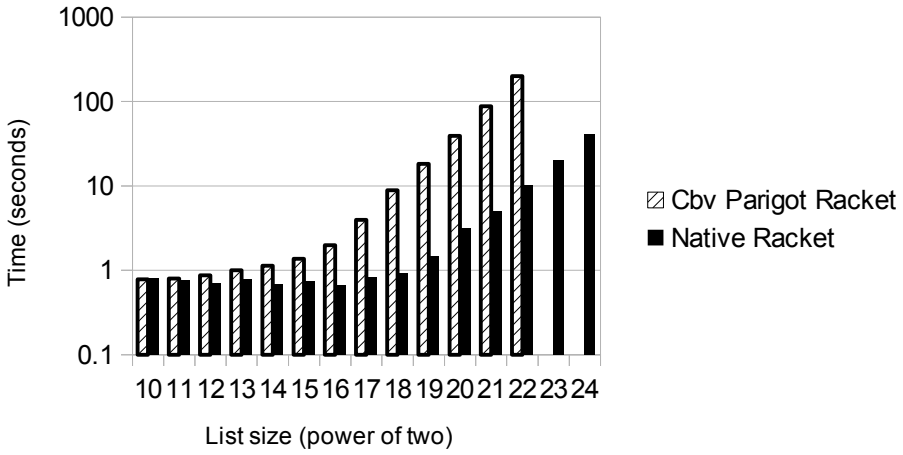

Fig. 22. Comparison of call-by-value Parigot and native Racket on the second sorting test (native integers from a range twice the length of the list)

7 Conclusion

This paper has introduced a new lambda encoding, called the embedded-iterators encoding, which has the expected asymptotic complexities for all basic operations on Peano numbers – including constant time for predecessor – but requires only quadratic space to represent each number. This improves substantially on the Parigot encoding, which requires exponential space for each numeral. Like the Parigot encoding, the embedded-iterators encoding is typable in F_ω^{rec}, an extension of System F_ω with global positive-recursive type definitions. This extension preserves strong normalization of the type theory, so all operations defined in this paper are statically confirmed by our `fore` implementation of F_ω^{rec} to be terminating. We also considered in detail how to implement a nontrivial algorithm – mergesort using Braun trees – with the Church, Parigot, and embedded-iterators encodings.

Finally, these encodings were evaluated using both eager (Racket) and lazy (Haskell) evaluation on a small suite of nontrivial benchmarks. The results generally have the Parigot encoding as the most performant with both eager and lazy evaluation, except for the exponentiation benchmark, where the Church encoding, particularly with Rosser's clever definitions of the basic operations, is significantly faster. The embedded-iterators encoding lags far behind the Parigot encoding on the mergesort benchmark with eager evaluation, but with lazy evaluation in Haskell, it scales similarly. Thus, if the size of normal forms is an important consideration, and if one is using lazy evaluation, we have seen empirical evidence that the new embedded-iterators encoding is currently the best lambda encoding available in total type theory.

If one wishes to use lambda-encoded data structures in practice, then designing optimizations specifically for improving runtime performance of lambda-encoded data is important future work. Another important direction is to improve dependently typed pro-

gramming with lambda encodings. Previous work showed how to define types like the natural numbers as their own *dependent* iterators (i.e., induction principles) (Fu & Stump, 2014). The next step is to extend the type theory to allow lifting term-level lambda-encoded data to the type level, for type-level computation and generic programming.

Acknowledgments. The authors thank the anonymous JFP referees for very helpful discussion that greatly improved the paper.

References

Abel, Andreas, & Matthes, Ralph. (2004). Fixed Points of Type Constructors and Primitive Recursion. *Pages 190–204 of:* Marcinkowski, Jerzy, & Tarlecki, Andrzej (eds), *Computer Science Logic (CSL), 18th International Workshop*.

Barendregt, Henk. (1985). *The lambda calculus: Its syntax and semantics*. North-Holland.

Böhm, Corrado, & Berarducci, Alessandro. (1985). Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, **39**(0), 135 – 154.

Church, Alonzo. (1941). *The Calculi of Lambda Conversion*. Princeton University Press. Annals of Mathematics Studies, no. 6.

Curry, Haskell, Hindley, Roger, & Seldin, Jonathan. (1972). *Combinatory Logic*. Vol. 2. North-Holland.

Felleisen, Matthias, Findler, Robert Bruce, & Flatt, Matthew. (2009). *Semantics Engineering with PLT Redex*. MIT Press.

Flatt, Matthew, & PLT. (2010). *Reference: Racket*. Tech. rept. PLT-TR-2010-1. PLT Design Inc. `http://racket-lang.org/tr1/`.

Fu, Peng, & Stump, Aaron. (2014). Self Types for Dependently Typed Lambda Encodings. *Pages 224–239 of:* Dowek, Gilles (ed), *25th International Conference on Rewriting Techniques and Applications (RTA) joint with the 12th International Conference on Typed Lambda Calculi and Applications (TLCA)*. Lecture Notes in Computer Science, vol. 8560. Springer.

Hinze, Ralf. (2005). Church numerals, twice! *J. funct. program.*, **15**(1), 1–13.

Jones, Simon L. Peyton. (1987). *The Implementation of Functional Programming Languages*. Prentice-Hall.

Koopman, Pieter, Plasmeijer, Rinus, & Jansen, Jan Martin. (2014). Church Encoding of Data Types Considered Harmful for Implementations. Plasmeijer, Rinus, & Tobin-Hochstadt, Sam (eds), *26th Symposium on Implementation and Application of Functional Languages (IFL)*. Presented version.

The Coq development team. (2014). *The Coq proof assistant reference manual*. LogiCal Project. Version 8.4.

Mogensen, TorbenÆ. (2001). An investigation of compact and efficient number representations in the pure lambda calculus. *Pages 205–213 of:* Bjørner, Dines, Broy, Manfred, & Zamulin, Alexandre (eds), *Perspectives of system informatics*. Lecture Notes in Computer Science, vol. 2244. Springer.

Norrell, Ulf, & the Agda Development Team. (2014). *The Agda Wiki*.

Okasaki, Chris. (1997). Three Algorithms on Braun Trees. *J. funct. program.*, **7**(6), 661–666.

Parigot, Michel. (1988). Programming with proofs: a second order type theory. *Pages 145–159 of:* Ganzinger, H. (ed), *European Symposium On Programming (ESOP)*. Lecture Notes in Computer Science, vol. 300.

Parigot, Michel. (1989). On the representation of data in lambda-calculus. *Pages 309–321 of:* Börger, Egon, Büning, HansKleine, & Richter, Michael (eds), *3rd Workshop on Computer Science Logic (CSL)*. Lecture Notes in Computer Science, vol. 440. Springer.

Philip Wadler. (1990). *Recursive types for free!* Available at `http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt`.