Church Encoding of Data Types Considered Harmful for Implementations

– Functional Pearl –

Pieter Koopman Rinus Plasmeijer
Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands
pieter@cs.ru.nl rinus@cs.ru.nl

Jan Martin Jansen
Netherlands Defence Academy (NLDA)
The Netherlands
jm.jansen.04@nlda.nl

Abstract

From the λ-calculus it is known how to represent (recursive) data structures by ordinary λ-terms. Based on this idea one can represent algebraic data types in a functional programming language by higher-order functions. Using this encoding we only have to implement functions to achieve an implementation of the functional language with data structures. In this paper we compare the famous Church encoding of data types with the less familiar Scott and Parigot encoding.

We show that one can use the encoding of data types by functions in a Hindley-Milner typed language by adding a single constructor for each data type. In an untyped context, like an efficient implementation, this constructor can be omitted. By collecting the basic operations of a data type in a type constructor class and providing instances for the various encodings, these encodings can co-exist in a single program. This shows the differences and similarities of the encodings clearly. By changing the instance of this class we can execute the same algorithm in a different encoding.

We show that in the Church encoding selectors of constructors yielding the recursive type, like the tail of a list, have an undesirable strictness in the spine of the data structure. The Scott encoding does not hamper lazy evaluation in any way. The evaluation of the recursive spine by the Church encoding makes the complexity of these destructors $O(n)$. The same destructors in the Scott encoding requires only constant time. Moreover, the Church encoding has serious problems with graph reduction. The Parigot encoding combines the best of both worlds, but in practice this does not offer an advantage.

Categories and Subject Descriptors D [*1*]: 1; D [*3*]: 3Data types and structures

Keywords Implementation, Data Types, Church Numbers, Scott Encoding, Parigot Encoding

1. Introduction

In the λ-calculus it is well-known how to encode data types by λ-terms. The most famous way to represent data types by functions in the λ-calculus is based on the encoding of Peano numbers by Church numerals [2, 3]. In this paper we review the Church encoding of data types and show that it causes serve problems complexity problems and spoils laziness. These problems can be prevented by using the far less known Scott encoding of data types.

Based on this approach we can transform the algebraic data types from functional programming language like Clean [30] and Haskell [17] to functions. These algebraic data types are first introduced in the language HOPE [10]. The algebraic data types are equivalent to the polynomial data types used in type theory. Representing all data types by plain functions simplifies the implementation of the programming language. The implementation only has to cope with plain higher order functions, instead of these functions and data types. Most abstract machines [22] used to implement programming languages contain special instructions to handle data types, e.g., the SECD machine [24], the G-machine [21, 29] and the ABC-machine [23]. When we represent data types by functions the transformed programs only contain functions. Since the implementation of the transformed programs does not have to cope with algebraic data types, they are easier to implement.

Barendregt [3] explains in chapter 6 how the Church encoding was transformed to the typed λ-calculus in papers of Böhm and various coauthors [5–9]. The Church encoding adapted to the typed λ-calculus is also known as the Böhm-Berarducci encoding. Barendregt introduces an alternative encoding called the *standard representation* in [2]. This is a two step approach to represent λ-terms. First, the λ-terms are represented as Gödel numbers. Next, these Gödel numbers are represented as Church Numerals.

The oldest description of an alternative encoding is in a set of unpublished notes of Dana Scott, see [12] page 504. This encoding is reinvented many times in history. For instance, the implementation of the language Ponder is based on this idea [13]. Also the language TINY from Steensgaard-Madsen uses the Scott encoding [32]. The representation of data types by Mogensen is an extension of the Scott encoding that enables reflection [3, 25]. For the implementation of the simple programming language SAPL we reinvented this encoding [18, 19]. SAPL is used to execute Clean code of an iTask program in the browser [31]. Naylor and Runciman used the ideas from SAPL in the implementation of the Reduceron [26]. Despite these publications the Scott encoding is still much less famous than the Church encoding. Moreover, we do not know a paper in which these encodings are thoroughly compared. Parigot proposed an encoding for data types that is a combination of the Church and Scott encoding [27, 28].

Using one additional constructor for each type and encoding, the encodings of the data types become valid types in the Hindley-Milner type system. This enables experiments with the encodings in typed functional languages like Clean and Haskell.

In this paper we use a type constructor class to capture all basic operations on algebraic data types, constructors as well as matching and selector functions. All manipulations of the data type in the source language are considered to be expressed in terms of the functions in this type class. By switching the instance of the type class, we obtain another encoding of the data type. We add a single constructor to the implementation of an algebraic data type by functions. This constructor is necessary to make the encoding typable in the Hindley-Milner type system. This is necessary in order to experiment with these encodings in a strongly typed language like Haskell or Clean. Moreover, the constructor enables us to distinguish the various encodings as different instances of a type class. Just like a `newtype` in Haskell there is no reason to use these constructors in an actual implementation based on these encodings. The uniform encoding based on type classes helps us to compare the various encodings of the data type.

In the next section we review the encoding of non-recursive data types in the λ-calculus. In Section 3 the λ-calculus is replaced by named higher-order functions. The named function makes the notation of recursive algorithms easier. We show how recursive data types can be represented by named function in Section 4. The Church and Scott encodings will be different instances of the same type constructor class. This makes the differences in the encoding very clear. Optimisations of the encodings are discussed in Section 5. In Section 6 we conclude that algorithmic complexity of the Church encoding is for recursive selector functions higher than the complexity of the Scott encoding.

2. Data Types in the λ-calculus

Very early in the development of λ-calculus it was known how to use λ-terms for manipulations handled nowadays by algebraic data types in functional programming languages. Since one does not want to extend the λ-calculus with new language primitives, λ-terms achieving the effect of the data types were introduced.

For an enumeration type with n constructors without arguments, the λ-term equivalent to constructor C_i, selects argument i from the given n arguments: $C_i \equiv \lambda\, x_1 \ \ldots x_n \,.\, x_i$. For each function over the enumeration type we supply n arguments corresponding to results for the constructors. The simplest nontrivial example is the type for Boolean values.

2.1 Booleans in λ-calculus

The Boolean values True and False can be represented by a data type having just two constructors T and F[1]. We interpret the first argument as true and the second one as false.

$$T \equiv \lambda\, t\, f \,.\, t$$
$$F \equiv \lambda\, t\, f \,.\, f$$

For the Boolean values the most famous application of these terms is probably the conditional.

$$cond \equiv \lambda\, c\, t\, e \,.\, c\; t\; e$$

Using Currying the conditional can also be represented as the identity function: $cond \ \equiv\ \lambda\, c \,.\, c$. We can even apply the Boolean value directly to the then- and else-part. This is used in the following definition of the logical and-operator.

$$and \equiv \lambda\, x\, y \,.\, x\; y\; F$$

If the first argument x is true the result of the and-function is determined by the argument y. Otherwise, the result of the and-function is F (false).

2.2 Pairs in λ-calculus

This approach can be directly extended to data constructors with non-recursive arguments. The arguments of the constructor in the original data type are given as arguments to the function representing this constructor. A pair is a data type with a single constructor. This constructor has two arguments. The functions `e1` selects the first element of such a pair and `e2` the second element. The λ-terms encoding such a pair are:

$$Pair \equiv \lambda\, x\, y\, p \,.\, p\; x\; y$$
$$e1 \equiv \lambda\, p \,.\, p\; (\lambda x\; y \,.\, x)$$
$$e2 \equiv \lambda\, p \,.\, p\; (\lambda x\; y \,.\, y)$$

With these terms we can construct a term that swaps the elements in such a pair as:

$$swap \equiv \lambda\, p \,.\, Pair\; (e2\; p)\; (e1\; p)$$

3. Representing Data Types by Named Functions

It seems to be straightforward to transform the λ-terms representing Booleans and pairs to functions in a functional programming language like Clean or Haskell. In this paper we will use Clean, but any modern lazy functional programming language with type constructor classes will give very similar results.

3.1 Encoding Booleans by Named Functions

The functions form Section 2.1 for the Booleans become[2][3]:

```
T :: a a → a
T t f = t

F :: a a → a
F t f = f

cond :: (a a → a) a a → a
cond c t e = c t e
```

3.2 Encoding Pairs by Named Functions

The direct translation of the λ-terms in Section 2.2 for pairs yields[4]:

```
Pair‘ :: a b → (a b → c) → c
Pair‘ x y = λp.p x y

e1‘ :: ((a b → a) → a) → a
e1‘ p = p (λx y.x)

e2‘ :: ((a b → b) → b) → b
e2‘ p = p (λx y.y)

swap‘ :: ((a a → a) → a) → (a a → b) → b
swap‘ p = Pair‘ (e2‘ p) (e1‘ p)
```

Unfortunately, the type for `swap‘` requires that both elements of the pair have the same type `a`. This is due to the fact that the type for `Pair‘` states the type of the access function as `a b→c` and

[1] In this paper we will use names starting with a capital for functions mimicking constructors and names starting with a lowercase letter for all other functions. Hence a boolean is a function with two arguments.

[2] For typographical reasons we generally prefer a function like `T t f = t` over the equivalent `T = λt f.t`.

[3] In Haskell the type `a a→a` is written as `a→a→a`.

[4] In Haskell the anonymous function `λp.p x y` is written as `\p->p x y`.

the Hindley-Milner type system requires a single type for such an argument [3]. The encoding with additional constructors and type constructor classes developed below in Section 4.1 will remove this limitation.

This restriction only limits the possibility to execute the encoding in functions in a strongly typed language. In an untyped context these functions will behave correctly.

4. Recursive Data Types

For recursive types we show two different ways to represent data types by functions. The first approach is a generalisation of the well-known Church numbers [2]. Here the recursive occurrences of a function representing a constructor are all equal. This implies that a recursive data structure mirrors an expanded fold as pointed out by Hinze in [16]. This is especially convenient in λ-calculus since recursive functions require that the function itself is passed around as an additional argument. This recursion is usually achieved by an application of the Y-combinator.

The second approach does nothing special for recursive arguments of constructors. Hence, it uses explicitly recursive manipulation functions for recursive data types, just like the algebraic data types in functional programming languages like Haskell and Clean. This arbitrary recursion pattern of these functions is not limited to folds of the Church encoding. The oldest source of this approach is a set of unpublished notes from Dana Scott, hence this approach is called the Scott encoding.

Since we have named functions, arbitrary recursion is no problem at all. For this reason we state both encodings of the data type and the associated manipulation functions directly as named functions. In order to compare both encodings easily and to be able to write functions that work for both encodings we construct type (constructor) classes for the data types in our description. The type constructor class will contain the constructors and the selection functions for elements of the constructors.

These type classes require a constructor in the functions representing data types. We already want to insert constructors to enable the use of the functional encodings in a strongly typed language. The constructors are still unnecessary when we fix the encoding and work in an untyped setting.

4.1 Pairs Revisited

In our new approach `Pair` is a type constructor class with a constructor `Pair` and two destructors `e1` and `e2`. These destructors select the first and second argument.

```
class Pair t where
    Pair :: a b → t a b
    e1   :: (t a b) → a
    e2   :: (t a b) → b
```

Using the primitives from this class, the `swap` function becomes[5]:

```
swap :: (t a b) → t b a | Pair t
swap p = Pair (e2 p) (e1 p)
```

Note that the use of the type class yields better readable types and eliminates the problem with the types of the arguments in the function `swap`. Here it is completely valid to use different types for the elements of the pair. In fact, it is even possible to yield a pair that is a member of another instance of the type constructor class `swap :: (t a b) → (u b a) | Pair t & Pair u`.

[5] The class restriction `| Pair t` in the type of the function `swap` states that this function works for any type `t` that is an instance of the type constructor class `Pair`. This ensures that `Pair`, `e1` and `e2` are defined for `t`. In Haskell such a class constraint is written as `swap :: (Pair t) ⇒ (t a b) → t b a`.

4.1.1 Pairs with Native Tuples

The instance of `Pair` for 2-tuples is completely standard.

```
instance Pair (,) where
    Pair a b  = (a, b)
    e1 (a, b) = a         // equal to the function fst from StdEnv
    e2 (a, b) = b         // equal to the function snd from StdEnv
```

4.1.2 Pairs with Functions

Since `Pair` is not recursive, both encodings of such a pair with functions coincide. We introduce a placeholder type `FPair` to satisfy the type class system. This also allows use to introduce a universally quantified type variable `t` for the result of manipulations of the type.

```
:: FPair a b = FPair (∀t: (a b→t) → t)
```

```
instance Pair FPair where
    Pair a b      = FPair λp.p a b
    e1 (FPair p) = p λa b.a
    e2 (FPair p) = p λa b.b
```

In order to give `FPair` the kind required by the type constructor class `Pair` the type of the result `t` is an universal quantified type in this definition. This makes the definition of `swap` typable in a Hindley-Milner type system, even without a type constructor class, while the definition of `swap‘` in Section 3.2 restricts the elements `e1` and `e2` of the pair to have identical types.

4.2 Peano Numbers

The simplest recursive data type is a type `Num` for Peano numbers. This type has two constructors. `Zero` is the non recursive constructor that represents the value zero. The recursive constructor `Succ` yields the successor of such a Peano number, it has another `Num` as argument. There are two basic manipulation functions, a test on zero and a function computing the predecessor of a Peano number.

```
class Num t where
    Zero   :: t
    Succ   :: t → t
    isZero :: t → Bool
    pred   :: t → t
```

It is of course possible to replace the basic type `Bool` by the type representing Booleans in this format introduced in Section 3. We use here a basic type to show that both type representation perfectly mix.

4.2.1 Peano Numbers with Integers

An implementation of these numbers based on integers is:

```
instance Num Int where
    Zero     = 0
    Succ   n = n+1
    isZero n = n == 0
    pred   n = if (n > 0) (n - 1) undef
```

4.2.2 Peano Numbers with an Algebraic Data Type

The implementation of `Num` with an ordinary algebraic data type `Peano` has no surprises. We use case expressions instead of separate function alternatives to make the definitions a little more compact.

```
:: Peano = Z | S Peano
```

```
instance Num Peano where
    Zero     = Z
    Succ   n = S n
    isZero n = case n of Zero = True; _ = False
    pred   n = case n of S m = m; _ = undef
```

4.2.3 Peano Numbers in the Church Encoding

The first encoding with functions is just the encoding of Church numbers in this format. The type `Peano` has two constructors. Hence, each constructor function has two arguments. The first argument represents the case for zero, the second one mimics the successor. A nonnegative number n is represented by n applications of this successor to the value for zero. The constructor `Succ` adds one application of this function.

The test for zero yields `True` when the given number `n` is `Zero`. Otherwise, the result is `False`. The predecessor function in the Church encoding is somewhat more challenging. The number n is represented by n applications of some higher order function `s` to a given value `z`. The predecessor must remove one of the function `s`. The first solution for this problem is found by Kleene while his wisdom teeth were extracted at the dentist [11]. The value zero is replaced by a tuple containing `undef`, the predecessor of zero[6], and the predecessor of the next number: `zero`. The successor function recursively replaces the tuple `(x,s x)` by `(s x,s (s x))` starting at `z`. The result of this construct is the tuple `(pred n,n)`. The predecessor function selects the first element of this tuple. Since the predecessor is constructed from the value zero upwards to n, the complexity of this operation is $O(n)$.

Just like in the representation of pairs we add a constructor `CNum` to solve the type problems of this representation in the Hindley-Milner type system of Clean. The universally quantified type variable `b` ensures that the functions representing the the constructors can yield any type without exposing this type in `CNum`. This type is very similar to the type cnat $:= \forall X.X \to (X \to X) \to X$ used for Church numbers in polymorphic λ-calculus, $\lambda 2$ [14].

The pattern `FPair _` in the `pred` function is an artefact of our encoding by type classes, it solves the overloading of `Pair`.

```
:: CNum = CNum (∀b: b (b→b) → b)

instance Num CNum where
  Zero   = CNum λzero succ.zero
  Succ n = CNum λzero succ.succ ((λ(CNum x).x) n zero succ)
  isZero (CNum n) = n True λx.False
  pred   (CNum n)
   = CNum λz s.e1 (n (Pair undef z)
                     (λp=:(FPair _).Pair (e2 p) (s (e2 p))))
```

Notice that the successor itself passes the values for `zero` and `succ` recursively to the given number `n`. Removing the constructor `CNum` from the argument of `Succ` is done in its right-hand side to prevent this argument that the argument is strict. This enables the proper evaluation of expressions like `isZero (Succ undef)`.

The function `Succ` has to add one application of the argument `succ` to the given number. Since all functions are equal this can be done in two ways. Above we add the additional application of `succ` to the encoding of `n`. It is in this encoding also possible to replace the given value of `zero` in the recursion by `succ zero`. That is `Succ (CNum n) = CNum λzero succ.n (succ zero) succ`. The direct access to both side of the sequence of applications of `succ` is unique for the Church encoding. We will use this below in Section 5 to optimise fold like operations over recursive types in the Church encoding.

4.2.4 Peano Numbers in the Scott Encoding

The type `SNum` to represent the Scott representation of numerals uses a constructor and a universally quantified type variable for exactly the same reasons as the Church encoding. This type is related to the types assigned to Scott numerals by Abadi et al., [1]. This type is again similar to snat $:= \forall X.X \to (\text{snat} \to X) \to X$ used for Scott numbers in $\lambda 2\mu$ [14]. Since we have recursive functions in our core language, the external recursion in this problem is no problem. In λ-calculus we need for instance a fixed point-combinator for the recursion.

The Scott encoding for the non-recursive cases `Zero` and `isZero` is equal to the Church encoding. For the recursive functions `Succ` and `pred` the Scott encoding is simpler than the Church encoding. The recursion pattern of the Scott encoding is very similar to the definitions for the type `Peano`.

```
:: SNum = SNum (∀b:b (SNum→b) → b)

instance Num SNum where
    Zero            = SNum λzero succ.zero
    Succ   n        = SNum λzero succ.succ n
    isZero (SNum n) = n True λx.False
    pred   (SNum n) = n undef λx.x
```

Since the implementation of `pred` in this Scott encoding is a simple selection of an element of a constructor its complexity is $O(1)$. This is much better than the $O(n)$ complexity of the same operator in the Church encoding.

4.2.5 Peano Numbers in the Parigot Encoding

Parigot proposed a different encoding of data types in an attempt to enable reasoning about algorithms as well as an efficient implementation of these algorithms [27, 28]. These papers do not mentioning the Scott encoding. In addition for a recursive type the constructors contain the Church-Style fold argument, as well as the Scott-style plain recursive argument. For numbers this reads:

```
:: PNum = PNum (∀b:b (PNum b→b) → b)

instance Num PNum where
    Zero            = PNum λz s.z
    Succ    p       = PNum λz s.s p ((λ(PNum n).n) p z s)
    isZero (PNum n) = n True λp x.False
    pred   (PNum n) = n undef (λp x.p)
```

It will be no surprise that this type resembles the type in $\lambda 2\mu$: pnat $:= \forall X.X \to (\text{pnat} \to X \to X) \to X$ used for Parigot numbers in [14] (called Church-Scott numbers there).

Notice that `pred` is implemented here in the more efficient Scott way. The second argument of `Succ` is more suited for a fold-like operation.

4.2.6 Using the Type Class Num

Using the primitive from the class `Num` we can define manipulation functions for these numbers. The transformation of any of these number encodings to one of the other encodings is given by `NumToNum`. This uniform transformation is a generalisation of the transformations between Church and Scott numbers in [20]. The context determines the encodings `n` and `m`. Using a very similar recursion pattern we can define addition for all instances of `Num` by the function `add`.

```
NumToNum :: n → m | Num n & Num m
NumToNum n | isZero n
    = Zero
    = Succ (NumToNum (pred n))

add :: t t → t | Num t
add x y | isZero x
    = y
    = add (pred x) (Succ y)
```

[6] Every now and then people use `zero` instead of `undef` as predecessor of `Zero`. This prevents runtime errors, but it does not correspond to our intuition of numbers and complicates reasoning. For instance, the property `pred n = pred m ⇒ n = m` does not hold since `pred (succ zero)` becomes equal to `pred zero`.

Using details of the encoding it is possible to optimise these functions. Although the definitions work for all instances, the algorithmic complexity depends on the encoding selected. In particular the processor function `pred` is $O(n)$ in the Church encoding and $O(1)$ for the other implementations of `Num`. In Section 5 we discuss how this can be improved for these examples.

4.3 Lists

In the Peano numbers all information is given by the number of applications of `Succ` in the entire data structure. Recursive data types that contain more information are often needed. The simplest extension of the Peano numbers is the list. The `Cons` nodes of a list corresponds to the `Succ` in the Peano numbers, but in contrast to the Peano numbers a `Cons` contains an element stored at that place in the list. This is modelled by type class `List`. Compared to `Num` there is an additional argument `a` in the constructor for the recursive case, and there is an additional primitive access function `head` to select this element from the outermost `Cons`.

```
class List t where
    Nil   :: t a
    Cons  :: a (t a) → t a
    isNil :: (t a) → Bool
    head  :: (t a) → a
    tail  :: (t a) → t a
```

4.3.1 List with the Native List Type

The instance for the native lists in Clean is very simple[7]

```
instance List [] where
    Nil       = []
    Cons a x  = [a:x]
    isNil  xs = case xs of [] = True;  _ = False   // isEmpty
    head   xs = case xs of [a:x] = a;  _ = undef   // hd
    tail   xs = case xs of [a:x] = x;  _ = undef   // tl
```

4.3.2 List in the Church Encoding

The instance inspired on the Church numbers is rather similar to the instance for `CNum`. The definition for `Nil` is completely similar to the instance for `Zero`. The constructor `Cons` has the list element to be stored as additional argument. Here it does matter whether we insert the new element at the head or the tail of the list. It is actually quite remarkable that we can add an element to the tail of the list without explicit recursion. Note that the arguments for `nil` and `cons` are passed recursively to the tail `x` of the list. The manipulation functions `isNil` and `head` directly yield the desired result by applying the function `xs` to the appropriate arguments. The implementation of `tail` is more involved. We use the approach known from `pred`. From the end of the list upwards a new list is constructed that is the tail of this list. Note that this is again $O(n)$ work with n the length of the list. Moreover, it spoils lazy evaluation by requiring a complete evaluation of the spine of the list. This also excludes the use of infinite list as arguments of this version of the `tail`.

```
:: CList a = CList (∀b: b (a→b→b) → b)

instance List CList where
 Nil      = CList λnil cons.nil
 Cons a x = CList λn c.c a ((λ(CList l).l) x n c)
 isNil  (CList l) = l True λa x.False
 head   (CList l) = l undef λa x.a
 tail   (CList l)
  = CList λnil cons.e1 (l (Pair undef nil)
        (λa p=:(FPair _).Pair (e2 p) (cons a (e2 p))))
```

[7] In Haskell the list `[a:x]` is written as `(a:x)`. The expression `[a:x]` is valid Haskell, but it is a singleton list containing the list `(a:x)` as its element.

4.3.3 List in the Scott Encoding

The implementation of lists based on Scott numbers differs at the recursive argument of the `Cons` constructor. Here we use a term of type `SList a`. In the list based on Church numbers this argument has type `b`, the result type of the list manipulation. As a consequence, we do not pass the arguments `nil` and `cons` as arguments to the tail `x` in the definition for the constructor `Cons`. This makes the access function `tail` a simple $O(1)$ access function.

```
:: SList a = SList (∀b: b (a→(SList a)→b) → b)

instance List SList where
    Nil              = SList λnil cons.nil
    Cons a x         = SList λnil cons.cons a x
    isNil (SList xs) = xs True λa x.False
    head  (SList xs) = xs undef λa x.a
    tail  (SList xs) = xs undef λa x.x
```

4.3.4 List in the Parigot Encoding

Just as for numbers the Parigot encoding of `Cons` contains an argument for Scott type of recursion (i.e. `x`), as well as for the Church type recursion (i.e. `((PList l).l) x n c`).

```
:: PList a = PList (∀b: b (a (PList a) b→b) → b)

instance List PList where
    Nil      = PList λnil cons.nil
    Cons a x = PList λn c.c a x ((λ(PList l).l) x n c)
    isNil (PList l) = l True λa t x.False
    head  (PList l) = l undef (λa t x.a)
    tail  (PList l) = l undef (λa t x.t)
```

4.3.5 Using the List Type Class

Using the primitives from `List` the list manipulations fold-right and fold-left can be defined in the well-known way.

```
foldR :: (a b→b) b (t a) → b | List t
foldR op r xs | isNil xs
    = r
    = op (head xs) (foldR op r (tail xs))

foldL :: (a b→a) a (t b) → a | List xs
foldL op r xs | isNil xs
    = r
    = foldL op (op r (head xs)) (tail xs)
```

Due to the $O(n)$ complexity of `tail` in the Church representation, the folds have $O(n^2)$ complexity in the Church encoding when the operators `op` is strict in both arguments. In the other instances of `List` the complexity is only $O(n)$. In Section 5 we show how this complexity problem can be fixed for `foldR`.

The transformation from one list encoding to any other instance of `List` is done in `ListToList` by an application of this `foldR`. The summation of a list is done by a fold-left since the use of an accumulator enables a constant memory implementation. This function works for any argument type `a` having an addition operator `+` and a unit element `zero`.

```
ListToList :: (t a) → u a | List t & List u
ListToList xs = foldR Cons Nil xs

suml :: (t a) → a | List t & +, zero a
suml xs = foldL (+) zero xs
```

4.3.6 Measuring Execution Time

Using these definitions we can easily verify the described behaviour of the implementations of the class `List`. Our first example is extremely simple; it takes the `head` of the `tail` of a list.

By just changing the type at a strategic place, here the function `headTail`, we enforce another implementation of `List`. When we replace `CList` in this function by `SList`, `PList` or `[]` that type instance of `List` is used. The length of the list is controlled by the definition of `m`.

```
headTail :: !(CList Int) → Bool
headTail l = head (tail l) == 2

fromTo :: Int Int → t Int | List t
fromTo n m | n > m
    = Nil
    = Cons n (fromTo (n+1) m)

Start = headTail (fromTo 1 m)
```

All experiments are done with 32-bit Clean 2.4 running on windows 7 in a virtual box on a MacBook Air with 1.8 GHz Intel Core i5 under OS X version 10.9.4. For reliable measurements the computation is repeated such that the total execution time is at least 10 seconds.

head (tail (fromTo 1 m))				
m	CList Int	SList Int	PList Int	[Int]
2	$6.5\,10^{-8}$	$3.3\,10^{-8}$	$5.5\,10^{-8}$	$1.1\,10^{-8}$
10	$2.8\,10^{-7}$	$3.3\,10^{-8}$	$5.5\,10^{-8}$	$1.1\,10^{-8}$
10^2	$2.5\,10^{-6}$	$3.3\,10^{-8}$	$5.5\,10^{-8}$	$1.1\,10^{-8}$
10^3	$2.6\,10^{-5}$	$3.3\,10^{-8}$	$5.5\,10^{-8}$	$1.1\,10^{-8}$
10^4	$2.8\,10^{-5}$	$3.3\,10^{-8}$	$5.5\,10^{-8}$	$1.1\,10^{-8}$
10^5	$4.7\,10^{-5}$	$3.3\,10^{-8}$	$5.5\,10^{-8}$	$1.1\,10^{-8}$

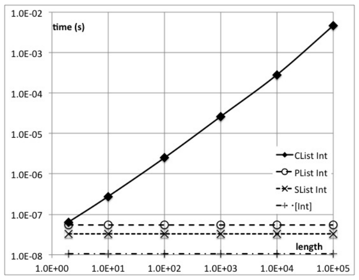

Figure 1. Execution time in seconds as function of the length for the encodings of `List`. Note the double logarithmic scale.

It is no surprise that the execution time for `SList Int` and `[Int]` is completely independent of the upper bound, `m`, of the list. Due to lazy evaluation the list is only evaluated until its second element. As predicted the execution time for `CList Int` is linear in the length of the list since `tail` enforces evaluation until the `Nil`. For very long lists in the Church representation, e.g. 10^5 elements, garbage collection causes an additional increase of the execution time.

In the second experiment we enforce evaluation of the entire list by computing the sum of the numbers 1 to m and check whether this sum is indeed $m(m-1)/2$ for various values of m. We use a tail recursive definition for the function `Sum`:

```
sum :: (t Int) → Int | List t
sum l | isNil l
    = 0
    = head l + sum (tail l)
```

sum (fromTo 1 m)				
m	CList Int	SList Int	PList Int	[Int]
1	$1.2\,10^{-7}$	$6.6\,10^{-8}$	$9.7\,10^{-8}$	$2.0\,10^{-8}$
10	$3.7\,10^{-5}$	$5.4\,10^{-7}$	$7.5\,10^{-7}$	$1.4\,10^{-7}$
10^2	$2.8\,10^{-2}$	$5.9\,10^{-6}$	$8.2\,10^{-6}$	$2.0\,10^{-6}$
10^3	$3.3\,10^{+1}$	$5.8\,10^{-5}$	$8.3\,10^{-5}$	$2.1\,10^{-5}$
10^4		$6.2\,10^{-4}$	$9.0\,10^{-4}$	$2.1\,10^{-4}$
10^5		$6.5\,10^{-3}$	$9.6\,10^{-3}$	$2.2\,10^{-3}$

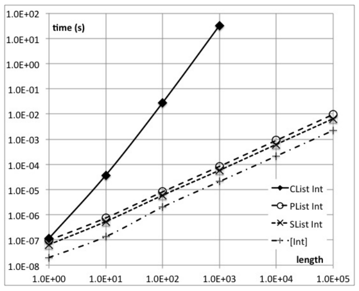

Figure 2. Execution time in seconds as function of the length for the encodings of `List`.

As expected the execution time for `SList Int` and `[Int]` grows linear with the the number of elements in the list. The version for `CList Int` is again much slower. Since the `tail` inside `sum` is $O(n)$ for a list in the Church representation, the `sum` itself is at least $O(n^2)$. The measurements show that the actual execution time grows as $O(n^3)$ in the Church representation. Since the `tail` in the Church representation yields a function application, the reduction of an application `tail l` cannot be shared. This expression is re-evaluated for each use of the resulting list. Each of these `tail` functions is $O(n)$. This makes the total complexity of `sum` $O(n^3)$ in the Church representation and $O(n)$ in the Scott representation and in the native lists of Clean.

In the final example we apply the quick-sort algorithm to a lists of pseudo random integers in the range $0..999$. Quick-sort is implemented for all list implementations in the class `List` by the function `qs`.

```
qs :: (l a) → (l a) | List l & <, == a
qs l | isNil l
    = Nil
    = append (qs (fltr (λx.x <  y) l))
          (append (fltr (λx.x == y) l)
              (qs (fltr (λx.y <  x) l))) where y = head l

append :: (t a) (t a) → (t a) | List t
append l1 l2 | isNil l1
    = l2
    = Cons (head l1) (append (tail l1) l2)

fltr :: (a→Bool) (l a) → l a | List l
fltr p l | isNil l
    = Nil
    | p x
        = Cons x (fltr p (tail l))
        =        fltr p (tail l)       where x = head l
```

qs (take m randomInts))				
m	CList Int	SList Int	PList Int	[Int]
1	$7.5\,10^{-7}$	$5.2\,10^{-7}$	$6.0\,10^{-7}$	$1.8\,10^{-7}$
10	$1.8\,10^{-4}$	$1.0\,10^{-5}$	$1.3\,10^{-5}$	$4.1\,10^{-6}$
10^2	$1.5\,10^{-1}$	$2.1\,10^{-4}$	$2.8\,10^{-4}$	$8.9\,10^{-5}$
10^3	$1.6\,10^{+2}$	$3.0\,10^{-3}$	$3.9\,10^{-3}$	$1.3\,10^{-3}$
10^4		$3.3\,10^{-2}$	$4.4\,10^{-2}$	$1.4\,10^{-2}$
10^5		$3.9\,10^{-1}$	$5.3\,10^{-1}$	$1.7\,10^{-1}$

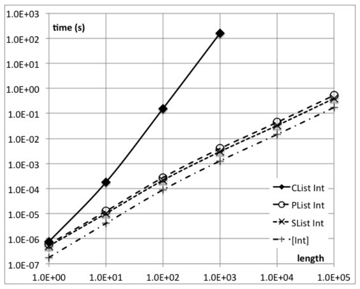

Figure 3. Relation between execution time in seconds and length of the list for four different implementations of `List`.

The measurements show the expected $O(n \log n)$ growth with the length of the list for the Scott representation of lists and native lists in Clean. The Church representation shows again a $O(n^3)$ growth with the length of the list. Since Quick-sort requires more list operations than `sum`, the increase in execution time is even bigger than for `sum`.

The Scott representation, `SList Int`, is on average a factor 2.8 slower than the native Clean lists, `[Int]`. This additional execution time is caused by the additional constructors `SList` need to convince the type system of correctness of this representation. This factor is independent of the size of the lists.

Functions like `sum`, `append` and `fltr` used in these examples can be expressed as applications of `fold`. It is possible to optimise a `fold` in the Church representation as outlined in Section 5. Since we study the representation of data structures by functions for a *simple* compiler, it is unrealistic to expect such a compiler to perform the required transformations. Moreover, this does not solve the problems with laziness, in examples like `head (tail (fromTo 1 m))`, and the complexity problems in situations where the `tail` function is not part of a `foldr`, like `qs_o2`.

4.4 Tree

We demonstrate how the approach is extended to data types with multiple recursive arguments such as binary trees. The constructor for nonempty trees, `Fork`, has now two recursive instances as argument instead of just one. There are now two selectors for the recursive arguments, `left` and `right`, instead of just `tail`.

```
class Tree t where
    Leaf    :: t a
    Fork    :: (t a) a (t a) → t a
    isLeaf  :: (t a) → Bool
    elem    :: (t a) → a
    left    :: (t a) → t a
    right   :: (t a) → t a
```

4.4.1 Tree with an Algebraic Data Type

The instance for a two constructor algebraic data type is again standard.

```
:: Bin a = Empty | Node (Bin a) a (Bin a)

instance Tree Bin where
    Leaf       = Empty
    Fork x a y = Node x a y
    isLeaf t   = case t of Empty = True; _ = False
    elem   t   = case t of (Node x a y) = a; _ = undef
    left   t   = case t of (Node x a y) = x; _ = undef
    right  t   = case t of (Node x a y) = y; _ = undef
```

4.4.2 Tree in the Church Encoding

The instance based on Church numbers passes the arguments for `leaf` and `node` now to two recursive occurrences in the constructor `Fork` for nonempty trees. The selector functions for the recursive arguments, `left` and `right`, use the same pattern as `pred` and `tail`. The difference is that there are two recursive cases. Fortunately, they can be handled with a single function. For readability we use tuples from Clean instead of a `Pair` as introduced in Section 2.2. Like above this selection function visits all n nodes in the subtree. Hence, its complexity is $O(n)$ while the version using ordinary algebraic data types does the job in constant time, $O(1)$.

```
:: CTree a = CTree (∀t: t (t a t→t) → t)

instance Tree CTree where
 Leaf = CTree λleaf fork
 Fork (CTree x) a (CTree y)
  = CTree λleaf fork. fork (x leaf fork) a (y leaf fork)
 isLeaf (CTree t) = t True λx a y.False
 elem   (CTree t) = t undef λx a y.a
 left   (CTree t)
  = CTree λe f.e1 (t (undef,e) (λ(s,t) a (x,y).(t,f t a y)))
 right  (CTree t)
  = CTree λe f.e1 (t (undef,e) (λ(s,t) a (x,y).(y,f t a y)))
```

4.4.3 Tree in the Scott Encoding

The version based on the Scott encoding of numbers is again much more similar to the implementation based on plain algebraic data types. In the constructor `Fork` for nonempty trees the arguments `leaf` and `node` are not passed to the recursive occurrences of the tree, `x` and `y`. This makes the selection of the recursive elements identical to the selection of the non recursive argument, `elem`. The complexity of the three selection functions is the desired $O(1)$.

```
:: STree a = STree (∀t: t ((STree a) a (STree a)→t) → t)

instance Tree STree where
    Leaf = STree λleaf node.leaf
    Fork x a y = STree λleaf node.node x a y
    isLeaf (STree t) = t True λx a y.False
    elem   (STree t) = t undef λx a y.a
    left   (STree t) = t undef λx a y.x
    right  (STree t) = t undef λx a y.y
```

4.4.4 Using the Tree Type Class

Using the primitives from `Tree` we can express insertion for binary search trees by the function `insertTree`.

```
insertTree :: a (t a) → t a | Tree t & < a
insertTree a t
    | isLeaf t  = Fork Leaf a Leaf
    | a < x     = Fork (insertTree a (left t)) x (right t)
    | a > x     = Fork (left t) x (insertTree a (right t))
    | otherwise = t // x == a
where x = elem t
```

Note that the recursion pattern in this function is dependent of the value of the element to be inserted, `a`, and the element in the current node, `elem t`.

When the selectors `left` and `right` operate in constant time the average cost of an insert in a balanced tree are $O(\log\ n)$, and in worst case the insert is $O(n)$ work. For the implementation based on Church numbers however, the complexity of the selectors `left` and `right` is $O(n)$. This makes the average complexity of an insert in a balanced tree $O(n \log\ n)$, in worst case the complexity is even $O(n^2)$. In testing basic properties of trees this is very well noticeable. Even with small test cases tests with the data structures based on the Church numbers take at least one order of magnitude more time than all other tests for the definitions in this paper together.

4.5 General Transformations

The examples in the previous sections illustrate the general transformation scheme from a constructor based encoding to a function based encoding. In the transformations below we omit the additional type and constructors used in this paper to handle the various versions in a single type constructor class. A type T with a arguments and n constructors named $C_1 \ldots C_n$ will be represented by n functions. The transformation $\mathcal{T}$ specifies the functions needed to represent a type T.

$$\begin{aligned}\mathcal{T}[\![\, T\, x_1 \ldots x_a = C_1\, a_{1_1} \ldots a_{1_m} | \ldots |C_n\, a_{n_1} \ldots a_{n_m} \,]\!] \\ = \mathcal{C}[\![\, C_1\, a_{1_1}\, \ldots\, a_{1_m} \,]\!]\, n \cdots \mathcal{C}[\![\, C_n\, a_{n_1}\, \ldots\, a_{n_m} \,]\!]\, n\end{aligned}$$

The function for constructor C_i with m arguments has the same name as this constructor and has $n + m$ arguments. $\mathcal{C}$ yields the function for the given constructor.

$$\begin{aligned}&\mathcal{C}[\![\, C_i\, a_{i_1} \ldots a_{i_m} \,]\!]\, n \\ &\quad = C_i\, a_1 \ldots a_m\, x_1\, \ldots\, x_n = x_i\, \mathcal{A}[\![\, a_1 \,]\!]\, n \ldots \mathcal{A}[\![\, a_m \,]\!]\, n\end{aligned}$$

For all arguments in the Scott encoding and the non recursive arguments in the Church encoding, the transformation $\mathcal{A}$ just produces the given argument.

$$\mathcal{A}[\![\, a \,]\!]\, n = a$$

For recursive arguments in the Church encoding however, all arguments of the function C are added:

$$\mathcal{A}_2[\![\, a \,]\!]\, n = (a\, x_1\, \ldots\, x_n)$$

These definitions show that a constructor is basically just a selector that picks the continuation corresponding to its constructor number. In a function over type T we provide a value for each constructor, similar to a case distinction in a switch expression.

$$\begin{aligned}&\mathcal{S}[\![\,\mathsf{case}\ e\ \mathsf{of}\\&\quad C_1\ a_{1_1}\ldots a_{1_m} = r_1;\\&\quad \ldots\\&\quad C_n\ a_{n_1}\ldots a_{n_m} = r_n;\ ]\!]\\&= e\ (\lambda\, a_{1_1}\ \ldots\ a_{1_m}\ .\, r_1)\ \ldots\ (\lambda\, a_{n_1}\ \ldots\ a_{n_m}\ .\, r_n)\end{aligned}$$

Due to recursive passing of arguments in the Church encoding, a recursive argument a_j will be transformed to an expression of the result type R of the case expression. In the Scott encoding it will still have type T. This implies that we can still decide in the body r_i whether we want to apply the function recursively or not. In the Church encoding the function is alway applied recursively.

5. Optimisations

In a real implementation of a functional language that represents data types by functions, the type constructor classes introduced here and the constructors required by those type classes should be omitted. They are only introduced in this paper to allow experiments with the various representations.

A version of the Church encoding for lists without additional constructors can be expressed directly in Clean.

```
:: ChurchList a r :== r (a r→r) → r

cnil :: (ChurchList a r)
cnil = λn c.n

ccons :: a (ChurchList a r) → (ChurchList a r)
ccons a x = λnil cons.cons a (x nil cons)

ctail :: (ChurchList a (r,r)) → (ChurchList a r)
ctail xs = λn c.fst (xs (undef,n) (λa (x,y).(y,c a y)))

clToStrings::(ChurchList a [String]) → [String]|toString a
clToStrings xs = xs ["[]"] λa x.[toString a, ": ": x]
```

Although these function are accepted by the compiler, there are severe limitations to this approach. The type system rejects many combinations of these function. This is due to the monomorphism constraint on function arguments. These problems are very similar those encountered by the function `swap'` in Section 3.2.

The Hindley-Milner type system does not accept the Scott encoding of data types without additional constructors, see Barendregt [3] or Barendsen [4] for a proof. Geuvers shows that this can be typed in $\lambda 2\mu$: $\lambda 2$ + positive recursive types [14]. Nevertheless, these functions work correctly in an untyped world with higher order functions.

```
snil      = λnil cons.nil
scons a x = λnil cons.cons a x
stail xs  = xs undef (λa x.x)
```

5.1 Using the Structure of the Type Representations

The destructors of the implementations of data constructors based on Church numbers are all very expensive operations, typically $O(n)$ where n is the size of the recursive data structure. When the shape of the computation matches the recursive structure of the Church encoding we can achieve an enormous optimisation by replacing definitions based on the interface provided by the type constructor classes by direct implementations. Since the encoding based on Church numbers is basically a `foldr`, as explained by Hinze in [16], these optimisations will work for manipulations that can be expressed as a `foldr`.

5.2 Peano Numbers

Many operations on Peano numbers can be expressed as a fold operation. For instance, a Peano number of the form `λzero succ.succ (succ .. zero)` can be transformed to an integer by providing the argument 0 for `zero` and the increment function, `inc`, for integers for `succ`. For the same operation on number in the Scott encoding we need to specify the required recursion explicitly. This is reflected in the tailor made instances of the class `toInt` for these number encodings.

```
instance toInt CNum where toInt (CNum n) = n 0 inc
instance toInt SNum where toInt (SNum n) = n 0 (inc o toInt)
```

The complexity of both transformations is $O(n)$. For the Church encoding this is a serious improvement compared to using `NumToNum` to transform a `CNum` to `Int`. Due to the $O(n)$ costs of `pred` for `CNum` the complexity of this transformation is $O(n^2)$. For the other encoding we can at best gain a constant factor. This can be generalised in a translation of `CNum` to other instances of `Num`.

```
CNumToNum :: CNum → n | Num n
CNumToNum (CNum n) = n Zero Succ
```

Similar clever definitions are known for the addition and multiplication of Church numbers. We express the optimised addition as an instance of the operator `+` for `CNum`. We achieve addition by replacing the `zero` of `x` by the number `y zero succ`. In exactly the same way we can achieve multiplication by replacing the successor `such` of `x` by `λz.y z succ`.

```
instance + CNum where
 (+) (CNum x) (CNum y) = CNum λz s.x (y z s) s
instance * CNum where
 (*) (CNum x) (CNum y) = CNum λz s.x z (λz2.y z2 s)
```

It is obvious that this reduces the complexity of these operations significantly. However, the addition is not the constant $O(1)$ manipulation it might seem to be. The result is a function and any application determining a value with this function will be $O(n \times m)$ work. This is of course a huge improvement to $O(n^2 \times m)$ for the addition for `CNum` using the function `add` from Section 4.2.6.

Those optimisations are only possible when the structure of the manipulation can be expressed by the structure of the encoding of `CNum`. No solutions of this kind are known for operations like predecessor and subtraction. For the predecessor it might look attractive to transform the encoding from `CNum` to `SNum` and perform the predecessor here in $O(1)$ instead of in $O(n)$ in `CNum`. Unfortunately, this transformation itself is $O(n)$, even using the optimised `CNumToNum` function. Nevertheless, such a transformation is worthwhile when we need to do more operation with higher cost in the `CNum` encoding that in the `SNum` encoding. This occurs for instance in subtraction m from n by repeated applications of the predecessor function, here the complexity drops from $O(n \times m)$ to $O(n + m)$. This is still more expensive that the $O(m)$ for the other encodings.

5.3 Lists

The Church encoding of lists is based on a fold-right. Also the Parigot encoding contains such a fold. By making an optimised fold function instead of the simple recursive version from Section 4.3.5 we can take advantage of this representation.

```
class foldR_o t :: (a b→b) b !(t a) → b

instance foldR_o CList where
 foldR_o f r (CList l) = l r f

instance foldR_o PList where
foldR_o f r (PList l) = l r λa t x.f a x
```

```
instance foldR_o SList where
 foldR_o f r (SList l) = l r λa x.f a (foldR_o f r x)

instance foldR_o []     where
 foldR_o f r l = case l of []=r; [a:x]=f a (foldR_o f r x)
```

For the Church and Parigot encoding of lists we directly use the given function `f` the the fold of this representation. The Scott encoding and the native lists of Clean does not have such a direct fold. Hence, we define an explicit recursive function.

5.4 Effect of the Optimisations

In order to determine the effect of the optimised fold implementation we replaced the functions `append` and `filter` in the function `qs` by their fold based variant shown above.

```
qs_o :: (t a) → (t a) | <, == a & foldRo, List t
qs_o l | isNil l
    = Nil
    = appendo (qs_o (fltro (λx.x < h) l))
          (appendo (fltro (λx.x == h) l)
             (qs_o (fltro (λx.h < x) l))) where h = head l

append_o :: (t a) (t a) → t a | foldRo, List t
append_o l1 l2 = foldR_o Cons l2 l1

fltr_o :: (a→Bool) (t a) → t a | foldRo, List t
fltr_o p l = foldR_o (λa x.if (p a) (Cons a x) x) Nil l
```

The results of this experiment are listed in Figure 4.

qs_o (take m randomInts))				
m	CList Int	SList Int	PList Int	[Int]
1	$3.2\,10^{-7}$	$4.5\,10^{-7}$	$3.5\,10^{-7}$	$1.1\,10^{-7}$
10	$5.8\,10^{-6}$	$6.2\,10^{-6}$	$6.5\,10^{-6}$	$2.1\,10^{-6}$
10^2	$1.2\,10^{-4}$	$1.3\,10^{-4}$	$1.3\,10^{-4}$	$4.7\,10^{-5}$
10^3	$1.7\,10^{-3}$	$1.9\,10^{-3}$	$1.9\,10^{-3}$	$7.2\,10^{-4}$
10^4	$1.8\,10^{-2}$	$2.0\,10^{-2}$	$2.0\,10^{-2}$	$7.9\,10^{-3}$
10^5	$2.4\,10^{-1}$	$2.9\,10^{-1}$	$2.9\,10^{-1}$	$1.0\,10^{-1}$

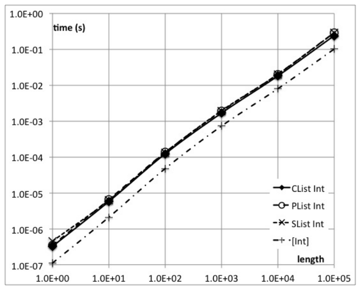

Figure 4. Execution time in seconds as function of the length for the encodings of `List`.

When all recursive list operations are replaced by an optimised fold-right all encodings of lists show very similar execution results. The small differences are explained by the additional arguments that have to be passed around in the Parigot encoding, and the additional constructors needed by this simulation of the Scott encoding.

These gains work only properly when every list manipulation is a fold-right. Introducing a single other operation can completely spoil the performance. Consider for instance a somewhat different formulation of our Quick-sort algorithm.

```
qs_o2 :: (t a) → (t a) | <, == a & foldRo, List t
qs_o2 l | isNil l
    = Nil
    = appendo (qs_o2 (fltro (λx.x < h) t))
      (Cons h (qs_o2 (fltro (λx.h ≤ x) t)))
where h = head l; t = tail l
```

The measurements in Figure 5 show that behaves similar to fold-based Quick-sort for most encodings. For small lists two instead of three filters over the list yields a gain. For long lists there will be many duplicates of the numbers between 0 and 999, hence the additional of equal elements yields a small gain. The overal complextity is $O(n\ \log n)$. For the Church-lists however, the single `tail` is a complete party breaker. For the same reasons as before the complexity is $O(n^3)$ in this Church encoding.

qs_o2 (take m randomInts))				
m	CList Int	SList Int	PList Int	[Int]
1	$2.1\,10^{-7}$	$2.2\,10^{-7}$	$1.8\,10^{-7}$	$7.2\,10^{-8}$
10	$1.2\,10^{-5}$	$4.3\,10^{-6}$	$3.7\,10^{-6}$	$2.2\,10^{-6}$
10^2	$5.6\,10^{-3}$	$9.4\,10^{-5}$	$9.4\,10^{-5}$	$5.8\,10^{-5}$
10^3	$5.6\,10^{0}$	$1.5\,10^{-3}$	$1.5\,10^{-3}$	$1.1\,10^{-3}$
10^4	$6.0\,10^{+3}$	$2.3\,10^{-2}$	$2.3\,10^{-2}$	$1.8\,10^{-2}$
10^5		$9.1\,10^{-1}$	$8.2\,10^{-1}$	$5.8\,10^{-1}$

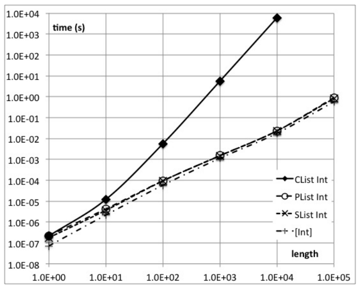

Figure 5. Execution time in seconds as function of the length for the encodings of `List`.

5.5 Optimization of other Operations

This kind of optimisation works only for operations that can be expressed as a fold-right. This implies that we cannot use it for many common list manipulations like, fold-left (`foldl`), `take`, `drop`, and insertion in a sorted list. For many operations that require a repeated application of `tail` it is worthwhile to transform the `CList` to a `SList`, perform the transformation on this `SList`, and finally transform back to the `CList` when we really want to use the Church encoding. This route via the lists in Scott encoding still force the evaluation of the spine of the entire list.

In some programming tasks it is possible to construct also in the Church encoding an implementation that executes the given task with a better complexity than obtained by applying the default deconstructs. For instance, the function to take the first `n` elements of a list using the functions from the class `List` is:

```
takeL :: Int (t a) → t a | List t
takeL n xs | n > 0 && not (isNil xs)
    = Cons (head xs) (takeL (n - 1) (tail xs))
    = Nil
```

For the Church encoding this has complexity $O(n \times L)$ where n is the number of elements to take and L is the length of the list. The complexity for the Scott encoding is $O(n)$, just like a direct definition in Clean. In contrast with the Church encoding, the Scott encoding is not strict in the spine of the list. Using a tailor-made instance of the fold in the Church encoding, the complexity of the take function can be reduced to $O(L)$:

```
takeC :: Int (CList a) → CList a
takeC n (CList xs)
 = fst (xs (Nil, xs 0 (λa x.x + 1) - n)
          (λa (ys, m).(if (m > 0) Nil (Cons a ys), m - 1)))
```

The expression `xs 0 (λa x.x+1)` computes the length of the lists. The outermost fold produces $length\ xs - n$ times `Nil`. When the counter `m` becomes non-positive, the fold starts copying the list elements. For finite lists, this is the same result as `takeL m`. It is not obvious how to derive such an optimised algorithm from an arbitrary function using the primitives from a class like `List`. Hence, constructing an optimised version requires in general non-trivial human actions. Even when this problem would be solved, the Scott encoding still has a better complexity and more appealing strictness properties.

5.6 Deforestation of Lists

Function fusion is a program transformation that tries to achieve the result of two separate functions f and g applied after each other by a single function h. That is we try to generate a function h such that $f \cdot g \equiv h \Rightarrow \forall x\,.\, f(g\ x) = h\ x$. Especially when fusion manages to eliminate a data structure this transformation will reduce the execution time significantly.

Wadler introduced a special form of fusion called deforestation [33]. In deforestation we recognise *producers* and *consumers*. For lists, any function forcing evaluation and processing the list like `foldr` is a good consumer. A producer is a similar recursive function that generates a list. When there is an immediate composition of a producer and a consumer these functions can be fused and the intermediate list is not longer needed. That is a function composition like `foldR g r (foldR (Cons o f) Nil xs)` can be fused to `foldR (g o f) r xs`. This is a standard transformation in many advanced compilers for functional programming languages, e.g. the GHC as described by Gill et. al. [15].

Since any `CList` is essentially a `foldR` that forces evaluation, all Church lists are good consumers. When the list is generated by a `foldRC` the function applied to as `Cons`-constructor can be written as `Cons o f`. Hence functions like `mapC` are good producers. This implies that there will be relatively many opportunities for deforestation in a program based on Church lists. Although the gain of deforestation can be a substantial factor, it does not change the complexity of the algorithm.

5.7 Optimisation of Tree Manipulations

The results from lists immediately carry over to trees. Any fold over a tree in the Church encoding can be optimised by to call of `foldTC`.

```
foldTC :: (b a b→b) b (CTree a) → b
foldTC f r (CTree t) = t r f
```

Using this fold we can collect all elements in a search tree by a single in-order traversal of the Church tree in a Church list by `inorderC`.

```
inorderC :: (CTree a) → CList a
inorderC t = foldTC (λx a y.appendC x (Cons a y)) Nil t
```

This works only for tree manipulations that can be expressed efficiently by a fold over the tree. As a consequence it does not solve the complexity problems for insertion, lookup and deletion from binary search trees.

6. Summary

For simple implementations of functional programming languages it is convenient to transform the data types to functions. By translating all data types to functions, we only need to implement functions on the core level in our implementation. The implemented language still provides algebraic data types like lists and trees to the user, but there is no runtime notion of these types needed. Such transformations are well known in λ-calculus. In this paper we use a language with named functions and basic types like numbers and characters, instead of pure λ-calculus. The named functions enable us to use recursion in an easier way than in the λ-calculus. In this paper we showed and compared three different implementation strategies for recursive data types by functions. The first strategy is an extension of the well known Church numerals. These Church numerals are treated in nearly all introductory texts about the λ-calculus. The second encoding is based on an idea originating from unpublished notes of Scott. Due to the lack of a good reference, this encoding is reinvented several times in history. The third encoding in a combination of the previous two known as the Perigot encoding.

The difference between these encodings is in the way they handle recursion. In the Church encoding the functions representing the data type contain a fold-like recursion pattern to process the list. In the Scott encoding the recursive manipulations are done by a recursive function that resembles the recursive functions in ordinary functional programming languages much closer.

By using some additional constructors we were able to implement instances of a type constructor class capturing the basic operations of lists or trees for an algebraic data type, an encoding based on Church numerals and an encoding of Scott numbers.

The comparison shows us that the functions producing the recursive branch in a constructor, like the tail of a list or a subtree of in a binary tree, are troublesome in the Church encoding. These operations become spine strict in the recursion. This undesirable strictness of the Church encoding ruins lazy evaluation and gives the selection operators an undesirable high complexity. The amount of work to be done is proportional to the size of the data structure instead of constant. This is caused by fold-based formulation of the selector functions. In the Scott encoding the selectors are simple non recursive λ-expressions, hence they do not have the strictness and complexity problems of the Church encoding.

The complexity problems of the Church representation are increased by the fact that the reduction of a recursive selector is not shared in graph reduction. A selector in the Church representation is a higher order function that needs the next manipulation as argument before it can be evaluated.

When the manipulation used is essentially a fold it is possible to optimise the functions implementing the data structure in the Church encoding to achieve the required complexity. For manipulations that are not a fold, the fold-like recursion pattern enforced by the Church encoding really hampers. Since many useful programs are not only executing folds over their recursive data structures, we consider the Church encoding of data structures harmful for the implementation purposes discussed in this paper.

The Perigot encoding contains both a Scott encoding and a Church encoding. Compared with the Church encoding it solves the complexity problems of selecting the recursive branch and it prevents the undesired strict evaluation. However, the Perigot encoding does not bring us the best of both worlds. The additional effort and space required to maintain both encodings spoils the potential benefits of the native fold-right recursion compared with Scott encoding.

Acknowledgments

Special thanks to Peter Achten from the Radboud University for useful feedback on draft versions of this work and stimulating discussions. Aaron Stump form the university of Iowa stimulated us to look again at the Parigot encoding. The feedback of anonymous referees helped us to improve the reading frame of this paper.

References

[1] M. Abadi, L. Cardelli, and G. D. Plotkin. Types for the scott numerals. Unpublished note, 1993. URL `http://lucacardelli.name/Papers/Notes/scott2.pdf`.

[2] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1985. ISBN 9780080933757.

[3] H. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013. ISBN 9780521766142.

[4] E. Barendsen. An unsolvable numeral system in lambda calculus. *J. Funct. Program.*, 1(3):367–372, 1991.

[5] A. Berarducci and C. Böhm. Automatic synthesis of typed Lambda-programs on term algebras. *Theoretical Computer Science*, 39 (820076097):135–154, 1985.

[6] A. Berarducci and C. Böhm. A self interpreter of Lambda-calculus having a normal form. In E. Börger, G. Jäger, H. Kleine Büning, S. Martini, and M. M. Richter, editors, *Computer Science Logic. 6th Workshop, CSL '92*, volume 702 of *LNCS*, pages 85–99. Springer, 1993. ISBN 978-3-540-56992-3.

[7] C. Böhm. The CUCH as a formal and description language. In T. Steel, editor, *Formal Languages Description Languages for Computer Programming*, pages 179–197. North-Holland, 1966.

[8] C. Böhm and W. Gross. Introduction to the CUCH. In E. Caianiello, editor, *Automata Theory*, pages 35–65, London, UK, 1966. Academic Press.

[9] C. Böhm, A. Piperno, and S. Guerrini. Lambda-definition of function(al)s by normal forms. In D. Sannella, editor, *ESOP*, volume 788 of *LNCS*, pages 135–149. Springer, 1994. .

[10] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pages 136–143. ACM, 1980.

[11] J. Crossley. Reminiscences of logicians. In J. Crossley, editor, *Algebra and Logic*, volume 450 of *Lecture Notes in Mathematics*, pages 1–62. Springer, 1975. ISBN 978-3-540-07152-5.

[12] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic, Volume II*. North-Holland, 1972.

[13] J. Fairbairn and U. of Cambridge. Computer Laboratory. *Design and Implementation of a Simple Typed Language Based on the Lambda-calculus*. Computer Laboratory Cambridge: Technical report. University of Cambridge, Computer Laboratory, 1985.

[14] H. Geuvers. The Church-Scott representation of inductive and coinductive data. Types 2014, Paris, Draft, 2014. URL `http://www.cs.ru.nl/~herman/PUBS/ChurchScottDataTypes.pdf`.

[15] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232. ACM, 1993.

[16] R. Hinze. Theoretical pearl church numerals, twice! *J. Funct. Program.*, 15(1):1–13, Jan. 2005. ISSN 0956-7968.

[17] P. Hudak, S. L. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. H. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, R. B. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson. Report on the programming language haskell, a non-strict, purely functional language. *SIGPLAN Notices*, 27(5):1–, 1992.

[18] J. Jansen, P. Koopman, and R. Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In H. Nilsson, editor, *Revised Selected Papers of the 7th TFP'06*, volume 7, pages 73–90, Nottingham, UK, 2006. Intellect Books.

[19] J. Jansen, P. Koopman, and R. Plasmeijer. From interpretation to compilation. In Z. Horváth, editor, *Proceedings of the 2nd CEFP'07*, volume 5161 of *LNCS*, pages 286–301, Cluj Napoca, Romania, 2008. Springer.

[20] J. M. Jansen. Programming in the λ-calculus: From Church to Scott and back. In P. Achten and P. Koopman, editors, *The Beauty of Functional Code*, volume 8106 of *LNCS*, pages 168–180. Springer, 2013. ISBN 978-3-642-40354-5.

[21] T. Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, 1987.

[22] W. Kluge. *Abstract computing machines: a lambda calculus perspective*. Texts in theoretical computer science. Springer, 2005. ISBN 3-540-21146-2.

[23] P. W. M. Koopman, M. C. J. D. V. Eekelen, and M. J. Plasmeijer. Operational machine specification in a functional programming language. *Software: Practice and Experience*, 25(5):463–499, 1995. ISSN 1097-024X.

[24] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[25] T. Æ. Mogensen. Efficient self-interpretations in lambda calculus. *J. Funct. Program.*, 2(3):345–363, 1992.

[26] M. Naylor and C. Runciman. The reduceron: Widening the von neumann bottleneck for graph reduction using an fpga. In O. Chitil, Z. Horváth, and V. Zsók, editors, *IFL*, volume 5083 of *LNCS*, pages 129–146. Springer, 2007. ISBN 978-3-540-85372-5.

[27] M. Parigot. Programming with proofs: A second-order type theory. In *Proc. ESOP '88*, LNCS 300, pages 145–159. Springer, 1988.

[28] M. Parigot. Recursive programming with proofs. *Theor. Comput. Sci. 94*, pages 335–336, 1992.

[29] S. L. Peyton Jones and J. Salkild. The spineless tagless g-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 184–201. ACM, 1989. ISBN 0-89791-328-0.

[30] R. Plasmeijer and M. van Eekelen. Clean language report (version 2.1). `http://clean.cs.ru.nl`, 2002.

[31] R. Plasmeijer, P. Achten, and P. Koopman. iTasks: executable specifications of interactive work flow systems for the web. In R. Hinze and N. Ramsey, editors, *Proceedings of the ICFP'07*, pages 141–152, Freiburg, Germany, 2007. ACM.

[32] J. Steensgaard-Madsen. Typed representation of objects by functions. *ACM Trans. Program. Lang. Syst.*, 11(1):67–89, Jan. 1989. ISSN 0164-0925.

[33] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248, Amsterdam, The Netherlands, The Netherlands, 1988. North-Holland Publishing Co.