Under consideration for publication in J. Functional Programming

FUNCTIONAL PEARL

Comprehensive Encoding of Data Types and Algorithms in the λ-Calculus

JAN MARTIN JANSEN
Faculty of Military Sciences,
Netherlands Defense Academy, Den Helder, the Netherlands
RINUS PLASMEIJER and PIETER KOOPMAN
Institute for Computing and Information Sciences (ICIS),
Radboud University Nijmegen, the Netherlands
(*e-mail:* jm.jansen.04@nlda.nl,{rinus, pieter}@cs.ru.nl)

Abstract

The λ-calculus is a well known basic universal programming language, but is not considered as a realistic option for expressing algorithms in a comprehensive way. In this paper we show that this bad qualification is mainly caused by the choice of the Church encoding for the representation of Algebraic Data Types. We show that, using a different encoding contributed to Scott, and with a little aid of a clever lay-out scheme, functional programs, like they are written in languages like Clean or Haskell, can be expressed using comprehensive and concise λ-expressions resembling their Haskell and Clean counterparts. For this purpose, we also use an alternative way to express recursion without the use of a fixed-point combinator. The resulting formalism not only allows for comprehensive and readable expression of algorithms, but also allows for an efficient implementation.

1 Introduction

Although the λ-calculus is considered to be the mother of all (functional) programming languages, programming in it is not considered to be very practical. Every course or textbook on λ-calculus (e.g. (Barendregt, 1984)) spends some time on showing how the well-known programming constructs can be represented in the λ-calculus. It commonly starts by explaining how to represent data types like natural numbers in the λ-calculus and how to define operations on them. In almost all cases the Church numerals are chosen as the way to encode them. The definition of Church numerals and operations on them shows that it is possible to use the λ-calculus for all kinds of computations and that it is indeed a universal programming language. The Church encoding can be generalized for the encoding of general Algebraic Data Types (see (Barendregt, 1997)). This encoding allows for a straightforward implementation of iterative (primitive recursive) or fold-like functions on data structures, but needs complex and inefficient constructions for the expression of general recursion. In this way one ends with an encoding that works but is also quite unreadable (and inefficient) in many cases.

It is less common knowledge that there are alternative encodings of numbers in the λ-calculus. In (Jansen *et al.*, 2006) we already introduced an alternative encoding for Algebraic Data Types and showed that this encoding allows for an efficient implementation of interpreters for functional languages with data types based on this encoding. While in our previous work the focus was on obtaining an efficient interpreter for an intermediate functional language, here we have an entirely different goal. We look at the λ-calculus from a programmers perspective and want to show that the use of this encoding also makes it possible to obtain comprehensible λ-expressions for the realization of data structures and algorithms.

Another issue to be dealt with when using the λ-calculus as a programming language is the representation of recursive functions. Because λ-expressions are nameless, they cannot refer to themselves, and a special construction is needed to express recursion. The standard way to do this is the use of a fixed point combinator. Here we show that we can express recursion without the use of fixed point combinators, with as only price a small change in the way recursive functions are called. A further gain of this representation of recursion is that it results in a more efficient implementation using fewer reduction steps than when using a fixed point combinator.

This paper is organized as follows: We start with describing the Scott encoding in Section 2. In Section 3 we sketch how recursion can be expressed without using a fixed-point combinator. In Section 4 we show how we can use the techniques from the previous sections to express complete programs as one λ-expression. We make a comparison of the Scott and Church encodings in Section 5 and end with some conclusions in Section 6.

2 Alternative Encoding of Algebraic Data Types

The encoding we use is relatively unknown, and independently (re)discovered by several authors (e.g. (Steensgaard-Madsen, 1989; Mogensen, 1994; Stump, 2008) and the first author), but originally contributed to Scott in an unpublished lecture which is cited in Curry, Hindley and Seldin ((Curry *et al.*, 1972), page 504) as: *Dana Scott, A system*

of functional abstraction. Lectures delivered at University of California, Berkeley, Cal., 1962/63. Photocopy of a preliminary version, issued by Stanford University, September 1963, furnished by author in 1968. [1] We will therefore call it the *Scott* encoding. The encoding results in a representation that is very close to algebraic data types as they are used in most functional programming languages. We illustrate this with some examples of well-known data types.

2.1 The Nature of Algebraic Data Types

Consider Algebraic Data Type (ADT) definitions in languages like Clean or Haskell such as tuples, booleans, temperature, maybe, natural (Peano) numbers, and lists:

```
data Boolean     = True | False
data Tuple a b   = Tuple a b
data Temperature = Fahrenheit Int | Celsius Int
data Maybe a     = Nothing | Just a
data Nat         = Zero | Suc Nat
data List t      = Nil  | Cons t (list t)
```

A type consists of one or more alternatives. Each alternative consist of a name, possibly followed by a number of arguments. Algebraic Data Types are used for several purposes:

- to make enumerations, like in `Boolean`;
- to package data, like in `Tuple`;
- to unite things of different kind in one type, like in `MayBe` and `Temperature`;
- to make recursive structures like in `Nat` and `List` (in fact to construct new types with an infinite number of elements).

The power of the ADT construction in modern functional programming languages is that one formalism can be used for all these purposes. Imperative formalisms like C and Java need several constructs (like enumeration types, records, pointers and inheritance) for achieving this. Algebraic Data Types also have a meaning in untyped and dynamically typed formalisms like Lisp. But in that case the packaging concept is the most important one. The packaging construct is needed for the assembly of composed results for functions and for the construction of arbitrary size data containers. Lisp uses the list as a kind of generic packaging construct.

If we analyse the construction of ADT's more carefully, we see that constructor names are used for two purposes. First, they are used to distinguish the different cases in a single type definition (like `True` and `False` in `Boolean` and `Fahrenheit` and `Celsius` in `Temperature`). Second, we need them for recognizing them as being part of a type and making type inferencing possible. Therefore, all constructor names must be different in a single functional program (module). For distinguishing the different cases in a function definition, pattern matching on constructor names is used.

In the next three sub sections we show how ADT's can be expressed as λ-expressions in a natural way, staying close to their original definitions.

[1] We would like to thank Matthew Naylor for pointing us to this reference.

2.2 Named λ-expressions

First, some remarks about the notation of λ-expressions. We will always give a λ-expression representing an ADT or a function a name:

True ≡ λa b . a

In this way it is possible to refer to this λ-expression. If in a λ-expression a name is in italics, then it refers to another λ-expression having this name. This is done for readability and for saving space. For example:

True (λf g . f g) (λf g . g f)

Should be read as:

(λa b . a) (λf g . f g) (λf g . g f)

By introducing an additional λ-abstraction and using the fact that (λtrue . true y z) x reduces to x y z, we can also write:

(λtrue . true (λf g . f g) (λf g . g f)) (λa b . a)

The last example shows a well known alternative way of introducing explicit names in λ-expressions (see also Section 4).

Named λ-expressions are only introduced for notational convenience. These definitions behave like macro definitions. The names are replaced by the corresponding body before any reduction is done. This implies that these definitions cannot be recursive.

2.3 Expressing Enumerations Types in the λ-calculus

The simplest example of such a type is Boolean. We already noted that we use pattern matching for recognizing the different cases (constructors). So we are actually looking for an alternative for pattern matching using λ-expressions. The simplest example of using a pattern match for booleans is the if-then-else construction:

ifte True a b = a
ifte False a b = b

But the same effect can easily be achieved by making True and False functions, selecting the left or right argument respectively and by making ifte the identity function. Therefore, the λ-calculus solution for this is straightforward:

True ≡ λa b . a
False ≡ λa b . b
ifte ≡ λt . t

This is also the standard (Church) encoding used for booleans in λ-calculus courses and text books. So far we learned nothing new yet!

2.4 Expressing a Simple Container Type in the λ-calculus

Tuple is the simplest example of a pure container type. If we group data into a container type, we also need constructions to get data out of the container (so-called projection functions). For Tuple this can be realized by pattern matching or by using the selection functions fst and snd. These functions can be defined in Haskell using pattern matching:

```
fst (Tuple a b) = a
snd (Tuple a b) = b
```

Containers can be expressed in the λ-calculus by using closures (partial applications). For `Tuple` the standard way to do this is:

```
Tuple  ≡ λa b f . f a b
```

A tuple is an function that takes 3 arguments. If we supply only two, we have a closure. This closure can take a third argument, which should be a 2 argument function. This function is then applied to the first two arguments. The third argument is therefore called a continuation (the function with which the computation continues). It is now easy to find out what the definitions of `fst` and `snd` should be:

```
fst    ≡ λt . t (λa b . a)
snd    ≡ λt . t (λa b . b)
```

If applied to a tuple, they apply the tuple to a two argument function, that selects either the first (`fst`) or second (`snd`) argument.

Again, this definition of tuples is the one that can be found in λ-calculus text books and courses. So again, we learned nothing new.

2.5 Expressing General Multi Case Types in the λ-calculus

It is now a simple step to come up with a solution for arbitrary ADT's. Just combine the two solutions from above. Let us look at the definition of the function `warm` that takes a `Temperature` as an argument:

```
warm :: Temperature → Boolean
warm (Fahrenheit f) = f > 90
warm (Celsius c)    = c > 30
```

We have to find encodings for `(Fahrenheit f)` and `(Celsius c)`. The first solution tells that we should make a λ-expression with 2 arguments that returns the first argument for `Fahrenheit` and the second argument for `Celsius`. The second solution tells that we should feed the argument of `Fahrenheit` or `Celsius` to a continuation function. Combining these two solutions we learn that `Fahrenheit` and `Celsius` should both have 3 arguments. The first one to be used for the closure and the second and third as continuation arguments. `Fahrenheit` should choose the first continuation argument and apply it to its first argument and `Celsius` should do the same with the second continuation argument. So their definitions now become:

```
Fahrenheit  ≡ λf a b . a f
Celsius     ≡ λc a b . b c
```

The definition of `warm` now becomes:

```
warm   ≡ λt . t (λf . f > 90) (λc . c > 30)
```

If we apply this strategy to the types `Nat` and `List` we obtain the following definitions for the constructors:

```
Zero ≡ λf g   . f
Suc  ≡ λn f g . g n
```

$Nil \equiv \lambda$f g . f
$Cons \equiv \lambda$x xs f g . g x xs

Note that in these definitions the fact that these data types are recursive is of no influence. Functions like predecessor, head and tail can now easily be defined:

$pred \equiv \lambda$n . n *undef* (λm . m)

$head \equiv \lambda$xs . xs *undef* (λx xs . x)
$tail \equiv \lambda$xs . xs *undef* (λx xs . xs)

pred and *tail* are here constant time functions, while in the Church encoding their definitions are linear in the size of n or xs (see Sec. 5). In partial functions like *hd*, *pred* and *tail* we use *undef* to indicate the part of the function that is not defined.

2.6 The General Case

In general the mapping of an ADT to λ-expressions is defined as follows. Given the following ADT definition in Haskell or Clean:

data type_name t_1 ... t_k = C_1 $t_{1,1}$ ... t_{1,n_1} | ... | C_m $t_{m,1}$ ... t_{m,n_m}

Then this type definition with m constructors can be mapped to m λ-expressions:

$C_1 \equiv \lambda v_{1,1}$ ... v_{1,n_1} f_1 ... f_m . f_1 $v_{1,1}$ ... v_{1,n_1}
...
$C_m \equiv \lambda v_{m,1}$ ... v_{m,n_m} f_1 ... f_m . f_m $v_{m,1}$ ... v_{m,n_m}

Consider the (multi-case) pattern-based function f in Haskell or Clean defined on this type:

f (C_1 $v_{1,1}$ ... v_{1,n_1}) = $body_1$
...
f (C_m $v_{m,1}$ ... v_{m,n_m}) = $body_m$

This function is converted to the following λ-expression using the Scott encoding of data types:

$f \equiv \lambda$x . x
($\lambda v_{1,1}$ ... v_{1,n_1} . $body_1$)
...
($\lambda v_{m,1}$ ... v_{m,n_m} . $body_m$)

This completes the description of our encoding of data types. In section 5 we compare the Scott representation with the widely used Church encoding of data types.

3 Defining Recursive functions

Now we know how to represent ADT's we can concentrate on functions. We already gave some examples of them above (ifte, fst, snd, head, tail, pred, warm). The more interesting examples are the recursive functions. The standard technique for defining a recursive function in the λ-calculus is with the use of a fixed point operator. Let us look for example at the addition operator for Peano numbers. In Haskell or Clean we express this by:

```
add Zero     m = m
add (Suc n) m = Suc (add n m)
```

Using the Scott encoding and recursion in the definition, this becomes:

$add_0 \equiv \lambda$n m . n m (λn . *Suc* (*add*$_0$ n m))

This definition is illegal because it uses a reference to the macro add_0 itself. With the use of the Y fixed point combinator to eliminate recursion this becomes:

$add_y \equiv Y$ (λadd n m . n m (λn . *Suc* (add n m)))
$Y \equiv \lambda$h . (λx . h (x x)) (λx . h (x x))

There is, however, another way to represent recursion. Instead of using a fixed point operator we can also give the recursive function itself as an argument (like this is done in the argument of Y):

add $\equiv \lambda$add n m . n m (λn . *Suc* (add add n m))

The price to pay is that each call of add should have add as an argument, as can been seen in the definition of *add*. The gain is that we do not need the fixed point operator anymore and that we can recognize recursive calls on the spot. This definition is also more efficient than the one with the fixed-point combinator, because it uses fewer reduction steps for evaluation. The following example shows how *add* can be used to add one to one:

(λadd . add add (*Suc Zero*) (*Suc Zero*)) *add*

3.1 Mutual Recursive functions

In case of mutual recursive functions, we have to add all mutual recursive functions as arguments for each function in the mutual recursion. An example to clarify this (we start with the Haskell definitions):

```
isOdd Zero     = False
isOdd (Suc n)  = isEven n
isEven Zero    = True
isEven (Suc n) = isOdd n
```

This can be represented by λ-expressions as:

isOdd $\equiv \lambda$isOdd isEven n . n *False* (λn . isEven isOdd isEven n)
isEven $\equiv \lambda$isOdd isEven n . n *True* (λn . isOdd isOdd isEven n)

All mutual recursive functions are now an argument of each of the recursive functions in the definition as well as in each applied occurrence.

4 Converting Clean and Haskell Programs to λ-calculus

We now have all ingredients ready for converting complete programs. The last step to be made is combining everything into a single λ-expression. For example, if we take the add 1 1 example from above, and substitute all macros, we obtain:

```
(λadd . add add ((λn f g.g n) (λf g.f)) ((λn f g.g n) (λf g.f)))
        (λadd n m . n m (λn . (λn f g.g n) (add add n m)))
```

Using ordinary β-reductions this reduces to a term equivalent to *Suc (Suc Zero)* that represents the desired value 2. As said before, we can introduce explicit names for `zero` and `suc` by abstracting out their definitions and obtain a maybe more comprehensive definition:

```
(λzero suc .
    (λadd  .
           add add (suc zero) (suc zero))
    (λadd n m . n m (λn . suc (add add n m)))
(λf g.f) (λn f g.g n)
```

Here we applied a kind of inverted λ-lifting (see Sec. 4.2). We have used some smart indentation to make the expression better readable. The main expression is indented most. Definitions are introduced by variable names before they are used. Their implementations are indented as much as the line where their names were introduced. Note the nesting in this definition: the definition of `add` is inside the scope of the variables `suc` and `zero`, because its definition depends on the definition of them. In this way also the macro reference *Suc* in the definition of `add` can be replaced by a variable `suc`.

As another example, the right hand side of the Haskell function:

```
main = isOdd (Suc (Suc (Suc Zero)))
```

can be written as:

```
(λisOdd isEven . isOdd isOdd isEven (Suc (Suc (Suc Zero)))) isOdd isEven
```

and after substituting all macro definitions and applying inverted λ-lifting:

```
(λtrue false zero suc .
            (λisOdd isEven .
                    isOdd isOdd isEven (suc (suc (suc zero))))
            (λisOdd isEven n . n false (λn . isEven isOdd isEven n))
            (λisOdd isEven n . n true  (λn . isOdd isOdd isEven n)))
(λa b.a) (λa b.b) (λf g.f) (λn f g.g n)
```

The conversion yields small λ-terms where the original functional version of the expression is easily recognizable.

4.1 Some Remarks on the Evaluation of Expressions

We use normal order reduction for the λ-expressions to achieve lazy evaluation similar to lazy functional languages like Haskell and Clean. In order to obtain recognizable results we treat λ-expressions like:

$\lambda x_1\ x_2\ \ldots\ x_n\ .\ e$

not as an abbreviation of:

$\lambda x_1\ .\ (\lambda x_2\ .\ (\ldots\ \lambda x_n\ .\ e\ \ldots))$

as usually in the λ-calculus. In contrast we have a special reduction rule for each `n`:

$(\lambda x_1\ \ldots\ x_n\ .\ e)\ a_1\ \ldots\ a_n \equiv (\ldots(e\ [x_1 = a_1])\ldots)\ [x_n = a_n]$

That is, only if the λ-expression has all its arguments it is reduced as an ordinary λ-expression. Without the proper number of arguments no reduction steps are applied (exactly the reduction behavior of Clean or Haskell).

As a consequence, `(λn f g . g n) (λf g . f)`, representing *Suc Zero*, is not considered to be a redex and will therefore not be reduced to `λf g . g (λf g . f)`.

4.2 Formalizing Inverted λ-lifting

Above we mentioned the operation of inverted λ-lifting. Here we make more precise what we mean with this. The expression of a functional program from Clean or Haskell as a λ-expression proceeds in a number of steps:

1. Remove all syntactic sugar (list notation, zf-expressions, where and let expressions, etc.).
2. Eliminate all algebraic data type definitions by converting them to functions using the Scott encoding.
3. Convert pattern-based function definitions to normal functions using the Scott encoding of algebraic data types (see Sect. 2.6).
4. Remove (mutual) recursion by the introduction of extra variables (as explained in Sec. 3).
5. Make a dependency sort of all functions, resulting in an ordered collection of sets (strongly connected components). So the first set contains the functions that do not depend on other functions (e.g. the Scott encoded ADT's). The second set contains the functions that only depend on the functions in the first set, etc. Hereby, a group of mutual recursive functions is treated as a single function and thus all functions in it must belong to the same dependency set. Note that we can do this because all possible cycles are already removed in the previous step.
6. Construct the resulting λ-expression by nesting the definitions from the different dependency sets. The outermost expression consists of an application of a λ-expression with as variables the names of the functions from the first dependency set and as arguments the λ-definitions of these functions. The body of this expression is obtained by repeating this procedure for the remainder dependency sets. The innermost expression is the main expression.

The result of this process is:

```
(λfunction_names_first_set .
  (λfunction_names_second_set .
    ...
      (λfunction_names_last_set .
        main_expression)
      function_definitions_last_set)
    ...
  function_definitions_second_set)
function_definitions_first_set
```

4.3 A More Complex Example

As a last, more interesting example, consider the following Haskell version of the Eratosthenes prime sieve program:

```
data Nat                 = Zero | Suc Nat
data Inflist t           = Cons t (Inflist t)
nats n                   = Cons n (nats (Suc n))
sieve (Cons Zero xs)     = sieve xs
sieve (Cons (Suc k) xs)  = Cons (Suc k) (sieve (rem k k xs))
rem p Zero     (Cons x xs) = Cons Zero (rem p p xs))
rem p (Suc k) (Cons x xs) = Cons x     (rem p k xs)

main = sieve (nats (Suc (Suc Zero)))
```

Here we use infinite lists for the storage of numbers and the resulting primes. `sieve` filters out the `zero`'s in a list and calls `rem` to set multiples of prime numbers to `zero`. Applying the first four steps of the conversion procedure results in:

```
Zero  ≡ λf g     . f
Suc   ≡ λn f g   . g n
Cons  ≡ λx xs g . g x xs
nats  ≡ λnats n . Cons n (nats nats (Suc n))
sieve ≡ λsieve ls . ls (λx xs . x (sieve sieve xs)
                                  (λk . Cons x (sieve sieve (rem rem k k xs))))
rem   ≡ λrem p k ls . ls (λx xs . k (Cons Zero (rem rem p p xs))
                                    (λk . Cons x (rem rem p k xs)))
main ≡ sieve sieve (nats nats (Suc (Suc Zero)))
```

The dependency sort results in:

```
[{zero,suc,cons},{rem,nats},{sieve},{main}]
```

Putting everything together in a single λ-expression yields:

```
(λzero suc cons .
 (λrem nats     .
  (λsieve       .
    sieve sieve (nats nats (suc (suc zero))))
  sieve)
 rem   nats)
Zero Suc Cons
```

And after substituting the λ-definitions for all macros:

```
(λzero suc cons .
 (λrem nats     .
  (λsieve       .
    sieve sieve (nats nats (suc (suc zero))))
  (λsieve ls  . ls (λx xs . x (sieve sieve xs)
                              (λk . cons x (sieve sieve (rem rem k k xs))))))
 (λrem p k ls . ls (λx xs . k (cons zero (rem rem p p xs))
                              (λk . cons x (rem rem p k xs))))
 (λnats n . cons n (nats nats (suc n))))
(λf g . f) (λn f g . g n) (λx xs g . g x xs)
```

Which is probably the most compact, completely self-contained, definition of a prime number generator. Even shorter (143 characters) using one letter identifiers:

$$(\lambda zsc.(\lambda rf.(\lambda e.ee(ff(s(sz))))(\lambda el.l\lambda ht.h(eet)\lambda k.ch(ee(rrkkt))))$$
$$(\lambda rpkl.l\lambda ht.k(cz(rrppt))\lambda k.ch(rrpkt))(\lambda fn.cn(ff(sn))))(\lambda fg.f)(\lambda nfg.gn)(\lambda htg.ght)$$

5 Comparison of the Church and Scott encoding

We already indicated that the Church and Scott encoding overlap for simple enumerations and simple (non-recursive) packaging types. They only differ for recursive types. Let us have a look at the Church definition of natural numbers:

$Zero_c \equiv \lambda \texttt{f x . x}$
$Suc_c \equiv \lambda \texttt{n f x . f (n f x)}$

As a reminder, above we had for the Scott encoding:

$Zero_s \equiv \lambda \texttt{f g . f}$
$Suc_s \equiv \lambda \texttt{n f g . g n}$

The functions $Zero_c$ and $Zero_s$ are both selection functions, but the definition of Suc_c is completely different from Suc_s. Instead of feeding only n to the continuation function `f` the result of of `n f x` is fed to the continuation function `f`. This is exactly the same thing as what happens in the `fold` function. Using the Scott encoding for natural numbers `fold` can be defined as (the recursion can be removed with the technique used earlier):

$foldNat \equiv \lambda \texttt{f z n . n z (}\lambda \texttt{n . f (}foldNat\ \texttt{f z n))}$

In (Hinze, 2005) Hinze states that Church numerals are actually folds in disguise. As a consequence only primitive recursive functions on numbers can be easily expressed using the Church encoding. An example of such a function is addition:

$add_c \equiv \lambda \texttt{n m . n}\ Suc_c\ \texttt{m}$

Which is comparable to the following Scott version using `foldNat`:

$add_s \equiv \lambda \texttt{n m .}\ foldNat\ Suc_s\ \texttt{n m}$

For functions that need general recursion (or functions for which the result for `suc n` cannot be expressed using the result for `n`) we run into troubles. Church himself was not able to solve this problem but Kleene found a way out (as described in (Barendregt, 1997)). A nice example of his solution is the predecessor function, which can be easily expressed using the Scott encoding, as we saw earlier:

$pred_s \equiv \lambda \texttt{n . n}\ undef\ (\lambda \texttt{m . m})$

To define it using the Church encoding Kleene used a construction with pairs.

$pred_c \equiv \lambda \texttt{n .}\ snd(\texttt{n (}\lambda \texttt{p .}\ pair\ (Suc_c\ (fst\ \texttt{p}))\ (fst\ \texttt{p}))\ (pair\ Zero_c\ Zero_c))$

Each pair combines the result of the recursive call with the previous element. A disadvantage of this solution, besides that it is hard to comprehend, is that $pred_c$ `n` has complexity *O(n)* while that of $pred_s$ `n` is *O(1)*. From a programmers point of view this is a serious drawback.

It is straightforward to convert Church and Scott encoded numbers into each other:

$toChurch \equiv \lambda \texttt{n f x .}\ foldNat\ \texttt{f x n}$
$toScott \equiv \lambda \texttt{n . n}\ Suc_s\ Zero_s$

This again, shows that the difference between the Church and Scott encoding is a fold!

5.1 Comparing the Scott and Church encoding for lists

The Church encoding for lists together with the functions `sum` and `tail` are given by:

$Nil_c \equiv \lambda$`f x . x`
$Cons_c \equiv \lambda$`h t f x . f h (t f x)`

$sum_c \equiv \lambda$`xs . xs` add_c $Zero_c$
$tail_c \equiv \lambda$`xs .` *snd* (`xs (`λ`x rs .` *pair* ($Cons_c$ `x` (*fst* `rs`)) (*fst* `rs`)) (*pair* Nil_c Nil_c))

Also here the definition of `cons` behaves like a fold (a `foldr` actually). Again, we need the pair construction for the non-primitive recursive function `tail`. The Scott version of `foldr` for lists and its application in the `sum` function are:

foldList $\equiv \lambda$`f d xs . xs d (`λ`h t . f h (`*foldList* `f d t`))
$sum_s \equiv \lambda$`xs .` *foldList* add_s $Zero_s$ `xs`

The conversions between the Church and Scott encoding for lists are given by:

toChurchList $\equiv \lambda$`xs f d .` *foldList* `f d xs`
toScottList $\equiv \lambda$`xs . xs` $Cons_s$ Nil_s

Note that these definitions are completely equivalent to those for the conversion of numbers. They only use a different fold function in *toChurchList* and different constructors in *toScottList*.

5.2 Discussion

We already indicated that the Scott encoding just combines the techniques used for encoding booleans and tuples in the Church encoding as described in standard λ-calculus text books and courses. The Scott and Church encodings only differ for recursive types. A Church encoded type just defines how functions should be folded over an element of the type. A fold can be characterized as a function that replaces constructors by functions. The Scott encoding just packages information into a closure. Recursiveness of the type is not visible at this level. Of course, this is also the case for ADT's in functional languages, where recursiveness is only visible at the type level and not at the element level.

The representation achieved using the Scott encoding is equivalent to that of ADT definitions in modern functional programming languages and allows for an similar realization of functions defined on ADT's. Also the complexity (efficiency) of these functions is similar to their equivalents in functional programming languages. This in contrast to their counterparts using the Church encoding that sometimes have a much worse complexity. Therefore, from a programmers perspective the Scott encoding is better than the Church encoding.

A disadvantage of the Scott encoding of ADT's is that the resulting functions cannot be typed using standard HM type systems, while Church encoded ADT's can be neatly typed. The encoding of recursive functions in combination with the absence of ordinary combinators is too complicated for the standard HM type systems. Therefore, from a mathematicians point of view the Church encoding is better, also because Scott and Church encoded type elements can be easily converted into each other.

An interesting question now is: Why did it took so long before the Scott encoding was discovered and why is this encoding still relatively unknown? The encoding is simpler

than the Church encoding and allows for a straightforward implementation of functions acting on data types. Of course, the way ADT's are represented in modern functional programming languages is rather new and dates from languages like ISWIN (Landin, 1966), HOPE (Burstall *et al.*, 1980) and SASL (Turner, 1979) and this was long after the Church numerals were invented. Furthermore, ADT's are needed and defined by computer scientists, who needed an efficient way to define new types, which is rather irrelevant for mathematicians who are less concerned with an efficient implementation of algorithms.

In (Jansen *et al.*, 2006) we showed that this representation of functional programs can be used to construct very efficient, simple and small interpreters for lazy functional programming languages. These interpreters only have to implement β-reduction and no constructors nor pattern matching.

Altogether, we argue that the Scott encoding also should have its place in λ-calculus textbooks and courses.

6 Conclusions

In this paper we showed how the λ-calculus can be used to express algorithms and Algebraic Data Types in a way that is close to the way this is done in functional programming languages. To achieve this, we used a rather unfamiliar encoding of ADT's contributed to Scott. We showed that this encoding can be considered as a logical combination of the way how enumerations (like booleans) and containers (like tuples) are normally encoded in the λ-calculus. The encoding differs from the Church encoding and the connecting element between them is the fold function.

For recursive functions we did not use the standard fixed-point combinators, but instead used a simple technique where an expression representing a recursive function is given (a reference to) itself as an argument. In this way the recursion is made more explicit and this also results in a more efficient implementation using fewer reduction steps.

We also sketched a systematic method for writing Haskell or Clean like programs in the λ-calculus.

Altogether we have shown that it is possible to express a functional program in a concise way as a λ-expression that is clearer than the standard Church representation of the functional program.

References

Barendregt, Henk. (1984). *The lambda calculus, its syntax and semantics (revised edition)*. Studies in Logic, vol. 103. North-Holland.

Barendregt, H.P. (1997). The impact of the lambda calculus in logic and computer science. *The bulletin of symbolic logic*, **3**(2), 181–215.

Burstall, R. M., MacQueen, D. B., & Sannella, D. T. (1980). *Hope: An experimental applicative language*.

Curry, H., Hindley, J., & Seldin, J. (1972). *Combinatory logic*. Vol. 2. North-Holland Publishing Company.

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

Jansen, Jan Martin, Koopman, Pieter, & Plasmeijer, Rinus. (2006). Efficient interpretation by transforming data types and patterns to functions. *Pages 73–90 of:* Nilsson, Henrik (ed), *Revised selected papers of the 7th Symposium on Trends in Functional Programming, TFP '06*, vol. 7. Nottingham, UK: Intellect Books.

Landin, P. J. (1966). The next 700 programming languages. *Commun. acm*, **9**(3), 157–166.

Mogensen, Torben Ae. (1994). Efficient Self-Interpretation in Lambda Calculus. *Journal of functional programming*, **2**, 345–364.

Steensgaard-Madsen, J. (1989). Typed representation of Objects by Functions. *ACM transactions on programming languages and systems*, **11**(1), 67–89.

Stump, Aaron. (2008). Directly reflective meta-programming. *Journal of higher order and symbolic computation*.

Turner, D. A. (1979). A new implementation technique for applicative languages. *Softw., pract. exper.*, **9**(1), 31–49.