Chapter 1

Efficient Interpretation by Transforming Data Types and Patterns to Functions

Jan Martin Jansen[1], Pieter Koopman[2], Rinus Plasmeijer[2]

Abstract: This paper describes an efficient interpreter for lazy functional languages like Haskell and Clean. The interpreter is based on the elimination of algebraic data types and pattern-based function definitions by mapping them to functions using a new efficient variant of the Church encoding. The transformation is simple and yields concise code. We illustrate the concepts by showing how to map Haskell and Clean programs to the intermediate language **SAPL** (**S**imple **A**pplication **P**rogramming **L**anguage) consisting of pure functions only.

An interpreter is described for SAPL, based on straightforward graph reduction techniques. This interpreter can be kept small and elegant because function application is the only operation in SAPL. The application of a few easy to realize optimisations turns this interpreter into an efficient one. The resulting performance turns out to be competitive in a comparison with other interpreters like Hugs, Helium, GHCi and Amanda for a large number of benchmarks.

1.1 INTRODUCTION

In this paper we present an implementation technique for lazy functional languages like Haskell [1] and Clean [16] based on the representation of data types by functions. Although it is well known that it is possible to represent algebraic data types as functions by using the Church encoding or variants of it (Berarducci and Bohm ([6] and [7]) and Barendregt [5]), these representations have never

[1]Netherlands Defence Academy, Faculty of Military Sciences, Den Helder, the Netherlands; E-mail: `j.m.jansen@forcevision.nl`

[2]Institute for Computing and Information Sciences (ICIS), Radboud University Nijmegen, the Netherlands; E-mail: {`pieter,rinus`}`@cs.ru.nl`

been used in implementations for efficiency reasons. Therefore, intermediate languages always contain special constructs for data types and pattern matching (see e.g. Peyton Jones [12] and Kluge [10]). In this paper we present a new variant of the Church encoding for algebraic data types. This variant uses named functions and explicit recursion instead of lambda expressions for the conversion. We show how to convert a pattern-based function definition to a single function without patterns using this encoding. The encoding results in a program in the intermediate language SAPL consisting of pure functions only. The encoding we use has important advantages over the Church encoding because it allows for destructor functions with complexity $O(1)$, instead of proportional to the size of the data structure (list, tree, etc.).

In the second half of this paper an interpreter is described that can handle the functions that are the result of this transformation. The interpreter is based on straightforward graph reduction techniques. To optimise the performance of the interpreter two types of function annotations are introduced. The first annotation enables an optimal instantiation of function bodies that are the result of translating pattern-based function definitions, and the second annotation enables the inline execution of certain local function definitions. The annotations can easily be added during the translation of a Haskell or Clean program to SAPL. It is also possible to add them during a static analysis of the translated programs without knowledge of the original data types and pattern definitions.

Summarizing, the contributions of this paper are:

- We introduce a new encoding scheme that transforms algebraic data types to simple function definitions in the intermediate language SAPL. The encoding uses named functions and explicit recursion which simplify the encoding considerably in comparison with known encodings.
- We show how to transform a pattern-based function definition to a single function without patterns using this encoding.
- We describe how an efficient interpreter can be realized for lazy functional programming languages using minimal and elementary effort. The interpreter takes as input the result of the transformation mentioned above. The implementation of the interpreter is considerably shorter than that of byte code based interpreters like Helium, Hugs and GHCi with a better performance. The better performance of the interpreter can be attributed to the simplicity of the intermediate formalism enabling a high-level abstract machine having large atomic actions with minimal interpretation overhead.

The structure of this paper is as follows. In Section 1.2 we introduce a new encoding of algebraic data types by functions and we compare this encoding with two existing encodings. In Section 1.3 we introduce the intermediate functional programming language **SAPL**. SAPL has, besides integers and their operations, no data types. SAPL is similar to the pure functional kernel of languages like Haskell and Clean. We show how to transform complex pattern-based function definitions to SAPL based on the representation of data types from Section 1.2.

In Section 1.4 we define an interpreter for this language based on straightforward graph-rewriting techniques. We show how the interpreter can be optimised by using two simple annotations that can be added to SAPL programs. The performance of the optimised interpreter is compared with other implementations in Section 1.5. In Section 1.6 we give some conclusions and discuss further research possibilities.

1.2 REPRESENTATION OF DATA TYPES BY FUNCTIONS

In the lambda calculus several representations of algebraic data types by functions (or lambda expressions) exist. In this section we introduce a new representation and compare it with the two most important existing representations. We use two examples to demonstrate the differences: the Peano representation of natural numbers with the addition and predecessor operations and lists with the length and tail operations. We use Haskell syntax for all definitions, although some functions cannot be typed.

1.2.1 A New Representation of Data Types by Functions

Consider the following algebraic data type definition in Haskell or Clean:

$$typename\ t_1\ ..\ t_k\ ::=\ C_1\ t_{1,1}\ ..\ t_{1,n_1}\ |\ ..\ |\ C_m\ t_{m,1}\ ..\ t_{m,n_m}$$

We map this type definition with *m* constructors to *m* functions:

$$\begin{array}{l} C_1\ v_{1,1}\ ..\ v_{1,n_1} = \lambda\, f_1\ ..\ f_m\ \rightarrow\ f_1\ v_{1,1}\ ..\ v_{1,n_1} \\ .. \\ C_m\ v_{m,1}\ ..\ v_{m,n_m} = \lambda\, f_1\ ..\ f_m\ \rightarrow\ f_m\ v_{m,1}\ ..\ v_{m,n_m} \end{array}$$

Each constructor is represented by a function with the same name. Now consider the Haskell (multi-case) function *f* with as argument an element of this data type:

$$\begin{array}{l} f\ (C_1\ v_{1,1}\ ..\ v_{1,n_1}) = body_1 \\ .. \\ f\ (C_m\ v_{m,1}\ ..\ v_{m,n_m}) = body_m \end{array}$$

This function is converted to the following function without patterns:

$$\begin{array}{l} f\ el\ =\ el \\ \qquad\qquad (\lambda\, v_{1,1}\ ..\ v_{1,n_1}\ \rightarrow\ body_1) \\ \qquad\qquad .. \\ \qquad\qquad (\lambda\, v_{m,1}\ ..\ v_{m,n_m}\ \rightarrow\ body_m) \end{array}$$

The body of each case is turned into a lambda expression that is placed as an argument of the data type element. The actual data type argument will select the correct lambda expression and apply it to the arguments of the constructor. Therefore we call a function corresponding to a constructor a *selector* function. The result of the transformation of recursive functions on recursive data types cannot be typed by Hindley-Milner type inference (see examples in the next section). This is not a problem because the functions can be typed before the transformation.

1.2.2 Examples

The Haskell definitions for the examples are (note that we defined *tail Nil* as *Nil* and *pred Zero* as *Zero* in order to have total functions):

$$\begin{aligned}
&\mathit{data\ Nat} &&= \mathit{Zero} \mid \mathit{Suc\ Nat}\\
&\mathit{add\ n\ Zero} &&= n\\
&\mathit{add\ n}\ (\mathit{Suc\ m}) &&= \mathit{Suc}\ (\mathit{add\ n\ m})\\
&\mathit{pred\ Zero} &&= \mathit{Zero}\\
&\mathit{pred}\ (\mathit{Suc\ n}) &&= n\\
\\
&\mathit{data\ List\ t} &&= \mathit{Nil} \mid \mathit{Cons\ t}\ (\mathit{List\ t})\\
&\mathit{length\ Nil} &&= 0\\
&\mathit{length}\ (\mathit{Cons\ x\ xs}) &&= 1 + \mathit{length\ xs}\\
&\mathit{tail\ Nil} &&= \mathit{Nil}\\
&\mathit{tail}\ (\mathit{Cons\ x\ xs}) &&= \mathit{xs}
\end{aligned}$$

Using the transformation to functions this becomes:

$$\begin{aligned}
&\mathit{Zero} &&= \lambda f\ g \rightarrow f\\
&\mathit{Suc\ n} &&= \lambda f\ g \rightarrow g\ n\\
&\mathit{add\ n\ m} &&= m\ n\ (\lambda\, \mathit{pm} \rightarrow \mathit{Suc}\ (\mathit{add\ n\ pm}))\\
&\mathit{pred\ n} &&= n\ \mathit{Zero}\ (\lambda\, \mathit{pn} \rightarrow n)\\
\\
&\mathit{Nil} &&= \lambda f\ g \rightarrow f\\
&\mathit{Cons\ x\ xs} &&= \lambda f\ g \rightarrow g\ x\ \mathit{xs}\\
&\mathit{length\ ys} &&= \mathit{ys}\ 0\ (\lambda\, x\ \mathit{xs} \rightarrow 1 + \mathit{length\ xs})\\
&\mathit{tail\ ys} &&= \mathit{ys\ Nil}\ (\lambda\, x\ \mathit{xs} \rightarrow \mathit{xs})
\end{aligned}$$

pred and *tail* both have complexity *O(1)*. The functions *Zero*, *Suc*, *Nil*, *Cons*, *pred* and *tail* can be typed, but *add* and *length* cannot be typed using Hindley-Milner type inference. In general, the encoding of recursive functions on recursive data types cannot be typed. The definitions of *add* and *length* are explicitly recursive. In general, to encode recursive functions over recursive data structures, we need explicit recursion. This is not a problem since we use named functions instead of lambda expressions in our encoding. The notation is easy to read and close to the original Haskell data type and function definitions.

1.2.3 Church Encoding

For this encoding we need pairs with the selection functions *fst* and *snd*. They can be represented by functions as follows:

$$\begin{aligned}
&\mathit{pair\ x\ y} &&= \lambda f \rightarrow f\ x\ y\\
&\mathit{fst\ p} &&= p\ (\lambda\, x\ y \rightarrow x)\\
&\mathit{snd\ p} &&= p\ (\lambda\, x\ y \rightarrow y)
\end{aligned}$$

The Church encoding is a generalization of the Church numerals. The representation described here is based on Berarducci and Bohm [6] and Barendregt [5]. For comparison reasons we use a slightly different notation than is generally used for describing Church numerals:

$$\begin{array}{lcl}
Zero & = & \lambda\, f\ g\ \rightarrow\ f \\
Suc\ n & = & \lambda\, f\ g\ \rightarrow\ g\ (n\ f\ g) \\
add\ n\ m & = & m\ n\ (\lambda\, rpm\ \rightarrow\ Suc\ rpm) \\
pred\ n & = & snd\ (n\ (pair\ Zero\ Zero)\ (\lambda\, p\ \rightarrow\ pair\ (Suc\ (fst\ p))\ (fst\ p)))
\end{array}$$

$$\begin{array}{lcl}
Nil & = & \lambda\, f\ g\ \rightarrow\ f \\
Cons\ x\ xs & = & \lambda\, f\ g\ \rightarrow\ g\ x\ (xs\ f\ g) \\
length\ ys & = & ys\ 0\ (\lambda\, x\ rxs\ \rightarrow\ 1\ +\ rxs) \\
tail\ xs & = & snd\ \ (xs\ \ (pair\ Nil\ Nil) \\
 & & \quad\quad (\lambda\, x\ pxs\ \rightarrow\ pair\ (Cons\ x\ (fst\ pxs))\ (fst\ pxs)))
\end{array}$$

In the *add* definition $add\ n\ (Suc\ m)$ can be defined using the result of $add\ n\ m$ (represented by *rpm*). The same holds for *length*. But in predecessor *pred (Suc n)* cannot be expressed in terms of *pred n*. Instead we need access to *n* in *Suc n* (we need to destruct *Suc n*). Kleene ([4]) found a way to overcome this by the use of pairs. In such a pair *n* is combined with the result of the recursive call, so access to *n* is also possible. For *tail* we also need this pair construction. Through this construction *pred n* has complexity *O(n)* and *tail xs* has complexity *O(length xs)*. In this encoding the recursion is put into the data structures. Therefore, functions on data structures do not have to be recursive themselves. A disadvantage is that this encoding only works fine for iterative and primitive recursive functions (see [7]). For destructor functions we need the pair construction. In the Church encoding data types and functions acting on them can be typed using Hindley-Milner type inference.

1.2.4 Representation according to Berarducci and Bohm

Another representation is described in Berarducci and Bohm [7] and Barendregt [5]. Again we adapted the notation to make a comparison with the other representations possible.

$$\begin{array}{lcl}
Zero & = & \lambda\, f\ g\ \rightarrow\ f\ f\ g \\
Suc\ n & = & \lambda\, f\ g\ \rightarrow\ g\ n\ f\ g \\
add\ n\ m & = & m\ (\lambda\, fz\ fs\ \rightarrow\ n)\ (\lambda\, pm\ fz\ fs\ \rightarrow\ Suc\ (pm\ fz\ fs)) \\
pred\ n & = & n\ (\lambda\, fz\ fs\ \rightarrow\ Zero)\ (\lambda\, pn\ fz\ fs\ \rightarrow\ pn)
\end{array}$$

$$\begin{array}{lcl}
Nil & = & \lambda\, f\ g\ \rightarrow\ f\ f\ g \\
Cons\ x\ xs & = & \lambda\, f\ g\ \rightarrow\ g\ x\ xs\ f\ g \\
length\ ys & = & ys\ (\lambda\, fn\ fc\ \rightarrow\ 0)\ (\lambda\, x\ xs\ fn\ fc\ \rightarrow\ 1\ +\ xs\ fn\ fc) \\
tail\ ys & = & ys\ (\lambda\, fn\ fc\ \rightarrow\ Nil)\ (\lambda\, x\ xs\ fn\ fc\ \rightarrow\ xs)
\end{array}$$

The basic idea in this representation is that the functions handling the different cases are propagated by the functions representing the data structures. Therefore, functions on data structures do not have to be recursive themselves. Here *pred n* and *tail xs* have complexity *O(1)*. In general, destructor functions have complexity $O(1)$, making this representation more powerful than the Church encoding. In this representation *Zero*, *Suc*, *Nil* and *Cons*, as well as the functions acting on them cannot be typed by Hindley-Milner type inference.

1.2.5 Conclusions

Our representation is more efficient than the Church encoding, because it realizes destructor functions with $O(1)$. Although this also holds for the representation of Berarducci and Bohm, the use of named functions and explicit recursion in our representation result in a simpler representation, which is suitable for an efficient implementation (see Section 1.4).

1.3 SAPL: AN INTERMEDIATE FUNCTIONAL LANGUAGE

SAPL is an intermediate language that can be used for the compilation and interpretation of functional programming languages like Haskell and Clean. The main difference between SAPL and the intermediate formalisms normally used is the absence of algebraic data types and constructs for pattern matching in SAPL. This makes SAPL a compact and simple language. In Section 1.4 we show that it is possible to make an efficient implementation for SAPL. SAPL is described by the following syntax:

$$\begin{array}{lll}
\mathit{function} & ::= & \mathit{identifier}\ \{\mathit{identifier}\}*\ {}'{=}'\ \mathit{expr} \\
\mathit{expr} & ::= & \mathit{application} \mid {}'\lambda'\ \{\mathit{identifier}\}+\ {}'{\rightarrow}'\ \mathit{expr} \\
\mathit{application} & ::= & \mathit{factor}\ \{\mathit{factor}\}* \\
\mathit{factor} & ::= & \mathit{identifier} \mid \mathit{integer} \mid {}'('\ \mathit{expr}\ ')'
\end{array}$$

A function has a name followed by zero or more variable names. An expression is either an application or a lambda expression. In an expression only variable names, integers and other function names may occur. SAPL function definitions start in the first column and can extend over several lines (as long as these are indented). SAPL is un-typed. The language has the usual lazy rewrite semantics (see Section 1.4). For efficiency we added integers and their basic operations to the language. In SAPL it is common that a curried application of a function is the result of a computation. This result will be presented as the application of the function name to the evaluated arguments.

SAPL's main difference with the lambda calculus is the use of explicitly named functions (enabling explicit recursion) which makes SAPL usable as a basic functional programming language and suitable for an efficient implementation.

For the use of SAPL as an intermediate language for implementing lazy functional languages like Haskell and Clean we must translate constructs from these languages to SAPL functions. Constructions like list-comprehensions, *where* and *let(rec)* expressions can be converted to functions with standard techniques as described in [12] and [14]. Algebraic data types and simple pattern-based functions are treated specially using the translation scheme from Section 1.2. In the next subsection the transformation of complex pattern-based functions is sketched.

1.3.1 Compiling Complex Pattern Definitions to Functions

In the implementations of Haskell and Clean pattern-based definitions are traditionally compiled to dedicated structures in a special pattern formalism that can be used to generate pattern-matching code (Augustsson [3] and Peyton Jones [12]). Here we transform a pattern-based function definition from Clean or Haskell to a single SAPL function without patterns. This function is capable of handling an actual call for the original pattern-based function. The conversion to a single function can be obtained using techniques similar to those used for the generation of pattern-matching code (see [3] and [12]). We use three examples to illustrate this conversion: *mappair* (*zipWith*), *samelength* and *complex*. Note that the pattern compiler introduces a name for every constructor (e.g. *as* in *mappair*) and uses existing names whenever possible (e.g. *ps* and *qs* in *samelength*).

$$\begin{array}{lll}
\mathit{mappair}\ f\ \mathit{Nil} & \mathit{zs} & = \mathit{Nil} \\
\mathit{mappair}\ f\ (\mathit{Cons}\ x\ \mathit{xs}) & \mathit{Nil} & = \mathit{Nil} \\
\mathit{mappair}\ f\ (\mathit{Cons}\ x\ \mathit{xs}) & (\mathit{Cons}\ y\ \mathit{ys}) & = \mathit{Cons}\ (f\ x\ y)\ (\mathit{mappair}\ f\ \mathit{xs}\ \mathit{ys})
\end{array}$$

$$\begin{array}{lll}
\mathit{samelength}\ \mathit{Nil} & \mathit{Nil} & = \mathit{True} \\
\mathit{samelength}\ (\mathit{Cons}\ x\ \mathit{xs}) & (\mathit{Cons}\ y\ \mathit{ys}) & = \mathit{samelength}\ \mathit{xs}\ \mathit{ys} \\
\mathit{samelength}\ \mathit{ps} & \mathit{qs} & = \mathit{False}
\end{array}$$

$$\begin{array}{ll}
\mathit{complex}\ (\mathit{Cons}\ a\ (\mathit{Cons}\ b\ (\mathit{Cons}\ c\ \mathit{Nil}))) & = a + b + c \\
\mathit{complex}\ (\mathit{Cons}\ a\ (\mathit{Cons}\ b\ \mathit{Nil})) & = 2 * a + b \\
\mathit{complex}\ (\mathit{Cons}\ a\ \mathit{Nil}) & = 3 * a \\
\mathit{complex}\ \mathit{xs} & = 0
\end{array}$$

The translation to SAPL results in:

$$\begin{array}{l}
\mathit{mappair}\ f\ \mathit{as}\ \mathit{zs}\ =\ \mathit{as}\ \mathit{Nil}\ (\lambda\, x\ \mathit{xs}\ \rightarrow\ \mathit{zs}\ \mathit{Nil}\ (\lambda\, y\ \mathit{ys}\ \rightarrow \\
\qquad\qquad\qquad\qquad \mathit{Cons}\ (f\ x\ y)\ (\mathit{mappair}\ f\ \mathit{xs}\ \mathit{ys})))
\end{array}$$

$$\begin{array}{l}
\mathit{samelength}\ \mathit{ps}\ \mathit{qs}\ =\ \mathit{ps}\ (\mathit{qs}\ \mathit{True}\ (\lambda\, y\ \mathit{ys}\ \rightarrow\ \mathit{False})) \\
\qquad\qquad\qquad\qquad (\lambda\, x\ \mathit{xs}\ \rightarrow\ \mathit{qs}\ \mathit{False}\ (\lambda\, y\ \mathit{ys}\ \rightarrow \mathit{samelength}\ \mathit{xs}\ \mathit{ys}))
\end{array}$$

$$\begin{array}{l}
\mathit{complex}\ \mathit{xs}\ =\ \mathit{xs}\ 0\ (\lambda\ a\ p1\ \rightarrow\ p1\ (\mathit{mult}\ 3\ a)\ (\lambda\, b\ p2 \rightarrow \\
\qquad\qquad\quad p2\quad (\mathit{add}\ (\mathit{mult}\ 2\ a)\ b) \\
\qquad\qquad\qquad\quad (\lambda\, c\ p3 \rightarrow\ p3\ (\mathit{add}\ (\mathit{add}\ a\ b)\ c)\ (\lambda\, p4\ p5 \rightarrow 0))))
\end{array}$$

1.4 AN INTERPRETER FOR SAPL

The only operations in SAPL programs are function application and a number of (built-in) integer operations. Therefore an interpreter can be kept small and elegant. The interpreter is implemented in C and is based on straightforward graph reduction techniques as described in Peyton Jones [12], Plasmeijer and van Eekelen [14] and Kluge [10]. We assume that a pre-compiler has eliminated all algebraic data types and pattern definitions (as described earlier) and all let(rec)- and where- clauses and lifted all lambda expressions to the global level. The interpreter is only capable of executing function rewriting and the basic operations on integers. The most important features of the interpreter are:

- It uses 4 types of memory Cells. A Cell corresponds to a node in the syntax tree and is either an: Integer, (Binary) Application, Variable or Function Call. To keep memory management simple, all Cells have the same size. A type byte in the Cell distinguishes between the different types. Each Cell uses 12 bytes of memory.
- The memory heap consists only of Cells. The heap has a fixed size, definable at start-up. We use a mark and (implicit) sweep garbage collection. Cells are not recollected, but the dirty bit is inverted after every mark.
- It uses a single argument stack containing only references to Cells. The C (function) stack is used as the dump for keeping intermediate results when evaluating strict functions (numeric operations only) and for administration overhead during the marking phase of garbage collection.
- The state of the interpreter consists of the stack, the heap, the dump, an array of function definitions and a reference to the node to be evaluated next. In each state the next step to be taken depends on the type of the current node: either an application node or a function node.
- It reduces an expression to head-normal-form. The printing routine causes further reduction. This is only necessary for arguments of curried functions.

The interpreter is based on the following 'executable specification' (without integers and their operations):

data Expr $=$ *App Expr Expr* | *Func Int Int* | *Var Int*

The first *Int* in *Func Int Int* denotes the number of arguments of the function, the second *Int* the position of the function definition in the list of definitions. The *Int* in *Var Int* indicates the position on the stack where the argument can be found.

The interpreter consists of three functions:

```
instantiate (App l r) es = App    (instantiate l es) (instantiate r es)
instantiate (Var n) es   = es !! n
instantiate x es         = x
```

```
rebuild e []          = e
rebuild e (x : xs)    = rebuild(App e x) xs

eval ::  Expr → [Expr] → [Expr] → Expr
eval     (App l r) es fs = eval l (r : es) fs
eval     (Func na fn) es fs
 = if    length es ≥ na
   then  eval (instantiate (fs !! fn) es) (drop na es) fs
   else  rebuild (Func na fn) es
```

Here *es* represents the stack and *fs* the list of function body definitions. One of the benchmarks in Section 1.5 is a SAPL version of the interpreter (including integers and their operations), which is the translation to SAPL of the Haskell version of the interpreter (a meta-circular implementation for SAPL). The C versions (including integers and operations on them) of *eval* and *instantiate* are straightforward implementations of this specification and fit on less than one page.

1.4.1 Optimising the SAPL Interpreter

For data-type-free programs the interpreter from the previous subsection has a performance comparable to Helium, GHCi and Amanda. But for programs involving algebraic data types the performance is worse. The difference depends on the number of alternatives and the complexity of the data type definition and varies from 30% slower for programs involving only if-then-else constructs, to several hundreds of times slower for programs involving complex data types and pattern matching (see section 1.5). This is not surprising because a pattern definition is converted to one large function containing all different cases. Instantiation of such a function is therefore relatively expensive, particularly because only a small part of the body will actually be used in a call for the function.

For optimising the SAPL interpreter we used both general optimisation techniques, commonly used for implementing functional languages, as well as techniques that are more specific for the way SAPL handles data types and pattern definitions.

General Optimisations

We use a more efficient memory representation for function calls with one or two arguments. For these function applications *APP* nodes are removed. This reduces the size of the bodies of functions and consequently copying overhead.

In the interpreter curried function calls are rebuilt. This can be prevented by keeping a reference to the top node of the application. If the number of arguments for a function call can be computed at compile time, the top node of a curried call can be marked. In this way an attempt to reduce a curried call can even be prevented.

Applying these two optimisations results in an average speed-up of 60% (see section 1.5). This speed-up is high since many functions have only 1 or 2 arguments and because SAPL programs contain many curried functions (due to the representation of data types by functions).

Specific Optimisations

We applied two specific optimisations. The first one addresses the instantiation problem for functions that are the result of the translation of pattern-based function definitions. The second one optimises the use of lambda expressions in these functions. Although the speed-up realized by these optimisations is significant, the implementation of them requires only small changes in the interpreter.

Selective Instantiation of Function Bodies The body of a transformed pattern-based definition consists of the application of a so-called selector function (see Section 1.3) to a number of arguments consisting of anonymous local function definitions. The selector function will select one of these local function definitions and apply it to the arguments of the corresponding constructor. All other arguments of the selector function will be ignored. In the *mappair* example below we have tagged the applications of selector functions with the keyword *select*.

$$\begin{aligned}&\mathit{mappair}\ f\ \mathit{as}\ \mathit{zs}\ = \\ &\qquad\qquad \mathit{select}\ \mathit{as}\ \mathit{Nil}\ (\lambda\, x\, \mathit{xs}\ = \\ &\qquad\qquad \mathit{select}\ \mathit{zs}\ \mathit{Nil}\ (\lambda\, y\, \mathit{ys}\ =\ \mathit{Cons}\ (f\ x\ y)\ (\mathit{mappair}\ f\ \mathit{xs}\ \mathit{ys})))\end{aligned}$$

The interpreter uses the *select* (semantically equivalent to the identity function) tag to optimise the instantiation of the body of *mappair*. Instead of copying the entire body, at first only the selector function part is instantiated (*as*) and depending on the result (*Nil* or *Cons x xs*), the correct remainder is instantiated. This is similar to evaluating the condition of an *if* expression before we decide to build the *then* part or the *else* part (but not both). In fact, in SAPL *True* and *False* are also implemented as selector functions. The optimisation is applied recursively to the bodies of all local definitions.

The optimisation realised in this way is significant. Varying from 30% faster for programs involving only if-then-else constructs, to up to 500 times faster for programs involving complex data type definitions like interpreters etc.

We can add the *select* tag during the transformation of the pattern-based function definition to SAPL, but it is also possible to infer the application of selector functions by a compile time analysis of a SAPL program. Selector functions must be recognized and the propagation of arguments and results of functions that are selector functions must be inferred. In this way this optimisation is a generic one and can even be used for the efficient reduction of lambda expressions.

Inlining of Local Definitions As a last optimisation we again consider the bodies of transformed pattern-based definitions. They contain local function definitions corresponding to the different cases. Normally these definitions are lambda lifted to the global level. During this lifting extra arguments are added to the function, causing extra stack operations at run-time. These local functions can also be reduced in the context of the reduction of the surrounding function call. This means that the local function is called (reduced) while the arguments of the main function are still on the stack and that at the end all arguments together are cleared from the stack. This can only be done because the reduction to head-normal-form of the local function call is necessary for the reduction to head-normal-form of the original function call, which is indeed the case for these transformed pattern-based functions. This optimisation results in an extra speed-up of about 10 to 25% for programs involving transformed pattern-based functions (see section 1.5). The optimisation is implemented by replacing $\rightarrow$ by $=$ in the local definition as a signal for the interpreter not to lambda lift this local function (see example in 1.4.1).

Again this optimisation can be applied not only for local definitions in translated pattern-based functions, but for all local function calls that must be reduced to head-normal-form while reducing the surrounding function call. But the gain for SAPL programs will be higher than for applying this optimisation for other functional languages, because SAPL programs, due to the translation scheme for pattern-based functions, contain more local function definitions.

1.5 BENCHMARKS

In this section we present the results of several benchmark tests for SAPL and a comparison of SAPL with other implementations. We ran the benchmarks on a 2.66 Ghz Pentium 4 computer with 512Mb of memory under Windows XP. SAPL was implemented using the Microsoft Visual C++ compiler using the -O2 option. The benchmark programs we used for the comparison are:

1. **Prime Sieve** The prime number sieve program, calculating the 5000th prime number.
2. **Symbolic Primes** Symbolic prime number sieve using Peano numbers, calculating the 280th prime number.
3. **Interpreter** An interpreter for SAPL, as described in Section 1.4 (including integers). As an example we coded the prime number sieve for this interpreter and calculated the 100th prime number.
4. **Fibonacci** The (naive) Fibonacci function, calculating *fib 35*.
5. **Match** Nested pattern matching (5 levels deep) like the *complex* function from section 1.3.1, repeated 2000000 times.
6. **Hamming** The generation of the list of Hamming numbers (a cyclic definition) and taking the 1000th Hamming number, repeated 4000 times.
7. **Twice** A higher order function (*twice twice twice twice (add 1) 0*), repeated 400 times.
8. **Sorting** Tree Sort (6000 elements), Quick Sort (6000 elements), Merge Sort (40000 elements, merge sort is much faster) and Insertion Sort (6000 elements).

TABLE 1.1. SAPL with/without Selective Instantiation (Time in seconds)

	Pri	Sym	Inter	Fib	Match	Twi	Sort	Qns	Kns	Parse	Plog
With	11.4	6.0	2.2	11.6	14.7	11.0	1.0	10.5	4.0	8.0	0.2
Without	21.5	107.0	53.0	19.2	23.0	10.9	17.8	16.0	6.1	16.0	106.0

9. **Queens** Number of placements of 11 Queens on a 11 * 11 chess board.
10. **Knights** Finding a Knights tour on a 5 * 5 chess board.
11. **Parser Combinators** A parser for Prolog programs based on Parser Combinators parsing a 17000 lines Prolog program.
12. **Prolog** A small Prolog interpreter based on unification only (no arithmetic operations), calculating ancestors in a four generation family tree, repeated 500 times.

For sorting a list of size *n* we used a source list consisting of numbers 1 to *n*. The elements that are 0 modulo 10 are put before those that are 1 modulo 10, etc.

Three of the benchmarks (*Interpreter*, *Prolog* and *Parser Combinators*) are realistic programs, the others are typical benchmark programs that are often used for comparing implementations. They cover a wide range of aspects of functional programming (lists, laziness, deep recursion, higher order functions, cyclic definitions, pattern matching, heavy calculations, heavy memory usage). All times are machine measured. The programs where chosen in such a way that they ran for at least several seconds (interpreters only). Therefore start-up times can be neglected. The output was always converted to a single number (e.g. by summing the elements of a list) to eliminate the influence of slow output routines.

The input for the SAPL interpreter is code generated by an experimental data type and pattern compiler from sources equivalent to the Haskell and Clean programs (only minor syntactic differences). This compiler also generates the annotations needed for the optimisations. The *inline* optimisation is only applied for the lambda expressions that are the result of encoding a pattern- based definition. The benchmarks programs can be found in [17].

1.5.1 Optimisations for SAPL

In table 1.1 we first compare SAPL with and without the selective instantiation optimisation. In this comparison the other optimisation are not applied. *Hamming* is missing because the version of the interpreter without selective instantiation does not support cyclic definitions. We conclude that the selective instantiation optimisation is essential. Because SAPL also uses selective instantiation to optimise the if- then-else construct there is a speed-up for all benchmarks except *twice* (the only benchmarks without if-then-else and data structures). In the other examples the speed-up varies from around 1.5 times (*Primes, Fibonacci, Match, Queens, Knights*), around 20 times (*Symbolic Primes, Interpreter, Sorting*) to more than 500 times for *Prolog* (due to the complicated *unification* function).

Table 1.2 shows the results of applying the other optimisations.

TABLE 1.2. Comparison Versions of SAPL (Time in seconds)

	Pri	Sym	Inter	Fib	Match	Ham	Twi	Sort	Qns	Kns	Parse	Plog
Full	6.1	17.6	7.8	7.3	8.5	6.4	7.9	5.9	6.5	2.0	4.4	4.7
Select	11.4	37.6	14.3	11.6	14.7	11.3	11.0	9.4	10.6	4.0	8.0	10.4
Mem	6.2	28.0	9.3	7.5	9.0	8.0	7.9	6.4	7.0	2.7	4.9	6.7
Inline	11.4	24.4	12.9	11.5	14.4	9.2	11.0	8.7	10.0	3.3	7.5	7.8

TABLE 1.3. Different Memory Configurations (Time sec, Heap/Stack kB)

	Pri	Sym	Inter	Fib	Match	Ham	Twi	Sort	Qns	Kns	Parse	Plog
Heap	223	47	2350	12	101	105	785	2350	43	18	9700	150
Stack	270	35	1100	1	1	1	1	200	1	1	200	4
10.8 Mb												
Time	6.7	17.1	13.0	8.0	9.2	6.9	9.1	6.7	7.0	2.1	17.0	5.2
% GC	15	12	46	13	18	14	18	21	17	14	76	17
nr GC	87	204	150	117	157	100	120	83	114	32	190	83
24 Mb												
Time	6.4	17.5	8.8	7.8	9.1	6.7	8.8	6.5	7.0	2.1	6.0	5.1
% GC	13	10	24	13	18	15	15	15	14	14	38	16
nr GC	38	91	61	53	70	45	52	37	51	15	40	37
60 Mb												
Time	6.4	18.6	8.3	7.6	9.1	6.6	8.5	6.5	6.9	2.1	5.0	5.1
% GC	13	10	18	13	16	15	13	16	14	14	24	16
nr GC	15	36	24	28	28	18	21	15	21	6	14	15

- **Full** The fully optimised interpreter (Select, Mem and Inline).
- **Select** The interpreter using only the selective instantiation optimisation.
- **Mem** The interpreter using selective instantiation and the efficient representation of functions with 1 or 2 arguments.
- **Inline** The interpreter using selective instantiation and inlining of lambda expressions in encoded pattern-based functions.

From this comparison we learn that the fully optimised version is about 1.8 times faster than the version using only selective instantiation, 1.2 times faster than the version with selective instantiation and memory optimisation and 1.6 times faster than the version with selective instantiation and inlining. The benefit from the inline optimisation is modest, but the implementation of it in the run-time system consists of only moving a stack pop operation to another line. The more efficient memory representation gives a significant speed-up.

In table 1.3 we compare the behaviour of SAPL for a number of memory configurations: 10.8 Mb (90000 Cells), 24 Mb (2000000 Cells) and 60 Mb (5000000 Cells). 900000 Cells is the minimal heap size needed to run all benchmarks. We also give peak heap and stack usage in Kb and percentage of time spent in GC and number of GC. Because heap and stack usage are only measured at GC the actual maximum values can be (slightly) higher than those measured. For these

TABLE 1.4. Run-Times (in seconds) for different Implementations

	Pri	Sym	Inter	Fib	Match	Ham	Twi	Sort	Qns	Kns	Parse	Plog
SAPL	6.1	17.6	7.8	7.3	8.5	6.4	7.9	5.9	6.5	2.0	4.4	4.7
Helium	13,6	17,6	16,3	12,2	17.4	12.8	23.2	10,4	9,7	3.4	8.4	7.1
Amanda	18.0	33.0	-	8.8	17.2	14.0	-	12.5	7.7	2.4	10.9	8.5
GHCi	18.0	19.5	25.0	38.6	35.3	23.5	19.3	13.8	24.0	7.0	8.7	11.9
Hugs	44.0	26.0	-	120.0	66.0	36.0	-	54.0	42.0	13.0	10.4	16.2
GHC	1.8	1.5	8.2	4.0	4,1	3.8	6.6	1.6	3.7	0.9	2.3	1.3
GHC -O	0.9	1.5	1.8	0.2	1.0	1.4	0.1	1.1	0.4	0.2	1.6	0.4
Clean	0.9	0.8	0.6	0.2	0.9	1.4	2.4	0.7	0.4	0.2	4.9	0.6

tests we used a garbage collector with an explicit sweep phase instead of the implicit sweep (during memory allocation). This is done to make it possible to give meaningful figures about time spent in garbage collection. The price to be paid is a small performance penalty ($< 10\%$) and the use of an administration array for the collected free cells.

We conclude that if the peak heap memory stays under 30% of the total heap size execution times do not differ too much. If peak heap usage rises above 50% of total memory, performance drops radically and the amount of time spent in garbage collection grows rapidly. Because SAPL has a fixed heap, the memory management overhead is lower than in implementations with a flexible heap. SAPL uses relatively few GC cycles, because SAPL has a fixed heap and only starts garbage collection if there are less than 1000 free cells left.

The stack usage of SAPL is modest. Note, however, that SAPL also uses the C stack. The maximum amount of C stack for SAPL is 8Mb.

1.5.2 Comparison with other Implementations

In this subsection we compare SAPL with several other interpreters: Amanda V2.03 [9], Helium 1.5 [15], Hugs 20050113 [2] and GHCi V6.4 [1] and with the GHC V6.4 and Clean V2.1 compilers. We used the same amount of (fixed or maximal) heap space (64 Mb) and stack space (8 Mb) for all examples whenever this was possible (for Amanda the stack size cannot be set). For *Interpreter* and *Twice* the Amanda results are missing because of a stack overflow. Hugs also could not run these examples (C stack overflow).

Run-Time Comparison

The run-time results can be found in table 1.4. The results show us that the SAPL interpreter is almost 2 times faster than Amanda and Helium, about 3 times faster than GHCi and between 1.5 and 15 times faster than Hugs.

For the compilers there is more variation in the results due to the different optimisations applied by them. Comparing SAPL with GHC, the average speed-up of GHC is less than 3 times. The speed-ups of GHC -O and Clean vary between 1.1 (*Parser Combinators* in Clean) and 80 (*Twice* in GHC -O).

TABLE 1.5. Comparison Max Heap (kB) usage (upper) and GC time (%) (lower)

	Pri	Sym	Inter	Fib	Mch	Ham	Twi	Sort	Qns	Kns	Parse	Plog
SAPL	223	47	2344	12	101	107	762	2344	43	17	9700	150
Helium	774	16000	3000	258	774	516	1800	9000	258	256	10700	500
GHC	140	21	1800	6	46	50	800	1600	7	6	7000	50
SAPL	13	10	24	13	18	15	15	15	14	14	38	16
Helium	47	7	45	5	25	25	59	7	12	46	47	17
GHC def	18	1	87	1	22	16	67	5	1	45	70	25
GHC 24M	1	1	23	1	1	1	4	1	1	5	59	1

Comparison of Heap Usage

In table 1.5 we compare the memory usage and the time spent in garbage collection of SAPL (24 Mb heap) with that of Helium (standard heap) and the GHC compiler (standard and 24Mb initial heap). For Hugs, GHCi and Amanda no meaningful figures about memory usage can be given. We do not include a stack size comparison because SAPL also uses an unknown part of the C stack.

We conclude that GHC and SAPL use roughly the same amount of heap but that Helium uses more heap. The difference between SAPL and GHC can be explained by the fixed Cell size of 12 bytes used by SAPL. The unexpected high value of Helium for *Symbolic Primes* is probably a memory leak.

The amount of time spent in garbage collection of SAPL is mostly slightly lower than that of Helium and lower than that of GHC (default heap) for memory intensive programs like *Interpreter* and *Parser*. Variations of the (initial) heap size have only a small effect on the SAPL and Helium performance, but have a big impact on the performance of GHC. Setting the initial heap to 24Mb gives an almost 3-time speed-up for *Interpreter* and *Twice*, but halves the speed of almost all other benchmarks.

1.5.3 Discussion about Interpreter Comparison

What is the source of the good performance of SAPL compared with GHCi, Helium, Hugs and Amanda? The simplified memory management contributes to this better performance, but cannot be the only source (see table 1.5). Helium performs an overflow check on integer operations, which slows down integer intensive programs. If we compare SAPL with Amanda we see that for (almost) data type free programs there is not much difference in performance (*Fibonacci*, *Queens* and *Knights*). The difference in performance appears for programs using data types and pattern matching. Amanda uses a similar implementation of graph reduction as SAPL, but has a less sophisticated implementation of pattern matching using case-by-case matching [8]. If we compare the performance of SAPL with that of GHCi, Helium and Hugs we see that SAPL already has a better performance for data type free programs (*Twice*, *Fibonacci*). This increase in speed remains about the same for programs using data types and pattern matching. Helium uses techniques based on the STG machine to generate LVM byte code [11].

This byte code is interpreted. GHCi also compiles to byte code and is based on the GHC compiler that also uses the STG machine [13]. The Hugs implementation is based on byte code interpretation too. The SAPL interpreter is based on graph rewriting only and has no special constructs for data types and pattern matching. This enables a simple, high- level abstract machine with few, relatively large, atomic operations. There is no need for a more low level intermediate (byte code) formalism. The main difference between an interpreter and a compiler is that an interpreter has to check what to do next at every step. Keeping this overhead as small as possible is important for the construction of efficient interpreters. The easiest way to keep this overhead small is to use large atomic steps in the interpreter. Byte code instructions are mostly quite small. SAPL has a simple structure and uses large atomic steps. As a result the interpretation overhead for SAPL is lower than that for byte code based interpreters. The atomic operations in the SAPL interpreter are:

- Push a reference on the stack.
- Instantiate a function body, clear its arguments from the stack and place the result at the top application node.
- Call a built-in function, clear arguments from stack and place result at top application node.
- For a function call with as body a selector function application: Partly instantiate the body, recursively call *eval* for this instantiation and use the result to select and instantiate the appropriate other part of the body.

Except for the *push* operation these are all relatively large operations. The only benchmark for which the SAPL interpreter is not significant faster than Helium and GHCi, is *Symbolic Primes*. For this example the bodies of the (local) functions are mostly very small. Therefore the interpretation overhead will be much higher and comparable to the overhead of GHCi, Helium and Hugs.

Benefits of the Functional Encoding for the Interpreter Performance

First of all, we already concluded that the selective instantiation optimisation is essential for an efficient implementation of pattern-based function definitions using this encoding. It is therefore useless to try to run a SAPL program using another interpreter or compiler that doesn't uses the selective instantiation optimisation. Furthermore, in the previous subsection we concluded that the extra efficiency of the SAPL interpreter is not a result of the functional encoding and its implementation, but is a result of the simpler structure of the interpreter using a high level abstract machine with minimal interpretation overhead. The functional encoding enables this simple structure. It is possible to implement a traditional pattern matcher along the same lines as the functional pattern matcher with comparable performance, because both are based on the same techniques for encoding the pattern-based definition (see section 1.3.1).

We conclude that the most important benefit of the functional encoding is that it enables an elegant implementation of algebraic data types and pattern matching entirely within a pure functional domain and that this implementation can be made efficient by applying generic optimisations to a basic graph- rewriting interpreter.

1.6 CONCLUSIONS AND FURTHER RESEARCH POSSIBILITIES

In this paper we have defined the minimal (intermediate) functional programming language SAPL and an interpreter for it, based on a new variant of the Church encoding for algebraic data types. SAPL consists of pure functions only and has, besides integers, no other data types. For SAPL we have achieved the following results:

- The representation of data structures as functions in SAPL is more efficient than the Church encoding and the encoding of Berarducci and Bohm. The use of explicitly named functions (enabling explicit recursion) instead of lambda expressions enables an efficient implementation of this representation. We also showed how to translate pattern-based function definitions to SAPL. This makes SAPL usable as an intermediate language for interpretation of programs written in languages like Clean or Haskell.
- We described an efficient interpreter for SAPL based on straightforward graph rewriting techniques. The basic version of the interpreter is an ideal subject for educational purposes and for experimenting with implementation issues for functional languages. After applying two optimisations to speed up the execution of functions that are the result of the translation of pattern-based function definitions, the interpreter turns out to be competitive in a comparison with other interpreters. The results show us that for interpretation a high-level abstract machine with large atomic operations yields better results than low-level byte code interpreters based on techniques used for compilers.

1.6.1 Future Work

We plan to investigate the following issues for SAPL:

- We want to investigate whether the techniques used for implementing SAPL are also usable for realizing a compiler. We did some small experiments for this. We hand compiled the internal SAPL data structures to C code for a few benchmarks. This eliminates interpretation overhead and makes it possible to hard code the instantiation of functions (instead of a recursive copy). Speed-ups of 2 to 3 times seem possible, but more experiments are needed.
- We want to extend SAPL with IO features for creating interactive programs. Because SAPL is an interpreter it is also possible to use SAPL only as a calculation engine for another environment that does the IO.
- We want to investigate applications of SAPL. For example, SAPL can be used at the client side of Internet browsers as a plug-in, or inside a spreadsheet application.

REFERENCES

[1] *The Haskell Home Page*. www.Haskell.org.

[2] *Hugs Online*. www.Haskell.org/hugs.

[3] L. Augustsson. Compiling pattern matching. In Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architectures, Nancy*, volume 201 of *Lecture Notes in Computer Science*, pages 368–381. Springer Verlag, 1985.

[4] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1981.

[5] H.P. Barendregt. The impact of the lambda calculus in logic and computer science. *The Bulletin of Symbolic Logic*, 3(2):181–215, 1997.

[6] A. Berarducci and C. Bohm. *A self-interpreter of lambda calculus having a normal form*, volume 702 of *Lecture Notes in Computer Science*, pages 85–99. Springer Verlag, 1993.

[7] C. Bohm and A. Berarducci. Automatic synthesis of typed λ–programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[8] D. Bruin. Personal communication.

[9] D. Bruin. The amanda interpreter. www.engineering.tech.nhl.nl/engineering/personeel/bruin/data/amanda203.zip.

[10] W. Kluge. *Abstract Computing Machines*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[11] D. Leijen. *The λ Abroad – A Functional Approach to Software Components*. PhD thesis, Department of Computer Science, Universiteit Utrecht, The Netherlands, 2003.

[12] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice-Hall, 1987.

[13] S.L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[14] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. International Computer Science Series. Addison-Wesley, 1993.

[15] Software Technology Group, the Institute of Information and Computing Sciences, Utrecht University, the Netherlands. *The Helium Project*. www.cs.uu.nl/helium.

[16] Software Technology Research Group, Radboud University Nijmegen. *The Clean Home Page*. www.cs.ru.nl/˜clean.

[17] Software Technology Research Group, Radboud University Nijmegen. *The SAPL Home Page*. www.cs.ru.nl/˜jmjansen/sapl.