Efficient Interpretation by Transforming Data Types and Patterns to Functions

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

[1] The Netherlands Ministry of Defence,
[2] Institute for Computing and Information Sciences (ICIS),
Radboud University Nijmegen, the Netherlands.
j.m.jansen@forcevision.nl, {pieter,rinus}@cs.ru.nl

Abstract

In this paper we present the stepwise construction of an efficient interpreter for lazy functional programming languages like Haskell and Clean. The interpreter is realized by first transforming the source language to the intermediate language **SAPL** (**S**imple **A**pplication **P**rogramming **L**anguage) consisting of pure functions only. During this transformation algebraic data types and pattern-based function definitions are mapped to functions. This eliminates the need for constructs for Algebraic Data Types and Pattern Matching in SAPL.
For SAPL a simple and elegant interpreter is constructed using straightforward graph reduction techniques. This interpreter can be considered as a prototype implementation of lazy functional programming languages. Using abstract interpretation techniques the interpreter is optimised. The performance of the resulting interpreter turns out to be very competitive in a comparison with other interpreters like Hugs, Helium, GHCi and Amanda for a number benchmarks. For some benchmarks the interpreter even rivals the speed of the GHC compiler.
Due to its simplicity and the stepwise construction this implementation is an ideal subject for introduction courses on implementation aspects of lazy functional programming languages.

1 INTRODUCTION

For modern lazy functional programming languages like Haskell [9], Clean [8] it is common to use an intermediate language to realize an implementation. These intermediate languages normally contain special constructs for handling algebraic data types and pattern-based function definitions (see e.g. Peyton Jones [15] and Kluge [13]). In this paper we show that these special constructs can be omitted. It is possible to translate languages like Clean and Haskell to an intermediate language containing only functions with no special constructs for handling data types and pattern matching. The transformation is based on a representation of algebraic data types by lambda expressions as described by Berarducci and Bohm ([4] and [5]) and Barendregt [3]. We use explicitly named functions instead of lambda expression to represent types, which makes this representation practical usable.

*The first author was partly supported by the Centre for Automation of Mission critical Systems, Force Vision, the Netherlands Ministry of Defence.

More important, contrary to popular believe, it is possible to make an efficient implementation for this intermediate language. We constructed an interpreter based on straightforward graph rewriting for pure functions. The interpreter is optimised by applying a static analysis to the transformed program using abstract interpretation techniques. These optimisations are applied on the level of pure functions without direct knowledge about the data types and pattern definitions of the source language. Because of its simplicity, this implementation is an ideal object, not only for studying and prototyping new language features, but also for teaching implementation issues for functional programming languages.

Summarizing, the contributions of this paper are:

- The transformation of algebraic data types and pattern-based function definitions to simple function definitions in the intermediate language SAPL. This transformation eliminates the need for special constructs for handling pattern matching in an interpreter.
- A demonstration that an efficient implementation for lazy functional programming languages can be realized with minimal and elementary effort. The implementation of the interpreter is considerably shorter than that of traditional interpreters with a comparable and often even better performance.

Although the main purpose of this paper is expository, the transformation of pattern based function definitions to a continuation scheme of functions is new. Also, the optimisations used in the interpreter are a generalization (not aimed at pattern matching in particular) of the techniques used to realize an efficient implementation of pattern matching.

The structure of this paper is as follows. In Section 2 we introduce the intermediate functional programming language **SAPL**. SAPL has, besides integers and their operations, no data types. We show how to represent algebraic data types as functions in this language and give a general translation scheme for this transformation. SAPL can be considered as the smallest and purest usable lazy functional programming language. In Section 3 we sketch how to transform pattern-based function definitions to functions in SAPL. In Section 4 we define a very simple interpreter for this language bases on simple graph-rewriting techniques. This interpreter is optimised using abstract interpretation techniques in Section 5. The performance of the resulting interpreter is compared with other interpreters in Section 6. It turns out to be very efficient. In Section 7 we will give some conclusions and discuss further research possibilities.

2 SAPL: AN INTERMEDIATE FUNCTIONAL LANGUAGE

SAPL is based on simple rewriting of function applications. It has the following syntax description:

```
function     ::= identifier {identifier}* '=' expr
expr         ::= application | identifier '->' expr
```

```
application ::= factor {factor}*
factor      ::= identifier | '(' expr ')'
```

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 and other function names may occur. SAPL is un-typed. The language has the usual lazy rewrite semantics (see Section 4). For usability we added integers and their basic operations like addition, subtraction and comparing to the interpreter.

2.1 Representation of Data Types in SAPL

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

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

We map this type definition to *m* functions:

$C_1\ v_{1,1}\ ..\ v_{1,n_1}\ =\ f_1\ \text{->}\ ..\ \text{->}\ f_m\ \text{->}\ f_1\ v_{1,1}\ ..\ v_{1,n_1}$
..
$C_m\ v_{m,1}\ ..\ v_{m,n_m}\ =\ f_1\ \text{->}\ ..\ \text{->}\ f_m\ \text{->}\ f_m\ v_{m,1}\ ..\ v_{m,n_m}$

A constructor is now represented by a function with the same number of arguments. A function with an argument of this data type can be defined as follows:

$f\ x\ =\ x$
$\quad (v_{1,1}\ \text{->}\ ..\ \text{->}\ v_{1,n_1}\ \text{->}\ \text{body}_1)$
$\quad ..$
$\quad (v_{m,1}\ \text{->}\ ..\ \text{->}\ v_{m,n_m}\ \text{->}\ \text{body}_m)$

Here $body_i$ depends on $v_{i,1}$ .. v_{i,n_i} for all $i \in \{1,\dots,m\}$ and calculates the result of the function for an element of kind $C_i\ v_{i,1}$ .. v_{i,n_i}. We will call these functions corresponding to constructors *selector* functions.

2.2 Cyclic Definitions

For efficiency reasons it is necessary to allow for cyclic (local) constant definitions. This is realized by adding the possibility of labelling (sub)expressions in the body of a function and using these labels as variables at other places in the body. This addition can also be used for sharing constant definitions. We have to adapt the *factor* rule in the definition of SAPL with:

```
factor ::= identifier | [identifier '@'] '(' expr ')'
```

An example of the use of cyclic definitions is the famous Hamming function:

```
hamming = h @ (Cons 1 (merge (merge (map (mult 2) h)
                                    (map (mult 3) h))
                                    (map (mult 5) h)))
```

2.3 Examples

2.3.1 Natural Numbers

We can define natural numbers using the Peano axioms and give a recursive definition for the addition operation. In Haskell we have:

```
data Nat = Zero | Suc Nat
add Zero     n = n
add (Suc m)  n = Suc (add m n)
```

In the SAPL the definitions are:

```
Zero     = f -> g -> f
Suc n    = f -> g -> g n
add mz n = mz n (m -> Suc (add m n))
```

Note that expressions like *Zero* and *Suc (Suc Zero)* are now functions!

2.3.2 Lists

An important and often used algebraic data type in functional programming languages is the list. In most languages the list is a pre-defined data type with special syntax. A definition for lists in Haskell, together with the function *length* could be:

```
data List t = Nil | Cons t (List t)
length :: List t -> Int
length Nil         = 0
length (Cons a as) = 1 + length as
```

The translation to SAPL results in:

```
Nil       = f -> g -> f
Cons x xs = f -> g -> g x xs
length xs = xs 0 (x -> xs -> 1 + length xs)
```

2.3.3 Higher Order Data Types

As a last example, we consider a more complex data type. The data type represents arithmetical expressions, including the operations to evaluate them.

```
data Expr = Oper String  (Int -> Int -> Int) Expr Expr |
            UnOper String (Int -> Int)       Expr      |
            Num Int
eval :: Expr -> Int
eval (Oper name op left right) = op (eval left) (eval right)
eval (UnOper name op e)        = op (eval e)
eval (Num n)                   = n
```

The translation to SAPL results in:

```
Oper na op left right = f -> g -> h -> f na op left right
UnOper na op e        = f -> g -> h -> g na op e
Num n                 = f -> g -> h -> h n
eval e
= e (na->op->left->right-> op (eval left) (eval right))
    (na->op->e->           op (eval e))
    (n->                   n)
```

2.3.4 Discussion

From the examples above we see that the transformation of algebraic data types to SAPL functions is a natural one. The resulting functions greatly resemble the original type definitions. Simple functions on data types are easy to transform and result in readable functions. In the next section we discuss the transformation of more complex functions involving pattern matching.

2.4 Comparison with Church Numerals

It is interesting to compare the representation of natural numbers by functions from Subsection (2.3.1) with the standard representation by Church numerals. Although Church numerals capture the repetitive aspect of natural numbers in a more natural way, they have important drawbacks. It is, for example, very hard and inefficient ($O(n)$) to represent an operation like predecessor for them, while this can be done very easily using the representation above.

```
pred n = n fail (n -> n)
```

Here *fail* represents an illegal result (predecessor of zero).

3 COMPILING PATTERN DEFINITIONS TO FUNCTIONS

If we want to use SAPL as an intermediate language for implementing lazy functional language we must be able to translate constructs from these languages to SAPL functions. Constructions like list-comprehensions, *where* and *let(rec)* expressions can be handled with standard techniques like they are described in [15] and [17]. Another important construct in these languages is the use of pattern-based function definitions. Traditionally, pattern definitions are compiled to a dedicated structure that can handle a pattern-match at runtime (Augustsson [1] and Peyton Jones [15]). Here we transfer pattern definitions to SAPL functions. The transformation proceeds in two steps. In the first step, the pattern definitions are translated to a decision tree representing the steps to be taken during a pattern match. In the second step SAPL function definitions are generated from the decision tree. For the formal definition of pattern-based functions we have to replace the *function* line in the definition of SAPL from Section 2 by:

```
function    ::= identifier {patorvar}* ‘=’ expr
patorvar    ::= identifier | ‘(’ pattern ‘)’
pattern     ::= identifier {patorvar}+
```

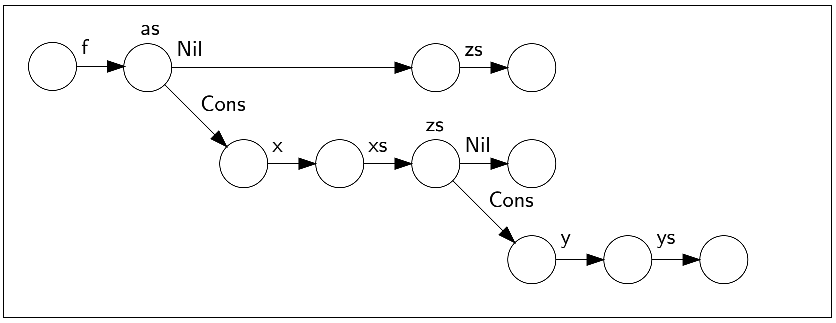

FIGURE 1. Decision Tree for *mappair1*

Here an identifier in *patorvar* must be a variable name or a zero argument constructor like *Nil*. The identifier in *pattern* must be the name of a constructor with one or more arguments. The number of arguments of a constructor and their types must correspond to the type definition of the constructor (see Section 2). We will call this version of SAPL extended SAPL.

3.1 Transformation of Pattern Definitions to a Decision Tree

The decision tree can be obtained by using a pattern-matching compiler as described in [15] and [17]. We use two examples to illustrate this process. The examples are two versions of the well known *mappair* (*zipWith*) function. In the definition of *mappair1* all cases exclude each other (no overlap).

```
mappair1 f Nil         zs           = Nil
mappair1 f (Cons x xs) Nil          = Nil
mappair1 f (Cons x xs) (Cons y ys)
                       = Cons (f x y) (mappair1 f xs ys)
```

Figure 1 represents the decision tree for *mappair1*. Each row in the tree corresponds to a case in the pattern definition. The resulting decision tree is optimal; all arguments have to be examined only once (there is no need for backtracking to previous arguments during a match). For performing a match one has to traverse the decision tree from the left to the right and follow the arrows corresponding to the actual arguments. If the last state in a row is reached the body corresponding to that row is the correct body. The second example represents the ‘inefficient’ version of *mappair* (a variable *as* instead of the pattern *(Cons x xs)* in the second case). The corresponding decision tree can be found in figure 2. In this figure dashed arrows indicate backtracking to a previous argument. Arrows pointing to nothing indicates that the entire match fails. Note also that in this example these *fail* arrows will never be traversed.

```
mappair2 f Nil         zs           = Nil
mappair2 f as          Nil          = Nil
mappair2 f (Cons x xs) (Cons y ys)
                       = Cons (f x y) (mappair2 f xs ys)
```

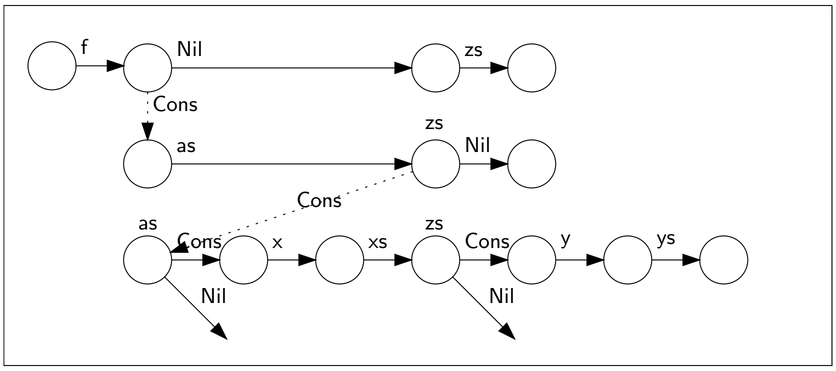

FIGURE 2. Decision Tree for *mappair2*

Comparing the results it is clear that *mappair1* is much more efficient than *mappair2*.

3.2 Generation of Functions

The generation of a SAPL function from a decision tree can be obtained by traversing the decision tree. Every *constructor* node (*var* nodes can be skipped) leads to an application of the label of the node to anonymous functions corresponding to the different alternatives for the type (in the correct order). The body of an anonymous function is the result for the corresponding successor node(s). If all constructor nodes in a row are handled the corresponding body can be filled in. The results for the two versions of *mappair* are:

```
mappair1 f as zs
= as  Nil  (x -> xs ->
      zs   Nil  (y -> ys ->
           Cons (f x y) (mappair1 f xs ys)))

mappair2 f as zs
= as Nil (v1 -> v2 ->
     zs  Nil (w1 -> w2 ->
         as  fail (x -> xs ->
             zs fail (y -> ys ->
                Cons (f x y) (mappair2 f xs ys)))))
```

As a more complex example we show the result for a pattern definition involving nested patterns. Consider the function *complex*:

```
complex (Cons a (Cons b (Cons c Nil))) = a + b + c
complex (Cons a (Cons b  Nil))         = 2 * a + b
complex (Cons a Nil)                   = 3 * a
complex xs                             = 0
```

The translation to SAPL results in:

```
complex xs =
xs 0 (a->p01->
       p01 (mult 3 a) (b->p011->
            p011 (add (mult 2 a) b) (c->p0111->
                 p0111 (add (add a b) c) (p01110->p01111->
                       0))))
```

In this case the translator has to invent names for many constructor nodes. Also for this function the result is optimal. There is no need to re-examine arguments.

In general we may conclude that translating pattern-based functions to SAPL results in compact and still readable functions.

4 A SIMPLE INTERPRETER FOR SAPL

The interpreter is kept as simple as possible. It is based on straightforward graph reduction techniques as described in Peyton Jones [15], Plasmeijer and van Eekelen [17] and Kluge [13]. We assume that a pre-compiler has eliminated all algebraic data types and pattern definitions (as described above) and all let(rec)- and where-clauses and lambda lifted all lambda expressions. 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 cells. A Cell 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. A Cell uses 12 bytes of memory.
- The memory heap consists only of Cells. The heap has a fixed, user definable size. Memory allocation is therefore very cheap.
- It uses mark and (implicit) sweep garbage collection.
- 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.
- It reduces an expression to head-normal-form. The printing routine causes further reduction.
- 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, an application or a function call node.

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 *eval* function is given by:

```
eval :: Expr -> [Expr] -> [Expr] -> Expr
eval (App l r)      es fs = eval l (push r es) fs
eval (Func na fn)   es fs
= if (length es >= na)
     (eval (instantiate (el fn fs) es) (drop na es) fs)
     (rebuild (Func na fn) es)

instantiate (App l r) es  = App (instantiate l es)
                                (instantiate r es)
instantiate (Var n)   es  = el n es
instantiate x         es  = x

rebuild e Nil             = e
rebuild e (Cons x xs)     = rebuild (App e x) xs
```

Here *es* represents the stack and *fs* the list of function body definitions. The actual C versions (including numbers and operations on them) of *eval* and *instantiate* are straightforward implementations of this specification and fit together on less than one page!

5 AN OPTIMAL INTERPRETER FOR SAPL

Despite its simplicity the interpreter from the previous section has already reasonable performance. But if we compare its performance with that of GHCi, Helium and Amanda it turns out to be 30 to 800% slower. This is not surprising because functions on algebraic data types tend to have large bodies (all cases are included), which implicates large instantiation overhead.

A careful look at the implementation of the *eval* function learns us that there are several options for optimisation:

- Representing nodes with less memory overhead
- Preventing rebuilding curried calls for functions
- Reducing the number of stack operations
- Reducing the size of the instantiations

All these issues are addressed in the optimisations for SAPL. We applied the following optimisations:

- A more efficient memory representation of function calls with one or two arguments. For function applications with one or two arguments the *APP* nodes are removed.

- Rebuilding of curried functions can be prevented by keeping a reference to the top node of the application. In many cases a static compile time check can detect a curried call and mark the top node. In this way an attempt to reduce a curried call can be prevented.

- In many function bodies applications of anonymous local functions occur. Normally a local function is lifted to the global level. In case evaluation of the local function is necessary for reduction to head-normal-form of the function body this lifting is not necessary. The local function call can be reduced (in-lined) in the context of the surrounding function. This reduces the number of stack operations, because lambda lifting introduces extra arguments in functions and calls for these functions.

  For example, consider the function:

  ```
  f x = (y -> add y x) 5
  ```

  Reduction of the local function is necessary for reducing a call for *f*. The anonymous function can therefore be reduced in the context of reducing *f*.

  This optimisation is especially useful for optimising the results of translating pattern-based definitions as we can see from the examples from Section 3. In the resulting functions often many lambda expressions occur.

- The must significant optimisation can be realized by reducing the size of the instantiations. We call a function that doesn't use all its arguments in its body a selector function. Functions that represent algebraic data types in SAPL are the most important examples of selector functions. A significant optimisation can be realized by statically checking if a function body consists of an (non-curried) application of a selector function. In that case it is not necessary to instantiate the entire body, but only the relevant part (depending on the actual selector function applied at runtime). To find out which functions are candidate for this optimisation an (abstract interpretation) analysis of the entire program is needed. The following steps are performed during this analysis:

  - First selector functions must be identified. This step is straightforward because selector functions can be recognized directly from their bodies.
  - It must be analysed which functions return selector functions. This requires some kind of fixed-point algorithm, because adding a function that returns a selector function can lead to new functions that return selector functions.

  - It has to be analysed which arguments of functions are of selector function type. This can be deduced from the results of the previous step and by analysing the propagation of selector function arguments.
  - As a last step it must be analysed which function bodies are an application of a selector function. As a result of this step new functions that return a selector function can be found. Therefore, the last three steps must be repeated until a fixed-point is reached.

  At run-time first the part of the body representing the selector function is instantiated and evaluated. Depending on the result the appropriate remainder part of the body is instantiated and evaluated. The result of this optimisation is significant and often leads to speed-ups up to a factor of 10.

Most of the effort for applying the above optimisations must be taken during the compilation phase of the SAPL program. The interpreter adds extra hooks to the resulting functions that are needed during the interpretation phase to perform the optimisation. The overhead in the interpretation phase (*eval* function) itself is very small (less than 60 lines of C)!

For example: the result of optimising the *mappair1* function is:

```
mappair1 f as zs
= select  as  Nil (x -> xs ->
              select  zs  Nil (y -> ys ->
                            Cons (f x y) (mappair1 f xs ys)))
```

The *select* directive indicates the application of a *selector* function. The anonymous functions in the body are not lifted to the global but executed inline.

The last optimisation involves a rather technical analysis of the result of the transformation of a source program to SAPL. It is also possible to add most of the optimisation hooks during the transformation of the source program to SAPL. In this case no complex analysis is needed, because we already know where to apply the hooks as can be seen from the example. Above we wanted to show that it is possible to apply the optimisations without direct knowledge of the original data types and functions and that they apply for a general functional programming language based on function rewriting only.

6 COMPARISON WITH OTHER IMPLEMENTATIONS

In this section we present the results of a comparison of SAPL with several other interpreters for functional programming languages: Amanda V2.03 [6], Helium 1.5 [10], Hugs 20050113 [12] and GHCi V6.4 [9]. We used 14 benchmark programs, divided over 11 categories, for the comparison:

- **Prime Sieve** The prime number sieve program.
- **Interpreter** A small interpreter for a SAPL like language.

- **Symbolic Primes** A symbolic version of the prime number sieve using Peano numbers instead of integers.
- **Fibonacci** The (naive) Fibonacci function.
- **Tree Generate** The generation of large binary trees from long lists.
- **Match** Deeply nested pattern matching (to measure the efficiency of the pattern matching implementation).
- **Hamming** The generation of the list of Hamming numbers (a cyclic definition).
- **Merge** Merging of very long sorted lists.
- **Twice** An example of the use of higher order functions (*twice twice twice twice (add 1) 1*).
- **Sorting** Quick Sort, Merge Sort and Insertion Sort.
- **Backtracking** Placement of Queens and Knights tour on a chess board.

We used the same amount of heap space for all examples (64 Mb). For Amanda and Helium it was not possible to set the stack size. For the other interpreters we made the stack large enough to run the benchmarks (8 Mb). The results for Hugs are not added to the figure because Hugs is much slower (10-20 times) for most examples. For *Interpreter* and *Twice* the Amanda results are missing because of a stack overflow. For *Tree Generate* the Helium result is missing because of an infinite recursion error message (probably also due to stack problems). The results can be found in figure 3. Here the SAPL results are normalized at 10. If we take the average of the results we see that SAPL is 2 times faster than Amanda and Helium and 3.4 times faster than GHCi.

We also compared the performance of SAPL with that of the GHC compiler (without the -O option). Surprisingly GHC is always less than 3 times faster (except for *Symbolic Primes* (10 times)) and often not more than 2 times faster (average 2.3 and 1.8 without *Symbolic Primes*). For *Interpreter*, *Tree Generate*, *Twice* and *Merge Sort* there is almost no difference in performance. GHC with the -O option turned on is mostly between 3 and 5 times faster than SAPL (average 4.6).

If we compare SAPL with Clean, we see that Clean is about 10 times faster than SAPL (not considering *Fibonacci*, 44 times). Clean does a very good job for programs not containing too many higher order functions. For *Twice* the difference is much smaller (4 times). *Twice* is the only benchmark for which Clean is slower than GHC -O. For the other benchmarks Clean is about 4 times faster than GHC and 2 times faster than GHC -O (again not considering *Fibonacci*).

From the results we see that SAPL has excellent performance. Especially for benchmarks involving heavy memory usage (*Primes, Merge, Tree Generate* and *Sort*) and for higher order examples (*Twice*), SAPL is doing very well. This shows

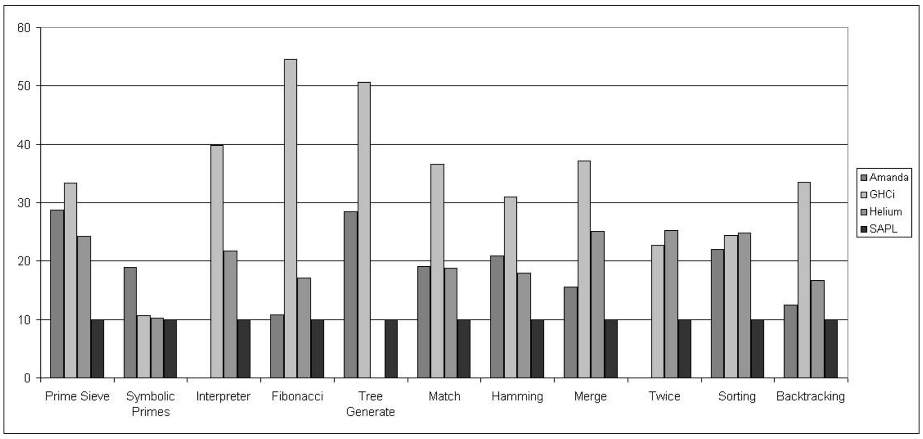

FIGURE 3. Performance Measures (Relative Time: SAPL Normalized at 10)

that the transformation of pattern matching to a continuation scheme of functions is not only elegant but can also be efficiently implemented.

6.1 Discussion about Interpreter Comparison

The SAPL interpreter turns out to be very efficient. The techniques used for its implementation are completely different from the techniques used in most compilers and interpreters for functional programming languages. Helium uses techniques based on the STG machine to generate LVM byte code [14]. This byte code is interpreted. Also GHCi compiles to byte code and is based on the GHC compiler that also uses the STG machine [16]. Amanda uses techniques similar to those of SAPL. Amanda has special constructs for data types and pattern-matching (like all other implementations). Amanda uses no ‘smart’ pattern compiler but tries to match every case separately until a matching one is found [7]. Amanda has lists and many operations on lists hard-coded in the interpreter. This increases the speed of list intensive programs almost by a factor 2. For the comparison above we used user defined lists and list operations to keep the comparison fair. It is, of course, also possible to hard code lists and list operations in SAPL with a similar speed-up.

The main difference between SAPL and the implementations for Helium, GHCi and Hugs is that SAPL is based on direct interpretation of graph rewriting, without the use of some kind of byte code. This is a bad idea when writing a compiler but turns out to be very efficient for an interpreter. 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 distinguishes efficient interpreters from less efficient ones. The easiest way to keep this overhead small is to use large atomic steps in the interpreter. Byte code instructions are mostly quite small. Because SAPL does not need special constructs for data types and pattern matching, it has a very simple structure and therefore there is no need for explicit intermediate (byte) code. The abstract machine of SAPL uses few and large atomic steps. The basic operations in the SAPL interpreter are:

- Push a reference on the stack. A small step.
- Instantiate a function body, clear its arguments from the stack and place the result at the top application node. Mostly a big step.
- Call a build-in function, clear the arguments from the stack and place the result on the top application node. A medium size step.
- 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. Mostly a big step.

The only poor performing benchmark is *Symbolic Primes*. For this benchmark SAPL is not really faster than Helium and GHCi and 10 times slower than GHC and GHC -O. A careful study of the SAPL code for this benchmark teaches us that the bodies of the functions are mostly very small. This means that the interpretation overhead is relatively large. This could explain the difference in performance for this benchmark.

6.2 Discussion about Compiler Comparison

It is also surprising that the difference between the performance of SAPL and the GCH compiler is small. Normally, compilers are thought to be at least ten times faster than interpreters. For imperative languages this is certainly true. But for functional (and probably also logic) languages this is not the case for the same reasons as mentioned above: SAPL has only very few basic operations of rather large size, so the interpretation overhead is small. Imperative languages are less suitable for realizing an interpreter with large basic steps.

7 CONCLUSIONS AND FURTHER RESEARCH POSSIBILITIES

In this paper we have defined a minimal (intermediate) functional programming language SAPL. This language consists of pure functions only and has, besides integers, no other data types. For SAPL we have achieved the following results:

- It is possible to represent data structures as functions in SAPL in a natural way. This representation is based on a representation of data types by lambda expressions described by Berarducci and Bohm ([4], [5] and [2]). The use of explicitly named functions instead of lambda expressions enables a practical implementation of this representation.

- Programs written in (or translated to) SAPL have comparable length to corresponding programs in functional programming languages like Haskell or Clean with only a small readability penalty.

- SAPL can be used as an intermediate language for interpretation of programs written in languages like Clean or Haskell. We have shown how to translate pattern-based function definitions to SAPL with an efficient result.

- We have constructed a very efficient interpreter for SAPL based on straightforward graph rewriting techniques. Due to the simplicity of the language the interpreter can be kept small and elegant. After applying some generic optimisations the interpreter turns out to be at least twice as fast as other interpreters.

7.1 Future Work

We have planned to investigate the following issues for SAPL:

- It is interesting 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 circumvents the test for taking the next step and makes it possible to hard code the instantiation of a function body (instead of a recursive copy). For the examples the saving is between a factor 2 and 3. The speed of SAPL is now almost equal to that of GHC (without -O).

  It is too early to draw general conclusion from this. More experiments for other examples are needed. It is also not clear yet whether the optimisations used in GHC -O can also be realized in a compiler based on SAPL techniques. Also for this further investigations 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 to be used from 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] L. Augustsson. Compiling Pattern Matching. *Conference on Functional Programming Languages and Computer Architectures, Nancy*, Jouannaud (editor), *Lecture Notes in Computer Science* **201**, pp 368-381, Springer Verlag, 1985.

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

[3] H.P. Barendregt. The impact of the Lambda Calculus in Logic and Computer Science. *The Bulletin of Symbolic Logic*, Volume 3, Number 2, pp 181 - 215, 1997.

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

[5] C. Bohm and A. Berarducci. Automatic synthesis of typed Λ–programs on term algebras. *Theoretical Computer Science* **39**, pp 135-154, 1985.

[6] D. Bruin. The Amanda Interpreter.
www.engineering.tech.nhl.nl/engineering/personeel/bruin/data/amanda203.zip.

[7] D. Bruin. Personal communication.

[8] The Clean Home Page. Software Technology Research Group, Radboud University Nijmegen, the Netherlands, www.cs.ru.nl/~clean.

[9] The Haskell Home Page, www.Haskell.org.

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

[11] J.R. Hindley and J.P. Seldin. Introduction to Combinators and λ–Calculus. *London Mathematical Society Student Texts*, 1986.

[12] Hugs Online, www.Haskell.org/hugs.

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

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

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

[16] S.L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming*, Volume 2, Number 2, pp 127 - 202, 1992.

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