Data Types and Pattern Matching by Function Application

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

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

Abstract. It is generally known that algorithms can be expressed using simple function applications only. In this paper we will show that it is possible to make a systematic translation of algebraic data types and pattern-based function definitions, like they are used in most modern functional programming languages, to function applications. The translated algebraic data types closely resemble the original type definitions. We developed an efficient interpreter for the resulting functions. The interpreter has a simple and elegant structure because there is no need for special handling of data structures and pattern matching. Despite the lack of optimisations its performance turns out to be competitive in a comparison with other interpreters like Hugs, Helium, GHCi and Amanda.

1 Introduction

It is common knowledge that computable algorithms can be expressed using function applications only. Examples can be found in every textbook on lambda calculus (Barendregt [2], Hindley/Seldin [10]). Modern functional programming languages like Haskell [8], Clean [7] use, besides functions, also algebraic data types to enrich the readability and the expressiveness of programs. Furthermore, they use pattern-based function definitions to enrich the readability of function definitions even more.

Algebraic data types and pattern-based definitions need a special implementation in compilers and interpreters for functional programming languages (see e.g. Peyton Jones [13] and Kluge [12]). In this paper we will show that it is not necessary to add special treatment for the implementation of algebraic data types and pattern-based function definitions, but that we can translate them to simple functions with as only operation function application. The way we represent data types by functions is considered to be part of the lambda calculus/functional programming folklore. Berarducci and Bohm ([4] and [5]) and Barendregt [3] describe a similar approach, that only differs for recursive data types. We can handle recursive data types easier since we have named functions instead of just lambda-terms. New is the use of this transformation to functions

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

to realize an efficient implementation for data structures and pattern matching. It turns out that a small adaptation to a straightforward interpreter for a pure functional language without data types can handle the resulting functions efficiently. The performance of this interpreter is comparable to that of other popular interpreters. The pure functional language that we will introduce can both be used as a programming language and as an intermediate language for interpreting higher functional languages like Haskell and Clean. Because of its simplicity, this interpreter 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 representation of algebraic data types by functions, containing only function applications, that closely resemble the original type definitions.
- A new elegant pattern compiling algorithm for transforming pattern-based function definitions to function applications.
- A simple but efficient interpreter for a pure functional language with as only operation function application that is optimised for handling transformed data types.

The structure of this paper will be as follows. In section 2 we will introduce a simple functional programming language with as only operation function applications. The language has no data types! We will show how we can represent algebraic data types as functions in this language and give a general translation scheme for this transformation. The language can be considered as the smallest usable functional programming language. In section 3 we will introduce a new pattern compilation algorithm that we will use to transfer pattern-based function definitions, as they are used in Haskell or Clean, to functions in the language of section 2. In section 4 we will define a very simple interpreter for this language. We will show that we can transform this interpreter into an efficient one by a straightforward optimisation. The performance of the interpreter will be compared with other interpreters in section 5. We will see that, despite its simplicity, the interpreter is very efficient. In section 6 we will give some conclusions and discuss further research possibilities.

2 SAPL: A Functional Programming Language with as only operation Function Application

The functional programming language we use has as only operation rewriting function applications. We will call it **SAPL** (**S**imple **A**pplication **P**rogramming **L**anguage). It has the following syntax description:

```
function ::= identifier {identifier}* ‘=’ expr
expr     ::= factor {factor}*
factor   ::= identifier | ‘(’ expr ‘)’
```

A function has a name followed by zero or more variable names. In an expression only variable names and other function names may occur. SAPL is un-typed. We will see that many functions we will define, cannot be typed using Hindley-Milner type inference. The language has the usual lazy rewrite semantics (see

section 4). In fact, SAPL is equivalent to un-typed lambda calculus with named expressions and (therefore) explicit recursion. For notational convenience we extend the formalism with anonymous or in-line functions (lambda expressions). The syntax description now becomes:

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

Using the technique of lambda-lifting we can transfer function definitions with lambda expressions to function definitions without lambda expressions.

2.1 Representation of Data Types by Functions

It is well known that the formalism described above is powerful enough to describe any computable algorithm ([2] or [10]). Many descriptions of implementations of functional programming languages use a similar language as an intermediate language or to illustrate implementation issues. But as soon as algebraic data types and pattern matching are described the formalism is extended with special constructs to handle these [13]. It is, however, perfectly possible to handle algebraic data types and pattern-based functions within the formalism of simple function applications. Consider the following algebraic data type definition in a language like Haskell or Clean:

$$\mathrm{typename}\ \mathrm{t}_1\ ..\ \mathrm{t}_k\ ::=\ \mathrm{C}_1\ \mathrm{t}_{1,1}\ ..\ \mathrm{t}_{1,n_1}\ |\ ..\ |\ \mathrm{C}_m\ \mathrm{t}_{m,1}\ ..\ \mathrm{t}_{m,n_m}$$

Here, *typename* is the name of the type, t_1 .. t_k are type parameters, C_1 .. C_m are constructor names and all $t_{i,j}$'s are type names or type variables. We map this type definition to m functions (one for every constructor):

$$\begin{array}{l}\mathrm{C}_1\ \mathrm{v}_{1,1}\ ..\ \mathrm{v}_{1,n_1}\ \mathrm{f}_1\ ..\ \mathrm{f}_m\ =\ \mathrm{f1}\ \mathrm{v}_{1,1}\ ..\ \mathrm{v}_{1,n_1}\\ ..\\ \mathrm{C}_m\ \mathrm{v}_{m,1}\ ..\ \mathrm{v}_{m,n_m}\ \mathrm{f}_1\ ..\ \mathrm{f}_m\ =\ \mathrm{f}_m\ \mathrm{v}_{m,1}\ ..\ \mathrm{v}_{m,n_m}\end{array}$$

The constructor names now have become function names. For each $i \in \{1,\ldots,m\}$ $v_{i,j}$ ($j \in \{1,\ldots,n_i\}$) are distinct variable names. For each $i \in \{1,\ldots,m\}$ f_k ($k \in \{1,\ldots,m\}$) are also distinct variable names, different from all $v_{i,j}(j \in \{1,\ldots,n_i\})$. There are as many f_k's in each definition as there are constructors (m). Using these functions a function with as argument one element of this algebraic data type can be defined as follows:

$$\begin{array}{l}\mathrm{f}\ \mathrm{tn}\ =\ \mathrm{tn}\\ \qquad\qquad (\mathrm{v}_{1,1}\ ->\ ..\ ->\ \mathrm{v}_{1,n_1}\ ->\ \mathrm{body}_1)\\ \qquad\qquad\ ..\\ \qquad\qquad (\mathrm{v}_{m,1}\ ->\ ..\ ->\ \mathrm{v}_{m,n_m}\ ->\ \mathrm{body}_m)\end{array}$$

Here $body_i$ depends on $v_{i,1}$ .. v_{i,n_i} for all $i \in \{1,\ldots,m\}$ and calculates the result of the function for an element of kind C_i $v_{i,1}$ .. v_{i,n_i}. This definition uses the fact that an element of the algebraic data type is now a function that selects of m functions exactly that function that corresponds to its constructor and applies this function to the arguments of the constructor. Therefore, we will call these

functions corresponding to constructors *selector* functions. If a function has more than one argument of an algebraic data type we have to examine all combinations of constructors. This makes the definitions less readable. Therefore, we will give in the next section a generic schema for the transformation of more readable pattern-based definitions to SAPL.

If we compare our transformation scheme to that of Berarducci and Bohm [4], we see that in our approach recursive data types are handled in a different, more simple, way. This can be done because we use explicitly named functions instead of anonymous lambda expressions.

2.2 Examples

We will now give some examples to illustrate the use of functions as data types. We first give the Haskell definition of the function and then the definition using the translation scheme defined above.

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))
```

We see that the definition of *add* has great resemblance to the Haskell definition. Note that an expression like *Suc (Suc Zero)* must be considered as both a function and as a value!

The most often encountered definition for natural numbers in the lambda calculus are the Church numerals. In Haskell the Church numerals can be defined as follows:

```
type Cnum t = (t -> t) -> (t -> t)
Zero :: Cnum t
Zero f = id
Suc :: Cnum t -> Cnum t
Suc cn f = f . cn f
```

Here *id* is the identity function and . represents the composition operation. The function for addition now becomes:

```
add cn dn f = cn f . dn f
```

We see that this definition is quite different from the definition above. Advantages of the Church numerals are that they can be given correct Haskell types, which is not possible for the SAPL definitions above. An important advantage of the SAPL definition of numbers is that an operation like predecessor can easily be described in it, while this is very hard and inefficient ($O(n)$) using Church numerals.

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 not using this syntax 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)
```

Again the definition of length greatly resembles the Haskell definition. Also *length* cannot be typed. Due to their continuation style it is in general not possible to type recursive functions on data types using simple Hindley-Milner types.

The standard way for defining containers by functions is the use of the *pair* function. This function can be defined in Haskell as:

```
pair :: t1 -> t2 -> (t1 -> t2 -> t3) -> t3
pair x y = \f -> f x y
```

The list containing the numbers *1*,*2* and *3* can now be defined as:

```
pair 1 (pair 2 3)
```

Booleans As a last example, we consider the Booleans. In Haskell we have:

```
data Boolean = True | False
```

In SAPL this becomes:

```
True x y = x
False x y = y
```

These definitions correspond exactly to the intuitive meaning of False and True. Note that switching True and False in the Haskell definition above results in counter-intuitive SAPL functions for True and False. As a convenient side effect, representing True and False by functions relieves us from the necessity to define the if-then-else construction.

3 Compiling Pattern Definitions to Function Applications

One of the intended uses of SAPL is as an intermediate language for compiling/interpreting higher functional programming languages like Haskell and Clean. Therefore we must be able to translate constructs from these languages to function applications. Constructions like list-comprehensions, *where* and *let(rec)* expressions can be handled with standard techniques like they are described

in [13] and [14]. Another important construct in these languages is the use of pattern-based function definitions. They enable the user to write concise function definitions. For the translation of these definitions we developed a new pattern compilation algorithm. New in this algorithm is the use of an intermediate data structure. This data structure can be used to generate pattern-matching code for a traditional interpreter or compiler, but also for the translation of pattern definitions to function applications. Furthermore, the use of this data structure simplifies the compilation algorithm for patterns considerably in comparison with the traditional approach (see Augustsson [1] and Peyton Jones [13]), while the results are comparable. This simplification is caused by the fact that the steps to be taken during a pattern match are made explicit in the data structure.

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}+
```

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 Description of the Intermediate Data Structure

In the pattern-based definitions we assume that left comparing is not allowed and that constant number patterns are not used. These issues are not relevant for our discussion and they can easily be added.

The essential step in our algorithm is transforming a pattern to be matched into a data structure representing the steps to be taken for matching a call. In a pattern only constructor and variable names occur. Constructor names have to be matched (recognized) and variables have to be bound to values during a pattern match. The following data structures describe this:

```
data MatchType     = MVar String RowNum RowNum |
                     MPat String RowNum [(String,Int,RowNum)]
data RowNum        = Fail | Row Int

type RowMatch      = [MatchType]
type MultiRowMatch = [RowMatch]
```

RowMatch describes the steps to be taken for a single pattern function match and *MultiRowMatch* for a multi pattern function match. The *RowMatch*'s in a *MultiRowMatch* correspond to the different cases in a multi-function pattern definition in the same order. The *String* in *MVar* represents the name of the variable. The first *String* in *MPat* represents the type name. The second *String* represents the constructor name. The *Int* represents the number of arguments

of the constructor. The first *RowNum* in *MVar* and in *[(String,Int,RowNum)]* indicates in which row to proceed after a successful match or binding. The other *RowNum* in *MVar* and *MPat* indicates in which row to proceed after a (future) failed match in the same row. If *RowNum* has the value *Fail* no match is possible and if there is no alternative available the entire pattern match will fail.

The transformation to *MultiRowMatch* of the patterns of a multi-pattern function proceeds in two steps. In the first step each case is converted to an element of *RowMatch* independently. Each *RowMatch* is capable to match a call that corresponds exactly to its case. In the second step the *RowMatch*'s are merged into a structure that is capable to match an arbitrary call in an efficient way.

3.2 Converting a Single Case Pattern

This conversion is achieved by applying a simple prefix tree walk algorithm to a pattern. For example, consider the definition of the following function f:

```
f (Cons a Nil) (Cons b bs) = a + b
```

The pattern *(Cons a Nil) (Cons b bs)* is translated to the sequence:

```
[MPat "list" Fail [("Nil",0,Fail),("Cons",2,Row 0)],
 MVar "a" (Row 0) Fail,
 MPat "list" Fail [("Nil",0,Row 0),("Cons",2,Fail)],
 MPat "list" Fail [("Nil",0,Fail),("Cons",2,Row 0)],
 MVar "b" (Row 0) Fail,
 MVar "bs" (Row 0) Fail]
```

The sequence tells us that, first *Cons* should be recognized (*Nil* will fail), then an arbitrary value is bound to variable *a*, then *Nil* must be recognized, then *Cons* should be recognized, then *b* must be bound and at last *bs* should be bound. Note that, because this *RowMatch* is for a single case function, all *RowNum*'s have the same row number.

3.3 The Merge Algorithm

The goal of merging is to minimize the number of times an argument in a call is examined. This minimization is realized by merging the *RowMatch*'s of the different cases until a state is found where a choice between cases can be made. Due to merging a match can continue at another row. When this happens it will always continue at the beginning of the other row. The extra *RowNum* in *MPat* and *MVar* indicates in which row to continue if a fail is encountered in the remainder of the current row. We will call this alternative a *fail alternative*. Fail alternatives are used for backtracking to other cases after an initial successful match in a row. For fail alternatives arguments have to be re-examined.

Merging is done by repeated merging of two sequences starting with the last two rows and working upwards to the first row. The result of a single merge is a tuple of two sequences. The first one contains the result of the merge. The second one is the continuation for the second case from the place where a choice between the two cases can be made.

```
merge :: RowMatch -> RowMatch -> (RowMatch, RowMatch)

merge (MVar varname firstrow _       : remrow1)
      (MVar _       _        failrow : remrow2)
= consfirst (MVar varname firstrow failrow)
            (merge remrow1 remrow2)

merge (MPat typename _ pats        : remrow1)
      (MVar varname varrow failrow : remrow2)
= (MPat typename varrow pats   : remrow1,
   MVar varname varrow failrow : remrow2)

merge (MVar varname (Row varrow) _ : remrow1)
      (MPat typename failrow pats  : remrow2)
= (MVar varname (Row varrow) (Row (varrow+1)) : remrow1,
   MPat typename failrow pats                 : remrow2)

merge (MPat typename _       pats1 : remrow1)
      (MPat _        failrow pats2 : remrow2)
| disjoint pats1 pats2
  = (MPat typename failrow (combine pats1 pats2) : remrow1,
     remrow2)
  = consfirst (MPat typename failrow (combine pats1 pats2))
              (merge remrow1 remrow2)

consfirst :: t1 -> ([t1],[t2]) -> ([t1],[t2])
consfirst a (as,bs) = (a:as,bs)

combine :: [(String,Int,RowNum)]->[(String,Int,RowNum)]->
           [(String,Int,RowNum)]
combine pats1 pats2 = mappair combpat pats1 pats2

combpat :: (String,Int,RowNum)->(String,Int,RowNum)->
           (String,Int,RowNum)
combpat (patname,nrargs,next1) (_,_,next2)
| next2 == Fail = (patname,nrargs,next1)
| next1 == Fail = (patname,nrargs,next2)
                = (patname,nrargs,next1)
```

Explanation The first elements of both sequences determine how to merge. We distinguish five cases:

Merging two *MVar*'s No choice can be made at this point, so the two variables are merged into one variable that gets the name of the first one (arbitrary choice) and we have to look at the following argument in both rows. A possible fail alternative from the second row is taken over.

Merging an *MPat* with an *MVar* If an actual argument matches the pattern, proceed in the first row, otherwise proceed in the second. But if the first row is chosen and at a later point a fail occurs we have to backtrack to the second row (fail alternative)!

Merging an *MVar* with an *MPat* The variable case should be chosen first and only a fail in the remainder of the variable case will cause backtracking to the pattern row (fail alternative).

Merging two *MPat*'s with excluding patterns We must continue depending on the actual argument. Thus, analysing the argument leads directly to the correct case. The function *combine* combines (unites) the transitions of both cases.

Merging two *MPat*'s with overlapping patterns We can merge the patterns like in the two variables case and we have to look at the following argument in both rows.

Remarks From the merge algorithm it follows immediately that fail alternatives only occur when merging a *var* with a *pattern* or vice versa. Furthermore, it is clear that for definitions without fail alternatives the order of the rows is irrelevant for the result of the matching process. Definitions without fail alternatives correspond with so-called uniform pattern definitions [13]. Uniform definitions lead to the most optimal solutions for pattern matching (each argument is examined only once).

3.4 Examples

We conclude our discussion about the merge algorithm with two examples. The same examples were used in Peyton Jones [13] to illustrate a pattern-matching compiler. The examples are two versions of the famous *mappair* function.

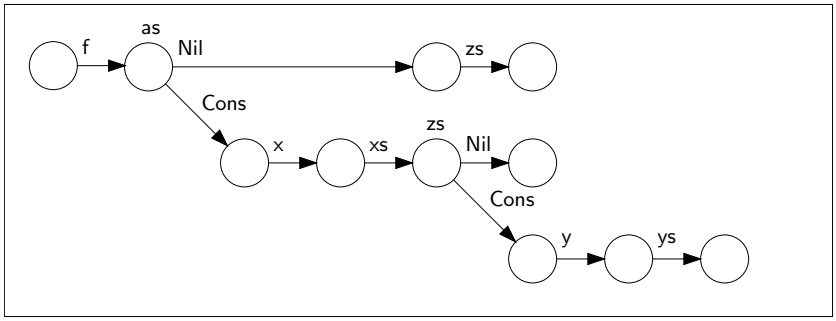

Fig. 1. Result of merge for *mappair1*

```
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)
```

In this example we see a combination of the pattern/pattern rule with and without overlap. The result of the transformation is visualised in figure 1 (the constructor node labels (*as*, *zs*) are added to enhance function generation). The result is optimal; all arguments are examined only once (there are no fail alternatives).

For the second example we use the 'inefficient' version of *mappair*. The difference with *mappair1* is the use of a variable *as* instead of the pattern *(Cons x xs)* in the second case. The result can be found in figure 2. In this figure dashed arrows are transitions caused by a fail alternative. Arrows pointing to nothing correspond with a failed match.

```
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)
```

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

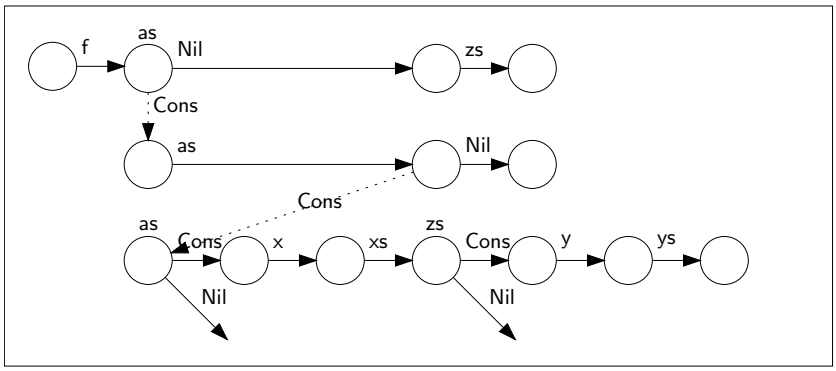

Fig. 2. Result of merge for *mappair2*

mappair2. Note that, a 'smart' pattern compiler can optimise *mappair2* in such a way that the results are equivalent to *mappair1*, because the variable *as* in *mappair2* can only be bound to a *Cons* pattern and can therefore be replaced by the pattern *(Cons x xs)* without changing the result of the match process! Note also that the *fail* arrows in the last row will never be traversed.

3.5 Generation of Functions

The last step is turning the result of pattern compilation into a function definition. The task of the function is to bind the actual arguments to variables and to feed the bounded variables to the correct case bodies. As a pre-condition we assume that all nodes are labelled with a name. Variable nodes with the name of the variable itself and constructor nodes by a generated name. Furthermore, we assume that all labels in the different rows are lined-up. This means that labels in different rows that correspond with the same (sub)argument must be the same. In the examples above this is already realized.

The arguments of the generated function now correspond to the names of the top-level arguments. The body of the function can be found by traversing the decision tree. Variable nodes can be skipped. For a constructor node we must write down the label of the node and process the descendant nodes in the order corresponding to the order used in the selector functions (e.g. for lists: first *Nil* then *Cons*). For every descendant we have to write down an anonymous

function with the same arity as the constructor and with as variable names the labels of the constructor arguments. For a fail alternative the variables can be given arbitrary names (different from all other names) because they are not used. For a fail arrow we have to write down the *fail* function. If the last node of a row is reached the corresponding body must 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)))))
```

Note that the resulting functions are very compact but still quite readable.

4 Interpretation

In the preceding sections we have shown how to transform constructs from higher functional programming languages to SAPL. To test the usefulness and efficiency of the results we developed a dedicated interpreter for SAPL. Programs in SAPL style cannot be executed by existing functional programming implementations since these programs in general do not obey the Hindley-Milner type rules.

We developed two versions of the interpreter. The first one is a straightforward interpreter for the basic version of SAPL (no lambda expressions) extended with integers and their basic operations. The second one uses knowledge about the generated functions to optimise memory overhead and execution speed.

4.1 Non-Optimised Interpretation

The interpreter is kept as simple as possible. It is based on straightforward graph reduction techniques as described in Peyton Jones [13], Plasmeijer and van Eekelen [14] and Kluge [12]. We assume that a pre-compiler has eliminated all algebraic data types and pattern definitions (as described above), all let(rec)- and where-clauses and all lambda expressions. The interpreter is only capable of executing function rewriting and the basic operations on integers. For an efficient implementation of constant *letrec* expressions we must allow for cyclic 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. 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 (default 1000000 Cells). 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.
- Is has a socket interface. This makes it is possible to interface with the interpreter from other programs. It offers a 'lazy' protocol for exchanging information.
- 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.

We now give a functional (and executable) specification of the interpreter. The Haskell definition of the main data structure is given by:

```
data Expr = App Expr Expr| Func Int Int| Var Int | Num Int |
            Oper (Int->Int->Int) | RelOp (Int->Int->Bool)
```

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. The *eval* function using extended SAPL 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)
eval (Oper op)      es fs = apply op (eval (el 0 es) Nil fs)
                                     (eval (el 1 es) Nil fs)
eval (RelOp op)     es fs = eval (apprel op (eval (el 0 es) Nil fs)
                                            (eval (el 1 es) Nil fs))
                                 (drop 2 es) fs
eval (Num n)        es fs = Num n

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

apply op  (Num n) (Num m) = Num (op n m)
apprel op (Num n) (Num m) = if (op n m) (Func 2 0) (Func 2 1)
```

Here *Func 2 0* and *Func 2 1* represent *True* and *False*, *es* the stack and *fs* the list of function definitions. In pseudo code *eval*, without numerals, is given by:

```
Cell eval(Cell top) {
  Cell e = top; int nr_args = 0;
  repeat {
    typeof(e) {
      App:      push(e->right); e = e -> left; nr_args++;
      Function: if (nr_args >= nrArgs(e)) {
                   e = instantiate(body(e));
                   updateroot(e);
                   pop(nrArgs(e)); nr_args -= nrArgs(e); }
                else {pop(nr_args);return top;}
    }
  }
}
```

The actual *eval* function fits on less than one A4! *Instantiate* recursively copies the body of the function and substitutes references to the arguments on the stack for the variables in the body. In this way sharing is realized. The result of *instantiate* overwrites the top node of the application. A numeric operation simply evaluates the top two elements on the stack, retracts the results from them, applies the operation and overwrites the top application node with the result. In spite of its simplicity, this interpreter has already reasonable performance. In the next section we will say something about its performance in comparison with the optimised interpreter.

4.2 Using Selective Instantiation to Optimise the Interpreter

There are many options for optimising the interpreter. Here we are only interested in optimisations that deal with the way algebraic data types and pattern definitions are translated. If we take a careful look at the translated functions we often encounter definitions of the following kind:

```
fname patarg arg1 ... argn = patarg case1 ... casem
```

Here *patarg* is a variable of an algebraic data type (selector function) with m cases. The actual selector function selects one of the m cases and applies it to its own arguments. The other $m-1$ arguments are not used. A large part of the body of the function *fname* is therefore copied and not used. This holds especially for data types with many cases, for example, data types that represent expressions. Because these functions are generated by the pattern compiler we know exactly which functions are of this type. We therefore modify the generator in such a way that it will generate:

```
fname patarg arg1 ... argn = case patarg case1 ... casem
```

The extra function *case* is semantically equal to the identity function but acts as a compiler directive. When the interpreter encounters a function call to a function with a body that is tagged with *case*, it will not copy the entire body of the function but evaluates only the left most element in the application (*patarg* in the example). The interpreter is modified in such a way that when evaluating a function call to a pattern function as a side effect it also returns its case

number (e.g. *0* for *Nil* and *1* for *(Cons x xs)*). This case number is used to select and copy the appropriate part of the function body and applying it to the constructor arguments. We will call this optimisation Selective Instantiation. In this way a large reduction in copying overhead is realized. Because copying of function bodies makes up a significant part of the time the interpreter uses, the speed-up gained by this optimisation can be large (see next section). We will call the optimised interpreter also SAPL.

5 Comparison with other interpreters

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 [9], Hugs 20050113 [11] and GHCi V6.4 [8]. We used three programs for the comparison:

- The prime number sieve program. This is a list intensive program making heavily use of lazy evaluation during the manipulation of infinite lists.
- A small evaluator/interpreter for a minimal functional programming language. This example makes heavily use of pattern matching for a data type with many cases.
- A symbolic version of the prime number sieve using Peano numbers instead of integers. In this example pattern matching and function rewriting are the only operations.

The code for the prime number generator is given by:

```
sieve (x:xs) = x : sieve (filter (nmz x) xs)
nmz x y      = mod y  x != 0
primes       = sieve (from 2)
```

The interpreter is an eager version (it evaluates arguments before pushing them on the stack) of the functional interpreter from section 4. Note that the resulting interpreter is still lazy, because SAPL itself is lazy. The interpreter is rather inefficient because function definitions have to be looked up in a list every time a call for a function has to be instantiated. For the test we coded the prime number sieve for it using the data types from section 4.

The program for the symbolic prime number sieve is the same as the prime number sieve above using the Peano numbers and their operations from section 2 instead of integers.

For all tests we did not use the built-in lists, but user-defined lists instead. The tests were done on a Pentium 4 2.66 GHz machine. We experimented with different heap sizes during the tests. Only Helium showed significant differences between the results. The final tests were done with a heap size of about 8Mbyte (giving optimal results for Helium). The results can be found in figure 3.

For the prime number generator the test we conducted was: Calculate the *4000th* prime number. Note that Hugs is a factor 15-20 slower than the other interpreters. Non-optimised SAPL was about 15% slower than optimised SAPL.

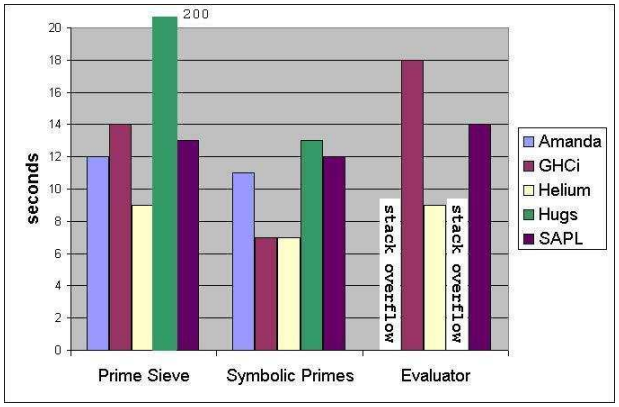

Fig. 3. Performance Measures

For the evaluator we coded the generation of the 80th prime number. For this test Amanda and Hugs had a stack-overflow. Using smaller examples Amanda had results comparable to Helium, Hugs was 3-5 times slower than the others. Non-optimised SAPL was about 8 times slower than optimised SAPL! This shows that the optimisation due to the *case* directive is significant and therefore necessary.

The test for the symbolic prime number generator was: Calculate the 200th prime number. This is the only example for which Hugs has competitive performance. This shows that Hugs has probably very inefficient handling of arithmetic expressions.

From the comparisons we may conclude that, despite its simplicity, SAPL has already acceptable performance.

5.1 Further Optimisations

Besides the optimisation due to the *case* directive SAPL uses no other optimisations. What are the possibilities for further improvements in performance?

First of all, the memory usage (and therefore execution speed) of SAPL can be reduced by using a more efficient coding of binary operations (like *add, sub*) that are used uncurried (using 3 cells instead of 5).

Also SAPL uses a check for the number of elements on the stack for every function call. In many cases this check can already be done at compile time.

These optimisations are already implemented in most of the above-mentioned interpreters. Some first experiments showed us that for SAPL a speed-up of at least 30% is possible.

6 Conclusions and further Research Possibilities

In this paper we have defined a minimal functional programming language SAPL. This language has function application as only operation and has no data structures. For SAPL we have achieved the following results:

- It is possible to represent data structures as functions in SAPL in a natural and data structure look alike way. Programs written in 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 an efficient interpreter for SAPL using Selective Instantiation. Due to the simplicity of the language the interpreter can be kept small and elegant. After adding integers and their operations the performance of the interpreter is comparable to state-of-the-art interpreters, while there is room for further performance improvements.

Recursive functions over data types cannot be typed using Hindley-Milner typing due to the lack of constructors. The Hindley-Milner type system assigns fixed types to constructors. SAPL has no constructors and hence these types are not assigned. We plan to develop a type system for SAPL. Such a type system will also indicate the spots where the optimisation of section 4.2 can be applied.

Finally we want to investigate whether the introduced implementation techniques can be used in a compiler.

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/amanda204.zip.
7. The Clean Home Page. Software Technology Research Group, Radboud University Nijmegen, the Netherlands, www.cs.ru.nl/~clean.
8. The Haskell Home Page, www.Haskell.org.
9. The Helium Project. Software Technology group, the Institute of Information and Computing Sciences, Utrecht University, the Netherlands, www.cs.uu.nl/helium.
10. J.R. Hindley and J.P. Seldin. Introduction to Combinators and λ–Calculus. *London Mathematical Society Student Texts*, 1986.
11. Hugs Online, www.Haskell.org/hugs.
12. W. Kluge. Abstract Computing Machines. *Springer-Verlag, Texts in Theoretical Computer Science*, 2004.
13. S.L. Peyton Jones. The Implementation of Functional Programming Languages. *Prentice-Hall International Series in Computer Science*, 1987.
14. R. Plasmeijer and M. van Eekelen. Functional Programming and Parallel Graph Rewriting. *Addison-Wesley, International Computer Science Series*, 1993.