From Interpretation to Compilation

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

[1] Netherlands Defence Academy,
Faculty of Military Sciences,
Den Helder, The Netherlands
jm.jansen.04@nlda.nl
[2] Institute for Computing and Information Sciences (ICIS),
Radboud University Nijmegen, The Netherlands
{pieter,rinus}@cs.ru.nl

Abstract. In this paper we sketch some experiments with the construction of a simple compiler for a high level intermediate lazy functional language, with C++ as a target language. Because the compiler is intended for educational and experimental use, simplicity and clearness of construction are considered to be more important than efficiency. Starting point for the construction is a simple interpreter. In a first step this interpreter is turned into a simple compiler in a straightforward manner. The performance of a number of compiled benchmarks is analysed in a comparison with the interpreter and the Clean and GHC compilers. This analysis leads to some suggestions for optimisations. Of these optimisations tail recursion optimisation and optimisation of numerical functions and numerical (sub)expressions in functions are implemented. It turns out that in many cases these optimisations suffice to obtain a competitive performance.

1 Introduction

The construction of efficient compilers for lazy functional programming languages like Clean [8] and Haskell [1] is a complex task. Compilers like GHC and Clean are large complicated systems that are too complex for study in introductionary courses on the implementation of functional programming languages. Therefore there is a need for simple compilers for educational purposes. Our main goal is to give the reader some insight in what kind of optimisations are important for obtaining an efficient implementation of lazy functional languages.

In [2] we constructed a simple but efficient interpreter for the lazy functional language SAPL. SAPL can be used as an intermediate language for the interpretation of languages like Clean and Haskell. We already constructed a Clean to SAPL translator. Several versions of the SAPL interpreter exist. One of these versions is a Java applet implementation that can be loaded in Internet Browsers and which makes it possible to run Clean programs at the client side of internet applications ([6] and [7]).

In this paper we investigate how we can extend the SAPL interpreter to a SAPL compiler with a reasonable performance. We use C++ as target language.

Z. Horváth et al. (Eds.): CEFP 2007, LNCS 5161, pp. 286–301, 2008.

The construction is made in two steps. In the first step we convert the interpreter into a straightforward but naive compiler. We then use a number of benchmarks to analyse the performance of the generated code in a comparison with the Clean and GHC compiler. It turns out that in some cases the performance is already quite good but that in other cases the performance is still very bad (more than 30 times slower). In an analysis of the characteristic of the poor performing benchmarks, it turns out that they often have some commonalities like the (heavy) use of tail recursive functions and the presence of many pure numeric functions or sub-expressions. Therefore, in the second step, we focus on improving the performance of the compiler by optimising tail recursions and numeric functions and sub-expressions. The resulting compiler is again compared with Clean and Haskell and the basic compiler using the same set of benchmarks. It turns out that the resulting performance is now acceptable in almost all cases.

Summarising, the contributions of this study are the stepwise construction of a simple compiler for a lazy (intermediate) functional programming language with the following characteristics:

- The compilers translates to concise and readable C++ functions (for a functional programmer knowing C++) that are in 1-1 correspondence with the original functions. The C++ functions give the programmer clear insight in how constructs from functional programming language are implemented.
- It gives the reader insight in what kind of optimisations are important for obtaining an efficient implementation of lazy functional languages.
- The user can easily add functions to the generated code and can modify generated functions to experiment with alternative optimisations.
- The performance of the resulting programs is in many cases competitive with that of Clean and Haskell.

The structure of this paper is as follows. In Section 2 we introduce the intermediate functional programming language **SAPL**. In Section 3 we sketch an interpreter for SAPL. This interpreter is the starting point for the construction of the compiler. The compiler is described in Section 4. We describe the compiler in a number of steps. First a basic version of the compiler is introduced that is a straightforward and simple extension of the interpreter. The performance of a set of benchmarks compiled with this compiler and the Clean and GHC compiler is used to make a comparison. The results of this comparison are analysed and this leads to the proposal of a number of candidate optimisations that are implemented. In the last section we give some conclusions.

2 The SAPL Programming Language

SAPL stands for **S**imple **A**pplication **P**rogramming **L**anguage. The basic version of SAPL has function application as only operation. SAPL is a simple functional programming language that can be used as an intermediate formalism for the interpretation of functional programming languages like Haskell and Clean. The main difference between SAPL and the intermediate formalisms normally used

for these languages is the absence of algebraic data types and constructs for pattern matching in SAPL. This makes SAPL a compact and simple language. More details about SAPL can be found in [2].

In [2] we also showed how to represent data types and pattern-based function definitions in SAPL. Here we shortly repeat the definition of the list data type together with the *length* function.

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

Now consider a pattern based Haskel function like *mappair*.

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

This definition can be transformed to the following SAPL function (using the above definitions of *Nil* and *Cons*).

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

3 An Interpreter for SAPL

The only operations in SAPL programs are function application and a number of (build-in) integer operations. Therefore an interpreter can be kept small and elegant. The interpreter is based on straightforward graph reduction techniques as described in Peyton Jones [4], Plasmeijer and van Eekelen [5] and Kluge [3]. We assume that a pre-compiler has eliminated all algebraic data types and pattern definitions (as described earlier), removed all let(rec)- and where- clauses and lifted all lambda expressions to the global level. Only constant let-expressions are allowed to enable sharing and cyclic 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 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 mark and 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).

- 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 pushes arguments on the stack until a function call is met. In that case the function body is instantiated while the arguments are substituted, the top application node is overwritten and evaluation continues on the new expression until we arrive at a curried call or an integer value.

3.1 Optimisations in the Interpreter

The interpreter can be optimised in several ways. Simple optimisations are the use of a more efficient memory representations of function calls with 1 or 2 arguments and the marking of curried calls (if possible) to avoid the useless evaluation of them. Applying these optimisations result in speed-ups up to 50%.

A more significant optimisation can be realized by marking the application of a function representing an algebraic data type element to its arguments by the keyword *select* (semantically equivalent to the identity function). This triggers the interpreter not to instantiate the entire function body at once, but first to evaluate the data type and only *select* and instantiate the relevant part of the remainder expression (more details can be found in [2]).

As a last optimisation, anonymous functions that are the argument of a *select* are not lifted to the global level, but are called inline (see [2]).

As an example we show how the *select* optimisation is applied in the *mappair* function (the lambda expressions in this example are not lifted to the global level).

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

The *select* optimisation is essential and may result in speed-ups of more than 100 times. Normally the *select* annotations are added while translating Haskell or Clean programs to SAPL, but it is possible to add the *select* annotations during a compile time analysis of a SAPL program. During this analysis it is determined where applications of data type functions to other arguments occur. This analysis can only be performed in case of complete programs and not for separately compiled files (modules). For example, if we consider the definition of *mappair* in isolation it is not clear that *as* and *zs* are *selectors*. One needs an example of the usage of *mappair* to determine that.

3.2 Considerations

The interpreter without the *select* optimisation and the integer operations is a pure graph reductor. The only operations are graph reduction (push arguments

on the stack until a function call is met) and graph instantiation (copy a function body and meanwhile substitute the arguments from the stack).

Numeric operations are strict in the sense that the arguments have to be evaluated before the operation can be performed. The same holds for the *select* optimisation. Also in this case the first argument of *select* has to be evaluated before the operation (selection of the appropriate argument) can take place. The optimisation prevents the instantiation of large graphs. In the remainder of this paper we show that many of the optimisations we implement in the compiler involve the use of strictness to prevent the instantiation of unnecessary graphs.

4 A SAPL Compiler

We present two versions of the compiler: a basic version and an optimised version. The optimisations are a result of an analyses of the performance of the basic version for a number of benchmarks.

The benchmarks we use for the comparison are the same we used for comparing the SAPL interpreter with several other interpreters and compilers in [2]. We briefly repeat the description of the benchmarks (their code can be found in [9]):

1. **Prime Sieve.** The prime number sieve program (*primes !! 5000*).
2. **Symbolic Primes.** Prime sieve using Peano numbers (*sprimes !! p280*).
3. **Interpreter.** A small SAPL interpreter. 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), repeated 2000000 times.
6. **Hamming.** The generation of the list of Hamming numbers (a cyclic definition) and taking the 1000th Hamming number, repeated 10000 times.
7. **Twice.** A higher order function (*twice twice twice twice (add 1) 0*), repeated 400 times.
8. **Queens.** Number of placements of 11 Queens on a 11 * 11 chess board.
9. **Knights.** Finding all Knight tours on a 5 * 5 chess board.
10. **Parser Combinators.** A parser for Prolog programs based on Parser Combinators parsing a 17000 lines Prolog program.
11. **Prolog.** A small Prolog interpreter based on unification only (no arithmetic operations), calculating all descendants in a six generations family tree.
12. **Sorting.** Quick Sort (20000 elements), Merge Sort (200000 elements) and Insertion Sort (10000 elements).

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.

We use C++ as a target language for our compiler. We do not use the object oriented properties of C++ (classes and member functions). But we use some specific features of C++ like reference variables. In all versions of the compiler

there is a one-to-one correspondence between SAPL and C(++) functions. Because we want to use the compiler for educational purposes we strive at readable and understandable generated code.

The generic structure of a translated function is:

int funcname(*Reduct t*) { *instantiate_body*; *return eval_body*; }

Here *funcname* is the name of the translated SAPL function. We assume that all arguments of a function are already on the stack when the function is called. The argument t of the function is a reference to the top node of the call for this function. To enable sharing we have to overwrite this top node with the result of the function. The function returns an integer. This is because functions that result in an algebraic data type have to return the selection number needed in a *select* construction. Because we want to use the same type signature for all functions, all functions have to return an integer. Note that we cannot give the C function the same arguments as the original function because we can make curried calls to a function which is, of course, not possible in C.

4.1 A Basic SAPL Compiler

If we take a closer look at the SAPL interpreter, the most obvious candidate for compilation is the instantiation of function bodies. The interpreter uses a recursive function *instantiate* to copy the body and substitute the arguments. It is straightforward to generate C++ code that does this instantiation directly.

Due to the *select* optimisation the body of a function containing a *select* is not copied at once but in parts. Therefore, in the translation to C++, we add the control structure (using *if* or *switch/case* statements) to enable this copying in parts. Also the generation of this control structure is entirely straightforward.

Examples. As an example consider the translation of the functions *sieve* and *el* from the prime number sieve program.

sieve xs =*cons* (*hd xs*) (*sieve* (*filter* (*nmz* (*hd xs*)) (*tl xs*)))
el n xs =*select xs error* (λ *a as* $\rightarrow$ *if* (*eq n* 0) *a* (*el* (*sub n* 1) *as*))

The translation of *sieve* results in:

```
int sieve(Reduct t) {
  testmem();
  setCell(t,SELB,newR(OPFUNC,get(0),0,9),newR(OFUNC,
     newR(BPFUNC,newR(OPFUNC,newR(OPFUNC,get(0),0,9),0,7),
     newR(OPFUNC,get(0),0,10),3),0,5),2);
  pop(1);
  return eval(t);
}
```

testmem() checks if garbage collection is necessary. This check is done before every body instantiation. *setCell(t,...)* overwrites *t*. Although the *setCell* call

looks quite complicated the only thing that is happening here is the allocation of a new graph in memory. Due to the memory optimisations for applications with one and two arguments and the marking of curried applications there are a large number of cell types (*SELB*, *OPFUNC*, etc.). *get(i)* returns a reference to the *i-th* element on the stack. *pop(i)* removes i elements from the stack. In the last line *eval(t)* recursively starts evaluating the resulting expression. The only thing the *eval* function does is pushing arguments on the stack and calling the resulting function.

The translation of *el* results in:

```
int el(Reduct t) {
  Reduct res = get(1);
  if(eval(res)) {
    pushs(res->r); pushs(res->l);
    testmem();
    res = newR(BINOPER,get(2),newR(NUM,Reduct(0),0),5);
    if(eval(res)) {
      testmem();
      setCell(t,BPFUNC,newR(BINOPER,get(2),
                      newR(NUM,Reduct(1),0),1),get(1),4);
      pop(4);
    }
    else {overwrite(t,get(0)); pop(4);}
  }
  else    {setCell(t,SFUNC,0,Reduct(0),0);   pop(2);}
  return eval(t);
}
```

In this example we see that the control structure of the original function is clearly reflected in the C++ function. In the first line *xs* is assigned to *res*. *res* is evaluated. In case the result is a *cons* (returns 1) the arguments of *cons* are pushed on the stack. Next the expression *eq n 0* is instantiated and evaluated. If *n != 0* the expression *el (sub n 1) xs* is instantiated and the stack is cleared. In case $n == 0$, t is overwritten with x. Also in this case the stack is cleared. The last *else* handles the case that the list was *nil*.

We conclude that the basic compiler results in concise code that clearly reflects how the graph reduction process is conducted. For a function acting on a data structure with 3 or more cases a C++ *switch* statement is generated. The adaptations to the interpreter needed to generate the C++ functions are modest. An interesting aspect is that the resulting C++ functions are integrated in the interpreter environment. The only difference for the user is the increase in speed (and an extra compilation round before starting the interpreter).

Although the Basic Compiler compiles to C++, it is essentially still an interpreter. The way graphs are reduced is the same as in the original interpreter.

In the remainder of this paper we sometimes abbreviate the instantiation of graphs with: `instantiate('expression')` or `overwrite(t,'expression')`.

	Pri	Sym	Inter	Fib	Match	Ham	Twi	Qns	Kns	Parse	Plog	Qsort	Isort	Msort
SAPL Int	6.1	17.6	7.8	7.3	8.5	15.7	7.9	6.5	47.1	4.4	4.0	16.4	9.4	4.4
SAPL Bas	4.3	13.2	6.0	6.5	5.9	9.8	5.6	5.1	38.3	3.8	2.6	10.1	6.7	2.6
GHC	2.0	1.7	8.2	4.0	4,1	8.4	6.6	3.7	17.7	2.8	0.7	4.4	2.3	3.2
GHC -O	0.9	1.5	1.8	0.2	1.0	4.0	0.1	0.4	5.7	1.9	0.4	3.2	1.9	1.0
Clean	0.9	0.8	0.8	0.2	1.4	2.4	2.4	0.4	3.0	4.5	0.4	1.6	1.0	0.6

Fig. 1. Comparison Speed of Basic Compiler (Time in seconds)

4.2 Performance of the Basic Compiler

In Fig. 1 we compare the performance of the basic compiler with that of the interpreter and of the GHC and Clean compilers. If we compare the basic compiler with the interpreter we see that the basic compiler is about 40% faster (speed-ups between 10 and 60%).

If we compare the basic compiler with GHC (without optimiser) we see that in three cases (*Interpreter*, *Mergesort* and *Twice*) the basic SAPL compiler is already faster. In the other cases GHC is mostly less than 2 times faster. Relatively slow SAPL benchmarks are *Symbolic Primes* (7 times) and *Prolog* (3.7 times).

Comparing the basic compiler with GHC -O and Clean we measure large differences in performance, varying from 10% faster (compared to *Parser Combinators* in Clean) to more than 30 times slower (*Fibonacci* for Clean, GHC -O and *Twice* for GHC -O).

4.3 Analysis of Basic Compiler

Compared with GHC (without optimiser) the Basic Compiler is already doing a reasonable job. The only poor performing benchmark is *Symbolic Primes*. This is an a-typical program, because there is no integer arithmetic in this example and the functions bodies are all very small. For SAPL this means a lot of interpretation overhead. More important, the performance dominating functions *Mod* and *Subtract* are tail recursive. In the sequel we show that, using tail recursion optimisation, the performance of this benchmark can be improved significantly.

If we take a closer look at the benchmarks for the comparison with GHC -O and Clean, we see that there is only one benchmark that performs good in this comparison: *Parser Combinators*. This is the most 'functional' of all benchmarks in the sense that it manipulates mostly higher order functions. For a compiler this means that a lot of closures must be maintained. Closures are represented by structures comparable to the graphs in SAPL. Every compiler should analyse (destruct) these closures at a certain moment in a way similar to the way the Basic SAPL compiler does this.

The worst performing benchmarks are: *Symbolic Primes*, *Fibonacci*, *Queens* and *Twice*.

- **Symbolic Primes** we already discussed above. It contains a number of tail recursive functions for which SAPL does no optimisations yet.

- **Fibonacci** is a pure numeric function (numeric arguments and numeric operations only). In SAPL every time the function is called in the recursion, a complete instantiation of the function body is made (on the heap). The Clean and GHC -O compilers optimise this function and do not use closures but instead only use the stack to execute it.
- **Queens** has a number of numeric sub-expressions and has a (hidden) tail recursion in function *safe*. Also in this case Clean and GHC -O use strictness analysis to eliminate the building of many closures.
- **Twice** is a special case. GHC -O has a much better performance than both SAPL and Clean. If we study the generated code for GHC -O we see that some very specific inline optimisations are made. We did not make any special optimisations for this example.

Conclusions and Plan for Optimisations. The basic compiler has already a nice performance for programs manipulating mostly higher order functions. Therefore, we may expect that the poorer performance is caused by the overhead involved in building instantiations (closures) that are not really necessary. The optimisations we apply are aimed at either preventing the building of closures or at building smaller closures. In the light of the discussion above we focus on tail recursive functions and on numeric functions and (sub)expressions, also because they can be recognized and optimised easily. But before that we look at some straightforward optimisations.

4.4 Reducing the Size of Closures and Removal of Interpretation Overhead

Consider the following function g:

$$g\ a\ b\ c\ d\ =\ f\ a\ (h\ b\ c)\ d$$

In the basic compiler this is compiled to:

```
int g(Reduct t) {
  testmem();
  setCell(t,APP,newR(APP,newR(APP,newR(FUNC,0,0,2),get(0)),
            newR(BFUNC,get(1),get(2),1)),get(3));pop(4);
  return eval(t);
}
```

In the body of g a large instantiation is build for which *eval* is called immediately. *eval* pushes the arguments of f on the stack and calls the function *f*. But if we already know this, we can hard code the pushing of the arguments and the call to *f*. In this way we both save instantiation and interpretation overhead.

```
int g(Reduct t) {
  testmem();
  Reduct a0,a1,a2;
  a0 = get(0);
  a1 = newR(BFUNC,get(1),get(2),1);
```

```
  a2 = get(3);
  pop(4);
  pushs(a2);pushs(a1);pushs(a0);
  return f(t);
}
```

In this example the number of allocated nodes is reduced from 4 to 1!

We apply this optimisation whenever possible. This means that an, at compile time, known function should be called with enough arguments.

4.5 Numerical Functions and Expressions

If a function has numeric arguments only and its body is a pure numerical expression we can avoid the creation of closures altogether. Consider for example the Fibonacci function:

$$\mathit{fib}\ n\ =\ \mathit{if}\ (n\ <\ 2)\ 1\ (\mathit{fib}\ (n\ -\ 1)\ +\ \mathit{fib}\ (n\ -\ 2))$$

The Basic SAPL compiler translates this to:

```
int fib(Reduct t) {
  Reduct res;
  testmem();
  res = newR(BINOPER,newR(NUM,Reduct(2),0),get(0),7);
  if(eval(res)) {
    testmem();
    setCell(t,BINOPER,newR(OPFUNC,newR(BINOPER,get(0),
                                  newR(NUM,Reduct(1),0),1),0,35),
                      newR(OPFUNC,newR(BINOPER,get(0),
                      newR(NUM,Reduct(2),0),1),0,35),0);
    pop(1);
  }
  else {
    setCell(t,NUM,Reduct(1),0);
    pop(1);
  }
  return eval(t);
}
```

In the optimised translation *fib* is translated to:

```
int fibh(int n) {
  if (n < 2) return 1;
  else return fibh(n-1) + fibh(n-2);
}

int fib(Reduct t) {
  eval(get(0));
  setCell(t,NUM,Reduct(fibh(getNum(get(0)))),0);
  pop(1);
  return 0;
}
```

fibh is a pure C++ function without any instantiations of cells and *fib* is a wrapper function for calling *fibh* from a functional context. The speed-up obtained in this way is more than 30 times. This version of *fib* now has a performance comparable to that of Clean and GHC -O.

Numerical expressions with a Boolean result. A special case of numeric expressions are those with a Boolean result. They often occur in the condition of an *if* statement. The *el* function we studied already before is an example of such a function. Using the numeric expression optimisation the compiled function becomes:

```
int el(Reduct t) {
  Reduct res = get(1);
  if(eval(res)) {
    pushs(res->r); pushs(res->l);
    eval(get(2));
    if(getNum(get(2) == 0){overwrite(t,get(0)); pop(4);}
    else {
      testmem();
      setCell(t,BPFUNC,newR(BINOPER,get(2),
                        newR(NUM,Reduct(1),0),1),get(1),4);
      pop(4);
    }
  }
  else    {setCell(t,SFUNC,0,Reduct(0),0);   pop(2);}
  return eval(t);
}
```

This saves allocation and interpretation overhead.

4.6 Optimising Tail Recursion Functions

Replacing tail recursions by while loops are a common optimisation also applied for strict functional and imperative languages. In these cases the optimisation is used to eliminate calling and stack overhead. But in the lazy functional context we have an extra benefit. Also the building of a closure (and the destruction of it) for the recursive call is prevented. Therefore, the speed-up is even higher.

Simple tail recursive functions have the form:

$$f\ a\ arg = if\ (cond\ a)\ (default\ a\ arg)\ (f\ (dec\ a)\ (update\ a\ arg))$$

The recursion runs over *a*. For the sake of simplicity we assume that there is only one other argument. The function contains a simple *if* construction at the top level. In the *else* case the same function is called with an *a* argument that is in some way smaller than the original argument. We compile this function to a C++ function containing a while-loop.

```
int f(Reduct t) {
  Reduct res  = instantiate('cond a');
  Reduct &a   = get(0);
```

```
  Reduct &arg = get(1);
  while(eval(res)) {
      arg  = instantiate('update a arg');
      a    = instantiate('dec a');
      res  = instantiate('cond a');
    }
  overwrite(t,'default a arg'); pop(2);
  return eval(t);
}
```

Note that we use reference variables for *a* and *arg*, so they remain on the SAPL stack, which is necessary for garbage collection purpose. In the while loop we instantiate the new versions of the arguments and the condition. The while condition determines if the recursion is finished. Because the arguments of the tail recursion are maintained by variables we can easily optimise numeric or Boolean arguments (see Subsection 4.5). As an example, consider the function *length* (note the use of an accumulating parameter).

$$length\ n\ xs\ =\ select\ xs\ n\ (\lambda\ a\ as\ \rightarrow\ length\ (n\ +\ 1)\ as)$$

This function is translated to:

```
int length(Reduct t) {
  eval(get(0));
  int n = getNum(get(0));
  Reduct &xs = get(1);
  while(eval(xs)) {
    n = n + 1;  xs = xs -> r;
  }
  overwrite(t,newR(NUM,Reduct(n),0)); pop(2); return 0;
}
```

Here the argument *n* is numerical and therefore assigned to the *int* variable *n*. The expression *n+1* is not instantiated, but directly translated to C. This saves an instantiation and a reduction. After the while loop we have to wrap the numeric result in a cell.

Note that this function also does not build the large closure *0+1+1+1+..* that is only evaluated at the end, which happens in the SAPL interpreter and the Basic Compiler. In this way a basic form of strictness analysis is realized. Furthermore, there is another optimisation. The arguments of *Cons* are not pushed on the stack, but can be found as the left and right child of *xs*. In the while loop of this function no instantiations are made.

A tail recursion may also runs over several arguments. In that case the condition is a conjunction of all the conditions. As an example, consider the following definitions of *Zero* and *Suc* and the tail recursive function *Sub* running over 2 arguments, all occurring in the *Symbolic Primes* benchmark:

$$Zero\ f\ g\ =\ f$$
$$Suc\ n\ f\ g\ =\ g\ n$$
$$Sub\ m\ n\ =\ select\ n\ m\ (\lambda\ pn\ \rightarrow\ select\ m\ Zero\ (\lambda\ pm\ \rightarrow\ Sub\ pm\ pn))$$

Sub is translated to:

```
int Sub(Reduct t) {
  Reduct &m = get(0);
  Reduct &n = get(1);
  while(eval(n) && eval(m)) {
    m = m -> 1;
    n = n -> 1;
  }
  if(eval(n)) {
    overwrite(t,'Zero');pop(2);return 0;
  }
  else {
    overwrite(t,'m');pop(2);return eval(t);
  }
}
```

Note that after the while we have 'to check' why the loop stopped to return the result of the right stopping case. Note also that we made use of the fact that the && operator in C++ is conditional (lazy). Again, no instantiations are made in the while loop.

Tail recursion that run over 3 or more variables are handled in a similar way.

Hidden Tail Recursions. Sometimes a function can be easily converted to a tail recursion. For example in the *safe* function used in the *Queens* benchmark an *and* condition with a recursive call to *safe* itself occurs.

$$\begin{aligned} safe\ xs\ d\ x\ =& select\ xs\ True \\ & (\lambda\ y\ ys\ \rightarrow\ \ and\ (and\ (neq\ x\ y)\ (neq\ (add\ x\ d)\ y)) \\ & \qquad (and\ (neq\ (sub\ x\ d)\ y)\ (safe\ ys\ (add\ d\ 1)\ x))) \end{aligned}$$

safe is translated to:

```
int safe(Reduct t) {
  Reduct xs = get(0);
  eval(get(1)); eval(get(2));
  int d = getNum(get(1));
  int x = getNum(get(2));
  int y;
  while(eval(xs) && (eval(xs -> l),y = getNum(xs -> l),x != y) &&
                    (x + d != y) && (x - d != y)) {
    xs = xs -> r;
    d = d + 1;
  }
  if (eval(xs)) {
      setCell(t,FALSE,0,0);
      pop(3);
      return 1;
    }
```

```
    else {
      setCell(t,TRUE,0,0);
      pop(3);
      return 0;
  }
}
```

Also in this case we make use of the conditionality of the && operator in C++.

4.7 Results and Discussion

Figure 2 gives the results of the comparison of the optimised compiler with the other compilers and the Interpreter. We see that the optimisations result in a significant speed-up in almost all cases. We briefly discuss the speed-up obtained for the benchmarks.

1. **Prime Sieve.** Speed-up 1.65: numeric optimisations and a tail recursion in *elem*.
2. **Symbolic Primes.** Speed-up 7.3: tail recursions in functions *Mod*, *Gt*, *Neq* and *Sub*.
3. **Interpreter.** Speed-up 1.82: tail recursions in *length*, *drop* and *elem* and several small numeric optimisations.
4. **Fibonacci.** Speed-up 33: pure numeric function.
5. **Match.** Speed-up 1.9: numeric optimisations.
6. **Hamming.** Speed-up 1.66: small numeric optimisations.
7. **Twice.** Speed-up 1.24: small numeric optimisations.
8. **Queens.** Speed-up 5.7: tail recursion in *safe* and several numeric optimisations.
9. **Knights.** Speed-up 2.1: numeric optimisations.
10. **Parser Combinators.** Speed-up 1.3: small numeric optimisations and minor tail recursions.
11. **Prolog.** Speed-up 2.0: tail recursions in several (minor) functions and some numeric optimisations.
12. **Sorting.** Quick Sort (1.7), Merge Sort (2.2) and Insertion Sort (2.7): numeric optimisations.

	Pri	Sym	Inter	Fib	Match	Ham	Twi	Qns	Kns	Parse	Plog	Qsort	Isort	Msort
SAPL Int	6.1	17.6	7.8	7.3	8.5	15.7	7.9	6.5	47.1	4.4	4.0	16.4	9.4	4.4
SAPL Bas	4.3	13.2	6.0	6.5	5.9	9.8	5.6	5.1	38.3	3.8	2.6	10.1	6.7	2.6
SAPL Opt	2.6	1.8	3.3	0.2	3.1	5.9	4.5	0.9	18.0	2.9	1.3	6.0	2.5	1.2
GHC	2.0	1.7	8.2	4.0	4,1	8.4	6.6	3.7	17.7	2.8	0.7	4.4	2.3	3.2
GHC -O	0.9	1.5	1.8	0.2	1.0	4.0	0.1	0.4	5.7	1.9	0.4	3.2	1.9	1.0
Clean	0.9	0.8	0.8	0.2	1.4	2.4	2.4	0.4	3.0	4.5	0.4	1.6	1.0	0.6

Fig. 2. Comparison Speed of Optimized Compiler (Time in seconds)

Even for the higher order examples *Twice* and *Parser Combinators* there is a (small) speed-up due to the numeric optimisations. The greatest speed-up is obtained for the *Fibonacci* benchmark. An interesting speed-up is obtained for the *Symbolic Primes* benchmark. This result could be obtained because the functions *Mod* and *Sub* are tail recursive and dominate the performance of the benchmark. Also for *Queens* a high speed-up is obtained because the tail recursive *safe* function dominates the performance.

Compared with GHC the optimised compiler is faster in almost all cases. Only for *Primes*, *Prolog* and *QSort* GHC is slightly faster. For *Fibonacci*, *Interpreter*, *Queens* and *Mergesort* the optimised SAPL compiler is much faster (more than 2.5 times).

Compared with GHC -O we see that only for *Twice* GHC -O is an order of magnitude faster (45 times). The GHC -O optimiser recognizes the repetition in this higher order function and replaces it with an iteration. Note that GHC -O is also much faster than Clean in this case. In all other cases the difference is less than 3 times and in several cases SAPL is even competitive. On the average the difference in performance stays within a factor of 2.

Compared with Clean we see that the greatest difference in performance stays within a factor of 6 (*Knights*). On the average Clean is about 2.5 times faster. For *Parser Combinators* the SAPL compiler is faster (1.5 times).

Considering only the more realistic applications (*Interpreter*, *Parser Combinators* and *Prolog*) we see that for *Parser Combinators* the SAPL compiler has competitive performance. For *Interpreter* the SAPL compiler is competitive with GHC and GHC -O but is 4 times slower than Clean. In case of *Prolog* the SAPL compiler is significant slower than all others. This is not surprising, because the performance dominating function *unify* in *Prolog* cannot be optimised with the techniques used in the SAPL compiler. Here more sophisticated optimisations based on strictness analyses are needed.

5 Conclusions

In this paper we presented a compiler for lazy functional languages for educational and experimental use, based on a straightforward interpreter. For optimising this compiler we did not use the more sophisticated techniques normally used for compilers but took a more opportunistic approach, applying only two easy to detect and apply optimisations. This has as an advantage that the generated functions have a simple structure. This makes it possible for the user to inspect how the optimisations are applied and it also enables the user to experiment with other (hand-made) optimisations.

The compiler generates comprehensible C++ code that gives the programmer clear insight in how contructs from functional programming languages are implemented. This in contrast with the GHC compiler that also uses C as an intermediate language, but for which the generated C code is difficult to understand and looks more like assembly than like an ordinary C program.

We have learned that sometimes applying simple optimisations result in significant speed-ups (e.g *Fibonacci* and *Symbolic Primes*), but in other cases the optimisations do not suffice. In these examples (e.g. *Prolog*) the difference with Clean and GHC is still too big. We also learned that optimising a function always boils down to trying to prevent the building of unnecessary graphs (closures). In our approach this was always realized by replacing 'functional code' by 'imperative code' in the generated C++ functions.

An interesting question is, if it is possible to extend the set of optimisations in such a way that the performance becomes competitive to that of GHC and Clean in all cases while maintaining readable and comprehensive generated code.

References

1. The Haskell Home Page, `http://www.Haskell.org`
2. Martin Jansen, J.M., Koopman, P., Plasmeijer, R.: Efficient interpretation by transforming data types and patterns to functions. In: Nilsson, H. (ed.) Proceedings Seventh Symposium on Trends in Functional Programming, TFP 2006, Trends in Functional Programming, Nottingham, UK, April 19-21, vol. 7. Intellect Publisher (2006)
3. Kluge, W.: Abstract Computing Machines. In: Texts in Theoretical Computer Science. Springer, Heidelberg (2004)
4. Peyton Jones, S.L.: The Implementation of Functional Programming Languages. International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1987)
5. Plasmeijer, R., van Eekelen, M.: Functional Programming and Parallel Graph Rewriting. International Computer Science Series. Addison-Wesley, Reading (1993)
6. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: Executable specifications of interactive work flow systems for the web. In: Ramsey, N. (ed.) Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming, CFP 2007 of International Conference on Functional Programming, Freiburg, Germany, October 1-3, 2007, pp. 141–152. ACM, New York (2007)
7. Plasmeijer, R., Jansen, J.M., Koopman, P., Achten, P.: Declarative Ajax and client side evaluation of workflows using iTasks. In: Principles and Practice of Declarative Programming PPDP 2008, Valencia, Spain (July 2008)
8. Software Technology Research Group, Radboud University Nijmegen. The Clean Home Page, `www.cs.ru.nl/~clean`
9. Software Technology Research Group, Radboud University Nijmegen. The SAPL Home Page, `home.hetnet.nl/~janmartinjansen/sapl`