

Thunk-lifting: Reducing heap usage in an implementation of a lazy functional language

A. Reza Haydarlou ^{*} Pieter H. Hartel [†]

Abstract

Thunk-lifting is a program transformation for lazy functional programs. The transformation aims at reducing the amount of heap space allocated to the program when it executes. Thunk-lifting transforms a function application that contains as arguments further, nested, function applications into a new function application without nesting. The transformation thus essentially folds some function applications. The applications to be folded are selected on the basis of a set of conditions, which have been chosen such that thunk-lifting never increases the amount of heap space required by a transformed program.

Thunk-lifting has been implemented and applied to a number of medium size benchmark programs. The results show that the number of cell claims in the heap decreases on average by 5%, with a maximum of 16%.

1 Introduction

Graph reduction [11] is a technique for implementing lazy functional languages. An expression is represented as a graph that is located in the heap. During each reduction step, the evaluator performs a transformation on the graph. The transformation process terminates as soon as there are no more reducible expressions left. Much of the creation and interpretation of a graph is realised at run-time, which requires time and space. Thus any method to avoid building graph in the heap and subsequently reducing it may improve the situation.

Thunk-lifting is such a method. It is an optimisation that is used with the FAST compiler. FAST (Functional programming on ArrayS of Transputers) is an optimising compiler [5] for a lazy functional language, which is basically a subset of Miranda¹ [15]. The FAST compiler translates lazy functional programs to a subset of C called *functional C* [9]. For each function in the functional program a corresponding C function is generated. The run-time system is based on the G-machine [8].

The Thunk-lifting program transformation lifts certain nested expressions, which at run-time will be represented as thunks, to the top level. This makes it possible for the compiler to avoid building graph (suspension) for thunks. A *thunk* is a special suspension that satisfies criteria that will be developed in Section 2. Section 3 presents some experiments. Section 4 compares thunk-lifting in the FAST compiler and the equivalent of thunk-lifting in the Spineless Tagless G-machine (STG [12]). Section 5 presents the conclusions.

^{*}stichting BAZIS, Postbus 901, 2300 AX Leiden, The Netherlands, e-mail: reza@bazis.nl

[†]Department of Computer Systems, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands, e-mail: pieter@fwi.uva.nl

¹Miranda is a trade mark of Research Software Ltd.

2 Think-lifting

Think-lifting is a transformation on a lazy functional program. It tries to achieve the following aim:

The generated code for the transformed program, when executed, must allocate less space than the code for the original program.

Think-lifting needs information which is provided by compile-time strictness analysis. Consider the following function definitions:

```
> h1 c d      = (plus (square c) (square d)):NIL
> plus (x) (y) = x + y
> square (x)  = x * x
```

Here the strict arguments, on the left-hand side of the definitions, have been annotated by enclosing the arguments in parenthesis. The FAST compiler translates the definitions into C code as follows:

```
h1(c,d)
{
  return cons(vap2(plus',vap1(square',c),vap1(square',d)),nil);
}
plus(x,y)                plus'(x,y)
{                          {
  return x + y;           return plus(reduce(x),reduce(y));
}                          }
square(x)                square'(x)
{                          {
  return x * x;          return square(reduce(x));
}                          }
```

The library functions *vap1*, *vap2*, ... build a suspension for the functions *plus'* and *square'*. The suspended functions are so called *prelude* functions, they are different from the original functions *plus* and *square*. The library function *reduce* evaluates a previously built suspension. To evaluate a suspension, of *square'* say, *reduce* calls the prelude function *square'*. The prelude function makes sure that the strict arguments are in reduced form. This is done by invoking *reduce* on all strict arguments (*x* in this case). When all strict arguments have been reduced, the prelude function calls the original function (*square*). This mechanism can be optimised in a number of ways [4], but for the present discussion this simplified description suffices.

The run-time graph built by the function *h1* is shown in Figure 1(a). The root node of the graph is a *cons* cell. Its right child (tail) is NIL and its left child (head) is a suspension which is represented by a *vap3*-node. A *vap*. . .-node is the graph representation of an expression which is built by the library function *vap*. . . The *vap3*-node has three children, which from left to right, are the name of the function *plus* (the pointer to the code for *plus*) and the suspensions for the expressions (*square c*) and (*square d*). The compiler knows that the constructor function *cons* is not strict in its arguments. Therefore code is generated, which builds a suspension for *plus* and its arguments, (*square c*) and (*square d*), and which thus postpones the evaluation of *plus* and *square*. We call the suspensions for (*square c*) and (*square d*) *thunks*. In Figure 1(a), we see that the thunks for (*square c*) and (*square d*) are inside the suspension for *plus*. The question is:

Is it possible to lift the thunks for (*square c*) and (*square d*) to the top level? In other words, is it possible to generate straight calls to the function *square* instead of building suspensions for them?

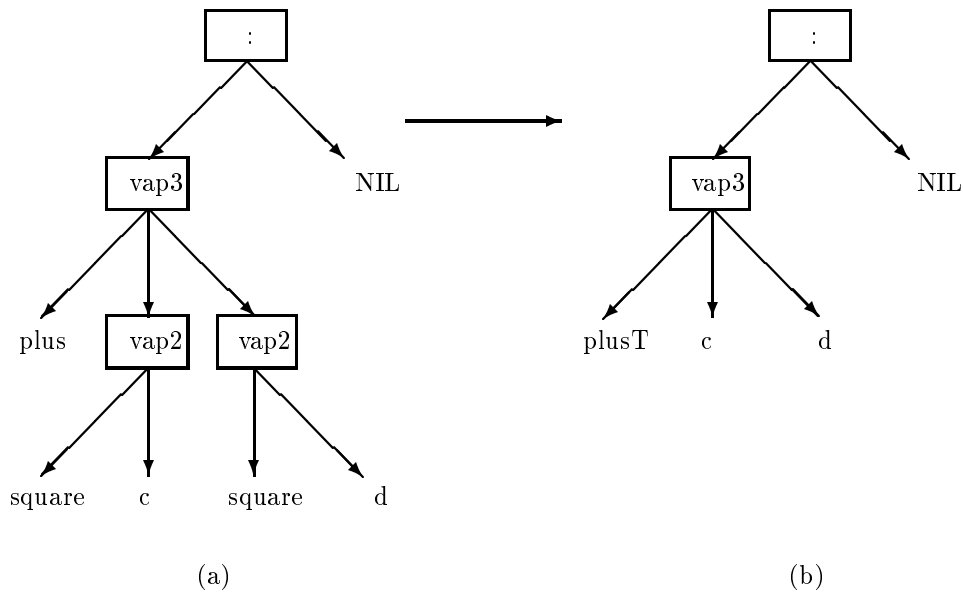


Figure 1: The run-time graph of the function $h1\ c\ d = (plus\ (square\ c)\ (square\ d)):NIL$ (a) before thunk-lifting (b) after thunk-lifting

The answer is yes. The idea is to generate a new function *plusT* and replace the expression:

```
> ... plus (square c) (square d) ...
```

with the expression:

```
> ... plusT c d ...
```

The arguments to the new function are the free variables occurring in the original expression. The body of the new function *plusT* is simply the call to the original expression. Figure 1(b) shows the run-time graph built by the transformed version *h1T* of the function *h1*. Thunk-lifting generates the new definitions below:

```
> h1T c d      = (plusT c d):NIL
> plusT (x) (y) = plus (square x) (square y)
```

The corresponding C code is:

```
h1T(c,d)
{
  return cons(vap2(plusT',c,d),nil);
}
plusT(x,y)          plusT'(x,y)
{                   {
  return plus(square(x),square(y));   return plusT(reduce(x),reduce(y));
}                                     }
```

Consider the generated C code for the transformed function *plusT*. The call to the function *plus* now occurs in a strict context, in the body of the function *plusT*. In addition, the function *plus* is strict in its arguments. Thus the compiler generates a straight imperative section of code for the expressions $(square\ x)$ and $(square\ y)$. The two run-time graphs in Figure 1(a) and 1(b) show that we have succeeded in building fewer suspensions and, in this case, also in allocating less space in the heap. As the original program may be obtained from the thunk-lifted version simply by unfolding the definition of *plusT*, the correctness of the transformation is immediate.

2.1 When is thunk-lifting beneficial?

Thunk-lifting may be performed when a function call occurs in a non-strict context. However thunk-lifting of an expression creates an extra function definition. Thus the code of the transformed program becomes larger and perhaps less efficient.

In the following sections, we give more examples which lead to the criteria on which the thunk-lifting transformation is based. The results are summarised in section 2.1.3.

2.1.1 A thunk must occur in a strict argument position

The arguments of a function f , whose call appears in a non-strict context, may contain suspensions. Not all these suspensions are thunks. Suspensions which occur in the non-strict arguments of f will also be built by the transformed version. Consider the following function definitions:

```
> h2 c d      = (first c (square d)):NIL
> first (a) b = a
```

The corresponding C code is:

```
h2(c,d)
{
  return cons(vap2(first',c,vap1(square',d)),nil);
}
first(a,b)                first'(a,b)
{                          {
  return a;                return first(reduce(a),b);
}                          }
```

Thunk-lifting yields:

```
> h2T c d      = (firstT c d):NIL
> firstT (x) y = first x (square y)
```

The corresponding C code is now:

```
h2T(c,d)
{
  return cons(vap2(firstT',c,d),nil);
}
firstT(x,y)                firstT'(x,y)
{                          {
  return first(x,vap1(square',y));  return firstT(reduce(x),y);
}                          }
```

The function *first* is strict in its first argument and non-strict in its second argument. Thunk-lifting creates a new function *firstT* whose body consists of a call to the function *first*. We see that the compiler makes a suspension for the expression (*square y*) occurring in the body of the new function *firstT*. In the generated C code for the original program (body of *h2*), a suspension has been made for the same expression (*square d*) as well. As a result, an extra function *firstT* is created, whereas we do not avoid building graph in the heap.

We conclude that if a suspension occurs in a non-strict position, thunk-lifting should not be applied to that suspension.

2.1.2 A transformed program must not claim more space

Each *vap*. . .-node (cell) contains a pointer to the code of the suspended function and pointers to the arguments of the function. To compute the size of each cell, we assume that each pointer occupies one heap location. In the FAST system this is a 32 bit word.

The transformed program may occupy more heap locations than the original version if the number of free variables occurring in the arguments of the suspended functions is large. Consider the following function definitions:

```
> h3 p q k s r      = (second (f3 q k s r) (square p)):NIL
> second a (b)      = b
> f3 (a) (b) (c) (d) = (a + b) - (c * d)
```

The corresponding C code is:

```
h3(p,q,k,s,r)
{
  return cons(vap2(second',vap4(f3',q,k,s,r),vap1(square',p)),nil);
}
second(a,b)                second'(a,b)
{                            {
  return b;                 return second(a,reduce(b));
}                            }
f3(a,b,c,d)                f3'(a,b,c,d)
{                            {
  return (a + b) - (c * d);  return f3(reduce(a),reduce(b),
                          reduce(c),reduce(d));
}                            }
```

Thunk-lifting yields:

```
> h3T p q k s r      = (secondT p q k s r):NIL
> secondT (p) q k s r = second (f3 q k s r) (square p)
```

The corresponding C code is now:

```
h3T(p,q,k,s,r)
{
  return cons(vap5(secondT',p,q,k,s,r),nil);
}
secondT(p,q,k,s,r)          secondT'(p,q,k,s,r)
{                            {
  return second(vap4(f3',q,k,s,r),    return secondT(reduce(p),q,k,s,r);
                square(p));
}                            }
```

Consider the generated C code for the original program. The function *second* occurs in a non-strict context. Therefore suspensions are built for the function *second* as well as its arguments (*f3 q k s r*) and (*square p*). The suspension for the function *second* occupies 3 heap locations and the suspensions for its first and second arguments occupy respectively 5 and 2 heap locations. As a result, the application of *second* in the original program occupies $3 + 5 + 2 = 10$ heap locations. The compiler knows that *second* is not strict in its first argument. In the generated C code for the transformed function *secondT*, a suspension is made for the first argument of *second*, namely (*f3 q k s r*) which occupies 5 heap locations. No suspension is built for its second argument (*square p*) because it occurs in a strict context. Thus we have succeeded in saving 2 heap locations. But in the generated C code for *h3T*, a suspension is built for the new function *secondT* which has 5 arguments and thus occupies 6 heap locations. The total number of heap locations that is used by the transformed program is $6 + 5 = 11$ which is more than for the original program.

$p ::= d_1 ; \dots ; d_p$	program
$d ::= f v_1 \dots v_d = b$	function definition
$b ::= \mathbf{let} v_1 = e_1 ; \dots ; v_b = e_b \mathbf{in} e$	top level let-expression
e	simple expression
$e ::= f e_1 \dots e_m$	function application
v	variable
c	constant

Figure 2: Abstract syntax of the thunk-lifter input language

2.1.3 The definition of a thunk

Now we can formulate the criteria for thunk-lifting which are guaranteed to improve the efficiency:

1. A lifted function must occur in a non-strict context.
2. At least one of the arguments of a lifted function must be a function application. This will be called a composite argument.
3. The lifted function must be strict in that argument.
4. The total number of heap locations allocated by the transformed program should be less than the number of heap locations allocated by the original program.

Each composite argument that satisfies the above criteria is defined to be thunk.

2.2 Developing a formula for thunk-lifting

The abstract syntax of the input language for the thunk-lifting transformation is a simple functional language as given in Figure 2. Programs in this form are produced as a result of compiling away more elaborate syntax. A thunk-lifter input program consists of a set of function definitions. Without loss of generality we may assume that **let**-expressions occur only at the top level. A constant can be a number, boolean, character or a string. No pattern matching is permitted for function arguments and no local recursive definitions are allowed. Data structures are built and accessed using primitive functions such as *cons* and *hd*. The results of strictness analysis are present in the form of annotations, which are not shown in the abstract syntax.

To guarantee that thunk-lifting is beneficial, the total size of the *vap*. . .-nodes built by the transformed program must be less than the size of the *vap*. . .-nodes built by the original program. The schemes in Figure 3 compute the total size of the *vap*. . .-nodes built by a program. These *vap*. . .-nodes are built only for expressions which are function applications and then only when they occur in a non-strict context.

To support the development of a formula on which to base the thunk-lifting strategy, we give three auxiliary functions. These definitions use the schemes in Figure 3. The first function S returns the list of expressions that appear in the strict argument positions of a function:

$$S :: e \rightarrow [e]$$

$$S \llbracket f e_1 \dots e_m \rrbracket = [e_i \mid f \text{ is strict in its } i\text{-th argument position}]$$

Thunk lifting should only take place if an expression appears in a strict argument position. Such an expression must not be a single constant or variable, but it must be a “composite”

$\mathcal{P} :: p \rightarrow IN$	
$\mathcal{P} \llbracket d_1; \dots; d_p \rrbracket$	$= \mathcal{D} \llbracket d_1 \rrbracket + \dots + \mathcal{D} \llbracket d_p \rrbracket$
$\mathcal{D} :: d \rightarrow IN$	
$\mathcal{D} \left[\left[\begin{array}{l} f \ a_1 \ \dots \ a_n = \mathbf{let} \ v_1 = e_1 \ ; \\ \qquad \qquad \qquad \qquad \qquad \qquad \vdots \quad \vdots \quad \vdots \quad \vdots \\ \qquad \qquad \qquad \qquad \qquad \qquad v_b = e_b \ ; \\ \qquad \qquad \qquad \qquad \qquad \qquad \mathbf{in} \ e \end{array} \right] \right]$	$= \mathcal{E} \llbracket e_1 \rrbracket + \dots + \mathcal{E} \llbracket e_b \rrbracket + \mathcal{E} \llbracket e \rrbracket$
$\mathcal{D} \llbracket f \ a_1 \ \dots \ a_n = e \rrbracket$	$= \mathcal{E} \llbracket e \rrbracket$
$\mathcal{E} :: e \rightarrow IN$	
$\mathcal{E} \llbracket f \ e_1 \ \dots \ e_m \rrbracket$	$= \mathcal{E} \llbracket e_1 \rrbracket + \dots + \mathcal{E} \llbracket e_m \rrbracket$ when $(f \ \dots)$ occurs in a strict context
$\mathcal{E} \llbracket f \ e_1 \ \dots \ e_m \rrbracket$	$= 1 + m + \mathcal{E} \llbracket e_1 \rrbracket + \dots + \mathcal{E} \llbracket e_m \rrbracket$ when $(f \ \dots)$ occurs in a non-strict context
$\mathcal{E} \llbracket v \rrbracket$	$= 0$
$\mathcal{E} \llbracket c \rrbracket$	$= 0$

Figure 3: Schemes to compute the total size of *vap*. . .-nodes built by a program. The schemes are informal with respect to the representation of strictness information.

expression. The second auxiliary function \mathcal{C} gathers the strict composite argument expressions in a list:

$$\begin{aligned} \mathcal{C} &:: e \rightarrow [e] \\ \mathcal{C} \llbracket f \ e_1 \ \dots \ e_m \rrbracket &= [e_i \mid e_i \in \mathcal{S} \llbracket f \ e_1 \ \dots \ e_m \rrbracket \wedge \mathcal{E} \llbracket e_i \rrbracket > 0] \end{aligned}$$

The third auxiliary function \mathcal{F} gathers the set of free variables of an expression in a set. This facilitates the determination of the number of arguments to a newly defined function:

$$\begin{aligned} \mathcal{F} &:: e \rightarrow \{v\} \\ \mathcal{F} \llbracket e \rrbracket &= \{v \mid v \text{ occurs free in } e\} \end{aligned}$$

With these three definitions in place let us look at a function h , with a function application $(f \ e_1 \ \dots \ e_m)$ appearing in a non-strict context:

$$h \ a_1 \ \dots \ a_n = \mathbf{let} \ v_1 = \dots; \dots \ v_b = \dots; \mathbf{in} \ \dots (f \ e_1 \ \dots \ e_m) \dots \quad (p)$$

Compare the program fragment (p) to new program fragment (pT) below, where we have assumed that no expression other than $(f \ e_1 \ \dots \ e_m)$ has been folded by the thunk-lifting transformation (We perform the thunk-lifting transformation expression by expression):

$$\begin{aligned} hT \ a_1 \ \dots \ a_n &= \mathbf{let} \ v_1 = \dots; \dots; v_b = \dots; \mathbf{in} \ \dots (fT \ b_1 \ \dots \ b_k) \dots \\ fT \ b_1 \ \dots \ b_k &= f \ e_1 \ \dots \ e_m \end{aligned} \quad (pT)$$

In the new program fragment (pT) the b_i are the free variables of the expression $(f \ e_1 \ \dots \ e_m)$ hence: $b_i \in \{a_1, \dots, a_n, v_1, \dots, v_b\}$ with $1 \leq i \leq k$.

The number of heap locations used by p is:

$$\mathcal{P} \llbracket p \rrbracket = CS + \mathcal{E} \llbracket f \ e_1 \ \dots \ e_m \rrbracket = CS + 1 + m + AS \quad (1)$$

Here $AS = \mathcal{E} \llbracket e_1 \rrbracket + \dots + \mathcal{E} \llbracket e_m \rrbracket$ and CS is the size of the *vap*. . .-nodes of the other expressions in the program.

The new program fragment pT uses heap locations to the amount of:

$$\mathcal{P} \llbracket pT \rrbracket = CS + \mathcal{E} \llbracket fT \ b_1 \ \dots \ b_k \rrbracket + (AS - AS') = CS + 1 + k + (AS - AS') \quad (2)$$

Here $AS' = \sum_{e_i \in \mathcal{C}} \llbracket f \ e_1 \ \dots \ e_m \rrbracket \ \mathcal{E} \llbracket e_i \rrbracket$.

Comparing the two expressions (1) and (2), we can formalise the criterion for thunk-lifting as follows:

$$\mathcal{P} \llbracket pT \rrbracket < \mathcal{P} \llbracket p \rrbracket$$

Simplification yields:

$$\begin{aligned} & \mathcal{P} \llbracket pT \rrbracket < \mathcal{P} \llbracket p \rrbracket \\ \equiv & \ \{\text{substitute (1) and (2)}\} \\ & CS + 1 + k + AS - AS' < CS + 1 + m + AS \\ \equiv & \ \{\text{subtract } (CS + 1 + AS) \text{ from both sides}\} \\ & k - AS' < m \\ \equiv & \ \{\text{rearrangement}\} \\ & k - m < AS' \end{aligned}$$

The inequality $(k - m < AS')$ denotes the criterion on which the thunk-lifting strategy is based. It means that thunk-lifting of $(f \ e_1 \ \dots \ e_m)$ is beneficial when the size of the *vap*...-nodes, occurring in the strict arguments of f , is larger than the difference between the number of free variables in $(f \ e_1 \ \dots \ e_m)$ and the number of arguments of f .

The formal thunk-lifting criterion governs the transformation of just one expression in an entire program p . The thunk-lifting of one expression forms a new program in which the next expression is considered. This gives rise to a chain of programs $p_0, p_1, p_2, \dots, p_n$. The differences between p_i and p_{i+1} are due to thunk-lifting of precisely one expression that occurs in p_i . Since the thunk-lifting criterion guarantees that p_{i+1} allocates less heap space than the program p_i , we may conclude that p_n allocates less heap space than p_0 .

It is impractical to actually implement thunk-lifting by considering just one function at the time. Our implementation considers all unrelated expressions at the same time, and repeats this process until no more expressions can be lifted. The net result of this procedure will not affect the final form of the thunk-lifted program.

Before moving on to the experiments we should like to point out that thunk-lifting optimises for space. This is often beneficial for the execution time as well. However, since a thunk-lifted program will contain more, but smaller functions, execution time may increase in some cases. To take into account the effects of breaking up large functions into smaller ones, a model would be required which allows the compiler to reason about the execution times. Such a model would not only account for say the cost of a function call, but it would also take into account the effects that breaking up the program has on the arrangement of the code in memory. This is necessary to deal with possible effects on the cache. This is perhaps not impossible, but it seems that before embarking on such an exercise one should consider first the scope of the effects that pure space oriented thunk-lifting has on some real programs. Should the effects be large, then further investigations are justified.

3 Experiments

To test the effect of the thunk-lifting, a number of benchmark programs [6] have been transformed, compiled and executed. The benchmark programs are applications from different areas. The benchmark set contains small and medium size programs, each of which runs on a realistic input data set. The largest program comprises 653 lines. There are a few numerical

programs	total function calls		total heap space		execution time	
	original	transformed	original	transformed	original	transformed
event	9383005	0.0%	14683802	-4.1%	23.6	+3.8%
wang	5158752	-15.8%	12648461	-16.4%	21.8	-17.0%
fft	1093650	-4.5%	2495157	-11.2%	7.9	-8.9%
genfft	2627196	0.0%	5693351	-0.1%	8.5	+4.7%
listcompr	1508013	+2.9%	4130002	-1.2%	8.6	+1.2%
wave4	8811231	0.0%	4774898	-0.1%	33.4	0%
sched	3592457	0.0%	5254741	-3.3%	9.7	-5.2%
ida	13339603	-0.4%	16037885	-2.6%	31.9	+0.6%
typecheck	27679958	+0.2%	35065003	-0.4%	61.6	-1.8%
solid	11673355	-9.4%	32570915	-14.1%	91.5	-15.0%
complab	13749717	0.0%	29405970	-0.1%	37.9	-2.1%

Table 1: The result of the thunk-lifting of the benchmark programs

applications (*wang* [17, 16], *fft* [7], *wave4* [16] and *solid* [1]). The *event* [10] program embodies the core of a simulation program. The programs *sched* [16] and *ida* [2] implement search algorithms typically found in artificial intelligence applications. An image processing application is present in the form of *complab* [14]. There are also programs that are parts of compilers (*listcompr* [11, Ch. 7] and *typecheck* [11, Ch. 9]).

Table 1 shows the result of thunk-lifting on the benchmark programs. For each original program the table gives:

- the total number of function-calls, including calls to runtime support functions such as *reduce*.
- the total number of heap locations (measured in 32 bit words) that is required by a program,
- the execution time in seconds.

The statistics for the transformed versions of the programs are relative to the original version. A negative percentage means fewer units, hence an improvement.

The two versions of a benchmark program have been compiled by the FAST compiler using the FCG [9] code generator. The “a.out” executables were timed on a SUN SPARC 4/690 UNIX system running SunOS 4.1.2. We have used `/bin/time`, taking the sum of user and system time as the total execution time. Each executable has been run 10 times with a heap size of 12 Mbyte, taking the best execution time as the ultimate performance measure.

The thunk-lifting transformation produces extra functions, however in most cases, after the transformation, the total number of function-calls either decreases or remains unchanged. The reason is that the transformation causes the number of calls to run-time support functions, such as *reduce* to be decreased.

Consider the total heap space required by both versions. Table 1 shows that thunk-lifting always saves heap space, although in some cases the gain is low. On average, the transformed versions of the benchmark programs require 5% less heap space with respect to the original versions. The numerical applications (*wang*, *fft* and *solid*) show the highest gains. In these programs, certain functions are called many times to perform arithmetic operations. Thunk-lifting lifts the arithmetic operators, which occur in the non-strict context, to the top level.

Table 1 shows that execution time is reduced when a significant amount of heap space is saved. This is the case for the numerical applications. In all other cases, the effect of thunk-lifting is too small to enable any sensible conclusion to be drawn from the execution time

measurements. Errors in the execution times are probably larger than 5%. The behaviour of a complex architecture is very difficult to capture in just two simple parameters. Caches for instance are notorious for causing this sort of behaviour. On one occasion we found that only a very slight modification of a large program, which consisted of the removal of two unused functions, caused it to run two times slower. Similar results are reported in [3].

4 The equivalent of thunk-lifting in the STG machine

There are several abstract machine designs to support functional languages. Examples are the G-machine [8] and the Spineless Tagless G-machine (STG) [12]. They have in common the property that each function, in a functional program, is compiled into an instruction sequence for these abstract machines. The two machine models differ in the way laziness is implemented.

It is interesting to see whether thunk-lifting applies as well to the STG machine as it does to the G-machine, for which it has been developed.

4.1 The STG language

The STG language (See [12] for the syntax of the STG language) is the abstract machine code for the Spineless Tagless G-machine. An STG program is just a collection of bindings. There is a special form of binding whose general form is:

$$f = \{v_1, \dots, v_n\} \setminus \pi \{x_1, \dots, x_m\} \rightarrow e$$

where (v_1, \dots, v_n) are the free variables occurring in e and (x_1, \dots, x_m) are the parameters of the function f . From an operational point of view, f is bound to a heap-allocated closure. A closure is entered by loading a pointer to it into a special register called *Node* and jumping to the code pointer in the closure. The code accesses its free variables via *Node*. The update flag $\setminus \pi$ indicates whether the closure should be updated when it reaches its normal form. The closure is updatable when the update flag is $\setminus u$ and it is non-updatable if the flag is $\setminus n$ [12].

The STG language supports boxed as well as unboxed values. An *unboxed value* is the bit-pattern representing the value itself, on which the built-in machine instructions operate. A *boxed value* is a pointer to a heap-allocated box containing an unboxed value [13]. In the STG language, the primitive integers $0\#, 1\#, \dots$ are unboxed values and the primitive operators $+\#, -\#, *\#$ and $/\#$ operate only on unboxed values. When a variable of unboxed type is bound, the expression to which it is bound must be evaluated immediately. Since *let* and *letrec* expressions always build closures, a variable of unboxed type can not be bound to these expressions. Instead, such a binding can be made using a *case* expression because *case* expressions always perform evaluation.

As an example, we give the Miranda and the STG version of the function *map*:

```
> map f [] = []
> map f (y:ys) = (f y) : (map f ys)

map = {} \n {f,xs} -> case xs of
  Nil {} -> Nil {}
  Cons {y,ys} -> let fy = {f,y} \u {} -> f {y}
                  mfy = {f,ys} \u {} -> map {f,ys}
                  in Cons {fy,mfy}
```

4.2 Implicit evaluation of strict arguments is made explicit

The function *map* is strict in its second argument which is of the algebraic data type *list*. In the STG version of the function *map*, the strict argument (*xs*) of *map* is evaluated explicitly using a *case* expression. When the strict arguments are numbers their evaluation is implicit. Consider the function *plus* and its (non-optimised) STG version:

```
> plus x y = x + y  
plus = {} \n {x,y} -> + {x,y}
```

The arguments *x* and *y* must be evaluated before they can be added, but this fact is implicit. To make the evaluation of *x* and *y* explicit, the following data type can be declared:

```
int ::= MkInt int#
```

This declares the data type of (boxed) integers, *int*, as an algebraic data type with a single constructor, *MkInt*. The latter has a single argument of type *int#*, the type of unboxed integer. So the value (*MkInt 2#*) represents the boxed integer 2 and *2#* stands for the unboxed constant 2, of type *int#*. Since the STG language supports unboxed values, the evaluation of the arguments of the function *plus* can be made completely explicit as follows:

```
plus = {} \n {x,y} -> case x of  
    MkInt x# -> case y of  
    MkInt y# -> case (plus# x# y#) of  
    t#       -> MkInt t#  
plus# = {} \n {x#,y#} -> x# +# y#
```

The arguments *x* and *y* are now explicitly evaluated by *case* expressions and *plus#* is used which produces an unboxed number. The final result is boxed again.

To exploit the information obtained from strictness analysis, a transformational framework has been presented in [13]. In this framework, a function is transformed to a semantically equivalent version in which the strict arguments are explicitly evaluated by *case* expressions. Using the rules of the transformational framework in [13], the function *plus* is split into two functions called *wrapper* and *worker*. The types of the two functions are given below:

```
wrapper function plus : int -> int  
worker function plus# : int# -> int#
```

The wrapper function takes a boxed type (integer), extracts the unboxed value from the box and gives it to the worker function. The latter does the real work. It explicitly evaluates all strict arguments before passing them to functions.

So far we have shown how the STG machine exploits the results of strictness analysis. In the following section, we will use the results of strictness analysis to decrease the number of closures built by *let* expressions.

4.3 Reducing the number of closures in the STG machine

In previous sections, we have considered thunk-lifting in connection with the FAST compiler, which is based on a variant of the G-machine. The goal of the transformation was decreasing the number of *vap*. . .-nodes that are constructed by a program at run-time. The aim of such a transformation in the STG machine can be decreasing the number of closures that are constructed in the heap at run-time. In the STG language, *let* expressions construct closures in the heap at run-time.

Consider the function *h1* as defined in Section 2:

```
> h1 c d = (plus (square c) (square d)):NIL
```

The non-optimised STG version of the function *h1* looks as follows:

```
h1 = {} \n {c,d} -> let ps = {c,d} \u {} ->
    let sc = {c} \u {} -> square {c}
        sd = {d} \u {} -> square {d}
    in plus {sc,sd}
    nil = {} \n {} -> Nil {}
in Cons {ps,nil}
```

The constructor function *cons* is not strict in its arguments, hence, closures for *ps* and for *nil* are constructed. If the first argument of *cons* is needed later, in evaluated form (head of the *cons* cell) the closure for *ps* must be entered. The code then builds closures for *sc* and *sd*. In cases such as this, the construction of the closures for *sc* and for *sd* can actually be avoided by thunk-lifting. The function *plus* is strict in its arguments and the arguments (*square c*) and (*square d*) can be evaluated explicitly using *case* expressions.

It is possible to apply the principles of thunk-lifting in several different ways. Let us consider the same route which has been taken in the previous sections, in relation with the FAST compiler. This means that we have to transform the function *h1* and generate a new function as follows:

```
> h1T c d      = (plusT c d):NIL
> plusT (x) (y) = plus (square x) (square y)
```

The corresponding STG versions of these functions are as follows:

```
h1T = {} \n {c,d} -> let ps' = {c,d} \u {} -> plusT {c,d}
    nil = {} \n {} -> Nil {}
    in Cons {ps',nil}
```

```
plusT = {} \n {c,d} -> case c of
    MkInt c# -> case d of
    MkInt d# -> case (square# c#) of
    sc#      -> case (square# d#) of
    sd#      -> case (plus# sc# sd#) of
    ps#      -> MkInt ps#
```

The functions *plus#* and *square#* are the worker functions which take the unboxed and evaluated arguments. No closures are built for the expressions (*square c*) and (*square d*). They are evaluated explicitly by *case* expressions.

Since the STG machine binds all top-level functions (globals) to a statically allocated closure, a new closure must be allocated for the new generated (global) function *plusT*. In addition, the generation of extra functions requires time. Now the question is:

Is it possible to generate an optimised STG code, without generating new functions?

The answer is yes. By simply unfolding the function *plusT*, which occurs in the body of the closure for *ps'*, we get the following optimised STG version of the function *h1*:

```
h1T = {} \n {c,d} -> let ps' = {c,d} \u {} ->
    case c of
    MkInt c# -> case d of
    MkInt d# -> case (square# c#) of
    sc#      -> case (square# d#) of
    sd#      -> case (plus# sc# sd#) of
    ps#      -> MkInt ps#
    nil = {} \n {} -> Nil {}
    in Cons {ps',nil}
```

When the closure for ps' is entered, a pointer to it is loaded into *Node* and then a jump is made to the code which begins with the evaluation of c . The important point is that the code can access the value of the free variables c and d via *Node*. Closures in the STG machine play the role of an environment via which the value of the free variables can be accessed.

In the case of the FAST compiler, thunk-lifting has to generate new functions to access the value of the free variables occurring in the code.

This is perhaps not the most natural way to apply thunk-lifting in the context of the STG-machine, but it does show the benefits of the method. If one were to implement thunk-lifting in a STG based compiler, slightly different criteria and transformations would be used instead.

5 Conclusions

Thunk-lifting transforms a function application that contains as arguments further, nested, function applications into a new function application without nesting. The transformation folds function applications, which are selected on the basis of a set of conditions.

The conditions take a number of properties of functions into account, such as the (non) strictness of the argument positions of the functions involved. Also the amount of space required to build suspended function applications and the number of arguments as well as the number of free variables in the expressions are taken into account by the conditions. This makes it possible to guarantee that thunk-lifting never increases the amount of heap space required by a program. On average, the transformed versions of a set of medium size benchmark programs require 5% less heap space than the original versions, with a maximum of 16%.

Thunk-lifting may on the other hand increase the number of function calls, though in our experiments we have found such increases to occur rarely, and if they occur the effect is small. Thunk-lifting causes most of the transformed programs to run faster.

Thunk-lifting is shown to be applicable to the G-machine as well as the STG-machine. These are both abstract machine designs underlying the implementation of many lazy functional languages. The transformation thus has a wide range of applicability.

Acknowledgements

We thank Marcel Beemster and the referees for their comments on a draft version of the paper. The FAST compiler represents joint work with Hugh Glaser and John Wild, which was supported by the Science and Engineering Research Council, UK, under grant No. GR/F 35081, FAST: Functional programming for ArrayS of Transputers.

References

- [1] J. R. Davy. *Using divide and conquer for parallel geometric evaluation*. PhD thesis, School of Computer Studies, Univ. of Leeds, England, Sep 1992.
- [2] J. Glas, R. F. H. Hofman, and W. G. Vree. Parallelization of branch-and-bound algorithms in a functional programming environment. In H. Kuchen and R. Loogen, editors, *4th Parallel implementation of functional languages*, pages 47–58, Aachen, Germany, Sep 1992. Aachener Informatik-Berichte 92-19, RWTH Aachen, Fachgruppe Informatik.
- [3] K. Hammond, G. L. Burn, and D. B. Howe. Spiking your caches. In K. Hammond and J. T. O'Donnell, editors, *Functional programming*, pages 58–68, Ayr, Scotland, Jul 1993. Springer-Verlag, Berlin.

- [4] P. H. Hartel, H. W. Glaser, and J. M. Wild. On the benefits of different analyses in the compilation of functional languages. In H. W. Glaser and P. H. Hartel, editors, *3rd Implementation of functional languages on parallel architectures*, pages 123–145, Southampton, England, Jun 1991. CSTR 91-07, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England.
- [5] P. H. Hartel, H. W. Glaser, and J. M. Wild. Compilation of functional languages using flow graph analysis. *Software—practice and experience*, 24(2):127–173, Feb 1994.
- [6] P. H. Hartel and K. G. Langendoen. Benchmarking implementations of lazy functional languages. In *6th Functional programming languages and computer architecture*, pages 341–349, Copenhagen, Denmark, Jun 1993. ACM.
- [7] P. H. Hartel and W. G. Vree. Arrays in a lazy functional language – a case study: the fast Fourier transform. In G. Hains and L. M. R. Mullin, editors, *2nd Arrays, functional languages, and parallel systems (ATABLE)*, pages 52–66. Publication 841, Dept. d’informatique et de recherche opérationnelle, Univ. de Montréal, Canada, Jun 1992.
- [8] T. Johnsson. *Compiling lazy functional languages*. PhD thesis, Dept. of Comp. Sci, Chalmers Univ. of Technology, Göteborg, Sweden, 1987.
- [9] K. G. Langendoen and P. H. Hartel. FCG: a code generator for lazy functional languages. In U. Kastens and P. Pfahler, editors, *Compiler construction (CC 92)*, LNCS 641, pages 278–296, Paderborn, Germany, Oct 1992. Springer-Verlag, Berlin.
- [10] H. L. Muller. *Simulating computer architectures*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, Feb 1993.
- [11] S. L. Peyton Jones. *The implementation of functional programming languages*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.
- [12] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal functional programming*, 2(2):127–202, Apr 1992.
- [13] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In R. J. M. Hughes, editor, *5th Functional programming languages and computer architecture*, LNCS 523, pages 636–666, Cambridge, Massachusetts, Sep 1991. Springer-Verlag, Berlin.
- [14] Q. F. Stout. Supporting divide-and-conquer algorithms for image processing. *Journal parallel and distributed computing*, 4(1):95–115, Feb 1987.
- [15] D. A. Turner. *Miranda system manual*. Research Software Ltd, 23 St Augustines Road, Canterbury, Kent CT1 1XP, England, Apr 1990.
- [16] W. G. Vree. *Design considerations for a parallel reduction machine*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, Dec 1989.
- [17] H. H. Wang. A parallel method for tri-diagonal equations. *ACM transactions on mathematical software*, 7(2):170–183, Jun 1981.