on the benefits of deforestation: a quantitative analysis

DIPLOMASCRIPTIE

door

Stefan Gruner

geboren te Wiesbaden in 1970

FACULTEIT DER WISKUNDE EN INFORMATICA

NL–1098 SJ Amsterdam-Watergraafsmeer

UNIVERSITEIT VAN AMSTERDAM

juli 1995

abstract. In this thesis an analysis program for estimating the deforestation gain for functional programs is developed. Several examples are analysed and the methods are discussed with respect to their suppositions and adequacy. The aim of deforestation is to remove intermediate data structures which arise when functions working on such structures are combined in a certain manner. However, a complete deforestation turns out to be impossible in realistic programs where lots of intermediate structures are shared.

Chapter 1

introduction

This thesis presents the result of my diploma project during the last six months on the benefits of deforestation. This introductory chapter gives the topic and outline of my work; but first of all some basic terms have to be explained because that is what the following chapters are built on.

1.1 what is deforestation?

A simple question! But look at some answers[1]:

> "A compositional style of programming is often advocated by functional programmers. However, there is a certain efficiency penalty involved in creating the requisite intermediate structures. Deforestation is a *program transformation technique* which removes such structures from such programs." [3]

> "Intermediate data structures are widely used in functional programs. Programs which use these intermediate structures are usually a lot easier to understand, but they result in loss of efficiency at run time. In order to reduce these run-time costs, a *transformation algorithm* called *deforestation* was proposed by Wadler which could eliminate intermediate structures." [7]

> "Deforestation is an *automatic transformation scheme* for functional programs which attempts to remove unnecessary intermediate data structures." [13]

> "Deforestation removes arbitrary intermediate data structures (including lists), but suffers from major drawbacks. [...] In this paper we present a cheap and easy *way of eliminating many intermediate lists* that does not suffer from these drawbacks." [4]

> "There is a *style of transformation* that reduce the space consumption of programs called deforestation. The name deforestation is used because the transformation eliminates intermediate data-structures (i.e. trees). The particular deforestation we are interested in is called foldr/build deforestation." [5]

Thus, the point at issue is surrounded, yet not hit by an exact definition. However, all authors quoted above agree that they perform algebraic transformations (according to certain rewrite rules) of a functional program from its source language into its source language before compilation, in order to save some runtime after compilation of this program. The novel approach presented here goes the other way round: looking at the events having taken place at runtime, we approximate the *a priori* possibility of deforesting the analyzed program for *any* transformation technique.

[1]The *slanted setting* is not used in the originals.

1.2 intermediate structures and garbage

Is there a relation between the concept 'intermediate structure' and the concept 'garbage in the heap'? Both intermediate and garbage cells are some kind of 'persona non grata' among those privileged cells which build the output structure of a functional program in the heap. Both are eventually superfluous, and the difference between them is not easy to see from a machine point of view. The reason for this difficulty is that the concepts of 'intermediate' and 'garbage' not only fail to be mutually exclusive, but are actually not even comparable. 'Being garbage' is simply a state of isolation in the heap; 'being intermediate', however, is one possible *reason for becoming* garbage later. 'Being garbage' means for a cell 'having done its duty' in the runtime history, whilst 'being intermediate' is the 'duty' itself (namely: to be produced by a function and to get consumed by another function).

1.3 deforestation techniques

Three main techniques for deforesting intermediate structures can be found in literature. The first one, called "the concatenate vanishes" [18], is presented for deforesting intermediate list structures caused by the list concatenating *append* function. This approach —based on four rewrite rules— seems to be a prototype: the terms 'intermediate' and 'deforestation' do not even appear in the treatise, and the technique has not been developed further. The second one, to be found in [19], could be called the 'classical' deforestation approach to arbitrary kinds of data structures. This approach is based on seven rewrite rules and is further extended in [7] and [13]. "The concatenate vanishes" may be regarded as a special case of this more general approach. Having discovered some properties of list production and list consumption, [4] present the third and completely different approach to list deforestation which is called "foldr/build" and is based on only one rewrite rule. Experiences made with it are reported in [5] and [6]. In this thesis the transformations themselves do not matter: we will be concerned with their results, of which examples will be given later.

1.4 topic and outline of this thesis

Have another look at the introductory statements on deforestation quoted above. Whilst the majority of them seem to be rather optimistic, as far as results are concerned, [13] speak more carefully of an *attempt* to remove *unnecessary* intermediate structures — which leads directly to the topic questions of this thesis: How much can be 'earned' with deforestation (assuming there *were* a method to remove *all* intermediate structures)? Are there *necessary* intermediate structures, not removable at all? In trying to give an answer, we will leave the 'plateau' of algebraic transformations to have a look into the 'cave' of the machine heap, where the structure building cells reside. The machinery used for this kind of deforestation analysis is presented in chapters 2, 3 and 4 in an abstract manner. Chapters 5, 6, 7 and 8 are concerned with small examples presented in the deforestation literature. In chapter 9 the question is asked, whether the results of those well-selected examples can be generalized, and deforestation analysis is performed on four bigger benchmark programs. Chapter 10 concludes, and some more details are to be found in the appendix together with two abstracts written in Dutch and German.

Chapter 2

an abstract view of deforestation

2.1 machinery for deforestation analysis

To study effects and possible gains of the deforesting program transformations described in the literature on functional programming, the use of a certain machinery seems inevitable. In this case we have: the language Miranda [16], [2] as a representative of a lazy functional programming language; this is the front end of our machinery and the point where deforesting transformations apply. Then, the FAST compiler [10], which takes a functional program written in a subset of Miranda as input and generates C code suitable for compilation on almost every machine, and finally a SPARC machine executing the C code to give the output of the Miranda program. The FAST compilation happens —variations in detail are not mentioned— according to the principles of the well known G-machine described by Johnsson *et* Augustsson [15]. Execution output of a FAST program is a *trace* (Fig.1, see also chapter 4 for further explanation), which documents relevant computation steps and changes being made to the heap during the run. Such a trace will be the basis of an automatic analysis done by a program which is described later in this thesis. The purpose of this program is to *replay* the events which have taken place in the heap and to detect and count cells which have been created as intermediate structures in the sense of the terminology of the deforestation literature. Therefore the program uses information extracted from the output trace.

2.2 abstracting the essentials

Abstracting from particular items in a complex situation means regarding a lot of information as irrelevant (or even distracting) to understanding the essentials of the situation. In our case it does not matter, for example, if constructor cells are represented in a boxed or unboxed manner nor are the values important, which the constructor cells contain. What matters is *that* they have been created at all. In a similar way, the analysis of the suspended application nodes being built does not depend on values being computed by the functions belonging to those nodes and how this evaluation has been done: we only want to know *if* suspensions have been allocated. Thus, deforestation analysis will be a ***structure analysis*** and therefore as independent from the structures' contents as possible. For this reason it seems practicable to abstract from the machinery described above (Fig.2), when we have to explain the generation of the output traces. Such an abstraction is described in the following. The front end language Miranda is represented by expressions of an abstract syntax as shown in Fig.3. A simple translation scheme $\mathcal{T}$ explains the generation of a pseudo code α for an abstract machine $\mathcal{A}$ (Fig.4 and Fig.5). In this machine everything which is not essential for the subsequent analysis is hidden in a 'black box', and only

what matters as far as generation and rearrangement of heap structures are concerned is visible. As $\mathcal{A}$ is meant to represent a simple model of the FAST system, $\mathcal{A}$ is also built using some of the principle ideas of the G-machine. In contrast to the detailed trace of the FAST machine, $\mathcal{A}$ outputs an abstract trace, which is nevertheless expressive enough to show all events of a machine run which are important for deforestation analysis.

```
                  reduce:  caf=52340
              _01_LIT2563:
         _01_NULLARY_PACK:  n=1 <512> at 590d8
              _01_LIT2563:  return_T=590d8
                   update:  root=52340 result=590d8 tag=512
              _01_PROJECT:  59084 !  1 at 0 box at 52340
     _11_threestate_cmp:  _01_x_T_0=052340 _00_y_T_0=0526d8
                     vap2:  fun=5249c,a1=526d8 at 590e0
                     vap3:  fun=5252c,a1=590e0,a2=2 at 590ec
                     vap4:  fun=524fc,a1=52248,a2=52240,a3=590ec at 590fc
                   reduce:  vap=590fc
prel_0001T_0001T_1100B_0:  proc=2ad8c arg=59104
                   reduce:  vap=590ec
      prel_1100I_1111I_1:  proc=2a600 arg=590f4
                   reduce:  vap=590e0
            prel_0100T_1:  proc=25970 arg=590e8
                   reduce:  vap=526d8
       prel_1111I_0_var:  proc=2511c arg=526e0
                _01_PACK:  n=1 <768> at 59110 box at 59110
                   update:  root=526d8 result=59110 tag=768
                 _01_TAG:  pack=526d8 tag=3 at=59118
                   update:  root=590e0 result=59118 tag=9
```

Fig.1 *example of a trace fragment from the FAST compiler*

Miranda subset → **FAST** compiler → **C** code → **SPARC** machine → **trace**

||||||||||||||| ABSTRACTION |||||||||||||||

abstract syntax → $\mathcal{T}$ scheme → α code → $\mathcal{A}$ machine → **abstract trace**

Fig.2 *concrete and abstract machinery basis for deforestation analysis*

$p \to d_1; \ldots; d_n;$	program
$d \to f\ v_1 \ldots v_n = e$	function definition ($n \geq 0$)
$e \to f_s\ e_1 \ldots e_n$	application in strict context
$\mid\ f\ e_1 \ldots e_n$	application in non strict context
$\mid\ e_1\ e_2$	partial application
$\mid$ let $v = e_1$ in e_2	local definition
$\mid\ v$	variable
$\mid\ c$	constant
$\mid\ f$	function name
$\mid$ eval e	enforced evaluation

Fig.3 *annotated abstract syntax of a lazy functional language for cell structure generation*

type $\mathcal{T}$ :: [*abstract syntax expression*] → (*identifier*) → [α *code*]

$\mathcal{T}[\![d_1; \ldots; d_n;]\!]$ ()	⇒	$\mathcal{T}[\![d_1]\!]$ () ⋮ $\mathcal{T}[\![d_n]\!]$ ()	case: program
$\mathcal{T}[\![f\ v_1 \ldots v_n = e]\!]$ ()	⇒	function f $v_1 \ldots v_n$; $\mathcal{T}[\![e]\!]$ (f) return;	case: function definition ($n \geq 0$)
$\mathcal{T}[\![f_s\ e_1 \ldots e_n]\!]$ (*ident*)	⇒	$\mathcal{T}[\![e_1]\!]$ (a_1) ⋮ $\mathcal{T}[\![e_n]\!]$ (a_n) *ident* ▷ $\mathcal{E}$ ($\mathcal{T}[\![f]\!]$ (), $a_1, \ldots, a_n$);	case: application in strict context where $f \neq$ cons, $f \neq$ if
$\mathcal{T}[\![f\ e_1 \ldots e_n]\!]$ (*ident*)	⇒	$\mathcal{T}[\![e_1]\!]$ (a_1) ⋮ $\mathcal{T}[\![e_n]\!]$ (a_n) *ident* ▷ vap ($\mathcal{T}[\![f]\!]$ (), $a_1, \ldots, a_n$);	case: application in non strict context where $f \neq$ cons, $f \neq$ if
$\mathcal{T}[\![$cons $e_1\ e_2]\!]$ (*ident*)	⇒	$\mathcal{T}[\![e_1]\!]$ (hd) $\mathcal{T}[\![e_2]\!]$ (tl) *ident* ▷ cons (hd, tl);	subcase: construction in any context

$\mathcal{T}[\![$if *bool* $e_1\ e_2]\!]$ (*ident*)	⇒	$\mathcal{T}[\![bool]\!]$ (bool)
		if (bool)
		{$\mathcal{T}[\![e_1]\!]$ (*ident*)}
		else
		{$\mathcal{T}[\![e_2]\!]$ (*ident*)}
			subcase: conditional expression

$\mathcal{T}[\![e_1\ e_2]\!]\ (ident)$	$\Rightarrow$	$\mathcal{T}[\![e_1]\!]$ (pap$_1$) $\mathcal{T}[\![e_2]\!]$ (pap$_2$) *ident* ⊳ @ (pap$_1$, pap$_2$);	case: partial application
$\mathcal{T}[\![$let $v = e_1$ in $e_2]\!]\ (ident)$	$\Rightarrow$	$\mathcal{T}[\![e_1]\!]$ (v) $\mathcal{T}[\![e_2]\!]$ (*ident*)	case: simple local definition
$\mathcal{T}[\![v]\!]\ (ident)$	$\Rightarrow$	*ident* ⊳ v;	case: variable
$\mathcal{T}[\![c]\!]\ (ident)$	$\Rightarrow$	*ident* ⊳ c;	case: constant
$\mathcal{T}[\![f]\!]$ ()	$\Rightarrow$	proc$_f$	case: function name
$\mathcal{T}[\![$eval $e]\!]\ (ident)$	$\Rightarrow$	$\mathcal{T}[\![e]\!]$ (*ident*$_E$) reduce;	case: enforced evaluation

Fig.4 ***translation to pseudo code α.*** Assume that all identifiers f, a_i, pap$_i$, hd, tl, let$v$ and bool are internally made unique by additional labels for scope separation

2.2.1 abstract syntax

Have a look at the abstract syntax as shown in Fig.3. A functional *program* is a set of function definitions, each of them concluded by a semicolon. This pattern is derived from a special demand of the FAST compiler to its input language. Of course function definitions may be recursive or mutually recursive. The definition sequence is completely irrelevant as long as no undefined functions occur. A function *definition* is an equation with a function name and a number of variables according to the defined function's arity on the left hand side, and a legal defining expression on the right hand side, which has especially to be well typed. An *expression* can be a variable, a constant, a local definition (using let) or a partial or full application on further expressions. Examples for constants are numbers, characters or NIL.
Beyond the usual abstract syntax schemes we would like to distinguish strict from non strict application contexts in order to simulate the handling of applications as closely as possible to the according property of the FAST compiler[1]. Assume hereby that we can get the necessary strictness information for free (of course it has to be deduced in reality). This distinction is provided by a little index $_s$ given to the function name of an application appearing in strict context. Function calls in strict context will not be suspended, but immediately evaluated instead. This is to avoid the creation of unnecessary suspension cells. Additionally, a *function name* is mentioned as a separate kind of expression, as function names will be used as special labels in α code later. There are also built in function names, for example null, cons, head and tail. The most unusual concept of the abstract syntax presented here is to make an *evaluation* explicit by giving it also a separate kind of expression. This concept influences the evaluation sequence of the abstract machine $\mathcal{A}$ and may be used, for example, for reasons of efficiency.

[1]The deforestation analysis developed in this thesis is, of course, also applicable in principle for compilers which do not perform strictness analysis. Then however, the deforestation analysis results would probably be different, as strictness analysis influences the allocation behaviour of a functional program.

2.2.2 translation to α code

Now have a look at Fig.4. For each case of the abstract syntax definition there has to be one case of the translation scheme $\mathcal{T}$. Here there are even more because constructions and conditional expressions, both subcases of applications, are mentioned separately. A program is translated by translating every one of its definitions. We do not have to be concerned with the way of storing a translated definition into —and finding it again, when needed, out of— the program memory of the machine, but we may assume instead that all those compile time and run time management operations, irrelevant from our abstract point of view, are done correctly by the machine[2].
The result of the translation is an α program with a structure as shown in Fig.5. It consists of a non empty command sequence to be executed step by step. The most obvious possibility is to do case analysis by selection of one command sequence out of two given alternatives according to some condition bool. The update command can not be reached by the translation scheme $\mathcal{T}$: this is because the graph reducing property is inherent to the machine, and not determined by the program. The meaning of the other commands, symbols and the identifiers *ident* is to be explained in the following chapter; this will be done by defining actions of the abstract machine $\mathcal{A}$ for every possible command of α.

α prog	::=	*command*
command	::=	*ident* ⊳ $\mathcal{E}$ (*ident* ... *ident*);
	|	*ident* ⊳ cons (hd, tl);
	|	*ident* ⊳ vap (*ident*,a_1, ..., a_n);
	|	*ident* ⊳ @ (pa$_1$, pa$_2$);
	|	*ident* ⊳ *ident*;

	|	if (bool)
		{ *command* }
		else
		{ *command* }

	|	function *ident* ... *ident* ;
	|	return;
	|	reduce;
	|	update;
	|	*command* *command*

Fig.5 *syntax of pseudo code α to be run by the abstract machine $\mathcal{A}$*

[2]$\mathcal{A}$ is not an oracle machine, because it does not need to be fed with transition advice.
In fact, $\mathcal{A}$ is *not implemented* for reasons as described above.

2.3 example

As an example for translation to α code consider the abstract syntax expression

main := let v = cons 1 NIL in cons v NIL

Following the translation scheme $\mathcal{T}$ (Fig.4) we obtain step by step:

$\mathcal{T}$[main := let v = cons 1 NIL in cons v NIL] () .... case: program

In this example the program consists of only one function definition, main, without any arguments.

```
function main; ..........................................................................
T[let v = cons 1 NIL in cons v NIL] (main) ..............................................
return; ................................................... case: function definition
```

In the context of (main) the let expression has to be translated.

```
function main;
T[cons 1 NIL] (v) .......................................................................
T[cons v NIL] (main) .................................. case: simple local definition
return;
```

It gives the order to connect one Cons cell to the head of another one.

```
function main;
T[1] (hd1) ..............................................................................
T[NIL] (tl1) ............................................................................
v ▷ cons (hd1, tl1); ............................................ case: construction
T[v] (hd2) ..............................................................................
T[NIL] (tl2) ............................................................................
main ▷ cons (hd2, tl2); ......................................... case: construction
return;
```

The (shared) value NIL is to be contained by (both) the Cons cells' tails.

```
function main;
hd1 ▷ 1; ..................................................... case: constant
tl1 ▷ NIL; ................................................... case: constant
v ▷ cons (hd1, tl1);
hd2 ▷ v; ..................................................... case: variable
tl2 ▷ NIL; ................................................... case: constant
main ▷ cons (hd2, tl2);
return;
```

Now the translation is finished. This piece of α code gives the order to return a Cons cell as the main result. The head of this Cons cell has to point to another Cons cell which has to be allocated first, and has to contain the value 1 as its head. The Cons cells' tails have to point to the (shared) value NIL. Both 1 and NIL are constant, v is a variable.

Chapter 3

the $\mathcal{A}$ machine at work

3.1 machine components

In the previous chapter a reason for an abstract view of analysis has been given and a rough sketch of such an abstraction has been drawn. In this chapter the generation of an abstract trace, reporting the cell allocation history of a program run by the abstract model machine $\mathcal{A}$, will be explained in more detail. The parts of $\mathcal{A}$ are the following:

a) Some program memory containing function definitions which have been translated to pseudo code α. A special one of these code sequences is the *main* program which starts a machine run.

b) A heap, being the memory segment for runtime cell allocations as used in all common implementations of functional programming. In the case of $\mathcal{A}$ the heap is to contain nothing but a skeleton of the usual heap content, namely the structure building nodes *Cons*, *Vap* and application nodes in an abstracted form. A Cons is a structure building entity consisting of a reference to somewhere which is usually called *head* and another such reference to some *tail*. We introduce $\bowtie$ (the mathematical symbol for a join) as the symbol for Cons because of its combining property. Vap are special application nodes with an arbitrary number of arguments. Vap also appear in the original G-machine; their name is a short form of Vector application node. An abstract Vap is an entity consisting of a corresponding function reference label proc_f and some further references to somewhere, depending on the number of arguments of the function f belonging to it. Let $\ominus$ be a symbol for a newly created Vap, and $\odot$ a symbol for a Vap for which the corresponding function f has already been evaluated by executing the procedure proc_f representing f in the program memory of $\mathcal{A}$. A $\odot$ is *ready* to become updated in a graph reducing step (but still it is a non updated Vap: in the model presented here, the evaluation of a function is separated from the graph reducing update of the suspension node of that function in order to augment intuitive clarity: first the evaluation, then the update). There are also ordinary application nodes, as in the G-machine, but these appear in the model described here only when partial applications are to be done; @ is the usual symbol for them.

c) A stack for references to the heap and the black box part of the machine $\mathcal{A}$. As it is not meant to deal with computation processing details here, the stack model is chosen to be very simple for clarity: offset counts and some other organization methods necessary for running real graph reducers are not essential for our question and can completely be avoided in this model.

d) A *dump* in order to save remaining machine states when the normal program flow is to be interrupted and stack rearrangements have to be done. This is the case when non strict function calls occur in order to solve lazy suspensions for complete results. In the model described here, the dump is another stack containing pairs consisting of a stack and some α code sequence. In contrast to the G-machine, $\mathcal{A}$ is not meant to perform efficient computation. A simple kind of dump in the model machine $\mathcal{A}$ has been chosen in order to represent only the idea of handling

interrupts to the normal program flow caused by new function calls.
e) Finally, there is an evaluation and storage management black box which $\mathcal{A}$ would not be a machine without. This black box provides all operations necessary to compute properly but which are out of the scope of interest of the questions to be asked here. These operations are variable referencing and dereferencing, finding the locations of arguments which a function is to be applied to, finding a function's code out of the program memory, evaluation of built in functions and strict function calls, reduction and update decisions and more. Fig.6 shows a conceptual sketch of $\mathcal{A}$.

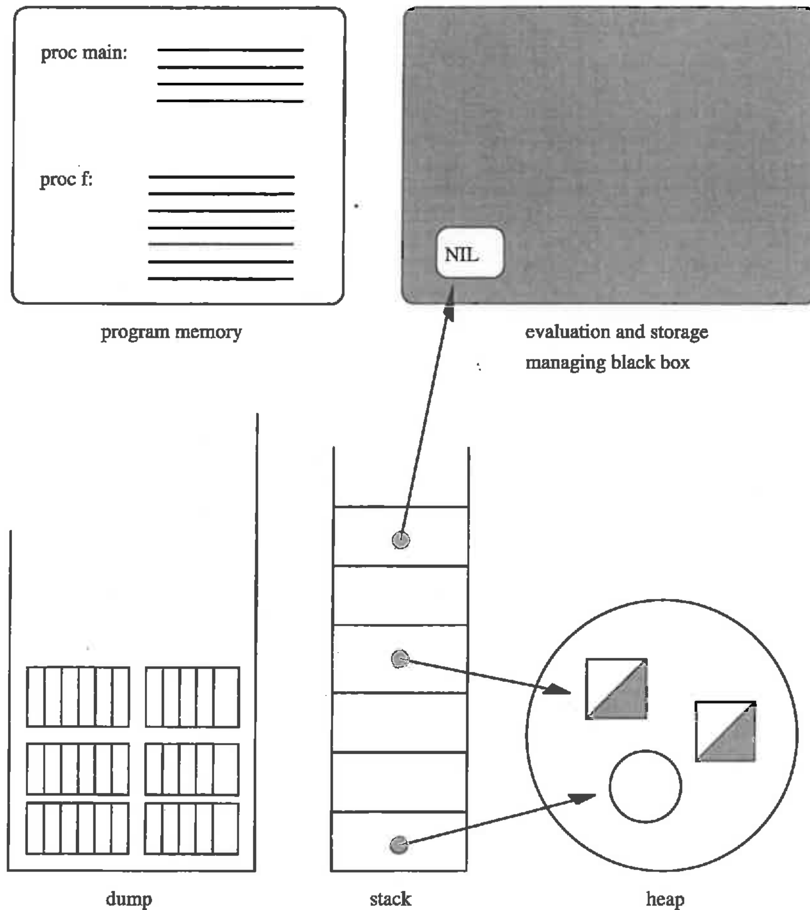


Fig.6 *conceptual sketch of $\mathcal{A}$ with program memory, stack, dump, heap and black box*

3.2 the meaning of α code

In the following, the translation to α code (Fig.4 and Fig.5) together with its meaning are first explained informally, then the state transitions of $\mathcal{A}$ which are the semantics of α code are presented. Two examples will conclude this chapter.
The syntax of α code is an abstraction of an *imperative* language. Translation to α code thus means finding a proper algorithmic description for a sequence of machine steps out of a functional specification written in abstract syntax, where no problem solving procedure has been made explicit. Fig.4 gives an idea of how a sequential code can be produced recursively by translating such specifications piecewise and putting the results in a row. This method is well known in the field of compiler design [1].
Commands beginning with '*ident* $\triangleright$' (Fig.5) force a pointer to be pushed onto the pointer stack. In the model described here, each pointer has a name *ident* to identify its 'membership' within a certain syntactic context. The translation scheme $\mathcal{T}$ (Fig.4) is able to transport such context information in the single brackets appended to the argument expression of $\mathcal{T}$. Function evaluation demands context information in order to work properly. The exact usage of *environments* as mathematical objects is described in the compiler literature; for the simple model here it is not necessary to be concerned with evaluation details: by naming the stack pointers according to the objects they refer to, an unambiguous representation of all structural coherency is provided.
Commands containing cons, vap or @ (Fig.5) create, additionally, the corresponding cells on the heap. Of course, a new cell must somehow be connected to the already existing cells, but again we may assume by abstraction that this will be done by the black box component, which only has to provide sufficient information about those connections for our analysis later.
The black box also has to be employed when a command '*ident* $\triangleright$ $\mathcal{E}$ (...)' for application in strict context occurs (Fig.5). This causes some evaluation out of our scope and leaves, as its result, a pointer on the top of the stack.
Nota bene: The black box evaluation of a function f_s in strict context may further affect the components of $\mathcal{A}$. The reasons for such possible effects are of course to be found in the procedure proc_f representing the definition of f. (Fig.4, case: application in strict context).
The command function *name* $v_1 \ldots v_n$ (or simply function *name*, if there are no variable function arguments) is the beginning of a new (non strict) function call (Fig.5). In the G-machine a function call is initialised by a rather sophisticated creation of a new stack frame. To remind us of these initial operations in reality, a symbolic fence # is pushed as a border line on the pointer stack in our simple model. # may be regarded as a special pointer to nowhere. The information about (eventual) arguments $v_1 \ldots v_n$ is kept in the black box to be given back later when needed for deforestation analysis.
In correspondence with function, the command return concludes a function call. The result pointers overwrite the topmost # and the superfluous stack content is popped. The mechanism is very simple, but not in opposition to the common usage in compiler design.
The abstract machine $\mathcal{A}$ dumps a complete stack, together with the remaining code sequence, when a function call occurs in order to dissolve a suspension. The code for the new computation part is imported from the program memory mentioned above and the new stack contains as its only element a pointer to the corresponding $\ominus$. In the event of there being old code together with old stack pointers, this code is restarted when the current remaining code sequence is empty.
Updates cause some rearrangements in the heap principally according to the corresponding behaviour of the known graph reduction machine. In our model the only visible change an update can do is to replace a suspension node by a Cons after some evaluation; an update of a suspension node with a structurally irrelevant value is not seen: in this case the suspension node will remain as a Vap (or @) node as before, but of course not representing an unsolved function any more. It is

the command reduce which causes those reduction events (Fig.5), but the order in which they are to occur is only a matter for the black box evaluator; the evaluation circumstances reside beyond the horizon of our abstract point of view.
A computation will finish if the dump is empty and no code sequence remains to be done.

3.3 state transitions

Some further notations have to be explained, which will be used in the description of the state transitions of $\mathcal{A}$. They are as similar as possible to the notation used in [15].
Let [] be a symbol for the empty heap, the empty dump, the empty stack or the empty code sequence.
S is the stack. If S is some stack, p:S is the result of pushing p onto S. Vice versa, if p:S is some stack, then S remains after popping one element. Elements of the stack are named pointers as described above.
C is a program consisting of a sequence of α commands. If c:C is a code sequence, whose first command is c, then C remains to be computed after c has been done.
D is the dump. A dump of which the top pair is (S,C), is written as (S,C):D. After a redump, D remains.
All stack tops appear on the left side.
H is the heap. Unlike a stack, the heap does not force its elements into a sequential order. If H is a heap in some state, we use H+n to denote the generation of a new node n in the heap. We use n:H to *focus attention* on a certain node n in the heap. (An equivalent statement would be $\exists n \in \{ \dots \}$).
Finally, T is a trace. If T is a trace documenting a current run, T:*act* denotes its state after the latest action *act* of interest has just taken place.
For better readability, states of the components of $\mathcal{A}$ may be numbered.
For example, if $\mathcal{K}_i \in$ {S, C, D, H, T} is an $\mathcal{A}$ component in some state i, and this state is being changed (for example e:$\mathcal{K}_i$), then this new state may be denoted by $\mathcal{K}_{i+1}$ in the following machine step.
Let now $\hat{\mathcal{A}}$ = <S, C, D, H, T> be a quintuple consisting of the five components stack, program, dump, heap and trace (in this order) of $\mathcal{A}$.

$\hat{\mathcal{A}}_{init}$ = <[], EVAL:C_{main}, [], [], []>

is then the initial state of $\hat{\mathcal{A}}$ before the computation of the corresponding main function. EVAL is not an α command, but a *characteristic property of* $\mathcal{A}$ and must not be mixed up with the eval expression in our abstract syntax. This eval expression is meant to force some *additional* evaluation where it is regarded as useful, whilst the EVAL property of $\mathcal{A}$ provides the main function evaluation and reduction.
Depending on the top command of the remaining code sequence, $\hat{\mathcal{A}}$ changes its states as follows:

<[], EVAL:C_{main}, [], [], []> $\Longrightarrow$
<[], C_{main}:reduce, [], [], []>

where C_{main} is the α code sequence for the main expression to be reduced.

<S, id $\rhd$ t:C, D, H, T> $\Longrightarrow$
<id:S, C, D, H, T>

where id is (the name of) a new pointer to some heap cell or black box value refered to by the variable or constant syntactic object t. Of course, a new pointer may point to a previously existing, not necessary newly created, heap object.

<S, id $\rhd$ $\mathcal{E}$ ($proc_f$...):C, D, H, T> $\Longrightarrow$
<id:S', C', D', H', T'>

where id is a pointer to the black box evaluation result of $proc_f$ in strict context. S'=S, C'=C, D'=D, H'=H and T'=T, if the black box evaluation of $proc_f$ has *no* further impact on stack, code, dump, heap or trace (which is of course determined by the code of f). Otherwise, additional changes to S, H and T, as described by the transition rules, are possible.

<S, id $\rhd$ cons (hd, tl):C, D, H, T]> $\Longrightarrow$
<id:S, C, D, H+ $\bowtie_{id}$, T:cons (adr_{hd}, adr_{tl}) at adr_{id}>

where adr_{id}, adr_{hd} and adr_{tl} are some heap addresses given by the black box according to its allocation decisions and id is a pointer identifying the new cell $\bowtie_{id}$.

<S, id $\rhd$ vap ($proc_f$,$a_1 \dots a_n$):C, D, H, T> $\Longrightarrow$
<id:S, C, D, H+$\ominus_{id}$, T:vap ($adr_1 \dots adr_n$) at adr_{id}>

where all adr_j and adr_{id} are addresses supplied by the black box according to the locations of the suspension node and its arguments, and id points to the new cell $\ominus_{id}$.

<S, id $\rhd$ @(pa_1,pa_2):C, D, H, T> $\Longrightarrow$
<id:S, C, D, H+$@_{id}$, T:ap(adr_1, adr_2) at adr_{id}>

where again all adr_j are address attributes given by the black box of $\mathcal{A}$.

<S, if(bool):{$commands_t$}:else:{$commands_e$}:C, D, H, T> $\Longrightarrow$
<S, $commands_t$:C, D, H, T> if bool is pointing to True,
<S, $commands_e$:C, D, H, T> if bool is pointing to False,

where the value of bool is given by the black box of $\mathcal{A}$, and $commands_t$ and $commands_e$ are sequences of α code.

<S, function *ident* $v_1 \dots v_n$:C, D, H, T> $\Longrightarrow$
<#:S, C, D, H, T:use $adr_1 \dots adr_n$>

where # has to be regarded as a special pointer to nowhere and all adr_j are given back by the black box as addresses of the heap objects or values used as function arguments (see section 3.2); in the case that $n = 0$ we have a constant applicative form (Caf).

<p_1:...:p_j:#:S, return:C, D, H, T> $\Longrightarrow$
<r:S, C, D, H, T:return adr_r>

where adr_r given by the black box identifies the result location of the returning function, all pointers $p_1 \dots p_j$ are different from #, r —not always a 'fresh' pointer— points to the function result. (In the case where the returning function has been solving a suspension, the corresponding $\ominus$ is marked at returntime as $\odot$, ready to become updated).

$$\langle p_1{:}\ldots{:}p_j{:}\#{:}S,\ \text{return}{:}C,\ D,\ H,\ T\rangle \Longrightarrow$$
$$\langle S,\ C,\ D,\ H,\ T{:}\text{output}\ adr_r\rangle$$

where the returning function is the main function and adr_r is given by the black box as address of the output structure root.

$$\langle S,\ \text{reduce}{:}C,\ D,\ \ominus_f{:}H,\ T\rangle \Longrightarrow$$
$$\langle p,\ code_f,\ (S,\text{update}{:}\text{reduce}{:}C){:}D,\ \ominus_f{:}H,\ T\rangle$$

where $\ominus_f$ is a suspension of f, the black box has decided to dissolve this suspension by evaluating f, p is a pointer to $\ominus_f$ and the corresponding $code_f$ is loaded from the program memory of $\mathcal{A}$. The update for the node just being reduced is prepared in the dump, but the reduce command remains while there are still further nodes to be reduced.

$$\langle S,\ \text{reduce}{:}C,\ D,\ H,\ T\rangle \Longrightarrow$$
$$\langle S,\ C,\ D,\ H,\ T\rangle$$

where the black box has decided that there is nothing more to reduce.

$$\langle S,\ \text{update}{:}C,\ D,\ \odot_u{:}H,\ T\rangle \Longrightarrow$$
$$\langle S,\ C,\ D,\ R_u{:}H,\ T{:}\text{update root}\ adr_u\ \text{result at}\ adr_{res}\rangle$$

which means $\odot_u$ is to be replaced by R_u where the function belonging to $\odot_u$ is already evaluated to the required kind of normal form (usually WHNF), the black box decided to update the root $\odot_u$ and u points to this updated root. R_u may be a $\bowtie$ or may be 'nothing', which is the case when the result is an invisible value residing in the black box: then $\odot$ simply remains as it is in this model. The address of R_u is adr_u.

$$\langle S,\ [],\ (S',C){:}D,\ H,\ T\rangle \Longrightarrow$$
$$\langle S{:}S',\ C,\ D,\ H,\ T\rangle$$

where the actual program sequence is empty.

$$\langle S,\ [],\ [],\ H,\ T\rangle \Longrightarrow$$
$$\langle [],\ [],\ [],\ [],\ T{:}\text{ready}\rangle = \hat{\mathcal{A}}_{finish}$$

where dump and program stack are empty.
A terminating run of $\mathcal{A}$ may be denoted as $\hat{\mathcal{A}}_{init} \stackrel{*}{\Longrightarrow} \hat{\mathcal{A}}_{finish}$. Otherwise we denote $\hat{\mathcal{A}}_{init} \stackrel{\infty}{\Longrightarrow} \bot$, which is the case when an input program with a non finite semantics is forced to be evaluated.

3.4 examples

As a first example consider the expression main := let v = cons 1 NIL in cons v NIL as introduced for a translation example in chapter 2, and we would now like to compute this expression in order to examine the heap structures which arise from it. First, we need to put α code for main into the program memory of $\mathcal{A}$. Following the translation scheme $\mathcal{T}$ (Fig.4) we obtain, as shown in chapter 2, the following α program:

1) function main;
2) $\text{hd}_1 \rhd 1$;
3) $\text{tl}_1 \rhd \text{NIL}$;
4) $\text{v} \rhd \text{cons}\ (\text{hd}_1,\ \text{tl}_1)$;
5) $\text{hd}_2 \rhd \text{v}$;
6) $\text{tl}_2 \rhd \text{NIL}$;
7) $\text{main} \rhd \text{cons}\ (\text{hd}_2,\ \text{tl}_2)$;
8) return;

For ease of reference and better readability, the particular commands of this program are represented in the following by their line numbers 1) to 8) as shown in the last translation step. With this program in its initial state $\hat{\mathcal{A}}$ will carry out the following state transitions:

$$\hat{\mathcal{A}}_{init} = \langle [],\ \text{EVAL}{:}1{:}2{:}3{:}4{:}5{:}6{:}7{:}8,\ [],\ [],\ []\rangle$$
$$\Longrightarrow \langle [],\ 1{:}2{:}3{:}4{:}5{:}6{:}7{:}8{:}\text{reduce},\ [],\ [],\ []\rangle$$
$$\Longrightarrow \langle \#,\ 2{:}3{:}4{:}5{:}6{:}7{:}8{:}\text{reduce},\ [],\ [],\ \text{use}\rangle$$
$$\Longrightarrow \langle \text{hd}_1{:}\#,\ 3{:}4{:}5{:}6{:}7{:}8{:}\text{reduce},\ [],\ [],\ T_1\rangle$$
$$\Longrightarrow \langle \text{tl}_1{:}\text{hd}_1{:}\#,\ 4{:}5{:}6{:}7{:}8{:}\text{reduce},\ [],\ [],\ T_1\rangle$$
$$\Longrightarrow \langle \text{v}{:}\text{tl}_1{:}\text{hd}_1{:}\#,\ 5{:}6{:}7{:}8{:}\text{reduce},\ [],\ []+\bowtie_{\text{v}},\ T_1{:}\text{cons}\ (adr_1,\ adr_{\text{NIL}})\ \text{at}\ adr_{\text{v}}\rangle$$
$$\Longrightarrow \langle \text{hd}_2{:}\text{v}{:}\text{tl}_1{:}\text{hd}_1{:}\#,\ 6{:}7{:}8{:}\text{reduce},\ [],\ \bowtie_{\text{v}},\ T_2\rangle$$
$$\Longrightarrow \langle \text{tl}_2{:}\text{hd}_2{:}\text{v}{:}\text{tl}_1{:}\text{hd}_1{:}\#,\ 7{:}8{:}\text{reduce},\ [],\ \bowtie_{\text{v}},\ T_2\rangle$$
$$\Longrightarrow \langle \text{main}{:}\text{tl}_2{:}\text{hd}_2{:}\text{v}{:}\text{tl}_1{:}\text{hd}_1{:}\#,\ 8{:}\text{reduce},\ [],\ \bowtie_{\text{v}} + \bowtie_{ret},\ T_2{:}\text{cons}\ (adr_{\text{v}},\ adr_{\text{NIL}})\ \text{at}\ adr_{ret}\rangle$$
$$\Longrightarrow \langle \text{main},\ \text{reduce},\ [],\ \bowtie_{\text{v}}{:}\bowtie_{ret},\ T_3{:}\text{output}\ adr_{ret}\rangle$$
$$\Longrightarrow \langle \text{main},\ [],\ [],\ \bowtie_{\text{v}}{:}\bowtie_{ret},\ T_4\rangle$$
$$\Longrightarrow \langle [],\ [],\ [],\ [],\ T_4;\text{ready}\rangle = \hat{\mathcal{A}}_{finish}$$

In our model it is not necessary to pop the pointers hd and tl as soon as they are not needed any more. For our purpose it is enough to remove them with the entire stack frame of the returning function. The resulting trace reports two Cons cells being created on the heap; the given address adr_v shows, that they are connected as intended by the main expression. (The NIL value is *shared*, but this is not interesting here). T is the abstract output trace reporting this history:

use
cons (adr_1, adr_{NIL}) at adr_{v}
cons (adr_{v}, adr_{NIL}) at adr_{ret}
output adr_{ret}
ready

These statements show the main expressions' effects on the heap. As no function using arguments has occured, the use statement in the trace shows no arguments, and as no suspensions have been built, the final reduce command fizzled out ineffectively. A picture of the result is given in Fig.7.

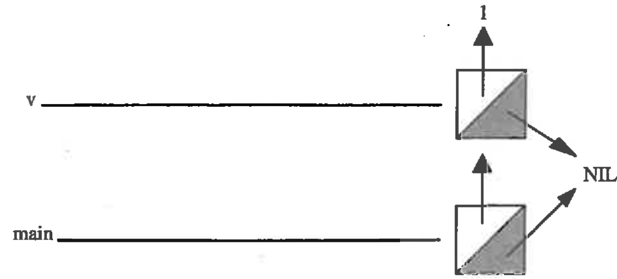


Fig.7 *two cons cells being built by the expression* main := let v = cons 1 NIL in cons v NIL

The second example is about to deal with a simple recursive function *copy*, defined as:

```
type copy :: [*] → [*]
copy ls = if (null_s ls) NIL (cons (head'_s ls) (copy (tail'_s ls)))
```

in abstract syntax, where $*$ denotes some arbitrary finite type and $[*]$ a list with elements of that type. The built in function null appears in strict context because the question, if ls denotes the empty list, has to be answered before one of both alternatives NIL or cons can be chosen. But why are head' and tail' attributed with a little s, though the contexts of cons and copy are generally non strict? The answer is that the special functions head' and tail', in contrast to their ordinary relatives head and tail, do *not* evaluate their argument ls to weak head normal form but *only* return a pointer to the corresponding substructure of ls. Therefore the expression given above translates via $\mathcal{T}$ to the α program:

```
1) function copy ls;
2) a ▷ ls;
3) bool ▷ ℰ (proc_null a);
4) if (bool)
5) {copy ▷ NIL;}
6) else
7) {b ▷ ls;
8) hd ▷ ℰ (proc_head b);
9) c ▷ ls;
10) d ▷ ℰ (proc_tail c);
11) tl ▷ vap (proc_copy, d);
12) copy ▷ cons (hd, tl);}
13) return;
```

The function arguments have been made unique by renaming their pointers a_1 as a, b, c, d, and again the commands are supplied with numbers for ease of reference and better readability later. Now suppose we would like to *copy* the constant list structure (A:(B:NIL)) (residing inside the black box) to examine the corresponding events in the heap. Therefore we define:

main = copy$_s$ (A:(B:NIL))

and translate this constant applicative expression via $\mathcal{T}$ to:

```
14) function main;
15) e ▷ (A:(B:NIL));
16) main ▷ ℰ (proc_copy, e);
17) return;
```

Nota bene: The call of copy is strict in the context of the main *expression because the* main *expression must always be evaluated completely.* Internally, the abstract machine $\mathcal{A}$ keeps the relation: ls = [1,2,NIL]. The examination run has to be started with the main code in the program stack of $\mathcal{A}$ and will then step through the following states:

```
Â_init = <[], EVAL:14:15:16:17, [], [], []>
  ⟹ <[], 14:15:16:17:reduce, [], [], []>
  ⟹ <#, 15:16:17:reduce, [], [], use>
  ⟹ <e:#, 16:17:reduce, [], [], T_1>
  ⟹ <main:e:#, 17:reduce, [], [], T_1>
  ⟹ <p_copy, 1:2:3:4:5:6:7:8:9:10:11:12:13, (main:e:#, 17:reduce), [], T_1>
  ⟹ <#:p_copy, 2:3:4:5:6:7:8:9:10:11:12:13, D_1, [], T_1:use adr_e>
  ⟹ <a:#:p_copy, 3:4:5:6:7:8:9:10:11:12:13, D_1, [], T_2>
  ⟹ <bool:a:#:p_copy, 4:5:6:7:8:9:10:11:12:13, D_1, [], T_2>
  ⟹ <bool:a:#:p_copy, 7:8:9:10:11:12:13, D_1, [], T_2>
  ⟹ <b:bool:a:#:p_copy, 8:9:10:11:12:13, D_1, [], T_2>
  ⟹ <hd:b:bool:a:#:p_copy, 9:10:11:12:13, D_1, [], T_2>
  ⟹ <c:hd:b:bool:a:#:p_copy, 10:11:12:13, D_1, [], T_2>
  ⟹ <d:c:hd:b:bool:a:#:p_copy, 11:12:13, D_1, [], T_2>
  ⟹ <tl:d:c:hd:b:bool:a:#:p_copy, 12:13, D_1, []+⊖_B, T_2:vap (adr_B) at adr_tl>
  ⟹ <copy:tl:d:c:hd:b:bool:a:#:p_copy, 13, D_1, H_1+ ⋈_ret, T_3:cons (adr_A, adr_tl) at adr_ret>
  ⟹ <copy:p_copy, [], (main:e:#, 17:reduce), H_2, T_4:return adr_ret>
  ⟹ <copy:p_copy:main:e:#, 17:reduce, [], H_2, T_5>
  ⟹ <[], reduce, [], H_2, T_5:output adr_ret>
  ⟹ <tl, 1:2:3:4:5:6:7:8:9:10:11:12:13, ([], update:reduce), ⊖_B:H_2, T_6>
  ⟹ <#:tl, 2:3:4:5:6:7:8:9:10:11:12:13, D_2, H_2, T_6:use adr_d>
  ⟹ <â:#:tl, 3:4:5:6:7:8:9:10:11:12:13, D_2, H_2, T_7>
  ⟹ <bool:â:#:tl, 4:5:6:7:8:9:10:11:12:13, D_2, H_2, T_7>
  ⟹ <bool:â:#:tl, 7:8:9:10:11:12:13, D_2, H_2, T_7>
  ⟹ <b̂:bool:â:#:tl, 8:9:10:11:12:13, D_2, H_2, T_7>
  ⟹ <ĥd:b̂:bool:â:#:tl, 9:10:11:12:13, D_2, H_2, T_7>
  ⟹ <ĉ:ĥd:b̂:bool:â:#:tl, 10:11:12:13, D_2, H_2, T_7>
  ⟹ <d̂:ĉ:ĥd:b̂:bool:â:#:tl, 11:12:13, D_2, H_2, T_7>
  ⟹ <t̂l:d̂:ĉ:ĥd:b̂:bool:â:#:tl, 12:13, D_2, H_2 + ⊖_NIL, T_7:vap (adr_NIL) at adr_t̂l>
  ⟹ <copy':t̂l:d̂:ĉ:ĥd:b̂:bool:â:#:tl, 13, D_2, ⊖_B:H_3+ ⋈_new, T_8:cons (adr_B, adr_t̂l) at adr_new>
  ⟹ <copy':tl, [], ([], update:reduce), ⊙_B:H_4, T_9:return adr_new>
  ⟹ <copy':tl, update:reduce, [], ⊙_B:H_4, T_10>
  ⟹ <copy':tl, reduce, [], ⋈_B:H_4, T_10:update root adr_tl result at adr_new>
  ⟹ <t̂l, 1:2:3:4:5:6:7:8:9:10:11:12:13, (copy':tl, update:reduce), ⊖_NIL:H_5, T_11>
  ⟹ <#:t̂l, 2:3:4:5:6:7:8:9:10:11:12:13, D_3, H_5, T_11:use adr_d̂>
```

$\Longrightarrow$ <á:#:$\hat{\mathsf{tl}}$, 3:4:5:6:7:8:9:10:11:12:13, D_3, H_5, T_{12}>
$\Longrightarrow$ <bool:á:#:$\hat{\mathsf{tl}}$, 4:5:6:7:8:9:10:11:12:13, D_3, H_5, T_{12}>
$\Longrightarrow$ <bool:á:#:$\hat{\mathsf{tl}}$, 5:13, D_3, H_5, T_{12}>
$\Longrightarrow$ <copy":bool:á:#:$\hat{\mathsf{tl}}$, 13, D_3, $\ominus_{NIL}$:H_5, T_{12}>
$\Longrightarrow$ <copy":$\hat{\mathsf{tl}}$, [], (copy':tl, update:reduce), $\odot_{NIL}$:H_5, T_{12}:return adr_{NIL}>
$\Longrightarrow$ <copy":$\hat{\mathsf{tl}}$:copy':tl, update:reduce, [], H_5, T_{13}>
$\Longrightarrow$ <copy":$\hat{\mathsf{tl}}$:copy':tl, reduce, [], H_5, T_{13}:update root $adr_{\hat{tl}}$ result at adr_{NIL}>
$\Longrightarrow$ <copy":$\hat{\mathsf{tl}}$:copy':tl, [], [], H_5, T_{14}>
$\Longrightarrow$ <[], [], [], [], T_{14};ready> = $\hat{\mathcal{A}}_{finish}$

All symbols S_i, D_i, H_i and T_i are short forms of previous states of stack, dump, heap and trace, for better readability. In an abstract manner the final trace T documents the history of copying the structure (A:(B:NIL)) from the value domain of the black box into the heap of $\mathcal{A}$:

use	(29)
use adr_e	(32)
vap (adr_B) at adr_{tl}	(36)
cons (adr_A, adr_{tl}) at adr_{ret}	(37)
return adr_{ret}	(38)
output adr_{ret}	(41)
use adr_d	(44)
vap (adr_{NIL}) at $adr_{\hat{tl}}$	(48)
cons (adr_B, $adr_{\hat{tl}}$) at adr_{new}	(49)
return adr_{new}	(50)
update root adr_{tl} result at adr_{new}	(51)
use $adr_{\hat{d}}$	(54)
return adr_{NIL}	(56)
update root $adr_{\hat{tl}}$ result at adr_{NIL}	(57)
ready	(60)

This trace reports a history as described in the following. (The line numbers in brackets are inserted for comparison with a similar trace in the next chapter). First, a strict call of *copy* was done by the main function. Hereby a Cons cell and a suspension node of the *copy* function were built so that the suspension node was the tail (pointed by tl) of the Cons cell, while its head pointed to the (irrelevant) value A. Then the function *copy* was evaluated in order to make its suspension node $\ominus_B$ ready for an update. During this evaluation another Cons cell, (containing a pointer to the value B in its head), and another suspension $\ominus_{NIL}$ of *copy* (pointed to by $\hat{tl}$ as tail of the new Cons cell) were built. After return, the first $\odot_B$ became updated to ⋈ (whereby the result of this update, namely the new Cons cell created secondly, became garbage; but this is not in the scope of our interest in this chapter). A similar procedure was then carried out to evaluate the second $\ominus_{NIL}$, but at this time no further heap cell had been created because the argument of *copy* was NIL, and after returning from *copy* the corresponding $\odot_{NIL}$ was updated with an 'invisible' value NIL residing in the black box evaluation segment of $\mathcal{A}$. Therefore the $\odot$ has remained unchanged in the heap. At last, three ⋈ and one $\odot$ remained in the heap, of which one was garbage and the other ones contained pointers to A, B; but these contents do not touch our interest for structures as purely as possible. Fig.8 shows the events caused by the example program discussed above.

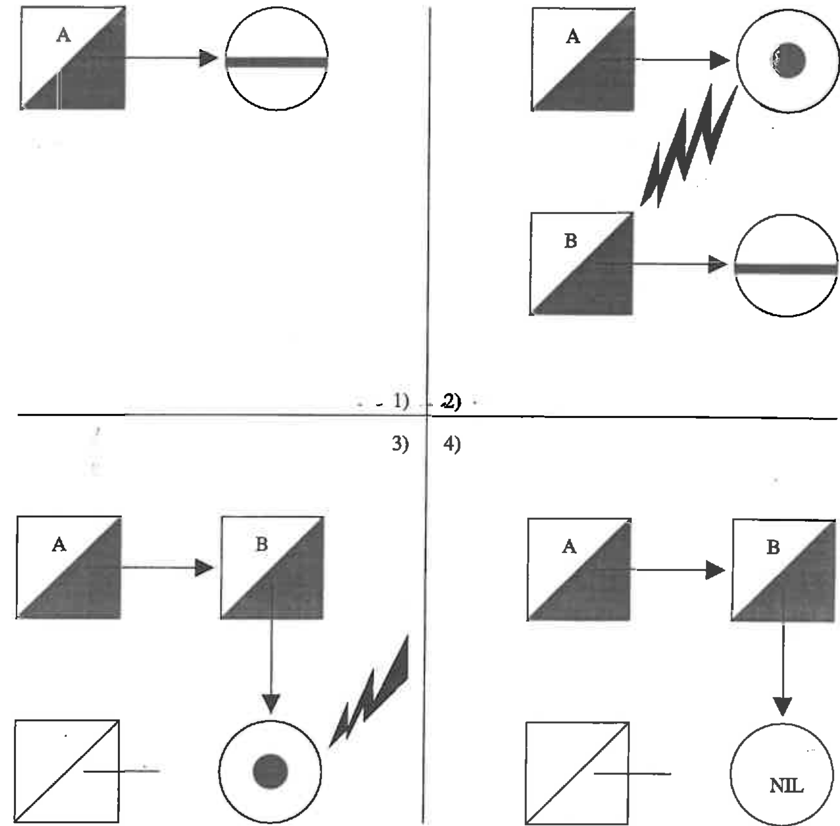


Fig.8 *heap events caused by the expression* main := eval copy$_s$ (A:(B:NIL)), *with 'flashes' depicting updates*

3.5 summary

In this chapter the translation from abstract syntax to α code has been explained and the semantics of it has been given more formally by defining a state transition of the abstract machine $\mathcal{A}$ for each α command. At least, two examples have been exercised, in order to give an intuition about the generation of heap cells by an (abstraction of a) functional program and the abstract output trace given by $\mathcal{A}$ to report those events.
Of course the FAST compiler, as used for investigations on the benefits of deforesting program transformations as described later in this thesis, is a sophisticated machinery and therefore may differ considerably in many cases from its simple abstract model, which the sketched machine $\mathcal{A}$ represents. Nevertheless $\mathcal{A}$ should be a sufficient explanation for the basis of the examination task as mentioned above. The abstraction was made mainly by ignoring unessential information. It is easy to find the way back from abstract traces to the concrete ones by 'covering the skeleton with flesh', which means, for example, replacing the abstract labels adr_i by natural numbers, considering 'boxed' or 'unboxed' variants of Cons cells, etc. This would mean having a *closer* look at the structures, but still the values are unimportant. The next chapter will explain important matters on real FAST output traces and introduce a way of analysing them with a program written in C [12], and also other data structures than lists built of Cons cells will be mentioned.

Chapter 4

analysis methods

4.1 output trace analysis

In the previous chapters it has been shown how output traces which report a history of heap events can be generated in principle. Now we have a look at a method to use the output trace information for reconstructing such histories. This reconstruction will be done by an analysis program in such a way that intermediate cells can be separated from cells belonging to the output structure or other cells. (From a 'meta' point of view, the output trace itself is an 'intermediate structure' between the 'functions' machine and analysis — and 'deforesting' this 'structure' would mean integrating this analysis into the compiler! We also give a reason in this chapter for not performing this kind of 'meta deforestation'.)
Having separated intermediate structures from the output structure, it is possible to reason about efficiency and to compare the run time behaviour of undeforested and deforested versions of the same functional program. The explanations given in this chapter will continue to use the abstract terminology developed in the previous chapters. Only some of the most important FAST properties will be mentioned in order to show what the abstraction is based on; more details concerned with the FAST compiler system can be found in [9] and in the appendix of this thesis.

4.1.1 some FAST properties in general

The FAST compiler performs strictness analysis to save unnecessary function suspensions by calling functions in a strict context immediately. Further, the heap objects usually (but not always) are represented as *boxed*. Boxing is implemented in the FAST system by using another kind of cell, namely the Box cell or just simply Box. A Box contains nothing but a pointer to the object to be boxed, which may be a structure cell as well as a value. The analysis of the FAST output traces must of course not ignore the Boxes, but they are irrelevant from our abstract point of view. Special built in *prelude functions* have to prepare certain function arguments for application according to the demands of the function to be applied: a prelude function evaluates an unevaluated object to the required kind of normal form (usually WHNF), and unboxes a boxed object if it needs to have unboxed form. Later in this thesis we will also have to pay regard to a further kind of cell called *Pack*, for data structures different from (linear) lists, but the principle of the analysis will still be the same.
Combinators without arguments are constant applicative forms called *Caf*. These are statically allocated (i.e. at compile time) and therefore do not reside in the heap. Even if the analysis of FAST traces is meant to reconstruct the dynamic allocations (i.e. the runtime behaviour) in the heap, the Caf cells must not be ignored, because they can become closely connected to the heap by update. One can find examples where the output structure built by a functional program starts

in the heap, then crosses the heap border to the Caf cells outside (in the area called 'black box' in the abstract machine terminology), and finally comes back into the heap again. Ignoring the Caf cells would mean losing the tail of such an output structure. As opposed to the abstract examples in the previous chapters, where the main function did not take argument, the FAST main function always takes one argument of the type 'number', (which may of course be a dummy and need not to be computed then). The FAST output type must always be a list of characters, thus:
type main :: num → [char] .

4.1.2 preparing trace information

The abstract output traces of $\mathcal{A}$ contain in principle all information needed for deforestation analysis. This is information about the allocation of cells, about the way those cells are connected to each other to build structures, about one special cell returned as the output structure root, the use of cells as function arguments, and about updates making changes to the structures during the run of the machine. Analyzing FAST traces, however, we have to keep in mind that there is no 'black box' any more, but one memory area which the heap is only one part of. For distinguishing cells that are allocated statically outside the heap from those ones allocated dynamically inside, we have to get information about the beginning of the heap and in fact we can find it reported in the FAST output trace. Any further information, apart from information about allocation, connection, update, function arguments, first address of the heap and output root, given in the FAST output trace is irrelevant and will be ignored.
All statements of an output trace can be regarded as instances of regular expressions and therefore it is easy to separate the relevant statements from the irrelevant ones automatically. The relevant statements will be translated into procedure calls in the syntax of C and then compiled together with the analysis program also written in this language. More details of the analysis technique and the analysis program itself can be found in the appendix.

4.1.3 automatic analysis

The deforestation analysis program contains a number of procedure definitions which can be divided into two groups. The first group are the *history repetition* definitions which match the output trace statements translated to function calls. The analysis program simulates heap *and* 'black box' in an array, in which 'cells' can be 'allocated', 'connected' and 'updated' according to the information in the trace. The second group is built by the *analysis* definitions, which will be activated as soon as the history repetition is finished. The analysis procedures separate the statically from the dynamically allocated cells (by using the trace information about the starting address of the heap), group the cells into structures (by using the trace information about connections), divide the structures into output structure (by using the trace information about the output root), garbage (by using the trace information about updates) and intermediate structures (the rest).
All heap cells are counted and the relation between the total number of cells and the number of intermediate cells is presented. With information about cells being used as function arguments (which is provided by the trace) further reasoning about the quality of different intermediate structures seems possible; this will be described later in this thesis. Fig.9 shows the analysis steps.

1) take Miranda program
2) compile it using FAST
3) let it run and get an output trace
4) save trace into a file
5) filter it with regular expression patterns
6) translate result to C procedure calls
7) compile and link with analysis program
8) run this program
9) get analysis of heap history caused by the Miranda program

Fig.9 *automatic analysis of the cell allocation history caused by a functional program*

Why should this deforestation analysis not be integrated in the FAST compiler system itself? Of course it could be, but the reasons for not doing this are firstly, not to mess with the already large and complicated FAST system and secondly, to have an adaptable module applicable to other compiler systems as well.

4.1.4 example

Given the function *copy* written in abstract syntax as shown in chapter 3, we now want to analyse the runtime behaviour of this function by following steps 1) to 9) shown in Fig.9. The *copy* program is specified in FAST Miranda as:

```
copy [] = [];
copy (l:ls) = (l:copy ls);

input = 0;
main dummy = copy ['A','B'];
```

(The numerical input demanded by the FAST compiler is a dummy in this case). Having compiled and run this small functional program, the following FAST output trace documenting its history appears:

```
1)   ap_rec     12 bytes
2)   box_rec    8 bytes
3)   caf_rec    12 bytes
4)   cons_rec   8 bytes
5)   double     8 bytes
6)   fun_rec    12 bytes
7)   pack_rec   4 bytes
8)   sd_rec     12 bytes
9)   vap_rec    88 bytes
10)  nde_rec    88 bytes
11)  heap       100000 ints from 3d000 to 9ea7c (start=3c0c0)
12)             version:  34,1,1,0,1,susp_box_ret,susp_unb_arg,normal_typing,
13)             no_nesting
```

```
14)               _01_259_I_:  numb=0 at 036170
15)                   _input:  caf=0032d8 busy=FALSE at 036178
16)                  _LIT281:  caf=0036f4 busy=FALSE at 036184
17)      _01_copy_1100LT_0_1:  proc=00342c prel=002370 arity=1 at 036190
18)        _01_TL_1111LT_0_1:  proc=0089a0 prel=0022b8 arity=1 at 03619c
19)        _01_HD_1111LT_0_1:  proc=006ca0 prel=0022b8 arity=1 at 0361a8
20) _01_CONS_0001T_0001LT_0_2:  proc=00448c prel=0024a4 arity=2 at 0361b4
21)                _11_275_LC:  cons=(0364ac,0361d0) at 0361c0
22)                _01_270_LC:  cons=(0364b4,036218) at 0361c8 box at 0361d0
23)                getrusage:  start_rusage=f7fffbc8
24)        _11_main _11_input:
25)                   reduce:  caf=36178
26)                _01_input:
27)                _01_input:  return_I=36170
28)                   update:  root=36178 result=36170 tag=9
29)                 _11_main:  _00_dummy_T_1=000000
30)                   reduce:  caf=36184
31)              _01_LIT281:
32)                 _01_copy:  _11_1A_LT_0=0361c0
33)                 _11_NULL:  361c0 = false
34)             _00_HD_COUNT:  361c0 at 364ac
35)             _00_TL_COUNT:  361c0 at 361d0
36)                     vap2:  fun=36190,a1=361d0 at 3d000
37)                 _01_CONS:  h=364ac t=3d000 at 3d00c box at 3d014
38)                 _01_copy:  return_LT=3d014
39)               _01_LIT281:  return_LC=3d014
40)                   update:  root=36184 result=3d014 tag=6
41)                 _11_main:  return_LC=3d00c                          A
42)                   reduce:  vap=3d000
43)          prel_1100LT_0:  proc=342c arg=3d008
44)                 _01_copy:  _11_1A_LT_0=0361c8
45)                 _11_NULL:  361c8 = false
46)             _00_HD_COUNT:  361c8 at 364b4
47)             _00_TL_COUNT:  361c8 at 36218
48)                     vap2:  fun=36190,a1=36218 at 3d01c
49)                 _01_CONS:  h=364b4 t=3d01c at 3d028 box at 3d030
50)                 _01_copy:  return_LT=3d030
51)                   update:  root=3d000 result=3d030 tag=6            B
52)                   reduce:  vap=3d01c
53)          prel_1100LT_0:  proc=342c arg=3d024
54)                 _01_copy:  _11_1A_LT_0=036210
55)                 _11_NULL:  36210 = true
56)                 _01_copy:  return_LT=36218
57)                   update:  root=3d01c result=36218 tag=8
58)                getrusage:  cur_rusage=f7fffbc0 sec=0.000000 microsec=70000.
59)
60)                            0.07 user + system seconds for copy.fast.out #1
```


The following lines of this FAST output trace represent an informal abstraction corresponding to the statements given by the abstract output trace of the second example in chapter 3:

29) call of the main function using its dummy argument
32) call of the copy function using its argument
36) dynamic allocation of a Vap cell for copy
37) dynamic allocation of a Cons cell for the copied 'A'
38) return of the copy function
41) return of the main function
44) call of the copy function using its (tail) argument
48) dynamic allocation of a Vap cell for copy
49) dynamic allocation of a Cons cell for the copied 'B'
50) return of the copy function
51) update of the first Vap cell
54) call of the copy function using its (NIL) argument
56) return of the copy function
57) update of the second Vap cell
60) program ready

(See appendix for details). Further we find in lines:

11) the starting address of the heap
15) static allocation of a Caf cell for the dummy 0 (irrelevant)
16) static allocation of a Caf cell for the program output
21) static allocation of a Cons cell for the input 'A'
22) static allocation of a Cons cell for the input 'B'
28) update of the dummy Caf (irrelevant)
40) update of the output Caf

The other lines may be completely ignored. Filtering the irrelevant statements out and translating the other ones into C procedure calls, we obtain as a formal abstraction:

```
heapstart(0X3d000);                                             (11)
caf(0X036178);                                                  (15)
caf(0X036184);                                                  (16)
cons(0X0364ac,0X0361d0,0X0361c0);                               (21)
cons(0X0364b4,0X036218,0X0361c8); box(0X0361c8,0X0361d0);       (22)
update(0X36178,0X36170);                                        (28)
use(0X000000);                                                  (29)
use(0X0361c0)                                                   (32)
vap(0X361d0,UNDEF,UNDEF,UNDEF,0X3d000);                         (36)
cons(0X364ac,0X3d000,0X3d00c); box(0X3d00c,0X3d014);            (37)
update(0X36184,0X3d014);                                        (40)
outputroot(0X3d00c);                                            (41)
use(0X0361c8);                                                  (44)
vap(0X36218,UNDEF,UNDEF,UNDEF,0X3d01c);                         (48)
cons(0X364b4,0X3d01c,0X3d028); box(0X3d028,0X3d030);            (49)
update(0X3d000,0X3d030);                                        (51)
use(0X036210);                                                  (54)
update(0X3d01c,0X36218);                                        (57)
```

and now the structural similarity to the abstract trace from the example given in chapter 3 is visible! When these calls are passed to the analysis program, the following output appears:

```
6 Nodes:  2 Box, 2 Cons, 2 Vap
2 VapUpd:  1 from inside, 1 from outside
0 Vap remain without being updated
1 cell became garbage updating inside Vap
1 cell became garbage updating outside Vap or Caf
4 cells belong to the output structure
```

Ignoring to the Box cells, this result corresponds perfectly to what we may expect after having studied the example of the abstract *copy* function in chapter 3. The analyser has found two Vap cells and two Cons cells being built; both Vap are found being updated, one with a structure cell from inside the heap, the other one with 'something' from outside the heap (which we know to be the NIL value). The simple Program main dummy = copy ['A','B'] has not created any intermediate structure.

4.2 α code analysis

The automatic analysis as introduced in the section above is not the only way of analysing the runtime behaviour of a functional program. We could also express a functional program in terms of our abstract syntax and observe the corresponding α-code in order to detect some runtime properties we are interested in, because all runtime properties of a functional program are determined by the machine code the program is translated into. (Examples of this code analysis method are given in the following chapters). If this is the case, at least now the question has to arise as to why the trace analysis machinery as presented above has been developed at all. The answer is that the possibilities of α code analysis are quite restricted in practice. Fig.10 shows a comparison between both analysis methods.

AUTOMATIC TRACE ANALYSIS:

theoretically applicable for all traces
practically applicable for a big subset of all traces
cell allocations *a posteriori* for one input
empirically tested

ALPHA CODE ANALYSIS:

theoretically applicable for all data independent functions
practically applicable only for small functions
cell allocations *a priori* for all possible inputs
mathematically sound

Fig.10 *properties of the analysis methods in comparison*

Chapter 5

the concatenate vanishes, 1987/89

In the previous chapters the relevance of pure structures for deforestation analysis has been emphasized and a method of automatically finding those structures created during the run of a functional program has been developed. In this and the following chapters, some examples found in the literature on deforestation will be examined. The analysis program introduced in the previous chapter is able to show the cell claim gain caused by the deforesting program transformations applied to these certain examples. Moreover, where these examples are simple enough to be analyzed with the α code method, there can be seen not only a correspondence between these results predicted by such a theoretical analysis and those found by the analysis program but also a certain limit of gain for each case, a limit which deforestation can never overcome. This limit is established by the implementation of the functional language itself, which is used for writing down those example programs to be examined.

5.1 the function reverse

5.1.1 ordinary version

As already mentioned in the introductory chapter of this thesis, Wadler's treatise *"the concatenate vanishes"* upon list processing shows how the list concatenating *append* function can be avoided. The following function *reverse* is shown to benefit from this avoidance. It can be defined straight forwardly in abstract syntax as:

type reverse :: [*] → [*]
reverse xs = if (null$_s$ xs) NIL (app (reverse$_s$ (tail'$_s$ xs)) (cons (head'$_s$) NIL))

with the strictness information provided as described in the previous chapters. The functions reverse and app are found in a composite context, whereby the inner function reverse will produce an intermediate list structure to be consumed by the outer function app, which is responsible for the output structure. This auxiliary list concatenating function is defined as:

type app :: [*] → [*] → [*]
app xs ys = if (null$_s$ xs) ys (cons (head'$_s$ xs) (app (tail'$_s$ xs) ys))

How many heap nodes will be created, when we are to *reverse* some list $l = [x_1, \ldots, x_n]$? How will the number of heap nodes depend on the length of the input list l? To answer these questions, we translate the functions and have a look at the resulting α code. Together with both function definitions above, we specify: main = reverse$_s$ [A,B,C] and translate it into α code, too.

Of course, the length of the input list [A,B,C] is only 3 and not some variable n; nevertheless we may hope to find information in the target code sufficient for predicting the cell claim behaviour for every input size n. These are the definitions given above, after being translated by $\mathcal{T}$:

```
function reverse xs;
b ▷ xs;
bool ▷ E(proc_null, b); ..................................*
if(bool)
{reverse ▷ NIL;} ........................................**
else
{d ▷ xs;
c ▷ E(proc_tail', d);
a1 ▷ E(proc_reverse, c); ...............................***
e ▷ xs;
hd ▷ E(proc_head', e);
tl ▷ NIL;
a2 ▷ cons(hd, tl); ....................................****
reverse ▷ E(proc_app, a1, a2);} .......................*****
return;

function app xs ys;
b ▷ xs;
bool ▷ E(proc_null, b);
if(bool)
{app ▷ ys;}
else
{c ▷ xs;
hd ▷ E(proc_head', c);
d ▷ xs;
a1 ▷ E(proc_tail', d);
a2 ▷ ys;
tl ▷ vap(proc_app, a1, a2);
app ▷ cons(hd, tl);}
return;

function main;
f ▷ [A,B,C];
main ▷ E(proc_reverse, f);
return;
```

and now we can describe the situation as follows:

main calls reverse in a strict context.
reverse does nothing if its argument is empty.
reverse otherwise claims one Cons (for the list head)
and calls the functions reverse and app in strict contexts.
app does nothing, if its first argument is empty.
app otherwise claims one Cons and one Vap for the function app.

This description informally represents an *a posteriori* strictness analysis, corresponding with the formal strictness analysis done by the FAST compiler. Knowing that the reduction of the main proceeds as follows:

```
main ⟹
reverse [A,B,C] ⟹
app (reverse [B,C]) [A] ⟹
app (app (reverse [C]) [B]) [A] ⟹
app (app (app (reverse []) [C]) [B]) [A] ⟹
app (app (app [] [C]) [B]) [A] ⟹
app (app [C] [B]) [A] ⟹
app (C:app [] [B]) [A] ⟹
app [C,B] [A] ⟹
(C:app [B] [A]) ⟹
(C:B:app [] [A]) ⟹
[C,B,A]
```

it is now easy to see that, if n is the length of the input list in general, there must occur:

$n+1$ calls of reverse
of which n are *cell productive*[1], and
n calls of app (done by reverse).
If $n > 0$, $n-1$ of them will cause a whole cascade of further calls as follows:
The first app causes no further cell productive call.
The second app causes 1 cell productive call of app.
...
The $n-1^{th}$ app causes $n-2$ cell productive calls of app.
The nth app causes $n-1$ cell productive calls of app.

Knowing the number of cells allocated in every cell productive call (each Cons and each Vap is one cell) we are now able to sum everything together. For this purpose we informally introduce a small function named 'Cells', which describes the allocation behaviour of a functional program, and takes arguments corresponding to those ones of the functional program to be described. Thus, for the example given above, we denote:

$$\begin{aligned}
\text{Cells}_{main}(n) &= \text{Cells}_{app}(n) + \text{Cells}_{rev}(n) \\
&= (2*1+2*2+\ldots+2(n-2)+2(n-1)) + (1*n) \\
&= 2(1+2+\ldots+(n-2)+(n-1)+n) - n \\
&= 2\tfrac{n(n+1)}{2} - n \\
&= n(n+1) - n \\
&= n^2+n-n \\
&= n^2.
\end{aligned}$$

So this is the formula for the cell allocation behaviour when a list of length n is to be *reversed*. We can see the heap growing in quadratic order, as Wadler remarks in his treatise. In the following section a proof for the cell claim formula developed above is given.

[1] We consider a function call as *cell productive*, if it will allocate at least one new cell in the heap.

5.1.2 runtime formula proof

The proof for the runtime behaviour n^2 of the list reverting program main = reverse$_s$ [...] is given by induction on the length l of the input list and based on the α code of the functions **reverse** and **app**.

Base case: $l = 0$, i.e. xs=[].
It is to show: Cells$_{main}$(0)=0.
Observing the α code for **reverse** we find out:
$\mathcal{E}$(proc$_{null}$, []) $\Longrightarrow$ TRUE. .... *
{reverse $\triangleright$ NIL;}. .... **
NIL is a value outside the heap.

Induction step: $l = n + 1$, i.e. xs=$[x_1, x_2, \ldots, x_{n+1}]$.
Hypothesis: let Cells$_{main}(n)$= n^2 be already proved.
It is to show: Cells$_{main}(n+1)$=$(n+1)^2$.
Observing the α code we find:
$\mathcal{E}$(proc$_{null}$, $[x_1, \ldots, x_{n+1}]$) $\Longrightarrow$ FALSE. .... *
$\mathcal{E}$(proc$_{reverse}$, $[x_2, \ldots, x_{n+1}]$) .... ***
$\mathcal{E}$(proc$_{app}$, $[x_{n+1}, \ldots, x_2]$, $[x_1]$) .... *****
cons(x_1, NIL) .... ****
Thus: Cells$_{main}(n+1)$ = Cells$_{main}(n)$ + Cells$_{app}(n)$ + 1.
With a similar induction proof it can be shown that: Cells$_{app}(n) = 2n$.
Therefore we conclude: $n^2 + 2n + 1 = (n+1)^2$.

Quod erat demonstrandum.

Nota bene: Because of the lazy evaluation, the cell claim formula n^2 as proved above is only valid if the **reverse** *function is forced to be evaluated completely, which is the case when that function is called directly by* **main**. The formula is not valid, for example, in main = head (reverse [...]), where the argument list is reverted only as far as the function **head** demands it.

Fig.11 shows the output of the automatic analysis of that list reverting functional program —this time written in Miranda and translated by the FAST compiler— with a input list length of 50: as the allocation formula predicts, $50^2 = 2500$ cells are found in the heap, if we do not pay regard to those $3727 - 2500 = 1227$ Boxes belonging to the irrelevant characteristics of the FAST system. The analysis program finds, further, as many cell structures as members of the input list, namely 50, of which one structure is output and 49 are intermediate. These are caused by the $n - 1$ cell productive calls of the intermediate **app** function by the function **reverse** as explained above. (For comparison: a program written in a traditional imperative style can do the same reverse job on a memory area consisting of 50 places).

5.1.3 deforested version

What is there to be gained by deforestation now? According to Wadler, the deforested version of the *reverse* function is given in abstract syntax as:

```
type reverse :: [*] → [*]
reverse xs = revhelp_s xs []
```



```
type revhelp :: [*] → [*] → [*]
revhelp xs ys = if (null_s xs) ys (revhelp_s (tail_s' xs) (cons (head_s' xs) ys))
```

The function **app** has disappeared and the former intermediate **reverse** has been 'lifted' to the output producing outer position. Again we translate the closed expression

main = reverse$_s$ [A,B,C];

together with the new definitions just given above, and find this time the following information in the corresponding α code produced by $\mathcal{T}$:

main calls **reverse** in strict context.
reverse calls **revhelp** in strict context.
revhelp does nothing if its first argument is empty.
revhelp otherwise claims a Cons cell
and calls itself in strict context.

Having exercised the analysis in the previous section, we can easily see now that n cells will be allocated in the heap, if some list of length n is to be *reversed* according to the new deforested definition; therefore we denote:

$$\text{Cells}_{defor}(n) = n.$$

Corresponding to this exact result, the analysis program can find for the FAST translated Miranda version of the deforested **reverse** function 50 Cons cells being allocated in the heap, where a list of 50 elements has been reversed; 50 Boxes have to be regarded as irrelevant FAST characteristics. Fig.11 shows the automatic analysis result. Abstracting from the Box cells, in both the ordinary and the deforested case (Fig.11 and Fig.12), the output structure contains 50 Cons cells as we expect, knowing that the input list contained 50 value elements.

5.1.4 deforestation gain

To obtain the ***absolute deforestation gain***, the function Cells$_{defor}(n)$ has to be subtracted from the function Cells$_{main}(n)$ given above:

$$\text{Gain}_{abs} = n^2 - n = n(n-1)$$

It is not surprising that this is exactly the same function as Cells$_{app}(n)$ from above, as just this *append* function was avoided by the concatenate vanishing transformation. According to Wadler's claim the transformation has changed the runtime behaviour from quadratic to linear order, thus: the longer the input, the more is to be gained.
It is also possible to measure a ***relative deforestation gain*** as:

$$\text{Gain}_{rel} = \frac{n^2}{n} = n$$

which is a term of pure linear order $\mathcal{O}(n)$ convergent to ∞ with increasing $n \in \mathcal{N}$, as expected.

```
3727 Nodes:  1227 Box, 1275 Cons, 1225 Vap
1225 VapUpd:  1225 from inside, 0 from outside
0 Vap remain without being updated
1225 cells became garbage updating inside Vap
1 cell became garbage updating outside Vap or Caf
99 cells belong to the output structure
2402 cells are intermediate and belong to
49 different intermediate structures
---
64.45 per cent of all cells are intermediate
```

Fig.11 *runtime behaviour of the ordinary reverse function with input length 50 when examined automatically*

```
100 Nodes:  50 Box, 50 Cons
1 cell became garbage updating outside Vap or Caf
99 cells belong to the output structure
0 cells are intermediate
```

Fig.12 *runtime behaviour of the improved reverse function with input length 50 when examined automatically*

5.2 quicksort

Quicksort is a common algorithmic solution of a list sorting problem and Wadler gives two versions of it as well, with and without making use of the *append* function. This example is difficult to analyse theoretically, as the runtime behaviour of the *quicksort* function does not only depend on the length of the input list given to it, but also on the element order *in* it. Already for short input lists there are hundreds and thousands of possibilities. Only the best-, worst- and average cases of the *quicksort* runtime are known as $\mathcal{O}(n)$, $\mathcal{O}(n^2)$ and $\mathcal{O}(n \log n)$. Thus, this time we do not analyse the α code generated out of definitions in abstract syntax, but use Miranda and the FAST compiler instead, and trust in the results of the analysis program.

5.2.1 ordinary version

The *quicksort* function and its auxiliary functions *above* and *below* are given in FAST Miranda as:

```
app :: [*] → [*] → [*];
app [] ys = ys;
app (x:xs) ys = (x:app xs ys);

below :: num → [num] → [num];
below x [] = [];
below x (y:ys) = (y:below x ys), if y < x;
               = below x ys, otherwise;
```



```
above :: num → [num] → [num];
above x [] = [];
above x (y:ys) = (y:above x ys), if y ≥ x;
               = above x ys, otherwise;

qsort :: [num] → [num];
qsort [] = [];
qsort (x:xs) = app (qsort (below x xs)) (app [x] (qsort (above x xs)));
```

To show the dependence on the content of the input list, we try out *quicksort* with three different input lists. We examine the call:

```
main :: num → [char];
main dummy = map decode (qsort list);
```

where map decode defined internally converts the qsorted list into a list of characters according to the demands of the FAST compiler's output type, and *list* will be replaced by one of the following list expressions, each of length 52:

a) [111,97,109,99,107,101,105,106,104,102,103,100,108,98,110,112,122,113,116,114,115,117,120, 118,121,119,79,65,77,67,75,69,73,71,72,70,74,68,76,66,78,80,90,81,84,82,83,85,88,86,89,87]

b) [122,85,90,67,122,103,112,109,114,99,82,101,72,73,78,97,114,107,120,97,114,99,78,79,102,119,104, 75,122,65,74,87,118,103,112,117,80,119,82,75,114,107,90,117,102,89,116,105,102,111,120,109]

c) [65,108,121,102,65,112,71,98,85,90,83,88,73,74,87,68,81,112,121,88,111,112,105,114,81,78,117, 106,85,112,109,98,103,74,97,122,99,66,67,106,115,120,71,76,111,78,71,106,99,82,75,122]

The numbers of list a) are unique, while repetitions occur in lists b) and c). Here are the results given by the analysis program:

for a) 1738 allocations altogether, of which 1070 are intermediate and 156 output
for b) 1706 allocations altogether, of which 1046 are intermediate and 156 output
for c) 1856 allocations altogether, of which 1145 are intermediate and 156 output

We can see the number of allocations in these three cases vary slightly, but of course the size of the output structure does not vary from case to case: it is $156 = 3 * 52$, where 52 has been the size of the input structure. How is this relation to be explained? As a result of the *quicksort* function, the FAST system will always return a list consisting of 104 cells, which are 52 Boxes and 52 Cons. (It is the auxiliary functions *below* and *above* which are responsible for the variation in the number of total allocations). But it must not be forgotten that **map decode** is applied to convert the list of numbers into a list of characters as the final step! Thereby also a list of altogether 104 Boxes and Cons cells has been built, but now the content of each Cons is not directly a value, but a Vap which later has been updated with a value (but still remains as an allocated heap cell). The output structure is shown in Fig.13.

5.2.2 deforested version

Now the definition of quicksort is going to be deforested. Wadler gives the appendless version as:

```
qsort :: [num] → [num];
qsort zs = quick zs [];

quick :: [num] → [num] → [num]; quick [] ys = ys;
quick (x:ys) ys = quick (below x xs) (x:quick (above x xs) ys);
```
[2]

Without using an additional list concatenating function, a better performance might be expected. With the analysis program we find:

for **a)** 1292 allocations altogether, of which 755 are intermediate and 156 output
for **b)** 1349 allocations altogether, of which 793 are intermediate and 156 output
for **c)** 1349 allocations altogether, of which 793 are intermediate and 156 output

Not the improved performance in all cases, but the changed relation between them should be surprising. The best case now turns out to be a) rather than b), while b) and c) accidentally show the same runtime behaviour. Still there are lots of intermediate heap cells, as the deforestation has not touched the (intermediate) auxiliary functions *below* and *above* at all.

5.3 summary

In the first deforestation example in this chapter —the function *reverse*— we have been able to show the possible decrease of cell claims caused by deforestation in a theoretic manner. This has been done by observing the α code of the corresponding function definition specified in abstract syntax. The analysis program has been found to give the same results as the analysis 'by hand' in this example. In the second example —*quicksort*— where the α code analysis has been impossible because of a data dependent function definition ("<", "≥"), the automatic analyser has found in three cases, that the deforesting program transformation technique results in "a slightly better performance" as Wadler claims. The outputs of deforested and ordinary versions of one and the same program *must* of course be the same, as an optimising transformation affecting the semantics of a program would be something other than an optimiser. But, as the example above shows in the differing size relations between the three cases in ordinary and deforested versions, the *way* to the result may be changed considerably by the deforesting transformation: Fig.14 gives an intuitive picture of the changes in *quicksort's* runtime behaviour caused by "the concatenate vanishes" described in [18]. The relative deforestation gain might be roughly estimated as $\text{Gain}_{rel} \approx 1800{:}1300 \approx 1.38$.

[2]The programming technique of collecting the results 'on the way' in an additional variable, here *ys*, is called *accumulator technique* — see also the *reverse* example given given before. In some cases it is possible to replace recursion by tail recursion, which is a recursive simulation of iteration, by using accumulator technique.

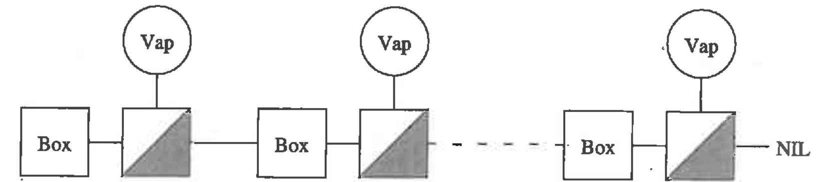


Fig.13 *output structure built by* main dummy = map decode (qsort *list*); *with 3 cells per each input value*

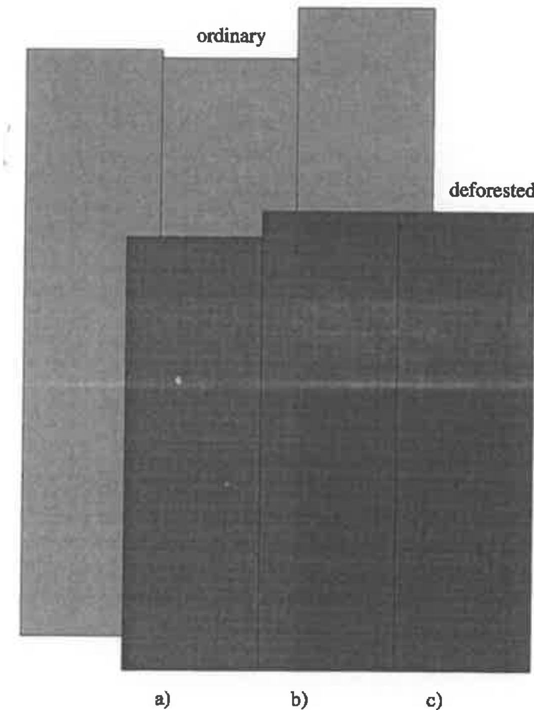


Fig.14 *three cases of quicksorted lists, each with and without use of the append function*

Chapter 6

deforestation: transforming..., 1988/90

Wadler's treatise *"deforestation: transforming programs to eliminate trees"* shows under which circumstances intermediate structures —not only lists but treelike structures in general— can be deforested, mainly by application of seven transformation rules. The main difference between the approaches *the concatenate vanishes* and *deforestation* from a program's point of view is that the first one 'lifts' an intermediate function to an output position by avoiding the, so to say, 'outermost' append function, whilst the second one avoids *'inner'* functions leaving the output function on the same level as it was before. Let us now have a look at one of the examples, which we regard as representative, in the treatise [19]. This example is interesting because it shows well how the classic method of deforestation works, but it is also possible to gain the same effects more cheaply in this case than by roaming through the forest of the seven rules.

6.1 an appending composite for three lists

Knowing a function to append two lists to each other, and wishing to append three lists into one, we could simply call:

app (app l_1 l_2) l_3

where all l_i are some list returning expressions and the function app is defined as in the previous chapter. The inner append concatenates the lists l_1 and l_2 to an intermediate list, which is consumed by the outer append for concatenationn with the list l_3. What runtime behaviour may be expected when the length of the list l_1 is n_1, the length of l_2 is n_2 and the length of l_3 is n_3? Knowing the allocation behaviour of app as $\text{Cells}_{app}(n) = 2n$ from an observation of the α code generated out of the abstract syntax definition as shown in the previous chapter, it is now easy to see that the cell claim formula of the appending function composite can now be written as:

$$\text{Cells}_{total}(n_1,n_2,n_3) = \text{Cells}_{inner}(n_1,n_2) + \text{Cells}_{outer}(n_1,n_2,n_3) = 2n_1 + 2(n_1 + n_2) = 4n_1 + 2n_2$$

independent of n_3 and for $n \geq 0$. The inner append has returned an intermediate list consisting of $\text{Cells}_{inner}(n_1,n_2) = 2n_1$ cells independent of n_2. Thus, if now the double append expression is to be deforested, a new cell claim behaviour described by:

$$\text{Cells}_{defor}(n_1,n_2,\bullet) = \text{Cells}_{outer}(n_1,n_2,\bullet) = 2n_1 + 2n_2$$

may sensibly be expected. Let us check now whether this expectation will be fullfilled.

6.2 deforested version

Out of a composite of two old *append* functions, each with *two* arguments, the deforestation transformation creates one new append function with *three* arguments. This new 'multi append' function is denoted in abstract syntax as:

```
type ap̂p :: [*] → [*] → [*]
ap̂p xs ys zs = if (null_s xs) (apphelp_s ys zs) (cons (head_s' xs) (ap̂p (tail_s' xs) ys zs))

type apphelp :: [*] → [*]
apphelp ys zs = if (null_s ys) zs (cons (head_s' ys) (apphelp (tail_s' ys) zs))
```

Nota bene: The function apphelp *is the same function as the old append 'modulo' renaming.*
What will happen if three lists l_1, l_2 and l_3 are to be appended with the new function?
Let us consult the α code of the new definition in order to answer this question.

```
function ap̂p xs ys zs;
b ▷ xs;
bool ▷ ℰ(proc_null, b);
if (bool)
{c1 ▷ ys;
c2 ▷ zs;
app ▷ ℰ(proc_apphelp, c1, c2);}
else
{d ▷ xs;
hd ▷ ℰ(proc_head, d);
e ▷ xs;
a1 ▷ ℰ(proc_tail, e);
a2 ▷ ys;
a3 ▷ zs;
tl ▷ vap(proc_app, a1, a2, a3);
app ▷ cons(hd, tl);}
return;
```

An analysis analogus to that one shown in the previous chapter reveals that, if n_1 is the length of l_1 and n_2 the length of l_2, there will occur:

$n_1 + 1$ calls of ap̂p, of which n_1 are cell productive, and
$n_2 + 1$ calls of apphelp, of which n_2 are cell productive.

The α code resulting from the translation by $\mathcal{T}$ tells us that both ap̂p and apphelp are claiming two cells —one Cons and one Vap— in each productive call. Therefore we can now describe the new runtime behaviour by the formula:

$$\text{Cells}_{defor}(n_1,n_2,\bullet) = 2n_1 + 2n_2$$

and can easily prove that $\text{Cells}_{defor} = \text{Cells}_{outer}$ as expected. (Of course all these runtime formulas are only valid if the corresponding functional expressions are completely evaluated).

6.3 deforestation gain

For better readability let us now replace n_1 by n and n_2 by m. Then the formula describing the absolute gain of the deforesting transformation from the appending composition to the new ap̂p function with its three arguments can easiliy be written down, only dependent on n, as:

$$\text{Gain}_{abs}(n,\bullet,\bullet) = \text{Cells}_{total}(n,m,\bullet) - \text{Cells}_{defor}(n,m,\bullet) = \text{Cells}_{inner}(n,\bullet) = 2n$$

More interesting than this is the relative deforestation gain, dependent on n and m, which is written as

$$\text{Gain}_{rel}(n,m,\bullet) = \frac{4n+2m}{2n+2m} = 1 + \frac{1}{1+\frac{m}{n}}$$

With this formula it is easy to estimate between which limits this relative Gain must be found.

6.3.1 m⟶ ∞ while n is fixed

$1 + \frac{1}{1+\frac{m}{n}} \longrightarrow 1 + \frac{1}{1+\infty} = 1 + 0 = 1$ for $m \longrightarrow \infty$

This result has to be regarded as a *speedup factor* and it means: the bigger m is, the less can be gained by deforesting the original composed expression.

6.3.2 n⟶ ∞ while m is fixed

$1 + \frac{1}{1+\frac{m}{n}} \longrightarrow 1 + \frac{1}{1+0} = 1 + 1 = 2$ for $n \longrightarrow \infty$

This means: however large n may be, the speedup factor gained by the deforestating transformation can never overcome the limit of constant 2. Thus, it is impossible to append three lists more than twice as fast with the new appending method than we have been able to do with the old composition method given above.

6.3.3 n,m⟶ ∞ while m = n ∗ k is constant

$1 + \frac{1}{1+\frac{m}{n}} \longrightarrow 1 + \frac{1}{1+k}$ for $n, m \longrightarrow \infty$, $\frac{m}{n} = k$ constant.

This will probably be the most frequent case in all practical applications of list concatenation. Assuming, l_1 and l_2 have almost the same length, then $k \approx 1$ and the deforestation speedup is about a factor of 1.5.

6.4 a more sophisticated appending composite

The expensive deforestation of the expression (app (app xs ys) zs) has been digging its own grave by delivering as byproduct "a proof that append is associative", as Wadler himself emphasizes. That means:

$$(\mathsf{app}\ (\mathsf{app}\ xs\ ys)\ zs) = (\mathsf{app}\ xs\ (\mathsf{app}\ ys\ zs))$$

and about the expression on the right hand side of this equation we read in [7]: "This term contains no intermediate structures because the structure append ys zs will appear directly in the result of the term".
And in fact, a cell claim analysis analogus to the ones shown above will result in the same formula

$$\mathrm{Cells}_{soph}(n_1,n_2,\bullet) = 2n_1 + 2n_2$$

for the sophisticated appending composite in the case of the deforested version.

6.5 summary

In this chapter, the deforestation of a composite of two ordinary *append* functions has been studied. The example has turned out to be somewhat unfortunate because it has made itself superfluous by returning a positive result. Nevertheless, we must not regard this misfortune as important as there are other examples without this 'self betraying' effect. A more important result is that there is a principal gain limit for the deforestation of the example studied here. As both the cell allocation formulae of the ordinary and the deforested version of the three list concatenation have order $\mathcal{O}(n)$, the relative deforestation gain between them must have order $\mathcal{O}(1)$, a number which is determined by the coefficients of n in the allocation formulae. This limit is factor 2 in the (unreachable) best case. Factor 1 —no gain— is the (also impossible) worst case, and in between these borders the actual deforestation gain is to be found: it may perhaps be about factor 1.5 in a lot of realistic applications. The decreasing speedup for $n = 100$ with m increasing from 50 to 500 is shown in Fig.15.

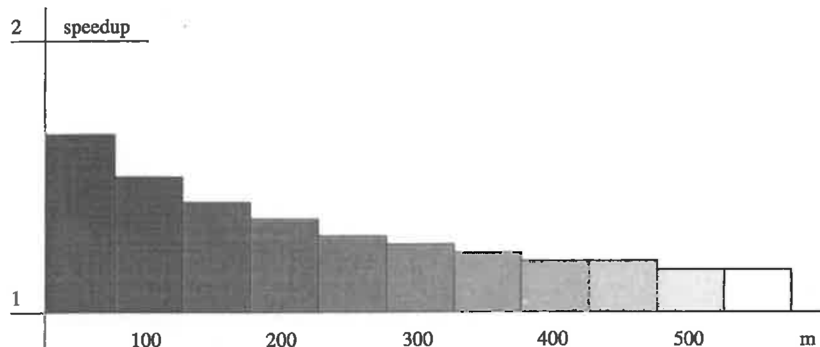


Fig.15 *decreasing deforestation speedup for increasing m with $n = 100$ constant*

Chapter 7

extending deforestation..., 1991

In their treatise *"extending deforestation for first order functional programs"* Hamilton *et* Jones present an example of a function composition which is not deforestable by Wadler's seven rules. The reason for this is that the intermediate function is not *creative*. A function is creative, if every evaluation of it will return a structure building cell, i.e. a Cons or a Pack. A function is not creative, for example, if it returns a suspension node. If an intermediate function is not creative one of Wadler's rules, the *unfold* rule, can not be applied with a useful result. Nevertheless it is possible to deforest this example composition by another algebraic transformation.

7.1 append-reverse, undeforestable version

Have a look at the example this chapter is about to deal with. It is a composed functional expression, denoted in abstract syntax as:

app (rev$_s$ xs ys) zs

where xs, ys, zs are list expressions. The rev function used in this expression is exactly the same function (modulo renaming) as revhelp in chapter 5, where it occured as a result of the deforestation of the function reverse. The definition of app is the same as in all previous chapters. It is possible to predict the runtime behaviour of this composed expression if the length of xs is n and the length of ys is m. (The length of zs is irrelevant because the connection to this last list structure is done by a single pointer). Translating the expressions via $\mathcal{T}$ to α code, as shown in the previous chapters, we can find information about their runtime behaviour as follows:

main calls rev in strict context at first.
main calls app in strict context afterwards.
rev does nothing if its first argument is empty.
rev otherwise claims a Cons cell and calls itself in strict context.
app behaves as described in chapter 5.

Therefore, it is now easy to see that:

$$\begin{aligned}\mathrm{Cells}_{tot}(n,m) &= \mathrm{Cells}_{rev}(n) + \mathrm{Cells}_{app}(n+m)\\ &= n + 2(n+m)\\ &= 3n + 2m.\end{aligned}$$

7.2 an algebraic transformation

Let us use the notation xs ++ ys instead of app xs ys for better readability in this section. It is possible to save some heap cells by applying a transformation based on the following

transformation rule: ((rev xs ys) ++ zs) = (rev xs (ys ++ zs))

The proof, which also can be found in [2], is given by complete induction over the length of xs. It makes use of the

associativity theorem: (xs ++ ys) ++ zs = xs ++ (ys ++ zs)

which is proved, for example, in [19]. Here is the proof of the validity of the rule given above:

Base case: xs = []
It is to show: (rev [] ys) ++ zs = rev [] (ys ++ zs).
(rev [] ys) ++ zs = (ys) ++ zs *per definitionem.*
rev [] (ys ++ zs) = (ys ++ zs) *per definitionem.*

Induction step: xs = (w:ws)
Hypothesis: let (rev ws ys) ++ zs = rev ws (ys ++ zs) be already proved.
It is to show: (rev (w:ws) ys) ++ zs = rev (w:ws) (ys ++ zs).
(rev (w:ws) ys) ++ zs = (rev ws (w:ys)) ++ zs *per definitionem.*
(rev ws (w:ys)) ++ zs = rev ws ((w:ys) ++ zs) *per hypothesis.*
rev ws ((w:ys) ++ zs) = rev ws (w:(ys ++ zs)) *per theorem.*
rev ws (w:(ys ++ zs)) = rev (w:ws) (ys ++ zs) *per definitionem.*

Quod erat demonstrandum.

7.3 append-reverse, transformed version

Using our experience from the α code analysis as shown in the previous chapters, we can see now immediately that the runtime behaviour of the new expression

rev$_s$ xs (app ys zs)

is this time describeable as

$$\text{Cells}_{trans}(n,m) = \text{Cells}_{app}(m) + \text{Cells}_{rev}(n) = 2m + n$$

if n is the length of xs and m the length of ys. Still we find both functions rev and app in a composite relation, nevertheless the optimisation is complete as the subterm (app ys xs) is not intermediate but belongs directly to the result of rev.

7.4 transformation gain

Again we describe the absolute and the relative transformation gains with the difference

$$\text{Gain}_{abs}(n,\bullet) = \text{Cells}_{tot}(n,m) - \text{Cells}_{trans}(n,m) = (3n + 2m) - (2m + n) = 2n$$

and the quotient

$$\text{Gain}_{rel}(n,m) = \frac{3n+2m}{2m+n} = 1 + \frac{2}{1+2\frac{m}{n}}$$

and again we are interested in between which limits the relative gain is to be found.

7.4.1 m$\longrightarrow \infty$ while n is fixed

In this case we may estimate

$$1 + \frac{2}{1+2\frac{m}{n}} \longrightarrow 1 + \frac{2}{\infty} = 1 + 0 = 1 \text{ for } m \longrightarrow \infty$$

which means: the bigger m, the less is to be gained by the algebraic transformation; this is the same result as in the chapter before.

7.4.2 n$\longrightarrow \infty$ while m is fixed

Now we have

$$1 + \frac{2}{1+2\frac{m}{n}} \longrightarrow 1 + \frac{2}{1+0} = 1 + 2 = 3 \text{ for } n \longrightarrow \infty$$

as a limit for the transformation speedup similar to the speedup result presented in the previous chapter, but greater in the present case.

7.4.3 n,m$\longrightarrow \infty$ while m = n $*$ k is constant

Similar to the example shown in the previous chapter, this is the average case which converges to

$1 + \frac{2}{1+2k}$ for $n, m \longrightarrow \infty$, $\frac{m}{n} = k$ constant.

Let, for example, $k = \frac{1}{2}$, then the transformation speedup will converge to 2 for increasing n, m.

7.5 summary

In this chapter an important example has been examined. For syntactical reasons, the classical deforestation with seven rules fails to be applicable to the non-creative function rev, as Hamilton *et* Jones state in their treatise. Nevertheless, the composed expression *has* been optimised by another transformation introduced in the present chapter. This means that Wadler's deforestation as presented in [19] is not complete. There is at least one case —and even a very simple one, as we have shown— which deforestation with seven rules is too weak to cope with, even though this case is deforestable. Compared to the result in the previous chapter, the deforestation gain of a factor 2 in the average case now seems high. Nevertheless, the new transformation applied in this chapter must not be regarded as the solution to all speedup problems since it has been constructed *ad hoc* to cover only one small class of abstract syntax expressions. However, here arises the question whether it could be possible to *combine* some well known algebraic laws (or other transformations from the source language into the source language) with the rather new deforestation techniques *systematically* in order to achieve better deforestation results, respectively to expand the deforestation domain into program classes which can not be improved by the deforestation techniques found in the literature so far. Future work may attempt to answer this question.

Chapter 8

"a short cut to deforestation", 1993

In the previous chapters, examples of deforestation with *the concatenate vanishes* and Wadler's *deforestation* have been discussed. There is yet another method of deforestation, presented as *foldr/build* by Gill, Launchbury *et* Peyton Jones [4]. This chapter will deal with deforestation by foldr/build. At first, an example showing the possibilities of this new deforestation method will be presented. Analogous to the result of the previous chapter, (where an easily optimisable expression has been discussed that turned out to be undeforestable by Wadler's seven rules), another example, this time not completely deforestable by foldr/build, is given.
For two reasons, no α code analysis will be performed in this chapter. The first example contains —like *quickSort* in chapter 5— data dependencies: thus the α code analysis is not applicable in this case. In the second example we are not interested in general program properties; only the failure of deforestation for one instance shall be shown: in this case, the α code analysis is too inconvenient and not at all necessary. Therefore we base our statements on the results of the automatic trace analysis alone.

8.1 successful application of foldr/build

8.1.1 ordinary list comprehension

Have a look at this FAST Miranda program:

```
f :: num → char;
f x = decode x;

g :: num → num;
g 0 = 0;
g x = 1 + g (x−1);

even :: num → bool;
even x = if x mod 2 = 0;

somefunction :: [num] → [char];
somefunction xs = [f x | x ← (map g xs); even x];

input :: num;
input = 0;
```

```
main :: num → [char];
main dummy = somefunct [65,66,67,68,69,70,71,72,73,74,75,76,77];
```

The purpose of this program is the following: the function f transforms an integer number into a character by calling the built in decode function. The function g is a hard working equivalent to the Identity function on natural numbers. The function even will return True if its input is even and False if its input is odd. The function somefunction takes a list xs of numbers, creates a new list (with the same content) by mapping g to each element of the input list; (the map function is built in). Then the function even is applied to each element x of the new list. If this application returns True, the corresponding x will be a member of a third list —the output— which is built out of all xs after applying the function f to each of them. All in all, somefunction returns a list of characters all having even code numbers. The function main is a closed expression calling somefunct with an argument list from 65 (the ASCII code of 'A') to 77 (the ASCII code of 'M'). The input 0 is an irrelevant dummy, but the FAST compiler demands it.
Nota bene: the subexpression map g xs *produces an intermediate list in the context of the list comprehension expression* [f x | x . . .].
To get an intuition about the runtime behaviour of somefunct let us do some tests with input lists of different lengths and compare the results of the analysis program about them.

a) Input length $l_1 = 13$
(numbers from 65 to 77, coding the characters from 'A' to 'M')

b) Input length $l_2 = 26$
(numbers from 65 to 90, coding the alphabet from 'A' to 'Z')

c) Input length $l_3 = 39$
(numbers from 65 to 90 and from 97 to 109, coding 'A'-'Z','a'-'m')

d) Input length $l_4 = 52$
(numbers from 65 to 90 and from 97 to 122, coding 'A'-'Z','a'-'z')

Here are runtime allocations of somefunction as found by the analysis program:

for **a)**
69 allocations altogether, which consist of:
39 intermediate ($= 3l_1$)
18 update garbage ($= l_1 + k - 1,\ k = 6$)
12 output structure ($= l_1 - 1$)

for **b)**
142 allocations altogether, which consist of:
78 intermediate ($= 3l_2$)
38 update garbage ($= l_2 + 2k$)
26 output structure ($= l_2$)

for **c)**
212 allocations altogether, which consist of:
117 intermediate ($= 3l_3$)
57 update garbage ($= l_3 + 3k$)
38 output structure ($= l_3 - 1$)

for **d)**
285 allocations altogether, which consist of:
156 intermediate ($= 3l_4$)
77 update garbage ($= l_4 + 4k + 1$)
52 output structure ($= l_4$)

These results[1] indicate an output structure consisting of half as many Cons cells as the corresponding input length; together with the Boxes there are as many output cells as the corresponding input length. The variation of -1 respectively $+1$ in the length of the output structure and the number of update garbage cells is a phenomenon of the data dependancy mentioned above, which can be explained by the changing number of even integers in the input list. The function somefunction shows a linear runtime behaviour: The output cell number is about the same as the input length, the intermediate number is three times the input length, and the garbage number increases in linear order with the input. Fig.16 shows a graph of the analysis result.

8.1.2 deforested version and speedup

Whilst the other functions f, g and even remain unchanged, the definition of somefunction is deforested by *foldr/build* to a new definition, named somefunct here. Dissolving the list comprehension according to [15] we would obtain:

```
somefunction [] = [];
somefunction xs = ((f y):somefunction ys)), if even y;
                = somefunction ys, otherwise
                  where (y:ys) = map g xs;;
```

Now it is intuitive that the intermediate structure map g xs can be removed by defining:

```
somefunct :: [num] → [char];
somefunct [] = [];
somefunct (x:xs) = ((f y):somefunct xs), if even y;
                 = somefunct xs, otherwise
                   where y = g x;;
```

This redefinition leads to the following runtime results:

for **a)**
18 allocations altogether, which consist of:
0 intermediate
6 update garbage ($= \frac{1}{3}all = \frac{1}{2}(l_1 - 1)$)
12 output structure ($= 2 * garb = in_1 - 1$)

for **b)**
39 allocations altogether, which consist of:
0 intermediate
13 update garbage ($= \frac{1}{3}all = \frac{1}{2}l_2$)
26 output structure ($= 2 * garb = in_2$)

[1] See appendix for how the analysis results are derived, and according to which assumptions the cells can be divided into the different classes 'output', 'intermediate' and 'update garbage'.

for c)
57 allocations altogether, which consist of:
0 intermediate
19 update garbage ($= \frac{1}{3}all = \frac{1}{2}(l_3 - 1)$)
38 output structure ($= 2 * garb = in_3 - 1$)

for d)
78 allocations altogether, which consist of:
0 intermediate
26 update garbage ($= \frac{1}{3}all = \frac{1}{2}l_4$)
52 output structure ($= 2 * garb = in_4$)

The deforestation has been successful: there are no intermediate structures left. The new somefunct definition is also linear in its runtime behaviour, but this time the graph line (Fig.16) is less steep than in the ordinary case. Therefore, the relative deforestation gain must be a limited speedup factor of order $\mathcal{O}(1)$, as we found in the earlier chapters. This factor is easy to estimate simply by dividing two of the corresponding total allocation numbers of the ordinary and the deforested version of somefunct:

for **a)** $\frac{69}{18} = 3.83$ for **b)** $\frac{142}{39} = 3.64$ for **c)** $\frac{212}{57} = 3.72$ for **d)** $\frac{285}{78} = 3.65$

So, roughly speaking, we gain a speedup factor of between 3 and 4 independent of the length of the input list. This is a considerably better result than in the earlier chapters, where a speedup of factor 2 or 3 was possible only for inputs of infinite length. The speedup factors estimated above depend on the FAST implementation and could be different if no Box cells were used.

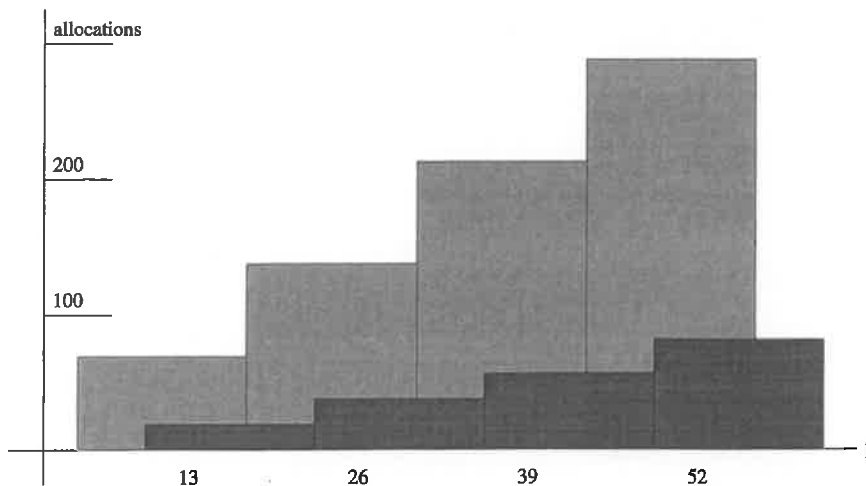


Fig.16 *runtime behaviour graphs of both versions of* somefunct *as found by the analysis program*

The reason for the improved speedup here, compared with the earlier examples, is not the different deforestation method, but the different size of the removed intermediate structure, compared to the size of the output structure.

8.2 foldr/build failing in a permutation

8.2.1 developing a tricky function

In this section we want to to show that also the foldr/build rule is, like Wadler's deforestation, not applicable in all cases. To get an idea how such an undeforestable —or at least: not completely deforestable— function composition can be constructed, let us repeat the foldr/build deforestation for list comprehension in general. The Miranda expression:

```
h :: [*] -> [* * *];
h xss = [f xs | xs <- (map g xss); pred xs];
```

transforms via foldr/build to

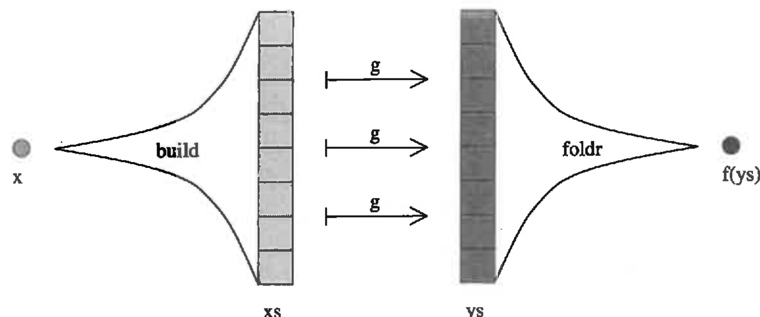
```
h :: [*] -> [* * *];
h [] = [];
h (xs:xss) = ((f ys):h xss), if pred ys;
           = h xss, otherwise
             where ys = g xs;;
```

and the inner functions f, g and pred are typed:

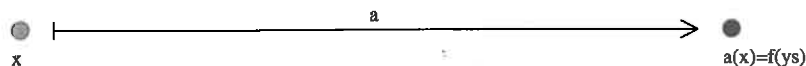
```
g :: * -> **;
f :: ** -> * * *;
pred :: ** -> bool;
```

Observing the once deforested expression above, we detect again some functions in composite situations! These are: f ys and pred ys, where ys = g xs. The questions are now: *can these composites always be further deforested with foldr/build? Are there cases where the deforested version of the* h *function as given above is the last link in a deforestation chain, though it may well be possible to improve this expression further in another way?* The answer to the first question is *no*, the answer to the second one is *yes*. In both function compositions of the deforested h expression g is the intermediate function — the consequent question is: *How must* g *be defined to prevent a further foldr/build deforestation?* Before we develop a sufficient definition of a cheeky_g function we should at first give an intuition of the situation.

This picture shows the situation *after the first* deforestation of the list comprehension:



The g function transforms a list xs into an intermediate list ys = g(xs). The function f *folds* the intermediate list into a single core value f(ys). To make the situation complete, we must assume that the list xs can be *built* out of another single core value x, for example by the expression xs = [1..x]. If now the creation of *both* list xs and ys is to be avoided we must find a function a so that a(x) = f(ys), thus:



And here comes the trick: *What about a function* g *containing an implicit 'meta knowledge' about the sequence of elements in the lists xs and ys? How should such 'meta knowledge' be extracted out of* g *and pushed into* a *by a simple syntactic transformation?* This seems really impossible, and in fact all examples for successful foldr/build applications given in literature deal with folding functions like sum and square which are *commutative*, i.e. the order of the list elements as arguments of those functions is irrelevant!
Thus if we want to make a further deforestation of the expression f (g xs) impossible, and if we assume that xs can be *built* and f can be expressed in terms of *foldr*, then the only possibility we have is to break the foldr/build chain between xs and f by providing a function g, which is not rewriteable in terms of the functions *foldr* and *build*[2]. Now we are able to define an example function cheeky_g, together with a *non* commutative folding function f, so that a further deforestation of the expression given above with foldr/build is impossible:

$$f\ [x_1, x_2, x_3, x_4] = 2^{x_1} * 3^{x_2} * 5^{x_3} * 7^{x_4}$$

This is a Gödel bijection based on prime numbers for coding sequences of natural numbers; it could easily be expressed in terms of foldr. The function

cheeky_g $[x_1, x_2, x_3, x_4]$ = shiftleft(shiftleft(shiftleft $[x_1, x_2, x_3, x_4]$))

permutes its argument $[x_1, x_2, x_3, x_4]$ three times.
The left shifting permutation is defined in FAST Miranda as:

[2]Such a function already has been mentioned in the previous chapter: the *reverse* function, which has been indeforestable for syntactic reansons, is a permutation. This property makes that function relevant for the present chapter as well.


```
shiftleft :: [*] → [*];
shiftleft [] = [];
shiftleft [x] = [x];
shiftleft (x:xs) = ((last xs):init (x:xs));

last :: [*] → *;
last [x] = x;
last (x:xs) = last xs;

init :: [*] → [*];
init [x] = [];
init (x:xs) = (x:init xs);
```

Without any possibility of making the implicit knowledge of cheeky_g explicit, namely:

cheeky_g $[x_1, x_2, x_3, x_4] = [x_2, x_3, x_4, x_1]$

we can be quite sure that the *foldr/build* deforestation as presented in literature can not generate a function a from a single value to another single value transporting the permutation 'knowledge' of the intermediate list. But how are *we* able to further improve the deforested h expression as given above? Not by a syntactic transformation, but *ad hoc* with our knowledge about permutations: on a list of length l = 4, one right shifting permutation is semantically equivalent to three left shifting permutations. Thus if we replace cheeky_g by the following function:

```
shiftright :: [*] → [*];
shiftright [] = [];
shiftright [x] = [x];
shiftright (x:xs) = append xs [x];
```

we are done because we have avoided two intermediate lists caused by the shiftleft function calls in cheeky_g.

8.2.2 results of the automatic trace analysis

Again we would like to construct cell allocation graphs for the different versions of the h expression. Therefore we have to provide inputs of different lengths and then use every input as an argument for each version of the expression h. We choose:

a) xss = [[4,3,2,1]] (length $l_1 = 1$)
b) xss = [[4,3,2,1], [4,3,2,1]] (length $l_2 = 2$)
c) xss = [[4,3,2,1], [4,3,2,1], [4,3,2,1]] (lenght $l_3 = 3$)

Yet some predicate function pred is needed, and we define for simplicity without loss of generality:

```
pred xs = True;
```

Now we let the analysis program work on the expression

```
h xss = [f xs | xs ← (map cheeky_g xss); pred xs];
```

and obtain the following results[3]:

for **a)**
47 allocations altogether, of which:
33 intermediate (= 11*output)
3 output structure (= $3l_1$)

for **b)**
95 allocations altogether, of which:
66 intermediate (= 11*output)
6 output structure (= $3l_2$)

for **c)**
143 allocations altogether, of which:
99 intermediate (= 11*output)
9 output structure (= $3l_3$)

Now we examine the deforested expression and, as we choose pred to always be true, we can now do without the if and otherwise statements:

```
h [] = [];
h (xs:xss) = ((f (cheeky_g xs)):h xss);
```

We obtain the following results from the analysis program:

for **a)**
45 allocations altogether, of which:
31 intermediate (= 10*output + l_1)
3 output structure (= $3l_1$)

for **b)**
90 allocations altogether, of which:
62 intermediate (= 10*output + l_2)
6 output structure (= $3l_2$)

for **c)**
135 allocations altogether, of which:
93 intermediate (= 10*output + l_3)
9 output structure (= $3l_3$)

In fact, some allocation decrease caused by the deforestation of the map expression in the list comprehension is recognisable, but only a small one. Let us finally replace cheeky_g by shiftright and examine

```
h [] = [];
h (xs:xss) = ((f (shiftright xs)):h xss);
```

Then we obtain the following, obviously better, results:

[3]for the demands of the FAST compiler the definition of the function f has been changed to f [x1,x2,x3,x4] = decode (($2^{x1} * 3^{x2} * 5^{x3} * 7^{x4}$)-100);

for **a)**
16 allocations altogether, of which:
8 intermediate (= 3*output − l_1)
3 output structure (= $3l_1$)

for **b)**
32 allocations altogether, of which:
16 intermediate (= 3*output − l_2)
6 output structure (= $3l_2$)

for **c)**
48 allocations altogether, of which:
24 intermediate (= 3*output − l_3)
9 output structure (= $3l_3$)

The following table shows the deforestation gains of foldr/build, respectively foldr/build plus replacement of cheeky_g by shiftright, applied to the former list comprehension.

gain	*foldr/build*	*plus replacement*
factor	1.06	2.98

8.3 summary

In this chapter, it has been shown how deforestation can be done with the transformation rule foldr/build. This method works well in cases of commutative foldings, where the element sequence of intermediate lists is irrelevant. The method does not work completely where the element sequence of the intermediate lists is some kind of meta information with influence on the folding function. In this case the generation of intermediate lists cannot be avoided without loosing this information for the folding function. Nevertheless, it is possible for us in some cases to improve such complicated expressions *ad hoc*. Replacing the function cheeky_g by shiftright has been such an *ad hoc* method in the example shown above. Thus as a result, we may state that great runtime improvements caused by foldr/build deforestation must not be expected in every function composition situation.
In their treatise "a short cut to deforestation" the authors themselves give important comments on the applicability of their new method. Functions of a certain type are definitively known to be resistant against foldr/build, but nothing is said about functions not of this type: their deforestability depends on whether they are expressable in terms of foldr/build or not. Yet we do not know whether this property is decideable and —if it were decideable— how to do a pre-transformation to foldr/build-form automatically to make this kind of deforestation possible. The example of cheeky_g presented in this chapter has shown one case, where a function probably can not be expressed into terms of foldr/build. As such difficult functions we consider in general: permutations, which change the sequence of the input lists' elements, filters, which decrease the input list length, and multiplexers, which increase the input list length.

Chapter 9

benchmark examples

In the previous chapters some representatives of those more or less successful deforestation examples found in the literature have been studied. The question is now whether deforestation can be successfully applied not only to those well-selected reasonable examples, but also —after all the purpose of the deforestation research— to more realistic functional programs.

9.1 empirical deforestation limits

Comparing the deforestation results of chapters 7 and 8 we have found that the actual deforestation gain depends on the size of the removed intermediate structure, compared to the corresponding output structure. Thus, where the intermediate structures are big compared to the output, the relative deforestation gain is bigger than for small intermediate structures — assuming, of course, the intermediate structures could be deforested at all. Size and number of intermediate structures produced by a functional program depend strongly on the 'density' of functionally composite situations occurring in such a program. The more a program is functionally composite, the more intermediate cells it will produce compared to its output cells, and the bigger could be the deforestation gain if there were a transformation to remove all those intermediate structures. Yet we have not read about deforestation gains of factors of 10 or 100, but found the following results in [6] instead: of 25 benchmark programs analyzed there, for only three the allocation gain was bigger than factor 2 with foldr/build; fifteen programs could not be deforested with an allocation gain of more than factor 1.1, and for seven programs the gain was between factor 1.1 and factor 2. However, the programs with a gain of more than factor 1.1 are only toys like the eight queens puzzle. These results are important for comparison in the following.

9.2 a new challenge for the analysis program

The analysis program has been only used for verification so far, which means either comparison with the results of the α code analysis, or comparison of the allocation behaviours of ordinary and deforested versions of a program without having performed α code analysis before, as shown in the previous chapters. But what about the case in which we can neither analyze a complicated program by hand, nor are supplied with an already deforested version of it for comparison? In this case we would like the analysis program to *predict* —or at least to guess— the maximal possible gain of applying a deforesting transformation, assuming such a transformation exists. For analysis applications, as done in the previous chapters, it was sufficient only to distinguish the cell classes 'intermediate', 'output', and 'update garbage' to find out how many cells have been allocated in total, and how many intermediate cells actually have been removed by deforestation. But for big

programs this distinction is too rough and simple. As explained above, big composed programs produce a lot more intermediate cells than output cells. Simply counting both and returning their numerical relation would mean predicting a huge deforestation gain, obviously in contradiction to the small or very small gains observed, as reported in the literature. Thus, the analysis program now has to reason further about those cells which it found to be 'intermediate'. This class shall further be divided into the subclasses *'deforestable'* and *'essential'*, in order to produce acceptable gain predictions. As byproduct of this further reasoning we found another cell class, of which the members first had been regarded as 'intermediate' in the former rough analysis, but then turned out to be *'not really intermediate'* at all. In the following we have to explain the new concepts 'essential' and 'not really intermediate' and have to tell something about the method of deriving these cell classes from the rough 'intermediate' class.

9.3 not really intermediate, essential or deforestable

In functional programs, some data structures are used as input arguments for more than one function, for example symbol tables in compiler design which are referred to many times. We assume that those structures often referred to can not be deforested. An example will illustrate this difficulty.

9.3.1 example

The following Miranda definitions simulate a small histogram table which is built out of an input list, and two functions called **most** and **fourtimes** which are able to select entries out of the table:

```
entry ::= Pair char num;

histogram :: [char] → [entry] → [entry];
histogram [] ents = ents;
histogram (x:xs) ents = histogram xs (make x ents);

make :: char → [entry] → [entry];
make x [] = [Pair x 1];
make x (e:ents) = ((Pair y (n+1)):ents), if x = y;
                = (e:make x ents), otherwise
                  where (Pair y n) = e;;

most :: [entry] → entry → char;
most [Pair x n] (Pair y m) = x, if n>m;
                           = y, otherwise;
most (e:ents) en = most ents e, if n>m;
                 = most ents en, otherwise
                   where (Pair x n) = e; (Pair y m) = en;;

fourtimes :: [entry] → [char];
fourtimes [] = [];
fourtimes ((Pair x n):ents) = (x:fourtimes ents), if n=4;
                            = fourtimes ents, otherwise;
```

The function **histogram** takes a list of characters and an accumulator list for entries as input and makes a new table (i.e. a list) of entries out of it. An **entry** is a pair consisting of a character (of the input list) and a number (according to how often this character occurs in the input list). The function **most** takes a histogram table and a comparison entry and returns the character with the highest number of occurrences, the function **fourtimes** returns a list of characters out of the table which all occur exactly four times. If we now compute the expression

```
main dummy = app [most (h) (Pair 'X' 0)] (fourtimes (h))
             where h = (histogram ['A','B','A','C','A','B'] []);;
```

we see that the intermediate table structure

```
[(Pair 'A' 3), (Pair 'B' 2), (Pair 'C' 1)]
```

produced by the expression h is *shared* between the functions **most** and **fourtimes**. (The main function's output is ['A']). Thus, if we wanted to deforest the intermediate table structure, we would have to do the deforestation transformation *twice*: first for the composition most(h) and second for the composition fourtimes(h). If there were ten functions operating on the histogram table, we would have to deforest ten times, etc. Therefore we claim the following hypothesis.
Hypothesis: *We call a multiply used intermediate structure 'essential' and assume it is undeforestable until the opposite is proved.*

9.3.2 separation method

When the analysis program has found all intermediate cells, it groups them into different cell structures according to the cell connection information given in the output trace. With trace information about the use of particular cells as function arguments, the intermediate cell structures in the heap are divided according to the following definitions.
Definition: *A cell is member of a structure, if either the cell is the structure root or the cell is contained by a Cons, Box or Pack cell which is itself a member of this structure.*
Definition: *An intermediate cell structure is* E*-essential, if at least one of its member cells has been used as a function argument at least* E *times. Thereby* E *is a natural number greater than 0, chosen according to our intuition about the power of deforestation.*
Definition: *A cell structure that is not output is not really intermediate if none of its member cells has ever been used as a function argument.*
Nota bene: The second definition can intuitively be derived from the 'definitions' of intermediate as given in the introductory chapter of this thesis. Not really intermediate 'structures' mostly turn out to contain not more than one single cell.
Definition: *Intermediate structures without the properties defined above will be regarded as deforestable in principle.*
Nota bene: deforestable in principle does not mean we actually posses a method for deforesting those cells.

9.3.3 continuation of the example

Two automatic analyses of the example given above shall show the suitability of our separation method. The **histogram** structure is used twice, thus we must expect that we are able to find this

structure by comparing two automatic analyses with E=1 and E=2, if we know that all the other intermediate structures occuring in the program are used only once. (These other structures are caused by app and the creation of the histogram itself). And indeed, we can find in the outputs of the analysis program the following statements for E=2:

```
70 Nodes:  17 Box, 11 Cons, 7 Pack, 35 Vap
2 cells belong to the output structure
58 cells are 'intermediate' (rough counting)
6 structures are 'really intermediate' and contain
34 deforestable intermediate cells
```

and for E=1:

```
5 structures are 'really intermediate' and contain
21 deforestable intermediate cells
```

with the other statements being the same as in case E=2. The 2 output cells (Cons and Box) are used for the most frequent value 'A'. The difference of 34 – 21 = 13 cells is exactly the size of the histogram structure, which contains 3 Cons cells to build the list, 3 Pack cells for the pairs, 6 Boxes and 1 not updated Vap for the operation +, as shown in Fig.17. It is thus possible to detect the essential intermediate histogram structure with help of the analysis program. We regard this result as a good argument for the suitability of the E–assumption concept as applied in the following advanced automatic benchmark analyses.

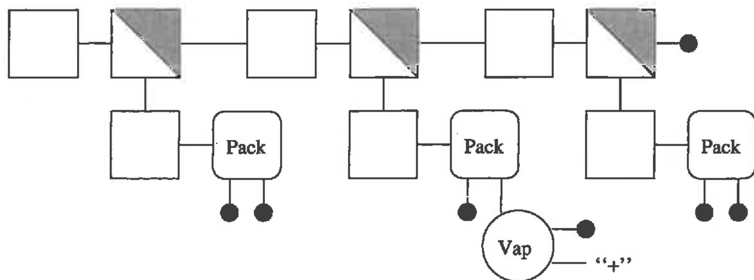


Fig.17 *the essential intermediate histogram structure consisting of 13 heap cells*

9.4 four benchmark analyses

In this section, four FAST benchmark programs called event [14], listcompr [15], sched [17] and wang [20] are to be analyzed. These programs are too big to be analyzed by hand: for the numerical input 2, their traces differ from 568 to 12889 lines; listcompr claims 5340 heap nodes for input 4 and 21078 nodes for input 8. None of the four programs exist in a deforested version, so that the analysis program is applied as forecaster of a possible deforestation gain. Analyses with different assumption numbers E will be compared to each other. (The programs are data dependent and therefore run with the small input 2 in order to avoid unnecessarily long traces).

9.4.1 sched

This is the output[1] of the analysis program for sched with E=2:

```
89 Nodes:  26 Box, 22 Cons, 6 Pack, 35 Vap
29 VapUpd:  15 from inside, 14 from outside
6 Vap remain without being updated
15 cells became garbage by updating inside Vap
6 cells became garbage by updating outside Vap or Caf
10 cells belong to the output structure
58 cells are intermediate and belong to
25 different intermediate structures
9 structures are 'really intermediate' and contain
33 deforestable intermediate cells
65.17 per cent of all cells are intermediate (rough counting)
37.08 per cent of all cells are deforestable with E=2.
```

'Rough counting' means regarding all cell which are neither output nor update garbage as deforestable intermediate, without separating them further as described above. We see, that the intermediate part decreases from about 65 per cent in rough counting (estimated speedup factor 2.85) to about 38 per cent (estimated speedup factor 1.60), if the not really intermediate structures are ignored the same way as the essential structures with an assumption number E=2. If we regard this number as too small, we analyze for E=3:

```
10 structures are 'really intermediate' and contain
34 deforestable intermediate cells
38.20 per cent of all cells are deforestable with E=3.
```

with the other analysis statements being the same as in the case above. For E=4 the result is:

```
10 structures are 'really intermediate' and contain
34 deforestable intermediate cells
38.20 per cent of all cells are deforestable with E=4.
```

Increasing E from 2 to 3 we regard now 38.2 per cent of all cells as deforestable intermediate; these are less than 2.0 per cent more than with E=2. There is no difference between the analyses for E=3 and E=4, which means that no cell is used as a function argument more than three times. The difference between E=2 and E=1 is not only quantitative, but also qualitative: an assumption of E=1 means regarding *any* shared structure as undeforestable. For E=1 we obtain:

```
6 structures are 'really intermediate' and contain
19 deforestable intermediate cells
21.35 per cent of all cells are deforestable with E=1.
```

In the following sections, the analysis results for the other benchmark programs with input=2 are presented in an analogous manner.

[1] It is given completely this time in order once to show the ability of the analysis program.

9.4.2 event

These are the automatic analysis results[2] for event with E=2, 3, 4:

```
802 Nodes:  218 Box, 212 Cons, 30 Pack, 342 Vap
589 cells are intermediate and belong to
132 different intermediate structures
40 structures are 'really intermediate' and contain
360 deforestable intermediate cells
73.44 per cent of all cells are intermediate (rough counting)
44.89 per cent of all cells are deforestable with E=2.

41 structures are 'really intermediate' and contain
364 deforestable intermediate cells
45.39 per cent of all cells are deforestable with E=3.

41 structures are 'really intermediate' and contain
364 deforestable intermediate cells
45.39 per cent of all cells are deforestable with E=4.
```

There is only a minimal difference in the cells regarded as deforestable intermediate under assumptions E=2 and E=3. No difference is found between the analyses under assumptions E=3 and E=4, which means that no intermediate cell is used as function argument more than three times. A rough counting returns about 70 per cent 'intermediate' cells in all cases. This would mean an estimated possible gain of factor about 3.3 which is probably too big, compared to our speedup reasoning in the sections and chapters before. But taking the essential structures and the not really intermediate structures out of our consideration, we find only about 45 per cent deforestable intermediate cells in all cases. The corresponding estimated speedup is now about factor 1.8 which may be regarded as a sensible forecast compared to the results presented in the chapters before. For the qualitatively different assumption E=1 we obtain finally:

```
30 structures are 'really intermediate' and contain
295 deforestable intermediate cells
36.78 per cent of all cells are deforestable with E=1.
```

9.4.3 listcompr

Analogue to the programs in the sections above, the analysis program finds now for listcompr:

```
5340 Nodes:  1453 Box, 1177 Cons, 281 Pack, 2429 Vap
4049 cells are intermediate and belong to
1002 different intermediate structures
152 structures are 'really intermediate' and contain
1108 deforestable intermediate cells
75.82 per cent of all cells are intermediate (rough counting)
20.75 per cent of all cells are deforestable with E=2.
```

[2]From now on, the less relevant statements will be omitted for clarity.


```
153 structures are 'really intermediate' and contain
1954 deforestable intermediate cells
36.59 per cent of all cells are deforestable with E=3.

155 structures are 'really intermediate' and contain
1966 deforestable intermediate cells
36.82 per cent of all cells are deforestable with E=4.
```

In this case the difference of the analyses between E=2 and E=3 is remarkable. Only one more structure is regarded as deforestable with E=3, but this structure contains $1886 - 1190 = 696$ cells! Nevertheless, even with assumption E=4 the predicted deforestation gain corresponding to the about 36 per cent deforestable intermediate cells is not bigger than factor 1.6 where the gain according to the rough counting would have been estimated as factor 4. If we assume that *any* shared intermediate structure is undeforestable then we obtain for E=1:

```
148 structures are 'really intermediate' and contain
1084 deforestable intermediate cells
20.30 per cent of all cells are deforestable with E=1.
```

9.4.4 wang

Finally, the wang program is analyzed with different assumptions as:

```
787 Nodes:  185 Box, 167 Cons, 23 Pack, 412 Vap
583 cells are intermediate and belong to
202 different intermediate structures.
27 structures are 'really intermediate' and contain
290 deforestable intermediate cells
74.08 per cent of all cells are intermediate (rough counting)
36.85 per cent of all cells are deforestable with E=2.

29 structures are 'really intermediate' and contain
302 deforestable intermediate cells
38.37 per cent of all cells are deforestable with E=3.

29 structures are 'really intermediate' and contain
302 deforestable intermediate cells
38.37 per cent of all cells are deforestable with E=4.

25 structures are 'really intermediate' and contain
250 deforestable intermediate cells
31.77 per cent of all cells are deforestable with E=1.
```

Again we find no cell being used as a function argument more than three times, and the estimated deforestation gain is for all E=1, 2, 3, 4 about factor 1.5. The following table presents all four

analysis results for comparison.

program	*predicted speedup factors with*				
	rough counting	E = 4	E = 3	E = 2	E = 1
sched	2.85	1.60	1.60	1.60	1.25
event	3.75	1.85	1.85	1.80	1.60
listcompr	4.15	1.55	1.55	1.25	1.25
wang	3.85	1.60	1.60	1.60	1.45

The big differences between the results for rough counting and E=4, and the small differences between the qualitatively different assumptions E=1 and E=2 (sharing 'forbidden' vs. 'allowed') show that the biggest waste of heap space is caused by the not really intermediate cells. Therefore, it seems not worth attempting the deforestation of multiply used intermediate structures, and in fact, none of the known works on deforestation treats or mentions examples of such multiply used structures.

9.5 result interpretation: mind the method!

In the introductory chapter we promised to *approximate* the possibilities of deforestation. The analysis program is able to find all cells allocated in the heap and their connections to each other. Thus: where is the approximation? In fact, there are two. One approximation is our guess E about the power of deforestation as mentioned above. The other one is founded, first in the impossibility of separating 'garbage' from 'intermediate cells' as mentioned in the introductory chapter, and second especially in the *lack of definition* shown in the vague statements quoted at the beginning of this thesis: we simply do not know exactly what an intermediate structure is. Speaking about 'intermediate', the quoted authors are mainly thinking about structures passed between *different* functions. But what about functions which create 'their *own*' intermediate structures by recursion? This situation has not been mentioned in the deforestation literature so far. Looking back at the histogram example, we find lots of cells being allocated transiently in order to create the histogram structure. Of course they are garbage afterwards[3], but are they intermediate? Our analysis program simply says: 'yes' — which is the second approximation! Should not at least those cells, which are transiently allocated by a function (and passed as arguments to the same function by recursion) in order to build up an undeforestable essential intermediate structure, also be regarded as undeforestable? At this point our program is too weak to perform further case analysis, but that does not matter: using the analysis program, we only have to keep in mind that the number of cells interpreted as 'deforestable intermediate' is —even after the refinement of the former rough counting— probably still somewhat overestimated, and as the benchmark analyses show, *even with* that overestimation the predicted possible deforestation gains seem to be no reason for expecting future runtime miracles.

[3]For deforestation analysis as performed in this thesis the garbage collector of the FAST compiler has to be switched off, because the structures could not be identified otherwise.

9.6 summary

In this chapter we have given a reason against an oversimplified trace analysis based only on counting the 'intermediate' cells. Such a counting would result in a bad estimation of a possible deforestation gain. Examples of observed deforestation gains have been presented for comparison at the beginning of this chapter. A better forecast results from a further separation of the 'intermediate' cells in the classes 'essential', 'not really intermediate' and 'deforestable in principle'. The size of the essential class of one analysis depends on our assumption about the possibility to deforest shared[4] structures. Sharing functional expressions wherever it is possible is a widely used programming technique in order to save superfluous runtime allocations. Thus, sharing may be regareded as 'deforestation a priori', however its disadvantage is, that it makes the 'genuine deforestation' a posteriori difficult.
In every benchmark analysis performed in this chapter, different assumption values E lead to similar deforestation gain predictions, and all these predictions are similar in size to the gain limits found for the small examples in the previous chapters. Both these similarities together seem to be an argument for the suitability of the approximative automatic trace analysis developed in this thesis. The estimated gain factors have to be regarded as maximum values, which could be reached if proper deforestation methods were supplied to remove all cells 'deforestable in principle'. If the deforestation techniques used are too weak, even the small predicted possible gain will not be reached.

[4]The concept 'shared' is used as 'being multiply used as a function argument' in this thesis. This is slightly different from the usual concept of sharing as 'being multiply referred to'.

Chapter 10

conclusions

In this thesis, an analysis program has been developed as a tool for efficiency analysis of functional programs. It is built in correspondence with the FAST compiler system, but is —because of modularity— adaptable in principle for any G-machine based and runtime tracing functional compiler as well. This analysis program is able to count the cells which the analyzed functional program has allocated in the heap. Therefore, the analysis program is suitable for comparing the behaviour of syntactically different but semantically equivalent versions of a functional program. Moreover, the analysis program is able to predict the possible gain which could be achieved by transforming the analyzed functional program by deforestation. Talking about 'deforestation', we mean all changes done to the definitions of a functional program in order to reduce intermediate structures which this functional program produces. Intermediate structures arise when functions, working on data structures, are composed. The basis of the automatic deforestation analysis is an output trace given by the machine which computes the functional program that is to be analyzed. With the information found in such a trace, the analysis program can replay the heap events in simulation and separate intermediate from output or garbage cells. Further reasoning on certain properties of those cells found to be intermediate makes it possible to predict the possible deforestation gain properly.
In the first part of this thesis, the genesis of output traces from functional programs is explained in an abstract manner. The functional language Miranda, which is used for the experiments in this thesis, is represented by an abstract syntax with strictness annotations. These strictness annotations make it possible to explain the situation with reference to relevant properties of the FAST compiler, which is used as the Miranda compiler in this thesis. The abstraction of this compiler is a translation scheme $\mathcal{T}$ which produces α code as abstraction of an imperative machine language. An abstract machine $\mathcal{A}$ takes α code as input, updates its heap according to the semantics of the corresponding abstract syntax program, and returns an abstract trace reporting that heap history as output. The abstraction is not only a didactic support for a better understanding of the analysis principles, but also an introduction to the approximating automatic analysis, which does not need to use all the information supplied by a FAST output trace.
In the second part, examples of different deforestation techniques presented in the literature have been analyzed. None of these techniques can cope with all situations of intermediate structure producing function composition, and in all examples analyzed —except one— a speedup limit has been found which can not be overcome in principle by any inventable deforestation technique. Finally, analyses of some medium sized benchmark examples have been presented. A comparison of the analysis results presented here with other benchmark tests in the literature shows that the analysis method developed here is sufficient for its purpose.
The deforestation techniques found in the literature seem to be too weak to improve realistic functional programs, which are considerably bigger than the specially chosen deforestation examples

in the literature. One reason for this weakness seems to be the deforestation techniques, which are bound by many restrictions: creativeness of the function definitions, appropriate function types and expressablility in terms of foldr/build are enumerated in the deforestation literature. The second reason can be found in the functional programs themselves: there seem to be essential intermediate structures which can not be deforested by any technique in practice because they are used as function arguments more than once. Further, the number of cells belonging to, so to say, 'crystalline' intermediate structures is not large, compared to the number of isolated cells residing in the heap as unstructured masses beyond the domain of the deforestation transformations. Perhaps further research on deforestation can develop techniques to remove essential intermediate structures, which still have to be regarded as undeforestable at this time, but our analysis results show that such attempts would not be very effective. We are still missing a deforestation theory which could order all these *ad hoc* techniques into a more general system, including even some suitable algebraic transformation laws which are already well known, although not mentioned in the deforestation literature so far.
Deforestation is not the only program transformation technique for saving superfluous heap allocations [8]. For example, 'thunk lifting' [11] is a transformation for avoiding suspension nodes, and also in this case, the transformation gain seems to be restricted. It may be that a certain efficiency price always has to be paid for the conceptual clarity of functional programs, compared to their spaghetti code relatives. Perhaps better runtime behaviour of functional programs could be achieved by combining deforestation with several of the many known other optimization techniques, however it could then happen that different techniques interfere adversely with each other. Nevertheless, our analysis program need not be useless: even if no deforestation or other transformation technique were available as an algorithm, an analysis result with more than a certain percentage of intermediate cells could inspire a programmer to think about her solution again and to rebuild an even better one, 'deforested' by the creativity of the human brain.

Appendix A

bibliography

A.1 literature on deforestation

[18] Philip Wadler. *the concatenate vanishes*
internal report (revised 1989), department of computing science, University of Glasgow, 1987

[3] A.B. Ferguson, Philip Wadler. *when will deforestation stop?*
proc. Glasgow workshop on functional programming, Rothesay(Sco) Aug.1988,
in: C.Hall, R.J.M.Hughes, J.T.O'Donnell (eds). Functional Programming, pp39-56,
research report 89/R4, department of computing science, University of Glasgow, 1989

[19] Philip Wadler. *deforestation: transforming programs to eliminate trees*
Theoretical Computer Science vol.73 nr.2, pp 231-248,
North Holland, Amsterdam, 1990 (earlier version 1988)

[7] G.W. Hamilton, S.B. Jones. *extending deforestation for first order functional programs*
Glasgow workshops in computing science, Portree(Sco) Aug.1991,
in: R.Heldal, C.Kehler-Holst, P.Wadler (eds). Functional Programming, pp134-145,
Springer Verlag, Berlin, 1991

[13] Simon Marlow, Philip Wadler. *deforestation for higher order functions*
Glasgow workshops in computing science, Ayr(Sco) Jul.1992,
in: J.Launchbury, P.Sansom (eds). Functional Programming, pp154-165,
Springer Verlag, Berlin, 1992

[4] Andrew Gill, John Launchbury, Simon L. Peyton Jones. *a short cut to deforestation*
Kopenhagen Jun.1993,
in: Arvind (ed). 6th Functional Programming Languages and Computer Architecture, pp223-232,
Lecture Notes in Computer Science,
ACM, New York, 1993

[5] Andrew Gill, Simon L. Peyton Jones. *building on foldr*
workshop, Ayr(Sco) Jul.1993
in: K.Hammond, J.T.O'Donnell (eds). Functional Programming, ppXIV.1-XIV.11,
report, department of computing science, University of Glasgow, 1993

[6] Andrew Gill, S.L. Peyton Jones. *cheap deforestation in practise: an optimiser for haskell*
workshop, Hamburg Aug.1994, in: proc. IFIP vol.1, pp581-586

A.2 literature on the FAST system

[8] Pieter H. Hartel, Hugh W. Glaser, J.M. Wild. *on the benefits of different analyses in the compilation of functional languages.* in: H.W. Glaser, P.H. Hartel (eds). 3rd implementation of functional languages on parallel architectures, pp123-145, report CSTR 91-07, department of electronics and computer science, University of Southampton, 1991

[9] Pieter H. Hartel, Hugh W. Glaser, J.M. Wild. *the FAST compiler user's guide* in: FAST: functional programming for arrays of transputers. The collected papers, pp247-264, report CSTR 93-15, department of electronics and computer science, University of Southampton report DOC 93/47, department of computing, Imperial College of science, technology and medicine, University of London, 1993

[10] Pieter H. Hartel, Hugh W. Glaser, J.M. Wild. *compilation of functional languages using flowgraph analysis* Software – Practice and Experience 24(2), pp127-173, 1994

A.3 background literature and benchmark references

[2] R. Bird, Philip Wadler. *introduction to functional programming,* Prentice Hall, New York, 1988

[15] Simon L. Peyton Jones. *the implementation of functional programming languages* Prentice Hall, Englewood Cliffs(NJ), 1987

[11] A. Reza Haydarlou, Pieter H. Hartel. *thunk lifting: reducing heap usage in an implementation of a lazy functional language.* technical report CS-93-07, vakgroep computersystemen, Faculteit der Wiskunde en Informatica, Universiteit van Amsterdam, 1993

[16] D.A. Turner. *miranda: a non-strict functional language with polymorphic types* in: proceedings of the IFIP international conference on functional programming languages and computer architecture, Nancy 1985. Lecture Notes in Computer Science 201, Springer Verlag, Berlin

[1] A.V. Aho, R. Sethi, J.D. Ullman. *compilers: principles, techniques and tools* Addison Wesley, Reading(Mass.), 1986

[12] Brian W. Kernighan, Dennis M. Ritchie. *the C programming language,* 2nd ed. Prentice Hall, Englewood Cliffs(NJ), 1988

[14] Henk L. Muller. *simulating computer architectures* (Dissertation) Fakulteit der Wiskunde en Informatica, Universiteit van Amsterdam, 1993

[17] Wim G. Vree. *design considerations for a parallel reduction machine* (Dissertation) Fakulteit der Wiskunde en Informatica, Universiteit van Amsterdam, 1989

[20] H. H. Wang. *a parallel method for tridiagonal equations* ACM transactions on mathematical software 7 (2), pp170-183, Jun.1981

Appendix B

trace information

B.1 extraction method

All statements printed in a FAST output trace are instances of patterns which can be described as regular expressions. Therefore, it is easy to identify those trace statements which are relevant for the analysis. The variety of trace statements is large; interested readers should consult the FAST literature. In the following, only a few examples of identifying trace statements with regular patterns are given. The script with all patterns necessary and sufficient for the examples analyzed in this thesis is given in the next section; finding *Pack* cells and the usage of arguments in functions calls is especially important for advanced analysis as done in chapter 9.

B.1.1 how to find the first heap address

The beginning of the heap is reported in every trace given by the FAST compiler. Looking at a trace, we may find for example a statement:

```
heap                    100000 ints from 3f000 to a0a7c (start=3e240)
```

`3f000` and `a0a7c` are addresses in hexadecimal notation. This means there is an area, `100000` times the size of the type `integer`, located between the addresses `3f000` and `a0a7c`. We are only interested in the starting address `3f000`, but nevertheless have to look out for the whole statement. Further, we have to consider the arbitrariness of all the numbers; only the words and symbols `heap`, `ints from`, `to`, `(start=` and `)` are fixed. We do not even have complete information about the size of the 'white space' between the words.
For these reasons, we have to use a general *pattern* on which a statement like that given above matches. Such a pattern can easily be written as a regular expression:

_*heap_*[0-9]*_ints_from_[0-9a-f]*_to_[0-9a-f]*_(start=[0-9a-f]*)

where _ denotes one 'unit' of 'white space', [0-9] is a short form of 0|1|...|9, [0-9a-f] is a short form of 0|1...|9|a|b|...|f and * is the common symbol for an arbitrarily often but finite repetition. As soon as a matching pattern is recognized, the number matching the expression `[0-9a-f]`, standing lexicographically directly behind `from`, is the desired heap starting address.

B.1.2 how to find the cons cells

The FAST compiler knows two versions of objects, boxed and unboxed. Here are two examples of Cons cells reported by a FAST output trace:

```
-11-535-LI: cons=(0381d0,0382c6) at 0382c5

-01-CONS: h=3f120 t=3f12c at 3f138 box at 3f140
```

They could hardly be more different from each other! But there are also some common items. The first digit (after -) shows if the Cons is boxed or unboxed. The first example statement begins with -1, which means unboxed. The second one begins with -0, therefore it is boxed: no wonder, box at 3f140 can be found at its end. The first Cons cell has been statically allocated and -LI indicates its membership to some List of Integers. The second one has been allocated dynamically by the built in function CONS. In both examples —though in different forms— the location of the cell at an address can be found, and the addresses of the heads and tails are given too. Four regular expressions scan Cons cell information from an output trace:

_*–11–[0-9]*–[A-Z]*:_cons=([0-9a-f]*,[0-9a-f]*)_at_[0-9a-f]*

_*–01–[0-9]*–[A-Z]*:_cons=([0-9a-f]*,[0-9a-f]*)_at_[0-9a-f]*_box_at_[0-9a-f]*

_*–11–CONS:_h=[0-9a-f]*_t=[0-9a-f]*_at_[0-9a-f]*

_*–01–CONS:_h=[0-9a-f]*_t=[0-9a-f]*_at_[0-9a-f]*_box_at_[0-9a-f]*

(Of course they could be combined into a single expression, but the separation has been chosen in order to make the differences obvious). There are some built in functions, e.g. APPEND or FROMTO, which also allocate Cons cells. These allocations are traced in yet another form so that even further regular expressions are necessary if these built in functions are to be used, but the principle is always the same (and anyway, every built in function could also be written 'by hand').

B.1.3 how to find the vap nodes

As Vap nodes may have an arbitrary number of arguments, there would have to be an equivalent number of patterns to detect them all, which is of course not practicable. Most application will probably not have more than three arguments, so we will not give more patterns than necessary for detecting these. Of course in special cases, more patterns for Vap with more arguments may easily be added. Their structure is almost the same as the ones shown below. Two examples taken from a trace are:

```
vap2:  fun=38208,a1=3826c at 3f000

vap4:  fun=3819c,a1=38254,a2=38214,a3=3814f at 3f024
```

For every node vap_i we can see the trace reporting $i+1$ addresses. The first one, which the 'structural view' is not interested in, is the location of the function code belonging to this suspension node. Then there appear $i-1$ addresses for function arguments a1 to a_{i-1} and the address identifying the location of the Vap itself at some certain place in the heap. The regular expressions

detecting these patterns are:

_*vap2:_fun=[0-9a-f]*,a1=[0-9a-f]*_at_[0-9a-f]*

_*vap3:_fun=[0-9a-f]*,a1=[0-9a-f]*,a2=[0-9a-f]*_at_[0-9a-f]*

_*vap4:_fun=...

and so on

As in the Cons case, even more structurally different expressions may be necessary if trace reports of Vap producing built in functions like APPEND are to be examined.
While the the trace statements shown above are reporting Vap being allocated dynamically in the heap, there are also applications allocated statically outside which are reported in yet again a different form. It should now be sufficient just to mention that they can also be found by matching some further regular expressions.

B.1.4 how to find the caf nodes

It is very important to find Caf (even if they are never allocated in the heap) because they can become 'bridges' between heap structures of the program being examined. Here is an example of a trace statement reporting a Caf:

```
-LIT541:  caf=00557c busy=FALSE at 381fc
```

The attribute busy tells us something about the evaluation state of this Caf, but for the analysis only the last address giving the location at some certain place in the heap is important. The corresponding regular expression

_*–LIT[0-9]*:_caf=[0-9a-f]*_busy=[A-Z]*_at_[0-9a-f]*

picks the information about Caf nodes out of a FAST output trace.

B.2 filter script to be applied under UNIX

B.2.1 description

The following sed-script contains the necessary regular expressions for identifying the relevant FAST output trace statements of all experiments done in this thesis. It is possible that these regular expressions are not enough for traces caused by other functional programs which could, for example, produce Vap nodes of bigger arity or may use special FAST built in functions not mentioned here. Nevertheless, it should be obvious that the corresponding rules can easily be added to the existing script.
If this sed-script is applied to a FAST output trace, the relevant trace statements will not only be filtered, but also translated into C procedure calls as demanded by the analysis program. If *foo.trac* is the name of a file containing a FAST output trace, *filter.exe* the name of a file containing the sed-script as described above, then the UNIX command

filter.exe < *foo.trac* > *information.c*

will produce a file named *information.c* which contains the relevant trace statements translated into C procedure calls. Here is the source code of our filter script:

B.2.2 script code

```
sed -n -e 's/ *heap *\([0-9]*\) ints from \([0-9a-f]*\) to \([0-9a-f]*\)
 (start=\([0-9a-f]*\))/heapstart(0X\2);/p' \
        -e 's/ *_[01]*_main: return_[A-Z]*=\([0-9a-f]*\)/outputroot(0X\1);/p' \
        -e 's/ *_[a-zA-Z0-9]*: caf=\([0-9a-f]*\) busy=\([A-Z]*\) at \([0-9a-f]*\)
/_caf(0X\3);/p' \
        -e 's/ *_01_[0-9a-f]*_[A-Z]*: cons=(\([0-9a-f]*\),\([0-9a-f]*\))
 at \([0-9a-f]*\) box at \([0-9a-f]*\)/_cons(0X\1,0X\2,0X\3);
 _box(0X\3,0X\4);/p' \
        -e 's/ *_11_[0-9a-f]*_[A-Z]*: cons=(\([0-9a-f]*\),\([0-9a-f]*\))
 at \([0-9a-f]*\)/_cons(0X\1,0X\2,0X\3);/p' \
        -e 's/ *_[_0-9a-zA-Z]*: ap_vap tag=[0-9]* (\([0-9a-f]*\),\([0-9a-f]*\),
\([0-9a-f]*\),\([0-9a-f]*\),\([0-9a-f]*\)) at \([0-9a-f]*\)
/_vap(0X\2,0X\3,0X\4,0X\5,0X\6);/p' \
        -e 's/ *_[_0-9a-zA-Z]*: ap_vap tag=[0-9]* (\([0-9a-f]*\),\([0-9a-f]*\),
\([0-9a-f]*\),\([0-9a-f]*\)) at \([0-9a-f]*\)
/_vap(0X\2,0X\3,0X\4,UNDEF,0X\5);/p' \
        -e 's/ *_[_0-9a-zA-Z]*: ap_vap tag=[0-9]* (\([0-9a-f]*\),\([0-9a-f]*\),
\([0-9a-f]*\)) at \([0-9a-f]*\)/_vap(0X\2,0X\3,UNDEF,UNDEF,0X\4);/p' \
        -e 's/ *_[_0-9a-zA-Z]*: ap_vap tag=[0-9]* (\([0-9a-f]*\),\([0-9a-f]*\))
 at \([0-9a-f]*\)/_vap(0X\2,UNDEF,UNDEF,UNDEF,0X\3);/p' \
        -e 's/ *update: root=\([0-9a-f]*\) result=\([0-9a-f]*\) tag=\([0-9]*\)
/update(0X\1,0X\2);/p' \
        -e 's/ *vap4: fun=\([0-9a-f]*\),a1=\([0-9a-f]*\),a2=\([0-9a-f]*\),
a3=\([0-9a-f]*\) at \([0-9a-f]*\)/_vap(0X\2,0X\3,0X\4,UNDEF,0X\5);/p' \
        -e 's/ *vap3: fun=\([0-9a-f]*\),a1=\([0-9a-f]*\),a2=\([0-9a-f]*\)
 at \([0-9a-f]*\)/_vap(0X\2,0X\3,UNDEF,UNDEF,0X\4);/p' \
        -e 's/ *vap2: fun=\([0-9a-f]*\),a1=\([0-9a-f]*\) at \([0-9a-f]*\)
/_vap(0X\2,UNDEF,UNDEF,UNDEF,0X\3);/p' \
        -e 's/ *vap: n=4 fun=\([0-9a-f]*\) a1=\([0-9a-f]*\) a2=\([0-9a-f]*\)
 a3=\([0-9a-f]*\) at \([0-9a-f]*\)/_vap(0X\2,0X\3,0X\4,UNDEF,0X\5);/p' \
        -e 's/ *vap: n=3 fun=\([0-9a-f]*\) a1=\([0-9a-f]*\) a2=\([0-9a-f]*\)
 at \([0-9a-f]*\)/_vap(0X\2,0X\3,UNDEF,UNDEF,0X\4);/p' \
        -e 's/ *_01_CONS: h=\([0-9a-f]*\) t=\([0-9a-f]*\) at \([0-9a-f]*\)
 box at \([0-9a-f]*\)/_cons(0X\1,0X\2,0X\3); _box(0X\3,0X\4);/p' \
        -e 's/ *_11_CONS: h=\([0-9a-f]*\) t=\([0-9a-f]*\) at \([0-9a-f]*\)
/_cons(0X\1,0X\2,0X\3);/p' \
        -e 's/ *_01_APPEND: \([0-9a-f]*\)=(\([0-9a-f]*\):\([0-9a-f]*\))
 ++ \([0-9a-f]*\) at \([0-9a-f]*\) box at \([0-9a-f]*\) (vap \([0-9a-f]*\)
 \([0-9a-f]*\) at \([0-9a-f]*\))/_cons(0X\2,0X\9,0X\5); _box(0X\5,0X\6);
 _vap(0X\7,0X\8,UNDEF,UNDEF,0X\9);/p' \
        -e 's/ *_11_APPEND: \([0-9a-f]*\)=(\([0-9a-f]*\):\([0-9a-f]*\))
 ++ \([0-9a-f]*\) at \([0-9a-f]*\) (vap \([0-9a-f]*\) \([0-9a-f]*\)
 at \([0-9a-f]*\))/_cons(0X\2,0X\8,0X\5);
 _vap(0X\6,0X\7,UNDEF,UNDEF,0X\8);/p' \
```



```
        -e 's/ *_01_FROMTO_I: from=\([0-9]*\),to=\([0-9]*\) at \([0-9a-f]*\)
 box at \([0-9a-f]*\) (vap \([0-9a-f]*\) \([0-9a-f]*\) at \([0-9a-f]*\))
/_cons(0X\1,0X\7,0X\3); _box(0X\3,0X\4); _vap(0X\5,0X\6,UNDEF,UNDEF,0X\7);/p' \
        -e 's/ *_11_FROMTO_I: from=\([0-9]*\),to=\([0-9]*\) at \([0-9a-f]*\)
 (vap \([0-9a-f]*\) \([0-9a-f]*\) at \([0-9a-f]*\))/_cons(0X\1,0X\6,0X\3);
 _vap(0X\4,0X\5,UNDEF,UNDEF,0X\6);/p' \
        -e 's/ *reduce_bind: n=2 a1=\([0-9a-f]*\) f=\([0-9a-f]*\) at
 \([0-9a-f]*\)/_vap(0X\1,UNDEF,UNDEF,UNDEF,0X\3);/p' \
        -e 's/ *bind_pap: n=3 fun=\([0-9a-f]*\) a1=\([0-9a-f]*\) a2=\([0-9a-f]*\)
 a3=\([0-9a-f]*\) at \([0-9a-f]*\)/_vap(0X\2,0X\3,0X\4,UNDEF,0X\5);/p' \
        -e 's/ *bind: n=2 a1=\([0-9a-f]*\) f=\([0-9a-f]*\) at \([0-9a-f]*\)
/_vap(0X\1,UNDEF,UNDEF,UNDEF,0X\3);/p' \
        -e 's/ *_01_NULLARY_PACK: n=1 <\([0-9]*\)> at \([0-9a-f]*\)
/_box(\1,0X\2);/p' \
        -e 's/ *_01_PACK: n=5 <\([0-9]*\)\/\([0-9a-f]*\),\([0-9a-f]*\),
\([0-9a-f]*\),\([0-9a-f]*\)> at \([0-9a-f]*\) box at \([0-9a-f]*\)
/_pack(0X\2,0X\3,0X\4,0X\5,0X\6); _box(0X\6,0X\7);/p' \
        -e 's/ *_01_PACK: n=4 <\([0-9]*\)\/\([0-9a-f]*\),\([0-9a-f]*\),
\([0-9a-f]*\)> at \([0-9a-f]*\) box at \([0-9a-f]*\)
/_pack(0X\2,0X\3,0X\4,UNDEF,0X\5); _box(0X\5,0X\6);/p' \
        -e 's/ *_01_PACK: n=3 <\([0-9]*\)\/\([0-9a-f]*\),\([0-9a-f]*\)>
 at \([0-9a-f]*\) box at \([0-9a-f]*\)/_pack(0X\2,0X\3,UNDEF,UNDEF,0X\4);
 _box(0X\4,0X\5);/p' \
        -e 's/ *_01_PACK: n=2 <\([0-9]*\)\/\([0-9a-f]*\)> at \([0-9a-f]*\)
 box at \([0-9a-f]*\)/_pack(0X\2,UNDEF,UNDEF,UNDEF,0X\3); _box(0X\3,0X\4);/p' \
        -e 's/ *_01_PACK: n=1 <\([0-9]*\)> at \([0-9a-f]*\) box at \([0-9a-f]*\)
/_box(0X\1,0X\3);/p' \
        -e 's/ *_01_TAG: pack=\([0-9a-f]*\) tag=\([0-9]\) at \([0-9a-f]*\)
/_box(0X\1,0X\3);/p' \
        -e 's/ *_[0-1][0-1]_[_0-9a-zA-Z]*: _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
 _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\) _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
 _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\) _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
 _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\) _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
/use(0X\1); use(0X\2); use(0X\3); use(0X\4);
 use(0X\5); use(0X\6); use(0X\7);/p' \
        -e 's/ *_[0-1][0-1]_[_0-9a-zA-Z]*: _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
 _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\) _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
 _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\) _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
 _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)/use(0X\1); use(0X\2); use(0X\3);
 use(0X\4); use(0X\5); use(0X\6);/p' \
        -e 's/ *_[0-1][0-1]_[_0-9a-zA-Z]*: _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
 _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\) _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
 _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\) _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
/use(0X\1); use(0X\2); use(0X\3); use(0X\4); use(0X\5);/p' \
        -e 's/ *_[0-1][0-1]_[_0-9a-zA-Z]*: _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
 _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\) _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
 _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
/use(0X\1); use(0X\2); use(0X\3); use(0X\4);/p' \
        -e 's/ *_[0-1][0-1]_[_0-9a-zA-Z]*: _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
```

```
 _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\) _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
/use(0X\1); use(0X\2); use(0X\3);/p' \
        -e 's/ *_[0-1][0-1]_[_0-9a-zA-Z]*: _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
 _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)/use(0X\1); use(0X\2);/p' \
        -e 's/ *_[0-1][0-1]_[_0-9a-zA-Z]*: _[0-1][0-1]_[_0-9a-zA-Z]*=\([0-9a-f]*\)
/use(0X\1);/p' \
```

Appendix C

the analysis program

C.1 description

The analysis program, as used in this thesis, has to be prepared for its run under UNIX as follows. Let *definitions.c* be the name of a file which contains the definitions of the data structures and procedures necessary for the analysis and the beginning of the main procedure. A file named *conclude.c* contains some procedure calls concluding the analysis and the final symbol } concluding the main procedure. As explained in appendix B, those procedure calls representing the FAST output trace of the functional program to be analyzed are found in another file named *information.c*, and all these three files together are the special analysis program for the original functional program. They are combined in UNIX with the command

cat *definitions.c information.c conclude.c* > *analysis.c*

into the file *analysis.c* which is the complete analysis program, ready to be compiled and run. The basic idea of it is described in the following:

First, a structured type `NODE` is defined. It is to simulate a heap node and therefore contains attributes which describe typical properties of such a node. These are the cell type (Box, Cons, Pack, Vap or Caf), connections to other nodes, reference count, update information, an address, and a *colour* to separate intermediate from other nodes.

Then a pseudo heap is initialised. This initial 'heap' is an array of `NODE`s whose members have only undefined attributes.

From now on, the program run is determined by the procedure calls from the part *information.c* of the analysis program.

The procedure `outputroot` writes the address of the root node of the output structure into a global variable.

Similarly, the procedure `heapstart` writes the address of the beginning of the heap into a global variable. This is to distinguish statically allocated cells (outside the real heap) from dynamically allocated cells (inside the real heap) which for simplicity are both represented in the pseudo heap array of the simulation.

The procedures `_caf`, `_box`, `_cons`, `_vap` and `_pack` update the pseudo heap array with new `NODE`s, having attributes according to the properties of those cells.

The procedure `update` copies relevant attributes from the `result` node into the `root` node, and overwrites the yet undefined update attributes. The cell being copied is marked as a member of the class 'update garbage' with the colour `YELLOW`. Different cases have to be considered where root or result reside inside or outside the real heap.

The procedure `use` increments the usage attribute of a `NODE` when it has just been used as a

function argument.
When all procedure calls of the part *information.c* are done, the dynamic heap history is reconstructed and the actual analysis can begin. The procedure calls necessary for this purpose are supplied by the part *conclusion.c* of the analysis program:
The procedure **deduce** separates the particular nodes into composed structures. For this purpose, the root notes of these structures are searched first. Then the structures are 'painted' from their roots to their leaf cells. The output structure is painted BLUE; the corresponding output root is already known globally. Intermediate structures are painted RED.
Nota bene: Because of 'update garbage' and 'intermediate' not being mutually exclusive (sharing!), it must be allowed to overpaint YELLOW *cells* RED. Thus, it is only forbidden to paint a BLUE node RED. The structures are stored in a 2-dimensional auxiliary array.
The procedure `refcount` counts the connections between the NODEs in the pseudo heap. A NODE with 0 references is a possible structure root and therefore stored in an auxiliary memory array.
The procedure `filterstructs` takes the auxiliary structure memory array and cuts out all NODEs which do not belong to the real heap, but without destroying the structures themselves; (we do this because only dynamically allocated cells are relevant for the question of deforestation). Therefore information about the beginning of the heap is needed. The filtered structures are stored in another 2-dimensional auxiliary memory array.
Now the procedure `deforestable` removes all 'essential' and all 'not-really-intermediate' structures from the second auxiliary memory by setting the corresponding first element to 0, and the results of the analysis are ready to be shown:
The procedure `statistic` prints the analysis result, which is derived from the NODEs' attributes, and the content of the auxiliary array with the 'deforestable' intermediate class.
The procedure `search` helps to find a NODE with a given address out of the pseudo heap array.

C.2 source code in ANSI C

This is the code contained in the *definitions.c* part of the analysis.

```
/* clarity first! the reader may improve efficiency ad libitum */
#include <stdio.h>

/* these variables have to be adapted from case to case: */
#define ASSUMPTION_E 3
#define HEAPSIZE 60000
#define MEMSIZE   3000
#define PRINTSTRUCTS 0

/* descriptions: procedures' side effects and purposes: */
#define PREPARE   void
#define NEWCELL   void
#define OPERATE   void
#define INFORM    void
#define ANALYS    void
#define HELP      void
#define SHOW      void
```



```
/* definitions: possible properties: */
#define UNDEF       -2
#define YES          1
#define NO          -1
#define RED          3
#define BLUE        -3
#define YELLOW       8
#define INTERN       4
#define INVISIBLE   -4
#define INSIDECOPY   5
#define OUTSIDECOPY -5

/* definitions: possible attribute classes: */
#define ADDRESS  long
#define IDENT    int
#define KIND     int
#define NUMBER   int
#define ACTION   int

/* specification: type heap node. maximal arity is 4. */
typedef struct { ADDRESS cellnumber;
                 NUMBER   reference;
                 NUMBER       usage;
                 IDENT       colour;
                 KIND         isbox;
                 ADDRESS      boxto;
                 KIND        iscons;
                 ADDRESS     headto;
                 ADDRESS     tailto;
                 KIND        ispack;
                 ADDRESS       p1to;
                 ADDRESS       p2to;
                 ADDRESS       p3to;
                 ADDRESS       p4to;
                 KIND         isvap;
                 ADDRESS     arg1to;
                 ADDRESS     arg2to;
                 ADDRESS     arg3to;
                 ADDRESS     arg4to;
                 KIND         iscaf;
                 ACTION  upd_active;
                 ACTION upd_passive; } NODE;
```

```
/* declarations: global variables:                                    */
ADDRESS heapbegin = UNDEF;    /* information: beginning of the heap    */
ADDRESS output    = UNDEF;   /*  information: output structure root   */
NUMBER firstfreeplace = 0;  /* current heapsize                       */
NUMBER memosize       = 0; /* current auxiliary memory size          */
NUMBER imdstructsize;     /* current position of struct. deduction */
NUMBER intermedsize ;    /* amount of intermediate structures     */
NUMBER deforstructs ;   /* amount of deforestable interm.structs.*/
NUMBER deforcells   ;  /* amount of deforestable interm. cells  */

/* declarations: data structures:                                     */
NODE    heap[HEAPSIZE];                  /* pseudo heap                */
ADDRESS memo[MEMSIZE][MEMSIZE];       /* aux.mem. for heap structures */
ADDRESS intermed[MEMSIZE][MEMSIZE]; /* aux.mem. intermediate strcts */

/* definitions: heap array access makros: */
#define OPEN  new=heap[firstfreeplace];
#define CLOSE heap[firstfreeplace]=new; firstfreeplace++

/* declarations: program procedure types: */
PREPARE initheap();
INFORM  heapstart(ADDRESS);
INFORM  outputroot(ADDRESS);
NEWCELL _caf(ADDRESS);
NEWCELL _box(ADDRESS, ADDRESS);
NEWCELL _cons(ADDRESS, ADDRESS, ADDRESS);
NEWCELL _vap(ADDRESS, ADDRESS, ADDRESS, ADDRESS, ADDRESS);
NEWCELL _pack(ADDRESS, ADDRESS, ADDRESS, ADDRESS, ADDRESS);
OPERATE update(ADDRESS, ADDRESS);
HELP    refcount();
HELP    saveroot(ADDRESS);
ANALYS  deduce();
ANALYS  paint(ADDRESS, IDENT, NUMBER);
NUMBER  search(ADDRESS);
INFORM  use(ADDRESS);
HELP    filterstructs();
ANALYS  deforestable();
SHOW    statistic();
```



```
/* definitions: program procedure bodies: */

PREPARE initheap() /* generate empty heap */
        {NUMBER i;
         NODE empty;
         empty.cellnumber  = UNDEF;
         empty.colour      = UNDEF;
         empty.reference   = UNDEF;
         empty.usage       = 0;
         empty.isbox       = UNDEF;
         empty.boxto       = UNDEF;
         empty.iscons      = UNDEF;
         empty.headto      = UNDEF;
         empty.tailto      = UNDEF;
         empty.isvap       = UNDEF;
         empty.arg1to      = UNDEF;
         empty.arg2to      = UNDEF;
         empty.arg3to      = UNDEF;
         empty.arg4to      = UNDEF;
         empty.iscaf       = UNDEF;
         empty.ispack      = UNDEF;
         empty.p1to        = UNDEF;
         empty.p2to        = UNDEF;
         empty.p3to        = UNDEF;
         empty.p4to        = UNDEF;
         empty.upd_active  = NO;
         empty.upd_passive = NO;
         for(i=0; i<HEAPSIZE; i=i+1)heap[i]=empty;
         firstfreeplace = 0;
         return;}


INFORM heapstart(ADDRESS begin) /* make heap begin known */
       {heapbegin=begin;
        return;}


INFORM outputroot(ADDRESS mainreturn) /* make output root known */
       {output = mainreturn;
        return;}
```

```
NEWCELL _caf(ADDRESS adr) /* insert new cell */
        {NODE new;
         OPEN;
         new.cellnumber = adr;
         new.isbox      = NO;
         new.iscons     = NO;
         new.isvap      = NO;
         new.ispack     = NO;
         new.iscaf      = YES;
         CLOSE;
         return;}


NEWCELL _box(ADDRESS adr, ADDRESS boxadr)
        {NODE new;
         OPEN;
         new.cellnumber = boxadr;
         new.isbox      = YES;
         new.boxto      = adr;
         new.iscons     = NO;
         new.isvap      = NO;
         new.ispack     = NO;
         new.iscaf      = NO;
         CLOSE;
         return;}


NEWCELL _cons(ADDRESS head, ADDRESS tail, ADDRESS adr)
        {NODE new;
         OPEN;
         new.cellnumber = adr;
         new.isbox      = NO;
         new.iscons     =  YES;
         new.headto     = head;
         new.tailto     = tail;
         new.isvap      = NO;
         new.ispack     = NO;
         new.iscaf      = NO;
         CLOSE;
         return;}
```



```
NEWCELL _vap(ADDRESS a1, ADDRESS a2, ADDRESS a3, ADDRESS a4, ADDRESS adr)
        {NODE new;
         OPEN;
         new.cellnumber = adr;
         new.isbox      = NO;
         new.iscons     = NO;
         new.isvap      = YES;
         new.arg1to     = a1;
         new.arg2to     = a2;
         new.arg3to     = a3;
         new.arg4to     = a4;
         new.ispack     = NO;
         new.iscaf      = NO;
         CLOSE;
         return;}


NEWCELL _pack(ADDRESS p1,ADDRESS p2,ADDRESS p3,ADDRESS p4, ADDRESS adr)
        {NODE new;
         OPEN;
         new.cellnumber = adr;
         new.isbox      = NO;
         new.iscons     = NO;
         new.isvap      = NO;
         new.iscaf      = NO;
         new.ispack     = YES;
         new.p1to       = p1;
         new.p2to       = p2;
         new.p3to       = p3;
         new.p4to       = p4;
         CLOSE;
         return;}
```

```
OPERATE update(ADDRESS root, ADDRESS result) /* graph reduction */
        {NUMBER ro;
         NUMBER re;
         NODE    k;
         NODE    l;
         ro = search(root);
         re = search(result);
         k  = heap[ro];
         if(re==UNDEF) /* case: result is not in heap array */
           {k.upd_active = INVISIBLE;
            heap[ro] = k;
            return;}
         l = heap[re]; /* case: result is in heap array */
         k.iscons = l.iscons;
         k.headto = l.headto;
         k.tailto = l.tailto;
         k.isbox  = l.isbox;
         k.boxto  = l.boxto;
         k.ispack = l.ispack;
         k.p2to   = l.p2to;
         k.p3to   = l.p3to;
         k.p4to   = l.p4to;
         k.arg1to = l.arg1to;
         k.arg2to = l.arg2to;
         k.arg3to = l.arg3to;
         k.arg4to = l.arg4to;
         if(l.cellnumber>=heapbegin)k.upd_active  = INTERN;
         if(l.cellnumber<heapbegin)k.upd_active   = INVISIBLE;
         heap[ro] = k;
         if(k.cellnumber>=heapbegin)l.upd_passive = INSIDECOPY;
         if(k.cellnumber<heapbegin)l.upd_passive  = OUTSIDECOPY;
         if(result!=output)l.colour = YELLOW;
         heap[re] = l;
         return;}
```



```
HELP refcount() /* reference count in pseudo heap array */
     {NUMBER count;
      NUMBER     i;
      NUMBER     j;
      NODE       k;
      NODE       l;
      for(i=0; i<firstfreeplace; i=i+1)  /* for every NODE: */
         {k=heap[i];
          count=0;
          for(j=0; j<firstfreeplace; j=j+1)   /* count ref. */
             {l=heap[j];
              if(l.boxto == k.cellnumber)count++;
              if(l.headto== k.cellnumber)count++;
              if(l.tailto== k.cellnumber)count++;
              if(l.p1to ==  k.cellnumber)count++;
              if(l.p2to ==  k.cellnumber)count++;
              if(l.p3to ==  k.cellnumber)count++;
              if(l.p4to ==  k.cellnumber)count++;}
          if(count==0 && k.colour!=YELLOW)saveroot(k.cellnumber);}
      return;}


HELP saveroot(ADDRESS root) /* store possible root into aux.mem. */
     {NUMBER i=0;
      NUMBER j;
      memo[memosize][0]=root;
      memosize++;
      return;}


ANALYS deduce() /* detect cell structures */
       {ADDRESS root;
        NUMBER     i;
        refcount();
        paint(output, BLUE, (MEMSIZE-1)); /* output blue */
        for(i=0; i<memosize; i=i+1)
           {imdstructsize=0;
            root=memo[i][0];
            paint(root, RED, i);}    /* intermediate red */
        return;}
```

```
ANALYS paint(ADDRESS ad, IDENT color, NUMBER structure)
       {NODE   k;
        NUMBER n;
        n = search(ad);
        if(n==UNDEF)return; /* case: nothing found */
        k = heap[n];
        if(k.colour!=BLUE && k.colour!=RED)
          {k.colour=color;
           heap[n]=k;
           memo[structure][imdstructsize]=k.cellnumber;
           imdstructsize++;
           if(k.isbox==YES)paint(k.boxto, color, structure);
           if(k.iscons==YES){paint(k.headto, color, structure);
                             paint(k.tailto, color, structure);}
           if(k.ispack==YES){paint(k.p1to, color, structure);
                             paint(k.p2to, color, structure);
                             paint(k.p3to, color, structure);
                             paint(k.p4to, color, structure);}}
        return;}


NUMBER search(ADDRESS cell) /* search cell position in heap array */
       {NUMBER i;
        NUMBER r = UNDEF;
        if(cell==UNDEF)return UNDEF;
        for(i=0; i<firstfreeplace; i=i+1)
           {if((heap[i]).cellnumber == cell)
             {r=i; i=firstfreeplace;}}
        return r;}  /* return cell position if found, else undef */


INFORM use(ADDRESS arg) /* increment usage attribute of arg */
       {NUMBER n;
        NODE   k;
        n=search(arg);
        if(n==UNDEF)return;
        k=heap[n];
        k.usage++;
        heap[n]=k;
        return;}
```



```
HELP filterstructs() /* filter static allocations out of structures */
     {ADDRESS  a;
      NUMBER x=0;
      NUMBER y=0;
      NUMBER u=0;
      NUMBER v=0;
      for(x=0;x<memosize;x=x+1)
         {v=0;
          for(y=0;y<MEMSIZE;y=y+1)
             {a=memo[x][y];
              if(a>=heapbegin && a!=output)
                {intermed[u][v]=a;
                 v=v+1;}}
         if(v>0)u=u+1;}
      intermedsize=u;
      return;}
```

```
ANALYS deforestable() /* keep only deforestable interm. structures */
       {ADDRESS a;
        NODE    k;
        NUMBER  i;
        NUMBER  j;
        NUMBER  n;
        NUMBER condition;
        NUMBER  onlynull;
        for(i=0;i<intermedsize;i++)
           {condition=1;
            onlynull =1;
            j=0;
            while(j<MEMSIZE && condition==1)
                 {a=intermed[i][j];
                  j=j+1;
                  if(a<heapbegin)condition=0;
                  n=search(a);
                  if(n!=UNDEF)
                    {k=heap[n];
                     if(k.usage>0)onlynull=0;
                     if(k.usage>ASSUMPTION_E)
                       {condition=0;
                        intermed[i][0]=0;}}}
            if(onlynull==1)intermed[i][0]=0;}
        n=0;
        for(i=0;i<intermedsize;i++)
           {if(intermed[i][0]>0)
              {n++;
               j=0;
               while(intermed[i][j]>0)
                    {deforcells++;
                     j++;}}}
        deforstructs=n;
        return;}
```



```
SHOW statistic() /* analysis results */
      {NUMBER box    = 0;
       NUMBER cons   = 0;
       NUMBER pack   = 0;
       NUMBER vap    = 0;
       NUMBER upd    = 0;
       NUMBER updstr = 0;
       NUMBER updval = 0;
       NUMBER out    = 0;
       NUMBER all    = 0;
       NUMBER imd    = 0;
       NUMBER gbgi   = 0;
       NUMBER gbgc   = 0;
       NUMBER rem    = 0;
       NUMBER structs;
       NUMBER   print;
       NUMBER  noness;
       NUMBER       i;
       NUMBER       j;
       NODE         k;
       float   speed1;
       float   speed2;
       for(i=0; i<firstfreeplace; i=i+1)
          {k=heap[i];
           if(k.cellnumber>=heapbegin) /* if not on heap, don't count */
             {if(k.isbox   == YES && k.isvap == NO)  box++;
              if(k.iscons == YES && k.isvap == NO) cons++;
              if(k.ispack == YES && k.isvap == NO) pack++;
              if(k.isvap   == YES)                   vap++;
              if(k.isvap==YES && k.upd_active==NO)  rem++;
              if(k.isvap==YES && k.upd_active!=NO)  upd++;
              if(k.isvap==YES && k.upd_active == INVISIBLE) updval++;
              if(k.isvap==YES && k.upd_active == INTERN)    updstr++;
              if(k.upd_passive==OUTSIDECOPY) gbgc++;
              if(k.upd_passive==INSIDECOPY)  gbgi++;
              if(k.colour == RED) imd++;
              if(k.colour==BLUE) out++;}}
       all = box + cons + vap + pack;
       speed2 = 100.0*((float)imd)/(float)all;
       filterstructs();
       structs = intermedsize;
       deforestable();
       noness  = deforstructs;
       speed1 = 100.0*((float)deforcells)/(float)all;
```

```
    printf("\n");
    printf("________________________________________________________\n");
printf("%d Nodes: %d Box, %d Cons, %d Pack, %d Vap\n",all,box,cons,pack,vap);
    printf("%d VapUpd: %d from inside, %d from outside\n",upd,updstr,updval);
    printf("%d Vap remain without being updated\n", rem);
    printf("%d Cells became garbage by updating inside Vap.\n", gbgi);
    printf("%d Cells became garbage by updating outside Vap or Caf.\n",gbgc);
    printf("%d Cells belong to the output structure.\n",out);
    printf("%d Cells are intermediate and belong to\n",imd);
    printf("%d different intermediate structures.\n",structs);
    printf("%d structures are 'really intermediate' and contain\n",noness);
    printf("%d deforestable intermediate cells, which are:\n",deforcells);
    printf("________________________________________________________\n");
    if(PRINTSTRUCTS)
     {print=1;
      for(i=0;i<intermedsize;i++)
        {if(print==1)printf("\n STRUCTURE: ");
         print=0;
         j=0;
         while(intermed[i][j]>0)
              {printf("%x,",intermed[i][j]);
               print=1;
               j++;}}
      printf("\n_____________________________________________________\n");}
    printf("%.2f per cent of all cells are intermediate.\n",speed2);
    printf("%.2f per cent of all cells are deforestable.\n",speed1);
    printf("The 'essential' assumption is %d.\n",ASSUMPTION_E);
    printf("________________________________________________________\n");
    return;}
```

```
main(){initheap();
```

The part *conclude.c* of the analysis contains nothing more than the calls:

```
    deduce();
    statistic();}
```

and the concluding curly bracket of the main procedure.

Appendix D

abstracts in dutch and german

D.1 samenvatting

Recentelijk zijn een aantal artikelen over ontbossende programmatransformaties verschenen, die aanzienlijke efficientieverbeteringen zouden moeten bewerkstelligen. Zulke resultaten worden in de weinige in de literatuur over ontbossing beschreven tijd- en geheugenruimtemetingen echter niet gevonden. De vraag naar de grootte van de door het ontbossen werkelijk optimaliseerbare deelverzameling van alle functionele programma's is in de literatuur nog niet gesteld. In deze scriptie wordt een algemene automatische methode ter afschatting van de winst van ontbossing voor ieder functionele programma ontwikkeld, onafhankelijk van de eventuele ontbossingstechniek. De methode wordt uitgelegd aan de hand van verschillende voorbeelden, en de geschiktheid van de methode wordt behandeld.
Onder ontbossing verstaan wij methoden die tussenstructuren verwijderen door middel van programmatekstveranderingen in het algemeen, en beperken het begrip niet alleen op de automatische transformaties, zoals die in de literatuur zijn voorgesteld. Tussenstructuren —bomen in het algemeen, lijsten in het bijzonder— treden op, als functies die op zulke datastructuren werken op een bepaalde manier met elkaar worden gecombineerd. Het hier ontwikkelde ontbossingsanalyseprogramma kan optredende tussenstructuren herkennen door de informatie van 'output traces' van de compiler te verwerken, nadat deze traces in een voor de analyse geschikte vorm gebracht zijn. Het analyse programma vergelijkt de aantallen van verschillende soorten knopen en voorspelt met behulp van twee redelijke ontbossingsaannamen de maximale winst die een willekeurige ontbossende transformatie op het geanalyseerde functionele programma kan bereiken.
In het eerste gedeelte van de scriptie worden de analyseprincipes en de creatie van de traces met behulp van een abstracte machine uitgelegd. In de abstractie weerspiegelt zich ook de hiervoor opgemerkte geschikte vorm van de trace. Het tweede gedeelte laat de toepassing van de in het eerste gedeelte voorbereidde methoden op sommige voorbeelden zien. De resultaten bevestigen de eerdere in de literatuur verschenen metingen, zodat een groot 'ontbossingwonder' in de nadere toekomst waarschijnlijk niet te verwachten is: een volledige ontbossing van alle door een functioneel programma gemaakte tussenstructuren blijkt in de meeste realistische gevallen onmogelijk, omdat vaak gebruikte structuren de dynamische relaties in het geheugen bijna willekeurig ingewikkeld maken.