

ELIMINATION OF VARIABLES
FROM
FUNCTIONAL PROGRAMS

O. de Moor

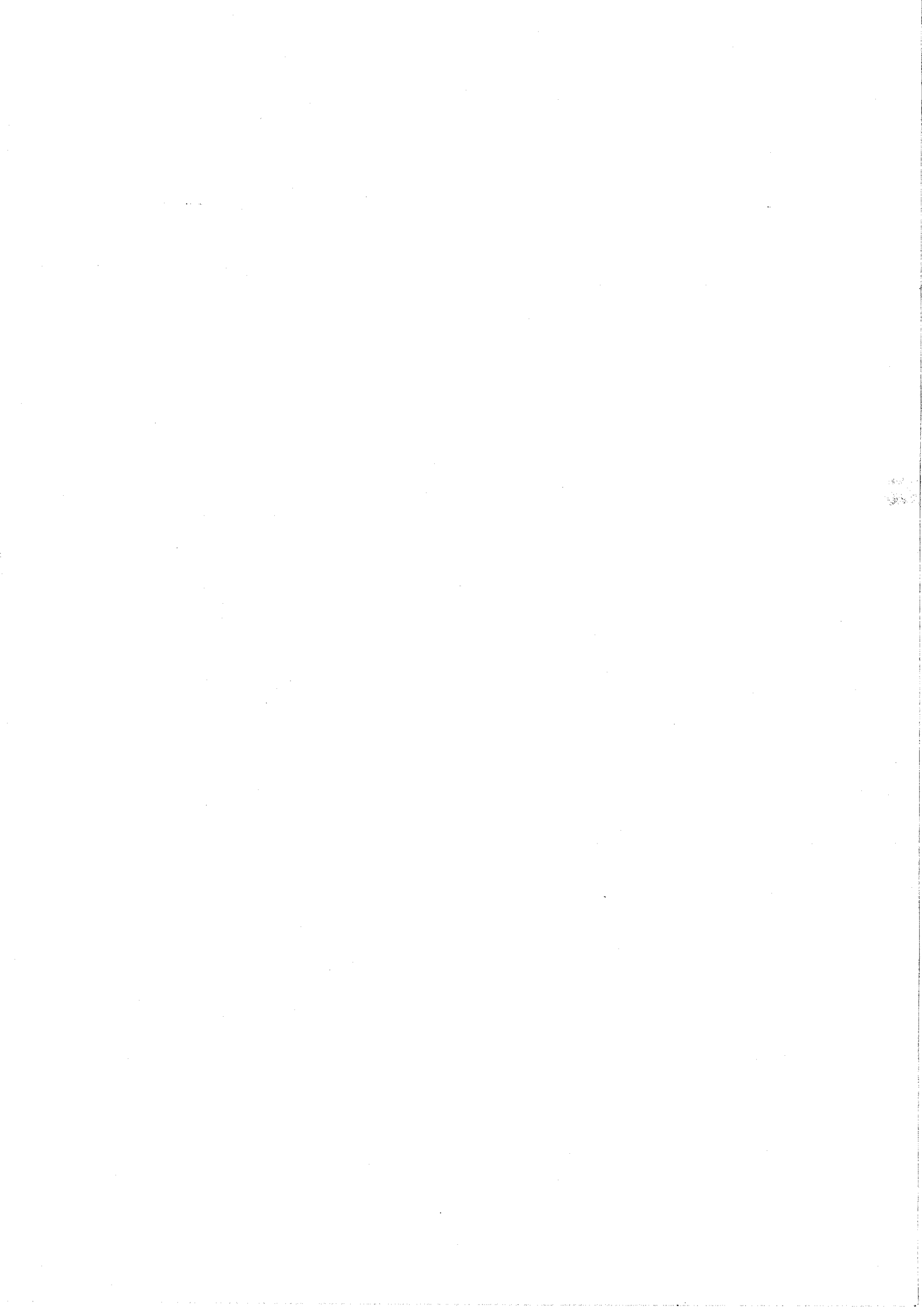
RUU-CS-87-24
November 1987



Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestiaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands



Abstract

After a brief introduction to functional programming, it is shown how by elimination of variables the substitutions involved in function application can be avoided. The variable-free functions resulting from the basic algorithm are however very large. Their size depends exponentially on the size of the original expression. The rest of the paper is devoted to the question of how to reduce the expression size. A well-known algorithm by David Turner is presented and analysed. Next, we turn to the 'best' solution. It is proved that only a weak form of improvement is attainable. An algorithm by Rick Statman that is optimal in this weak sense is discussed. Clearly, both the algorithm of Statman and that of Turner could be improved. We discuss systematical ways of doing so. This results in an algorithm combining the optimal results of Statman's method with the good average behaviour of Turner's. This algorithm works for a restricted, but very large class of functions. It highly resembles a method proposed by Warren Burton.

Contents

1	Introduction	3
1.1	Functions	4
2	Elimination of variables	16
3	Turner's algorithm	22
3.1	The algorithm	22
3.2	Caf size for applicative forms	27
3.2.1	A simplified algorithm: T'	27
3.2.2	Changing the tree representation	28
3.2.3	Construction of a worst case	33
3.2.4	Worst case for the original algorithm (T)	41
3.3	Generalization to λ -terms	43
4	Optimum code size	45
4.1	No general solution	45
4.2	Weakened optimality	46
4.2.1	Lowerbound on $wcl_a(n)$	46
4.2.2	An optimal algorithm	49
5	Possible improvements	57
5.1	Introduction of combinators	59
5.1.1	Adding extra combinators to Turner's algorithm	59
5.1.2	Adding combinators to Statman's algorithm	61
5.2	The redundancy of new combinators	63
5.3	Abstraction of expressions	64
5.3.1	Conditions on abstraction	65
5.3.2	The optimal abstraction sequence	67
5.3.3	Abstractions in Statman's algorithm	68

5.3.4	Turner on the chopping block	72
5.3.5	Abstraction of maximal free expressions	80
6	Discussion	83
	Acknowledgements	84
	Bibliography	85
	Index	87

Chapter 1

Introduction

Since the early days of computer science, it is a hotly disputed question what a ‘good’ programming language should look like. In the past decade, more and more people have argued that functional languages, which do not exhibit the cumbersome features of assignment, are best suited to the future [Bac78,Tur82]. Few people will deny that from a mathematical point of view such programs are elegant. However, there is one serious drawback: functional programs tend to be very inefficient, consuming lots of memory and computer-time. Therefore, it is important to study their implementation.

A technique that has received considerable attention during the past few years is the translation of functional programs into what has been called ‘combinator code’ [Tur79b,Tur79a,Hug82,HG85]. Few people have addressed the problem of minimizing the code-size. It is this problem we seek to solve here. A survey of the efforts made was drawn up by Hans Mulder [Mul85]. An in-depth study of the subject is provided in [JRB85]. At the time of this research, the author was unaware of the latter work. The idea to use abstraction of subexpressions, described further on in this paper, was suggested to me by Doaitse Swierstra.

The reader of this paper should possess a minimal background in mathematics. Though some of the proofs use advanced techniques, they are not vital to understanding the main issues. Some programming experience is tacitly assumed. Acquaintance with a functional language like SASL [Tur76] may be helpful.

1.1 Functions

In mathematics, functions are often regarded as some special kind of *relation*, where each argument is associated with a unique result. In common speech, a function denotes an *activity* of a person or thing. This contrasts with the non-operational mathematical approach.

Most of those who have done some programming will feel that functions are pretty strong tools. It may come as a surprise to them, though, that anything computable can be expressed in functions alone. No assignment, no control-structures like *'if'* or *'while'* are needed, just functions and application of functions. This unproved fact is known as Church's thesis. It cannot be proved, since that would require a similar statement defining 'computability'. The thesis is assumed to be correct because it is proven to be equivalent to all other commonly accepted notions of computation.

Functions are well studied mathematical objects. This speaks strongly in favour of expressing algorithms in a functional way. Since none of the notorious problems connected with the assignment such as aliasing (distinct variables denoting the selfsame object in memory) or side-effects (alteration of global variables in a procedure-body) block our insight in the structure of programs, it makes an elegant way to reason about algorithms.

To support his conjecture, Alonzo Church invented a formal system to talk about functions, the λ -calculus. We shall use this notation here. For a much more thorough introduction to the subject, see [Bar84]. Basically, there are two operations: λ -abstraction (introducing a parameter) and application of a function to its argument.

An example will clarify this. Suppose we want to describe the value of $(x + y)^2$, given x and y . In ordinary high-school notation, one would write something like this:

$$f(x, y) = (x + y)^2 \tag{1.1}$$

In the λ -calculus, it is written:

$$f = (\lambda x. (\lambda y. (\underbrace{\text{square}(\underbrace{(\underbrace{\text{plus } x} y)}_{E_3})}_{E_4})}_{E_2}))_{E_1} \tag{1.2}$$

E_3 stands for $x + y$. We write all operations in prefix notation: first the

operator, then the arguments. One might wonder what E_4 is. It signifies the *function* that adds x to its argument. This is our first example of a *higher-order function*: *plus* applied to one argument yields a function as a result. We will return to this phenomenon in a moment. E_2 denotes $(x+y)^2$. Again, the prefix-convention is used. Both *plus* and *square* are predefined functions. In E_1 , y is bound to E_2 as an argument-name. Likewise λx in E_0 names x as the first argument of the whole expression.

As can be seen, functions of two arguments may be written as one with a single argument using higher-order functions. This process is called *currying* (after the logician H.B. Curry). The concept of a higher-order function (having a functional argument and/or result) may seem a bit strange, but in fact it is quite common. Familiar examples are differentiation and integrating over a variable-length interval.

Now let us take a close look at what forms a λ -expression E may assume. The following definition gives rules for all possibilities.

Definition 1 (*λ -expression*)

A *λ -expression* is a sentence from the language described by

$$\begin{aligned} E &::= (EE) && \text{application} \\ E &::= (\lambda x.E) && \lambda\text{-introduction} \\ E &::= c_i && \text{constant } i = 0, \dots, m \\ E &::= x_i && \text{variable } i = 0, \dots \end{aligned}$$

Consider the application of our example (1.2) to the constant 2:

$$((\lambda x.(\lambda y.\text{square}(\text{plus } x)y))2) \tag{1.3}$$

Upon evaluation this appears to result into:

$$(\lambda y.\text{square}(\text{plus } 2)y) \tag{1.4}$$

This λ -term is equivalent to the function g , where $g(y) = (2 + y)^2$. We obtained it through substituting the argument 2 for the occurrences of x . As the reader will expect, the evaluation of a λ -expression may be defined by textual substitution. We shall try to put this more precisely.

Consider the following expression (which equals 5).

$$(\lambda x.(\text{plus}(\lambda x.x) 4) x) 1) \tag{1.5}$$

Clearly, one should not simply substitute a value for all occurrences of x . Only 'free' occurrences may be used. A variable is *free* if it is not *bound*, this is to say, in $\lambda x.(\text{plus } x) x$ x is bound, whereas in $((\text{plus } x) x)$ it is free.

Definition 2 (*free variables in a λ -term*)

Let t be a λ -term. The set of free variables in t , $freevars(t)$, is defined by:

$$\begin{aligned}
 freevars(t) := & \text{ if } t = (E_1 E_2) \rightarrow freevars(E_1) \cup freevars(E_2) \\
 & \square t = (\lambda x_i. E) \rightarrow freevars(E) - \{x_i\} \\
 & \square t = c \rightarrow \emptyset \\
 & \square t = x_i \rightarrow \{x_i\} \\
 & \text{fi}
 \end{aligned}$$

Definition 3 (*substitution*)

Let t be a λ -term, and let x denote a variable. The substitution of t for x in an expression E ($E[t/x]$) is defined by:

$$\begin{aligned}
 E[t/x] := & \text{ if } E = (E_1 E_2) \rightarrow (E_1[t/x] E_2[t/x]) \\
 & \square E = (\lambda y. E_1) \text{ and } \\
 & \quad y \neq x \text{ and } \\
 & \quad y \notin freevars(t) \rightarrow (\lambda y. E_1[t/x]) \\
 & \square E = (\lambda y. E_1) \text{ and } \\
 & \quad y = x \rightarrow E \\
 & \square E = (\lambda y. E_1) \text{ and } \\
 & \quad y \in freevars(t) \rightarrow E \\
 & \square E = c \rightarrow c \\
 & \square E = x \rightarrow t \\
 & \text{fi}
 \end{aligned}$$

Some may wonder why the special case for $y \in freevars(t)$ is included. We shall discuss this in a moment. Let us first define the execution (evaluation) of a λ -expression.

Definition 4 (*evaluation of a λ -expression*)

Let E be a λ -expression. Its evaluation is defined as follows:

$$\begin{aligned}
 eval(E) := & \text{ if } E = ((\lambda x. E_1) E_2) \rightarrow eval(E_1[E_2/x]) \\
 & \square E = (E_1 E_2) \rightarrow eval(eval(E_1) eval(E_2)) \\
 & \square E = (\lambda x. E_1) \rightarrow (\lambda x. E_1) \\
 & \square E = c \rightarrow c \\
 & \square E = x \rightarrow \text{undefined} \\
 & \text{fi}
 \end{aligned}$$

Now return to the question raised just before we formulated the above def-

inition. Why is the clause

$$E = (\lambda y.F) \text{ and } y \in \text{freevars}(t) \rightarrow E \quad (1.6)$$

necessary in the definition of substitution? An example will illustrate this. Consider

$$((\lambda y.(\lambda x.(\lambda y.((\text{plus } y)x))1)y)2) \quad (1.7)$$

If 1.6 was just ignored one could get both

$$((\text{plus } 1) 1) \quad (1.8)$$

and

$$((\text{plus } 1) 2) \quad (1.9)$$

as a result of the evaluation. 1.6 prohibits 1.8 as a correct execution-output, which is what we want. However, the condition is a little too restrictive. This is clear from the evaluation of:

$$(\lambda x.(\lambda y.(((\lambda x.(\lambda y.(\text{square}((\text{plus } x)y))))(\text{square } x))(\text{square } y)))) \quad (1.10)$$

We are not getting very far. Yet 1.10 is a perfectly legal expression, intuitively equal to $(x^2 + y^2)^2$. One solution to avoid such name-clashes is the renaming of variables:

$$(\lambda x.(\lambda y.(((\lambda a.(\lambda b.(\text{square}((\text{plus } a)b))))(\text{square } x))(\text{square } y)))) \quad (1.11)$$

is equivalent to 1.10 but it is evaluable. In the sequel, it is assumed that within $(\lambda x.E)$ no proper subexpression $(\lambda x.D)$ occurs. If this is the case, suitable renaming solves the problem.

Some readers may be in doubt whether the evaluation of an expression has been defined properly. In fact, the evaluation of an expression may result in expressions that differ in form. However such differing expressions may be turned into the same expression if some evaluation within function bodies (b in $\lambda x.b$) is done. This delightful phenomenon is called the *Church-Rosser property*.

The evaluation-step $((\lambda x.E_1)E_2) \Rightarrow E_1[E_2/x]$ is called a *reduction*. The question that remains is what order of reductions we should use in a deterministic evaluation process. If $((\lambda x.E_1)E_2) \Rightarrow \text{eval}(E_1[E_2/x])$ is always executed in favour of $(E_1E_2) \Rightarrow \text{eval}(\text{eval}(E_1) \text{eval}(E_2))$ then we are doing what has been called *lazy evaluation*. If it is the other way round, the process is called *compositional evaluation*. Most conventional programming

languages use the latter strategy: evaluate the components of an expression before the expression itself.

The Church–Rosser property tells us that there is no fundamental difference between both methods. In practice, there is an important difference. Compositional evaluation may go on forever in cases where lazy evaluation would stop. Unless otherwise stated, lazy evaluation is assumed to be the reduction strategy throughout the rest of this paper. We did not touch on the subject of evaluation of predefined functions. Special evaluation–rules (also called *reductions*) could be added for them.

The syntax of λ –expressions suggests an alternative to the obscure parenthesized formulas that were used up to now. We could think of a λ –term as a tree. The correspondence is depicted in figure 1.1. Some examples are to be found in figure 1.2

Lazy evaluation is done by running down the tree from the top node, always choosing the left branch, until in some node N a variable–introduction (λx) is found. Assume that it is a son to some application node, say P (arent). The left son of P , N , is a tree representing a function, the right son is the argument (A) (see figure 1.3). The free occurrences of x below N are replaced by the argument A . To this end, the whole subtree has to be searched, something quite costly. When the substitutions are completed, P is replaced by the function–body below N . N , P and M are superfluous by now, hence they are discarded. If P does not exist (N is the root of the tree), there is nothing to evaluate.

Clearly the substitution process makes evaluation expensive. Variable–free expressions may be computed efficiently, as illustrated by the following pascal fragment.

Algorithm 1 (*evaluation of a variable–free λ –expression*)

```

TYPE tree      = ^treenode;
   treenode = RECORD
       value           : con_expr;
       left_son, right_son : tree
   END;

```

```

FUNCTION execute(root : tree) : con_expr;

```

```

   PROCEDURE descent(VAR father, current : tree);

```

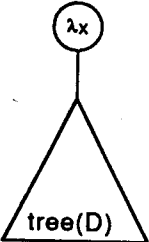
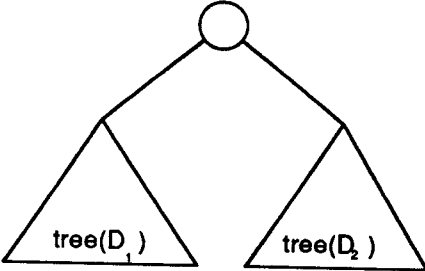
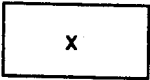
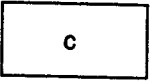
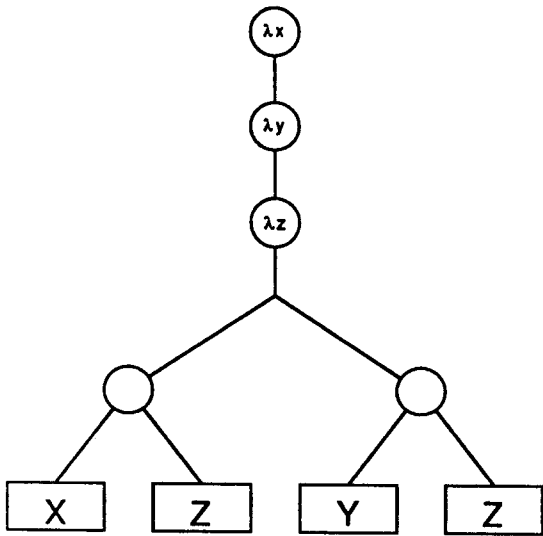
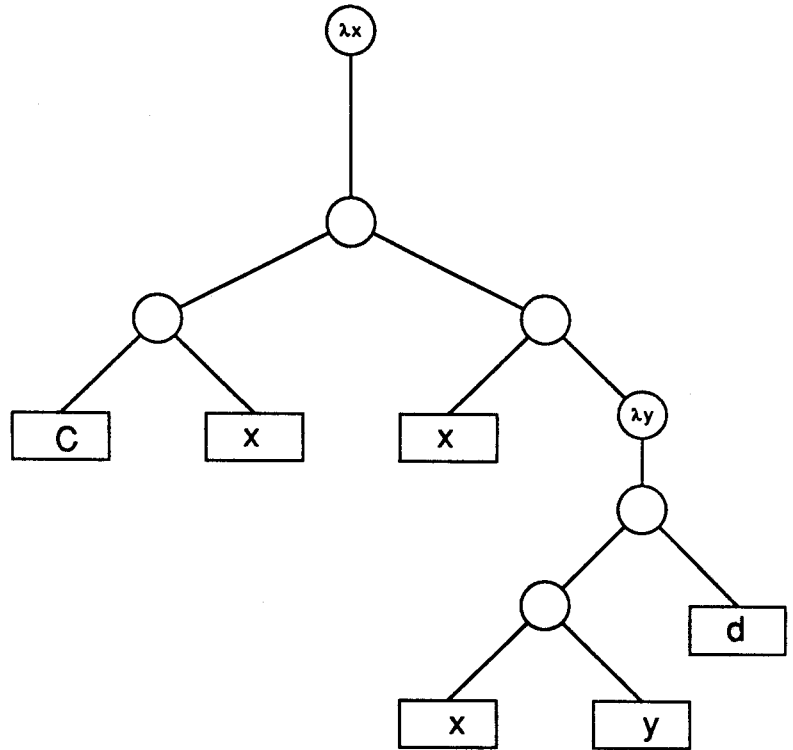
formula	tree
$(\lambda x . D)$	
$(D_1 D_2)$	
X	
C	

Figure 1.1: Formula and tree representations of λ -terms.



tree representation of
 $(\lambda x.(\lambda y.(\lambda z.(x z) (y z))))$



tree representation of
 $(\lambda x.(c x) (x (\lambda y.(x y) d)))$

Figure 1.2: Example tree representation.

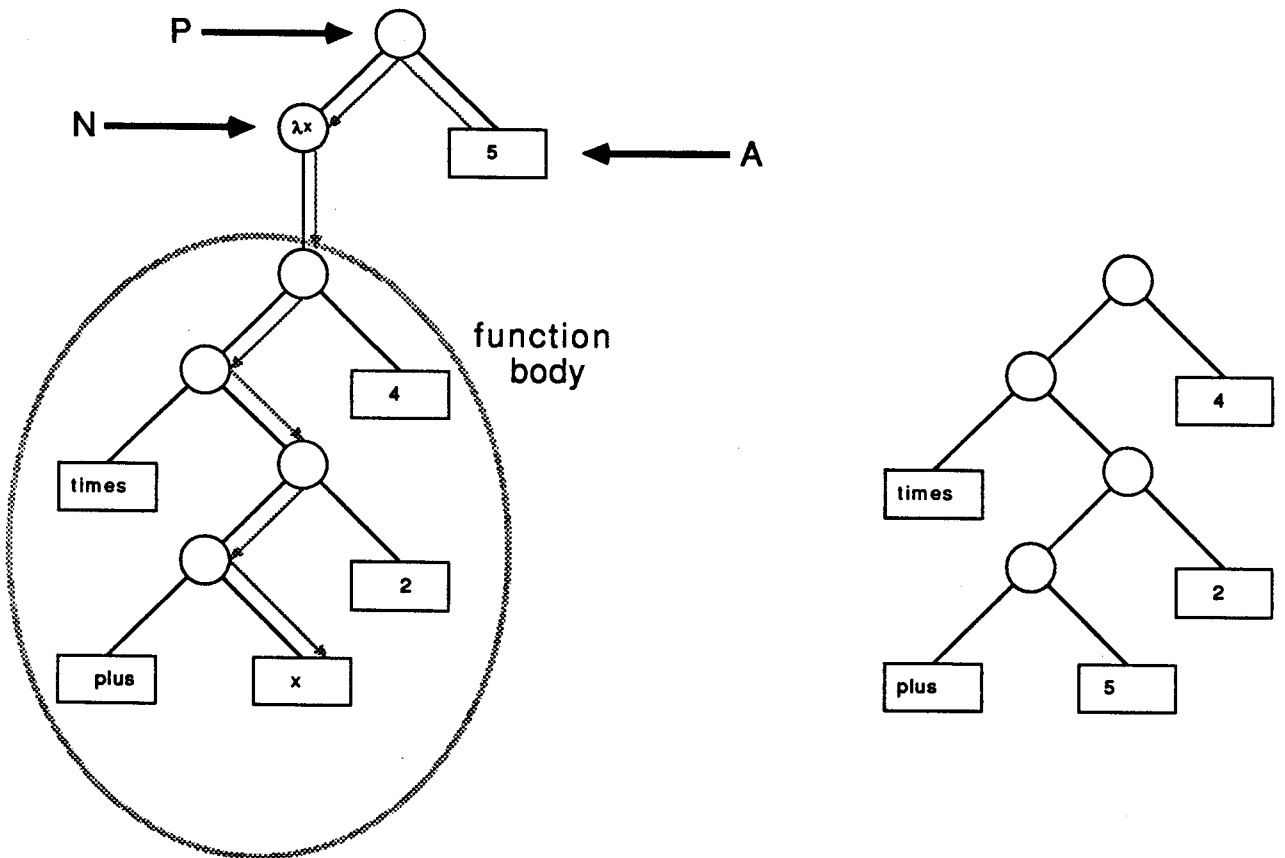


Figure 1.3: The evaluation process.

```

VAR
  scribble : tree;
BEGIN
  WHILE NOT is_leaf(current) DO
    descent(current, current^.left_son);
    scribble := father^.right_son;
    father := apply(current^.value, scribble);
    delete(current);
    delete(scribble)
  END;

BEGIN
  IF NOT is_leaf(root)
  THEN descent(root, root^.left_son);
  execute := tree^.value
END;

```

End of algorithm 1

Here, we do not have to know intricate things about free variables and substitutions; the changes to the graph are entirely local. If it would be possible to transform each λ -term into an expression without variables, this simple evaluation scheme might always be used.

At first sight, one might deem it impossible: doing without variables! Yet, it can be done by carefully choosing the constants. This is the subject of the next chapters. An algorithm is presented and we try to improve it, in order to minimize the length of the resulting λ -term.

However, before we are ready to embark upon a discussion of variable-eliminating algorithms, some additional nomenclature is indispensable. Some special kinds of λ -terms are introduced first:

- combinators or closed terms
- combinatory terms or applicative forms
- proper combinators or variable applicative forms
- constant applicative forms

Regarding an expression as a computer-program, we demand that all variables are declared. A λ -term E with this property ($\text{freevars}(E) = \emptyset$) is called a *combinator* or a *closed term*. We shall use ‘combinator’ only for predefined, closed terms (constants).

A special kind of combinator is the *applicative form* (*combinatory term*). Here all variables are declared at the outermost level. An applicative form looks like this:

$$(\lambda x_1. (\lambda x_2. (\lambda x_3. (\dots (\lambda x_n. A) \dots)))) \quad (1.12)$$

where A does not contain λ -introductions. If A does not contain any constants, we have a *variable applicative form*. If the applicative form has no parameters ($n = 0$) and thus no variables occur in A , it is called a *constant applicative form*. Such an expression consists of nothing but constants combined by application. We want to translate any closed λ -term into an equivalent constant applicative form, a *caf* for short. The following definition summarizes the terminology on special kinds of λ -terms.

Definition 5 (*special kinds of λ -terms*)

Let E be a λ -term. E is called a

- *combinator* if $\text{freevars}(E) = \emptyset$.
- *applicative form* if $E = (\lambda x_1. (\lambda x_2. (\dots (\lambda x_n. A) \dots)))$, where A does not contain λ -introductions and $\text{freevars}(E) = \emptyset$.
 - *constant applicative form (caf)* if E contains constants only ($n = 0$).
 - *variable applicative form (vaf)* if A contains variables only.

Up to now, we were quite vague about the ‘size’ of a λ -expression. In the comparison of methods to remove variables, a precise complexity measure is needed. Two definitions of size obviously present themselves.

Definition 6 (*size of a λ -term*)

1. The *tree size* of a λ -term is the number of nodes in its tree representation. (*tree measure*)
2. The *flat size* of a λ -term is the number of symbols occurring in it, not counting dots and parentheses. (*flat measure*)

In this paper, it is assumed that all identifiers have *unit length* (1). If n is the number of λ -introductions and a is the number of applications in E ,

(2) says that the size of E is $2n + a + 1$. In a caf, $n = 0$, and hence (2) tells that the size is $a + 1$. Applying the first definition to a caf yields $2a + 1$. To check this, observe that the tree representation of a caf is a binary tree. a is exactly the number of internal nodes. In a binary tree, the number of leaves is the number of internal nodes plus one. Hence the total size amounts to $(2a + 1)$.

The circumstance that a caf may be represented as a binary tree is important. It enables us to compute the number of cafs of given length, assuming that we know how many constants may be used. This property is shared by the bodies of applicative terms. The number may be immediately computed from the following theorem.

Theorem 1 (*number of binary trees*)

The number of binary trees with p leaves is

$$\xi_p = \frac{1}{p} \binom{2p-2}{p-1}$$

the p^{th} Catalan number (Eugène Catalan, 1814–1894).

This statement is proved by generating functions. The interested reader should consult [RND77]. Here we are only interested in the consequences.

Corollary 1.1 (*number of cafs*)

Using the tree measure The number of cafs of tree size m using k constants is

$$k^{(m+1)/2} \cdot \xi_{(m+1)/2}$$

Using the flat measure The number of cafs of flat size n using k constants is

$$k^n \cdot \xi_n$$

Proof: Let m be the total number of tree nodes, and a the number of applications

$$m = 2a + 1 \Leftrightarrow \frac{m-1}{2} = a \Leftrightarrow \frac{m+1}{2} = a + 1 = \text{number of leaves}$$

Theorem 1 yields the desired result, since we have k choices to fill each leaf.
□

Corollary 1.2 (*number of applicative forms*)

Using the tree measure The number of applicative forms of tree size s using k constants and v variables is

$$(k + v)^{(s-v+1)/2} \cdot \xi_{(s-v+1)/2}$$

Using the flat measure The number of applicative forms of flat size s using k constants and v variables is

$$(k + v)^{s-2v} \cdot \xi_{s-2v}$$

Proof: The contribution of the λ -introductions is subtracted from the size. We may apply the same method as in the proof of the previous corollary. □

In the sequel, we will use only the flat size measure. Anyway, conversion of results to the tree measure is usually easy enough.

Chapter 2

Elimination of variables

In this chapter, a straightforward algorithm for eliminating variables is presented. It has a most serious disadvantage: The size of the resulting λ -expression depends exponentially on the size of the input.

Consider the λ -expression in figure 2.1. Evaluating this expression, we could send 4 down the left subtree, to all leaves *plus*, x , and 2. Only x is replaced by 4. Note that this another way of describing the substitution process defined in the previous chapter. However, we would like the test whether a leaf is x to be done at compile-time. After all, the human observer is able to identify the path that an argument should take immediately just by looking at the expression tree. We have three functions S , K and I characterized by:

$$\begin{aligned} ((Sf)g)x &= (fx)(gx) \\ (Ky)x &= y \\ Ix &= x \end{aligned} \tag{2.1}$$

Using these functions one could rewrite figure 2.1 to the expression in figure 2.2. You could interpret this as follows:

- S — send argument into subtrees
 - K — block argument propagation:
don't send into subtree
 - I — here is where the argument should be
- λ -expressions are used to define S , K and I formally:

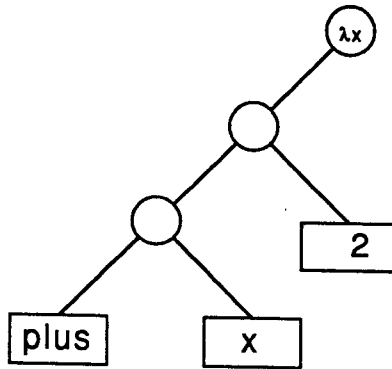


Figure 2.1: Example λ -expression.

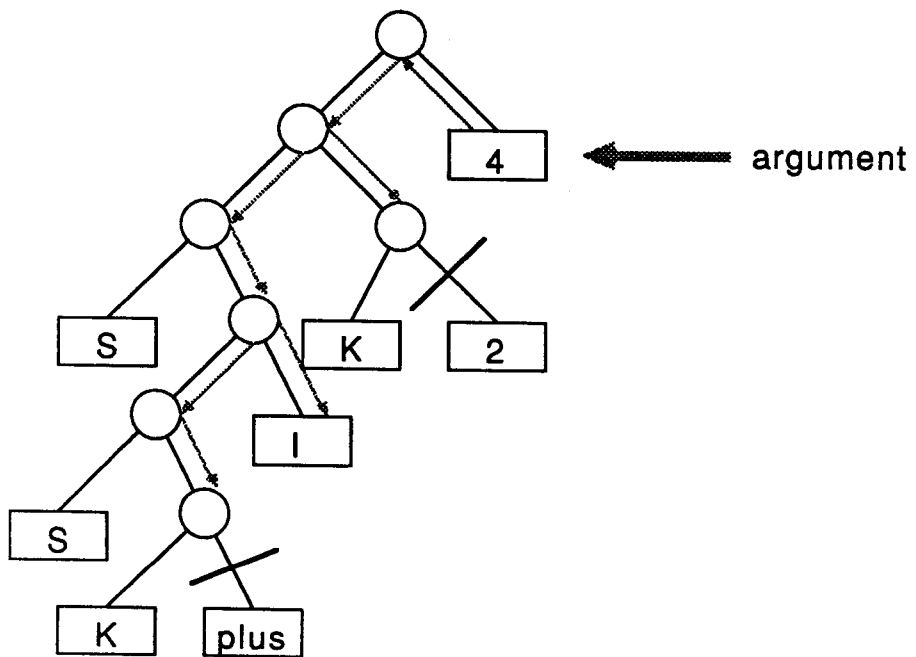


Figure 2.2: Translation of figure 2.1.

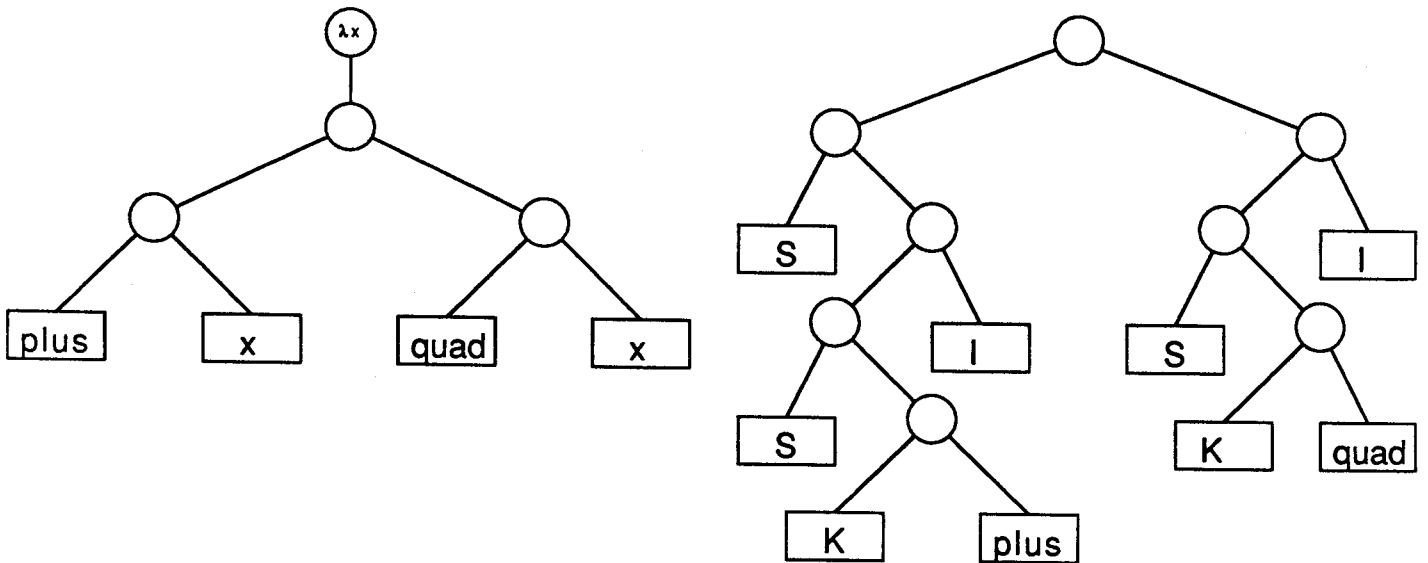


Figure 2.3: Example of Curry's algorithm.

Definition 7 (*S, K and I*)

$$S := (\lambda x.(\lambda y.(\lambda z.((xz)(yz))))))$$

$$K := (\lambda y.(\lambda x.y))$$

$$I := (\lambda x.x)$$

The names of these constant-functions betray their German origin [Sch24].

S = Schönfinkel's Verschmelzungsfunktion

K = Konstanzfunktion

I = Identität

They provide a way of eliminating variables. This is done by expressing the substitution steps explicitly in the code. To further elucidate what is intended, we add some more examples in figures 2.3 and 2.4. One may

```

((S((S(KS))((S((S(KS))((S(KK)) (KS))))))
(S((S(KS))((S((S(KS))((S(KK)) (KS))))))
((S((S(KS))((S(KK)) (KK))))((S(KK)) I))))((S(KK)) (KI))))))
(S((S(KS))((S((S(KS))
(S(KK)) (KS))))
(S((S(KS))((S(KK)) (KK)))) (KI))))
(S(KK)) (KI)))

((S((S(Kc)) I))
(SI)((S((S(KS))
(S((S(KS))((S(KK)) I)))
(S(KK)) (Ky))))((S(KK)) (Kd))))

```

Figure 2.4: Translations of expressions in 1.2.

define this method by the following algorithm:

Algorithm 2 (*Curry's algorithm*)

```

CurryTrans (λx.E) = removevar x (CurryTrans E)
CurryTrans (E1 E2) = ((CurryTrans E1)(CurryTrans E2))
CurryTrans c = c
CurryTrans x = x
CurryTrans comb = comb

removevar x (λy.E) = does not occur
removevar x (E1 E2) = ((S (removevar x E1))(removevar x E2))
removevar x y = if x = y → I
                □ x ≠ y → (K y)
                fi
removevar x c = (K c)
removevar x comb = (K comb)

```

End of algorithm 2

Here, *comb* denotes a combinator *S*, *K* or *I*. It was not taken as a constant for clarity and because the distinction between the predefined combi-

nators and ordinary constants will be needed later on.

From figure 2.4 it may be gathered that the expression size is exploding during translation. We will state exact bounds for this phenomenon.

Lemma 1 (*upperbound on variable-removal*)

Let E be an applicative form with a applications.

$$|(\text{removevar } x \ E)| \leq 3a + 2$$

This bound can be attained.

Proof: Induction on E .

- E atomic.

If $E = x$ then $|\text{removevar } x \ E| = |I| = 1$. If $E \neq x$ then $|\text{removevar } x \ E| = |K \ E| = 2$.

- $E = (A \ B)$, where A contains b applications, and B contains c applications.

Hypothesis $|\text{removevar } x \ A| \leq 3b + 2$ and $|\text{removevar } x \ B| \leq 3c + 2$.

$$\begin{aligned} |\text{removevar } x \ E| &= \\ |\text{removevar } x \ (A \ B)| &= \\ |((S (\text{removevar } x \ A)) (\text{removevar } x \ B))| &= \\ 1 + |\text{removevar } x \ A| + |\text{removevar } x \ B| &\leq \\ 1 + (3b + 2) + (3c + 2) &= \\ 3(b + c + 1) + 2 &= \\ 3a + 2 & \end{aligned}$$

□

Theorem 2 (*complexity of the Curry translation*)

Let $D = (\lambda x_1.(\lambda x_2.(\dots(\lambda x_k.E)\dots)))$ be an applicative form with a applications.

$$|(\text{CurryTrans } D)| \leq (2 \cdot 3^k \cdot a + 3^k + 1)/2$$

This bound can be attained.

Proof: By induction on k . For brevity, $CT = \text{CurryTrans}$.

$k = 1$ By lemma 1:

$$2|CTD| = 2(3a + 2) = 2 \cdot 3^1 \cdot a + 3^1 + 1$$

$k > 1$ **Hypothesis**

$$2|CT(\lambda x_2.(\dots(\lambda x_k.E)\dots))| = 2 \cdot 3^{k-1} \cdot a + 3^{k-1} + 1$$

$$\begin{aligned} 2|CTD| &= \text{(def. } D) \\ 2|CT(\lambda x_1.(CT(\lambda x_2.(\dots(\lambda x_k.E)\dots))))| &= \text{(lm. 1)} \\ 2[3(|(CT(\lambda x_1.(\lambda x_2.(\dots(\lambda x_k.E)\dots))))| - 1) + 2] &= \text{(i.h.)} \\ 6 \left((2 \cdot 3^{k-1} \cdot a + 3^{k-1} + 1) / 2 - 1 \right) + 4 &= \\ 2 \cdot 3^k \cdot a + 3^k + 1 & \end{aligned}$$

□

Chapter 3

Turner's algorithm

Among the variable-eliminating algorithms David Turner's is undoubtedly the best-known. In [Tur79b] he introduced the concept as an implementation technique for functional languages. A more refined algorithm was presented in [Tur79a].

Both articles lack a solid basis for complexity analysis. The algorithm is defined informally and new combinators are introduced in an ad hoc manner. For this reason, we closely follow the approach taken by J.R. Kennaway in [Ken84]. He proves that in the worst case the size of the resulting caf is proportional to the square of the input size. In the first part of the analysis, we will confine ourselves to applicative forms.

3.1 The algorithm

Why does Curry's algorithm perform so poorly? Recall that the combinators make the substitution process explicit. They 'direct' arguments down the expression tree. Curry's way of doing this does not seem very sophisticated. The directing goes on to atomic level, even if a subtree does not contain any occurrence of the variable in question. Hence, we conclude that the main problem lies in the introduction of K . We should replace an expression E without free occurrences of x by $(K E)$, blocking superfluous transformations of E .

Another rather crude director is S . In most cases, an argument need not be directed into both branches of an application node. However, with Curry's method S always does exactly that. This would seem to justify the introduction of two new combinators B and C , with B channeling the

argument into the right-hand subtree, C into the tree on the left-hand side.

When we encounter an expression of the form $(E x)$, where x is the variable being eliminated and E does not contain x , we might as well replace this expression by E . The application does not yield any effect and is superfluous.

The new translation of applications is depicted in figure 3.1. This way of translating λ -terms into cafs, attributed to M. Schönfinkel [Sch24] is a definite improvement over Curry's algorithm. The translation of

$$(\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))) \quad (3.1)$$

immediately yields S with Schönfinkel's method. Curry's algorithm in contrast, yields a baffling number of 73 combinators!

One could push the refinements even further by introducing combinators reaching deeper down the tree. Up to now, we were only concerned with the immediate subtrees of an application node. Why not look beyond this border, to also include cases like those depicted in figure 3.2?

Another good question is: Why do we not consider deeper subtrees on the right-hand side? In eliminating more and more variables, expressions tend to develop like this:

$$(((\dots \text{combinators} \dots)E)F) \quad (3.2)$$

If we were able to reach across the first subtrees, as exemplified by the special cases in figure 3.2, we would be able to prevent the algorithm from tunneling through lots of brackets, spewing combinators as it proceeds.

Definition 8 (B, C, S', B', C')

The combinators in Turner's algorithm are S, K, I , and those defined by the following λ -terms.

$$\begin{aligned} B &:= (\lambda x.(\lambda y.(\lambda z.(x(yz)))))) \\ C &:= (\lambda x.(\lambda y.(\lambda z.((xz)y)))) \\ S' &:= (\lambda k.(\lambda x.(\lambda y.(\lambda z.((k(xz))(yz)))))) \\ B' &:= (\lambda k.(\lambda x.(\lambda y.(\lambda z.((kx)(yz)))))) \\ C' &:= (\lambda k.(\lambda x.(\lambda y.(\lambda z.((k(xz))y)))))) \end{aligned}$$

By now, we have arrived at Turner's algorithm in its proper form. It is summarized below [Ken84].

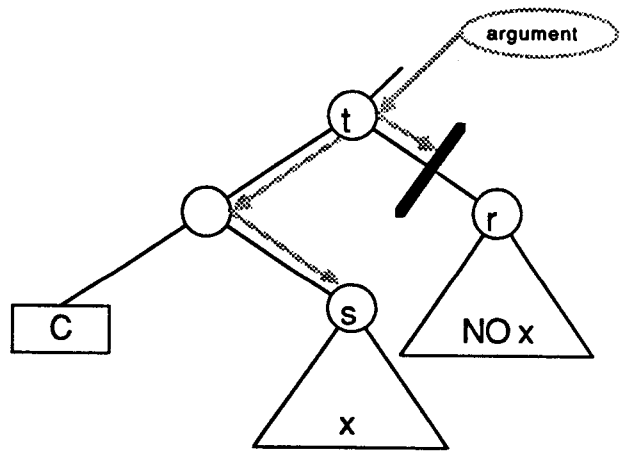
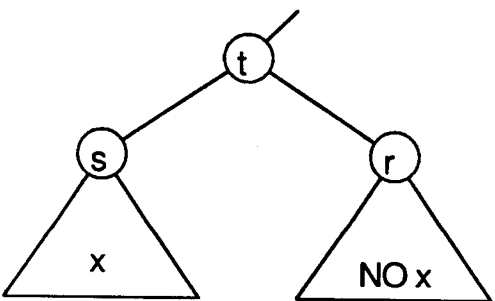
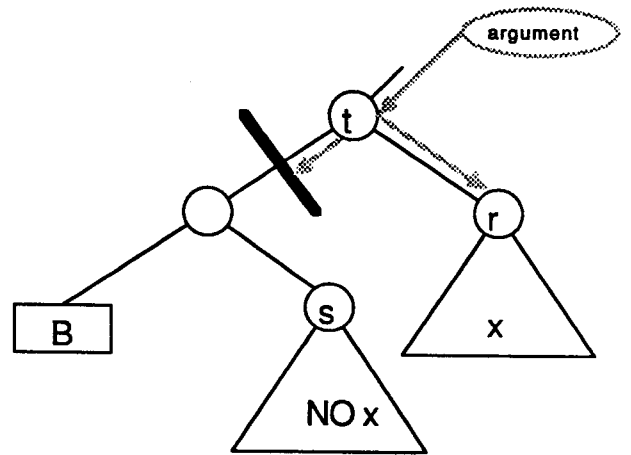
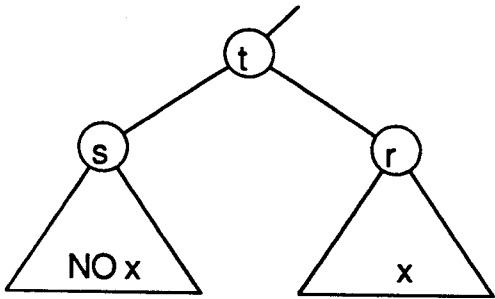
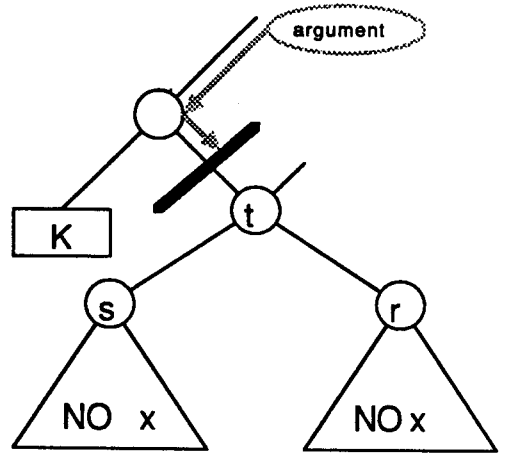
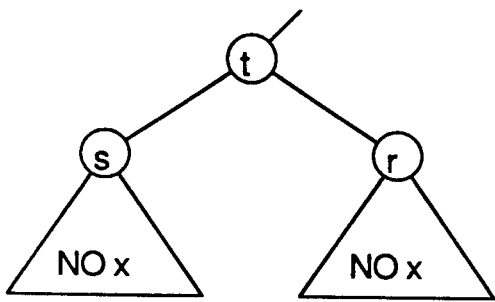


Figure 3.1: Schönfinkel's translation of application nodes.

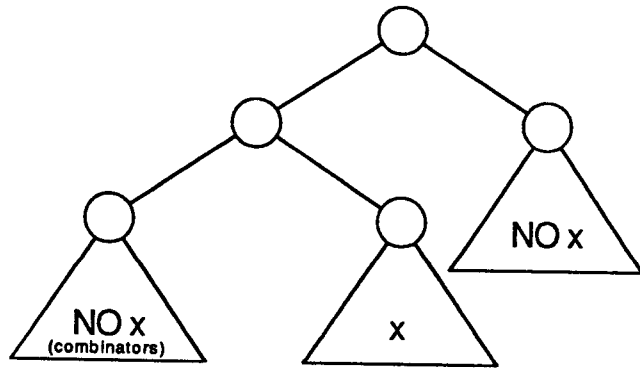
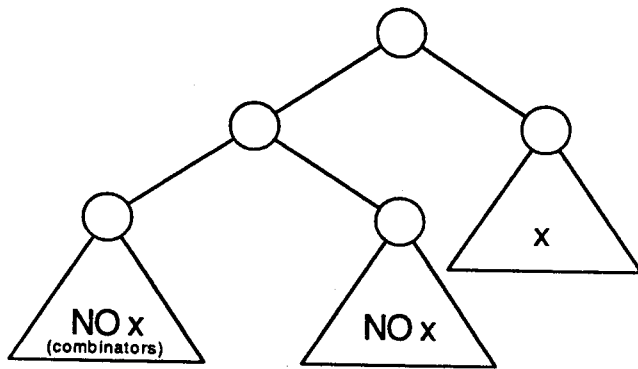
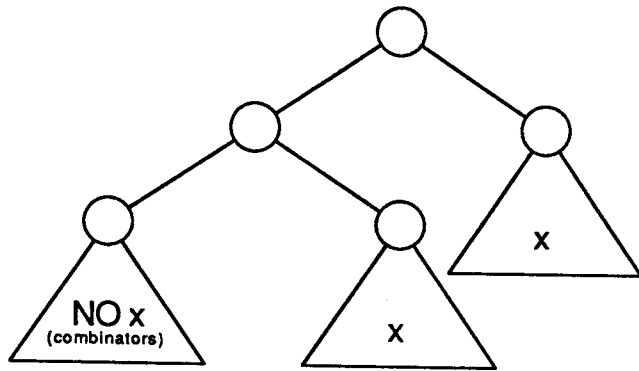


Figure 3.2: Reaching deeper down the tree.

Algorithm 3 (*Turner's algorithm*)

$$\begin{aligned}
 \text{TurnerTrans } (\lambda x.E) &= \text{removevar } x (\text{TurnerTrans } E) \\
 \text{TurnerTrans } (E_1 E_2) &= ((\text{TurnerTrans } E_1)(\text{TurnerTrans } E_2)) \\
 \text{TurnerTrans } c &= c \\
 \text{TurnerTrans } x &= x \\
 \text{TurnerTrans } \text{comb} &= \text{comb} \\
 \\
 \text{removevar } x E &= T x E
 \end{aligned}$$

For any expression E , $(T x E)$ is defined by the first of the following cases which applies. E_x , F_x and G_x stand for expressions in which x occurs at least once, and E , F and G for expressions in which x does not occur.

- (1) (a) $T x x = I$
 (b) $T x E = (K E)$
- (2) When E contains no variables
 - (a) $T x ((E x)F_x) = (S E)(T x F_x)$
 - (b) $T x ((E x)F) = (C E)F$
 - (c) $T x ((E F_x)G_x) = ((S' E)(T x F_x))(T x G_x)$
 - (d) $T x ((E F)G_x) = ((B' E)F)(T x G_x)$
 - (e) $T x ((E F_x)G) = ((C' E)(T x F_x))G$
- (3) (a) $T x (E x) = E$
 (b) $T x (E_x F_x) = (S (T x E_x))(T x F_x)$
 (c) $T x (E F_x) = (B E)(T x F_x)$
 (d) $T x (E_x F) = (C (T x E_x))F$

End of algorithm 3

The algorithm may be readily derived from our discussion, except for the cases 2^a and 2^b. They are examples of compile-time evaluation, like 3^a. They prevent the more expensive choice of a dashed combinator. Another 'irregularity' is the condition in case 2. One would expect "when E contains no occurrence of x " instead of "when E contains no variables". Clearly, the latter statement implies the former. Hence, the algorithm is still correct. The condition stems from the motivation that dashed combinators should reach across the first term *which consists of combinators only*.

As you see, we are faced with lots of special cases. Estimating the resulting expression size might be difficult. The next section is devoted to this task.

3.2 Caf size for applicative forms

In this section, the translation from general applicative forms to constant applicative forms is considered. Recall that an applicative form is:

$$(\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.E)\dots))) \quad (3.3)$$

where E does not contain any λ -introductions. A constant applicative form is made up of constants combined by application. In a caf, λ -introductions do not occur at all.

3.2.1 A simplified algorithm: T'

As we just pointed out, the present algorithm is abundant in special cases, and one may wonder whether analysis is feasible. If one tries to construct a worst case for the abstraction of a single variable, it turns out that the next cycle is not so bad at all. It seems that the problem is caused by two properties of the algorithm.

1. Several combinators may be introduced by more than one rule.
2. The tree representation does not reflect the ‘look-ahead’ characteristics of dashed combinators.

We follow [Ken84] in solving these problems. An outline of the analysis is to be found in figure 3.3

To circumvent the first problem, we define a modified version T' of T , which is easier to analyse. One obtains T' from T by dropping the irregular cases 2^a , 2^b and 3^a . In addition to that we strengthen the condition in 2 to: “when E does not contain any variables *or constants*”. This makes the motivation for dashed combinators explicit in their introduction: They should only reach across a combinator tree. To accommodate the use of T' and other versions of T that will be discussed in the sequel, we define the function *Trans*.

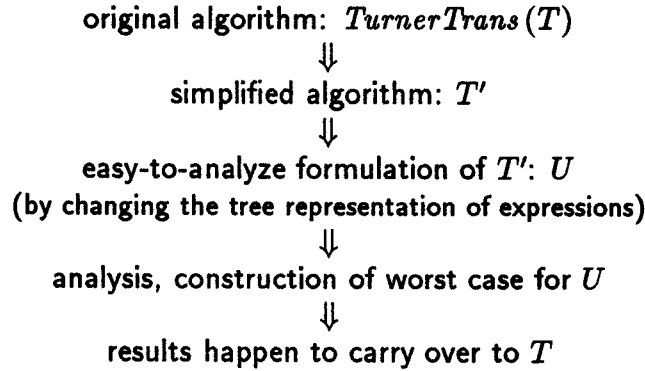


Figure 3.3: An outline of the complexity analysis.

Algorithm 4 (*general variable-eliminating algorithm*)

$$\begin{array}{lcl}
\textit{Trans RVfunc} & (\lambda x.E) & = \textit{RVfunc } x (\textit{Trans RVfunc } E) \\
\textit{Trans RVfunc} & (E_1 E_2) & = ((\textit{Trans RVfunc } E_1)(\textit{Trans RVfunc } E_2)) \\
\textit{Trans RVfunc} & c & = c \\
\textit{Trans RVfunc} & x & = x \\
\textit{Trans RVfunc} & \textit{comb} & = \textit{comb}
\end{array}$$

End of algorithm 4

Note that $\textit{TurnerTrans } E = \textit{Trans } T E$. Since this notation is still a little cumbersome, we will write $\textit{RVfunc } E$ meaning:

$$\textit{RVfunc } E \equiv \textit{Trans RVfunc} (\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.E)\dots))) \quad (3.4)$$

Where x_1, \dots, x_n are the parameters of the function with body E .

3.2.2 Changing the tree representation

Recall that Turner's set of combinators provides some sort of look-ahead capability in the syntax tree. This renders the use of binary tree representations awkward. We would like to transform trees as depicted in figure 3.4. In such a way all combinators have only a *local* directive function in the tree.

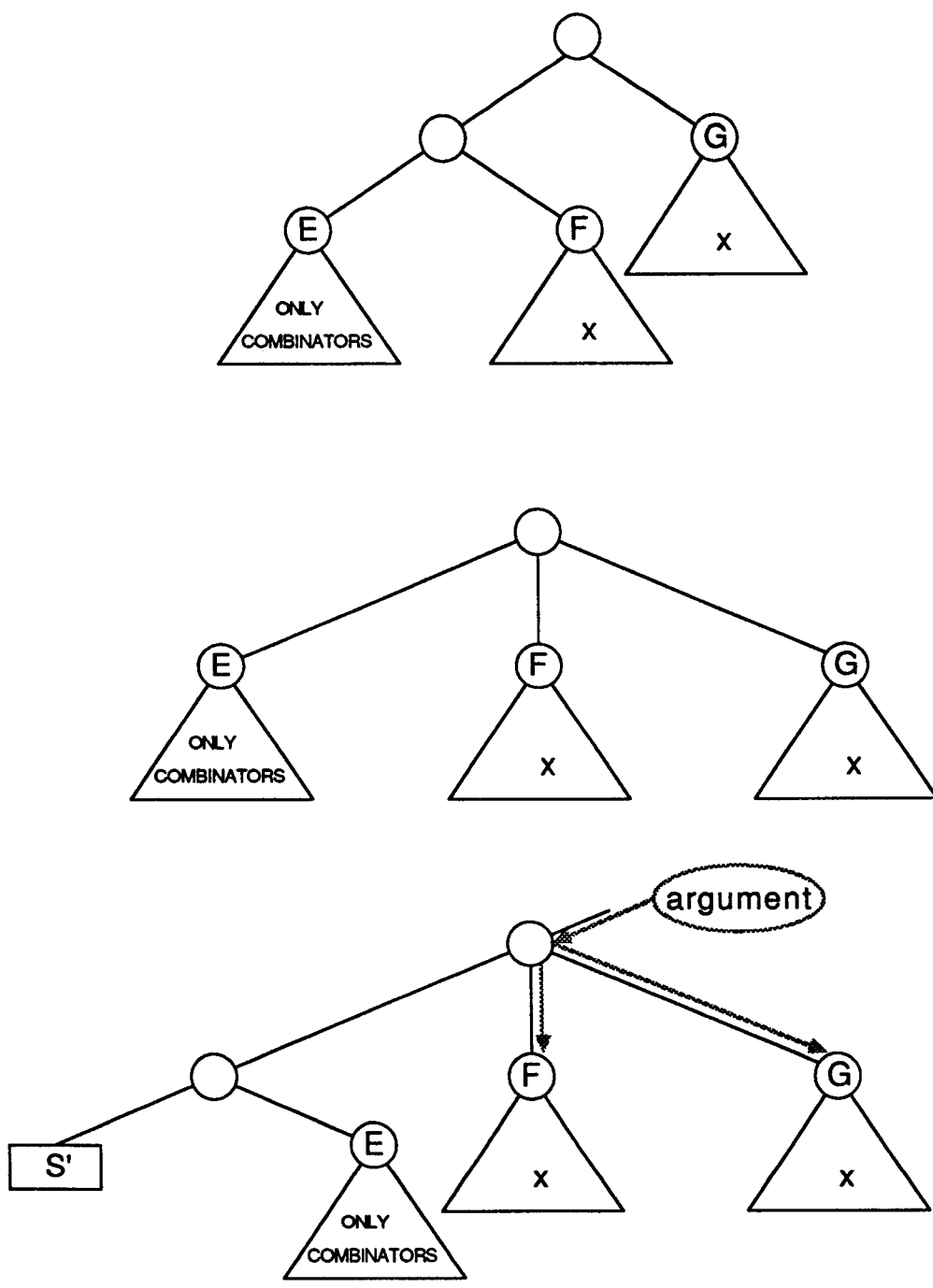


Figure 3.4: A more natural way of representing dashed combinators in a tree.

Hence, we want the bodies of applicative forms to obey to the following grammar :

$$\begin{aligned}
 E & ::= Z E E \mid E E \mid x \mid c \mid comb \quad (\text{expressions}) \\
 Z & ::= Z Z \mid comb \quad (\text{combinators only})
 \end{aligned}$$

($::=$ means: 'has the form', \mid means 'or is of the form') Using this grammar, the tree representation of a not fully parenthesized expression is not uniquely determined. In other words, the grammar is ambiguous. The old representations are mapped to the new ones by the function t .

Algorithm 5 (*translation to new tree representation*)

$$\begin{aligned}
 t \quad (\lambda x.E) & = (\lambda x.(t E)) \\
 t \quad ((E F) G) & = ((t E)(t F)(t G)) \quad \text{if } E \text{ consists} \\
 & \quad \text{of combinators only} \\
 t \quad (E F) & = ((t E)(t F)) \quad \text{if the preceding} \\
 & \quad \text{rule does not apply} \\
 t \quad a & = a \quad \text{for all atoms} \\
 & \quad \text{(variables, constants} \\
 & \quad \text{and combinators)}
 \end{aligned}$$

End of algorithm 5

The following algorithm, U , removes variables from the new representation trees. We follow the same conventions as in the definition of T .

Algorithm 6 (*easy-to-analyze simplification of Turner's algorithm*)

$$\begin{array}{ll}
 (1) \quad (a) \quad U \ x \ x & = \ I \\
 & (b) \quad U \ x \ E & = \ (K \ E) \\
 (2) \quad (a) \quad U \ x \ (E \ F_x \ G_x) & = \ (S' \ E)(U \ x \ F_x)(U \ x \ G_x) \\
 & (b) \quad U \ x \ (E \ F \ G_x) & = \ (B' \ E)F(U \ x \ G_x) \\
 & (c) \quad U \ x \ (E \ F_x \ G) & = \ (C' \ E)(U \ x \ F_x)G \\
 (3) \quad (a) \quad U \ x \ (E_x \ F_x) & = \ S(U \ x \ E_x)(U \ x \ F_x) \\
 & (b) \quad U \ x \ (E \ F_x) & = \ B \ E(U \ x \ F_x) \\
 & (c) \quad U \ x \ (E_x \ F) & = \ C(U \ x \ E_x)F
 \end{array}$$

End of algorithm 6

Obviously U is a restatement of T' in the new representation. This fact is formally expressed as:

Lemma 2 (*algorithms on old and new tree representation*)

$$\begin{array}{l}
 \mathbf{a)} \quad t(T' \ x \ E) = U \ x \ (t \ E) \\
 \mathbf{b)} \quad t(T' \ E) = U \ (t \ E)
 \end{array}$$

Proof:

- a) By inspecting the eight expression templates.
- b) From a, by induction on the number of parameters of the applicative form. $(U \ E)$ is defined analogously to $(T \ E)$.

□

Let us return to the central idea in Turner's algorithm. Arguments are channeled down the tree by combinators. The only relevant paths with respect to x are those from the body-root to occurrences of x in the leaves. Together these paths form *the spanning tree of x* . See figure 3.5. These spanning trees exhibit two useful properties if we submit them to U .

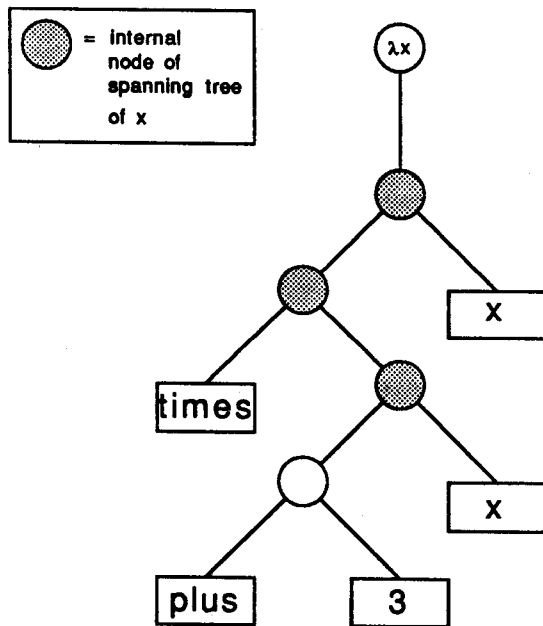


Figure 3.5: Example spanning tree.

1. No spanning tree is altered in the translation process.
2. The number of combinators introduced in the process of eliminating variable x is exactly the number of internal nodes in the spanning tree (figure 3.6).

Here is the starting point for a sound complexity analysis. We can predict the size of a caf coming from U .

Lemma 3 (*caf size and spanning tree sizes*)

Let E be the body of the applicative form
 $(\lambda x_1.(\lambda x_2.(\dots(\lambda x_k.E)\dots)))$

$$|(U E)| - |E| = \sum_{i=1}^k is(x_i, E)$$

where $is(y, E)$ denotes the number of internal nodes in the spanning tree of y in E .

3.2.3 Construction of a worst case

We try to construct a special case, maximising the sum in lemma 3. To this end, two lemmas on tree shape and output size are proved. First, we ask which tree from figure 3.7 will produce the longest code. Intuitively, the spanning trees of the one on the right-hand side seem larger, and we expect this tree will yield the longer code.

Lemma 4 (*tree representation and caf size*)

$$|U (E F G)| \leq |U ((E F)G)|$$

Proof: Call the expression on the left hand side L and the other R . Consider any parameter x of $(\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.L)\dots)))$ and $(\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.R)\dots)))$.

From figure 3.8 and lemma 3, we may conclude that the lemma is correct.

□

This lemma forces us to conclude that the worst possible input is a binary tree. Can we say anything on its shape except that it should be binary? It should be as unbalanced as possible, listing to the right. (Figure 3.9.) In such a tree, each leaf has to be considered separately. One cannot look ahead

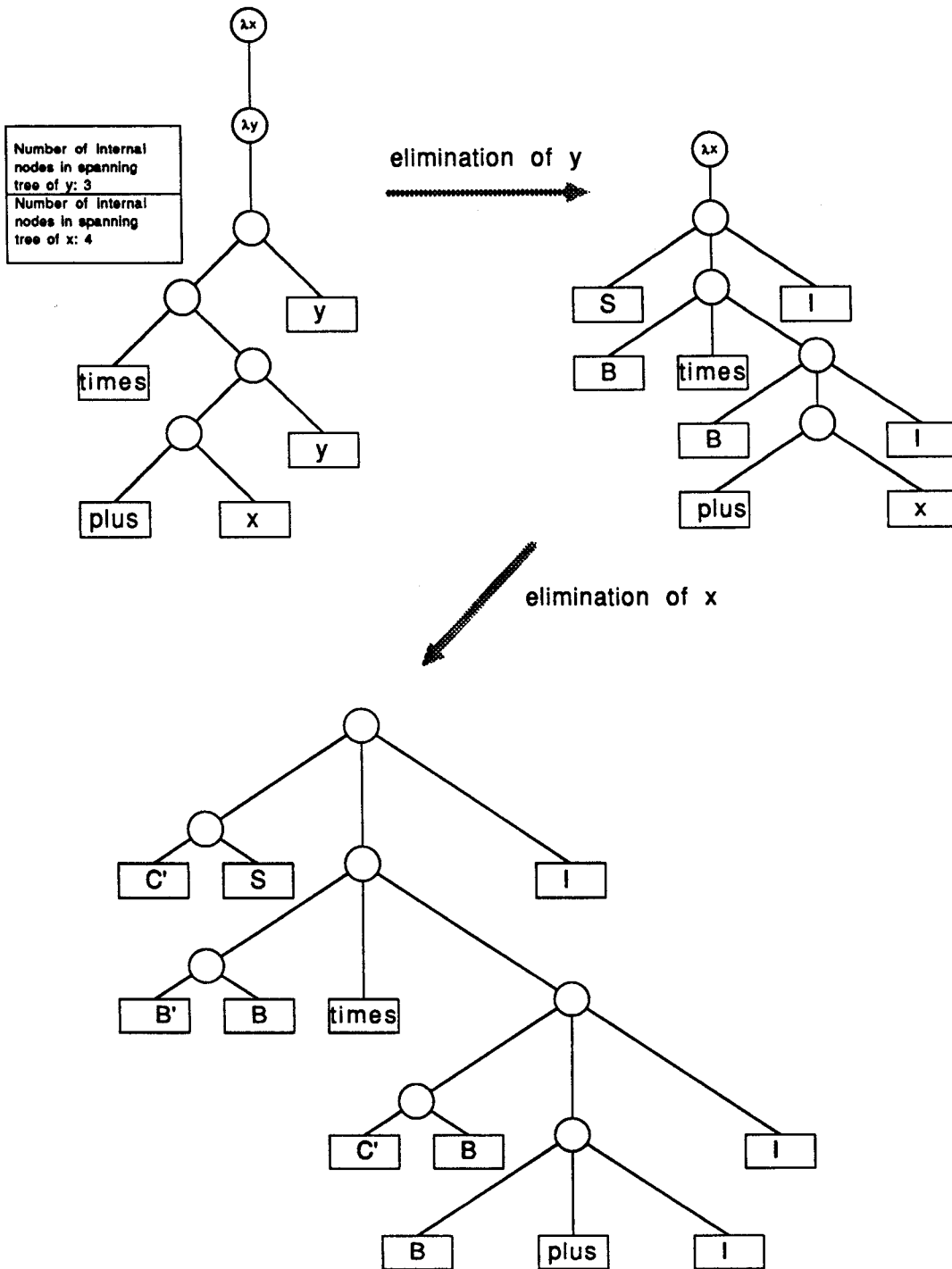


Figure 3.6: Code size and spanning trees.

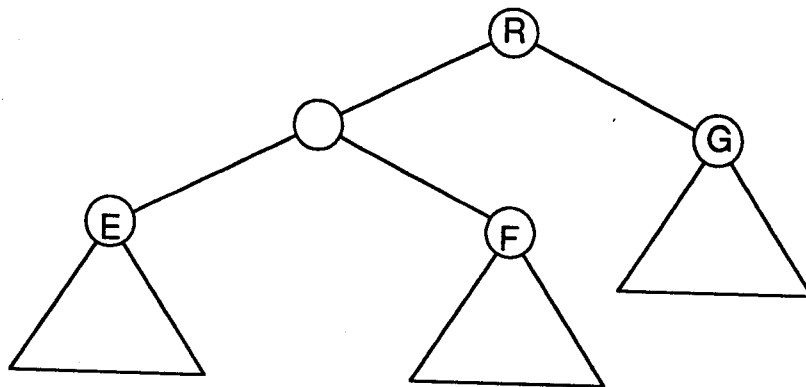
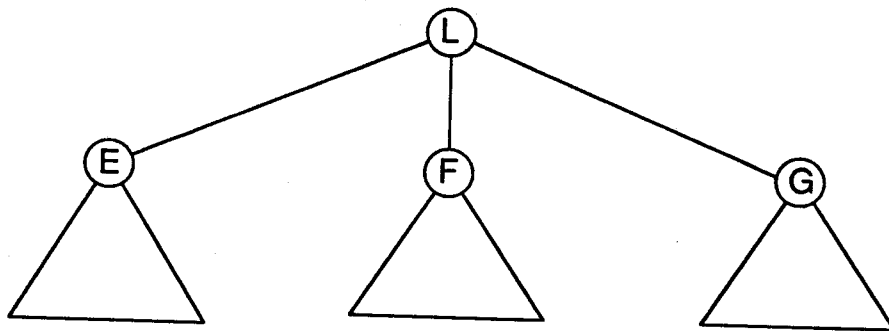


Figure 3.7: Which tree produces the largest code?

x occurs in	$is(x, R) - is(x, L)$
-	0
E	1
F	1
G	0
E, F	1
E, G	1
F, G	1
E, F, G	1

Figure 3.8: Differences in spanning trees from figure 3.7.

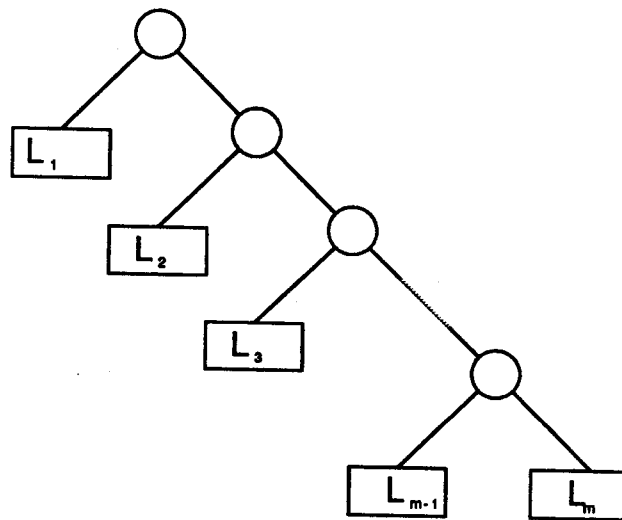


Figure 3.9: A worst case input.

across a number of irrelevant ones in a single step. In the next lemma, our intuitions are fed to the mathematical engine. The trees in figure 3.10 are compared with respect to code size.

Lemma 5 (*expression form and caf size*)

Let

$$E_1 = ((F_1 F_2)(F_3 F_4))$$

$$E_2 = (F_1(F_2(F_3 F_4)))$$

$$E_3 = (F_3(F_4(F_1 F_2)))$$

(See figure 3.10) Then $|UE_2| \geq |UE_1|$ or $|UE_3| \geq |UE_1|$.

Proof: As in the previous lemma, we construct a table for occurrences of parameter x . It is shown in figure 3.11.

Define: $v_i =$ variables occurring in F_i ($i = 1, 2, 3, 4$) This enables us to compress the results from the table into:

$$|UE_2| - |UE_1| = |(v_3 \cup v_4) - (v_1 \cup v_2)| - |v_1 - (v_2 \cup v_3 \cup v_4)|$$

$$|UE_3| - |UE_1| = |(v_1 \cup v_2) - (v_3 \cup v_4)| - |v_3 - (v_4 \cup v_1 \cup v_2)|$$

If the sum of the right hand sides is non-negative, we have established our lemma.

$$v_3 - (v_4 \cup v_1 \cup v_2) \subseteq (v_3 \cup v_4) - (v_1 \cup v_2)$$

$$v_1 - (v_2 \cup v_3 \cup v_4) \subseteq (v_1 \cup v_2) - (v_3 \cup v_4)$$

yields this fact. \square

One may conclude that U produces the worst results with an input that looks like figure 3.9. The only remaining question is: How have the constants and variables to be placed in the leaves L_i ? For any x occurring in L_i ($i < n$) $is(x, E) \geq i$. For x in L_n $is(x, E) \geq n - 1$. Hence, the variables should be placed as much to the right as possible.

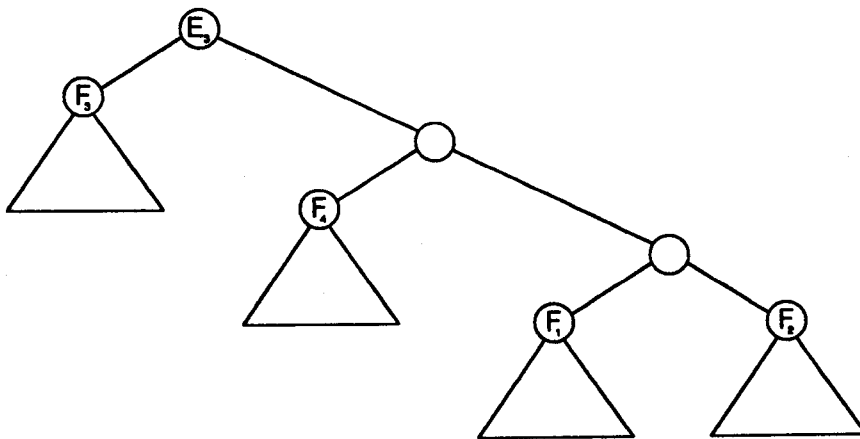
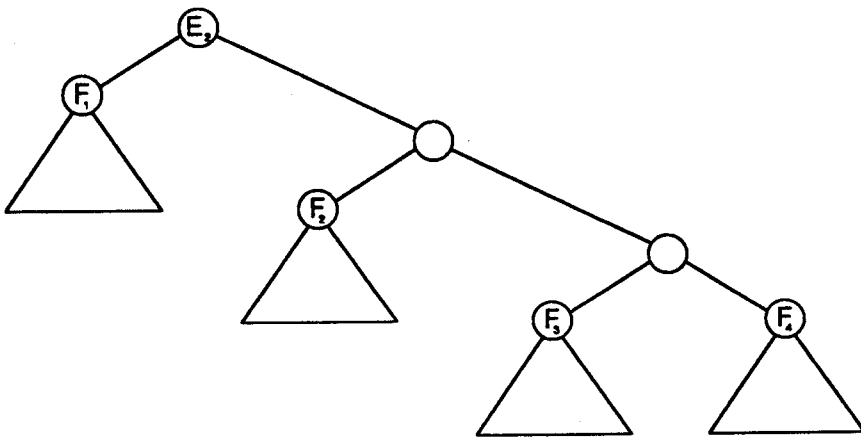
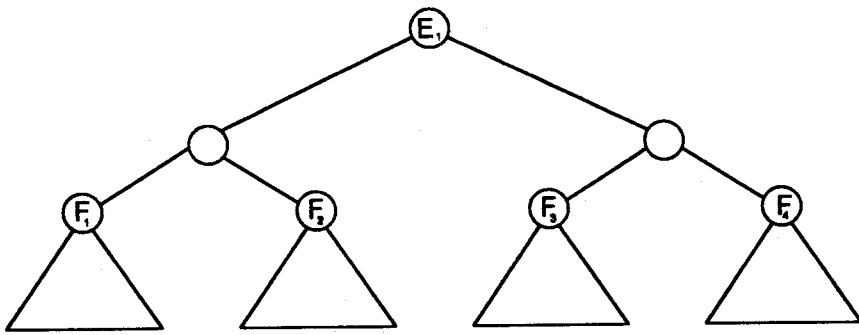


Figure 3.10: Expressions from lemma 5.

F_1	F_2	F_3	F_4	$is(x, E_2) - is(x, E_1)$	$is(x, E_3) - is(x, E_1)$
				0	0
			+	+1	+1
		+		+1	-1
		+	+	+1	0
	+			0	+1
	+		+	0	0
	+	+		0	0
	+	+	+	0	0
+				-1	+1
+			+	0	0
+		+		0	0
+		+	+	0	0
+	+			0	+1
+	+		+	0	0
+	+	+		0	0
+	+	+	+	0	0

Figure 3.11: Differences in the spanning trees from figure 3.10.

Theorem 3 (*caf size for applicative forms using U*)

Let E be the λ -free body of an applicative form with k parameters in the new representation. E has flat size n and k distinct variables occur in it.

if

$$k = 0 \rightarrow |U E| = n$$

$$\square \quad k > 0 \rightarrow |U E| \leq n + k(2n - k + 1)/2 - 1$$

fi

The upperbound can be attained by expressions in the old representation (which do not contain nodes with three descendants).

Proof: From lemmas 4 and 5: $|U E|$ is largest if E is of the form depicted in figure 3.9. Lemma 3 yields for such E :

$$|U E| - |E| = \sum_{i=1}^k is(x_i, E)$$

This sum is largest if the variables all occur as deep as possible. Then,

$$\begin{aligned} \sum_{i=1}^k is(x_i, E) &= \\ (n-1) + (n-1) + (n-2) + \dots + (n-k+2) + (n-k+1) &= \\ k(2n-k+1)/2 - 1 & . \end{aligned}$$

One may conclude that

$$|U E| = n + k(2n - k + 1)/2 - 1.$$

Since the worst input E is a binary tree with no combinators, $E = t E$. In other words, E is an expression with an 'old' tree representation. If $k = 0$, nothing needs to be done since the expression already is a caf, and $|U E| = n$. \square

3.2.4 Worst case for the original algorithm (T)

Theorem 4 (*caf size for applicative forms using T*)

Let E be the λ -free body of an applicative form with k parameters. E has flat size n , and k distinct variables occur in it.

$$|T E| \leq n + k(2n - k + 1)/2 - \min(k, 3)$$

This bound can be attained.

Proof: Consider the expressions in figure 3.12. The translation t from old representations to new ones leaves them unaltered:

$$E = t E \text{ and } F = t F$$

Hence (lemma 3 and the fact that t does not alter the expression size)

$$|T' E| = |t(T' E)| = |U(t E)| = |U E|$$

so $|T' E| = |U E|$ and analogously $|T' F| = |U F|$

T' differs only from T in that cases 2^a , 2^b and 3^a are dropped. Note that 2^a and 2^b can never apply for input E or F . We abstract variables x_k, x_{k-1}, \dots, x_1 from E and F in that order. Here we are not sure that F is always the worst case. In removing x_1 , the expression is of the form $((\text{constants } \& \text{ combinators})x_1)$. Thus, case 3^a is applied, and we get 2 combinators less than in the translation by T' . We conclude that

$$|T F| = n + k(2n - k + 1)/2 - 3$$

Input E does not suffer the combinator-loss and here:

$$\begin{aligned} |T E| &= \\ |U E| &= \\ n + (n - k) + (n - k + 1) + \dots + (n - 1) &= \\ n - (k(2n - k + 1))/2 - k & \end{aligned}$$

Since 3^a can only apply during removal of x_1 , the worst output size is the maximum of $|T F|$ and $|T E|$. \square

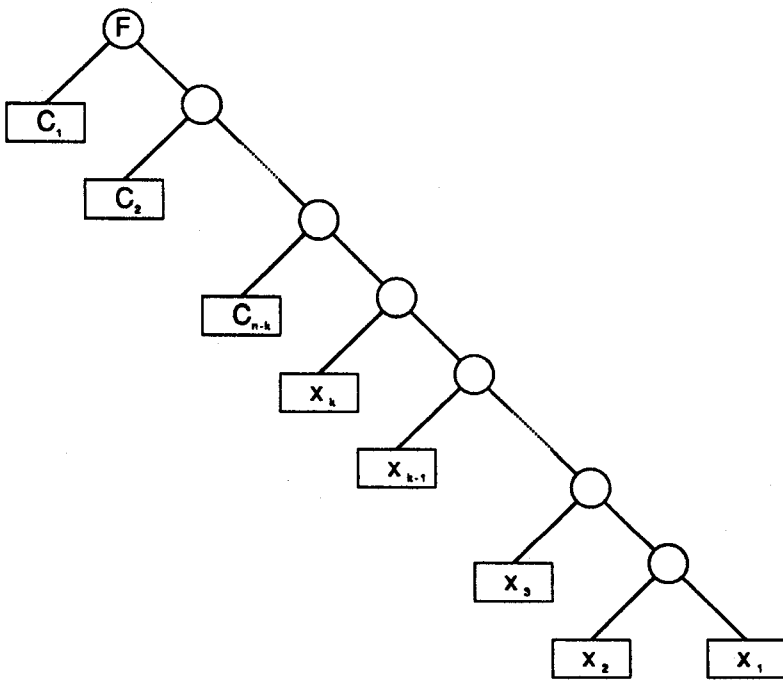
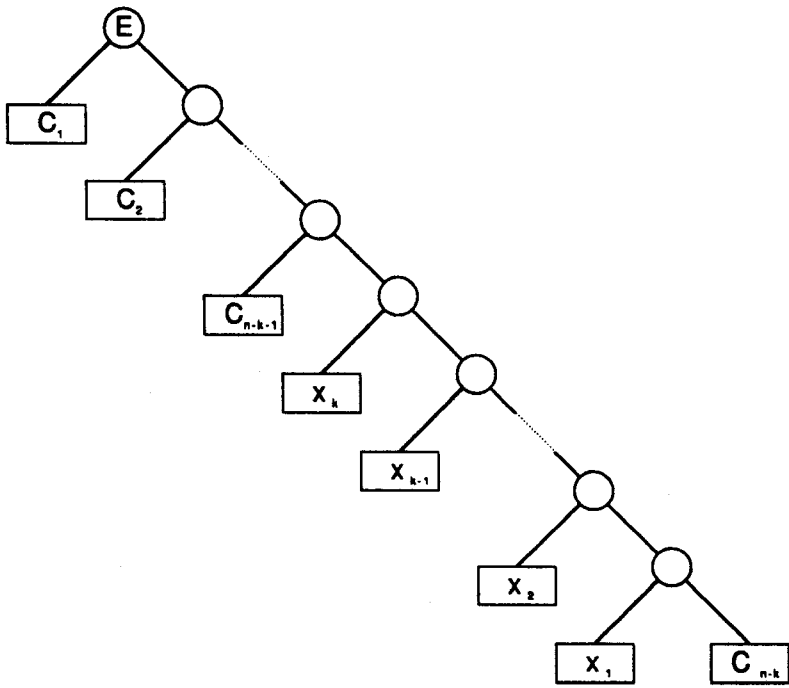


Figure 3.12: Cases in the proof of theorem 4.

3.3 Generalization to λ -terms

Now that we got restricted results on applicative forms, we should look at the performance of Turner's method on full λ -terms.

Theorem 5 (*caf size for arbitrary λ -terms using Turner's algorithm*)

Let E be a λ -term of size n .

$$|TurnerTrans E| \leq \text{if} \quad \begin{array}{ll} n \leq 17 & \rightarrow (-5k^2 + (2n - 5)k + 2n)/2 \\ & \text{where } k \text{ the nearest integer to } (2n - 5)/10 \\ \square \quad n \geq 17 & \rightarrow (-5k^2 + (2n - 3)k + 2n - 6)/2 \\ & \text{where } k \text{ the nearest integer to } (2n - 3)/10 \end{array} \quad \text{fi}$$

This bound can be attained.

Proof: Again, we go in search of the worst case. The following two fact provide the necessary information on how we should construct the input.

- If $x \notin \text{freevars}(E)$ then $TurnerTrans (\lambda x.E) = T x E' = K E'$, $TurnerTrans (c E) = (c E')$, where $E' = (TurnerTrans E)$.
So $|TurnerTrans (\lambda x.E)| = |TurnerTrans (c E)|$.
- If $x \in \text{freevars}(F)$ then $|TurnerTrans (D (\lambda x.F))| = |TurnerTrans ((\lambda x.F) D)| \leq |TurnerTrans (\lambda x.(D F))|$

We conclude that the worst input has to be an applicative form

$$E = (\lambda x_1.(\lambda x_2.(\dots(\lambda x_k.F))))$$

with $\text{freevars}(F) = \{x_1, \dots, x_k\}$. Note that for such E and F (by notational convention) $TurnerTrans E = (TF)$. We may apply the previous theorem to F . The flat size of F is $(n - 2k)$. We get: there exists a λ -term G such

that: $|TurnerTrans G| =$

$$\begin{array}{l}
 \text{if} \\
 \quad k \leq 3 \rightarrow (n - 2k) + \frac{k(2(n-2k)-k+1)}{2} - k \\
 \square \quad k \geq 3 \rightarrow (n - 2k) + \frac{k(2(n-2k)-k+1)}{2} - 3 \\
 \text{fi} = \\
 \text{if} \\
 \quad k \leq 3 \rightarrow n + \frac{k(2n-5k+1)}{2} - 3k \\
 \square \quad k \geq 3 \rightarrow n + \frac{k(2n-5k+1)}{2} - 2k - 3 \\
 \text{fi}
 \end{array}$$

Maximizing over k yields $k = (2n - 5)/10$ (for $n \leq 17$) and $k = (2n - 3)/10$ (for $n \geq 17$). \square

The theorem states that the worst case caf size is approximately behaving like $n^2/10 + 3n/4$. In the next chapters, it will be shown that significant improvements over this bound are possible.

Chapter 4

Optimum code size

In this chapter, we examine to what extent the minimization of combinator code is possible. As the reader might expect, it appears that the problem cannot be solved in general terms. Hence it is mandatory that we define some weaker notion of optimality. In this chapter we follow the approach of Statman [Sta83], based on worst case behaviour. An algorithm that is optimal with respect to his criterion is presented.

4.1 No general solution

Obviously, the translation of λ -terms into combinator code should be optimal. However, what is meant by 'optimal' in this connection? It would be most fortunate if the following optimality principle could be satisfied:

Let \mathcal{A} be an optimal translation from λ -terms into combinator code. Let E be a closed λ -term. Then there is no ccf C equivalent to $\mathcal{A}(E)$ such that $|\mathcal{A}(E)| > |C|$.

Regrettably, this ultimate goal cannot be attained. Let \mathcal{A} be an optimal translation. \mathcal{A} yields optimal code for unary functions with integer range. Clearly, the shortest code possible for such a function applied to an argument is a single integer constant. The existence of such \mathcal{A} is contradicted by the fact that the halting problem is beyond computation. The halting problem can be formulated as follows: Given an arbitrary function f , will it yield a result on the arbitrarily chosen input i ? $\mathcal{A}(f(i))$ always ends its evaluation, so the answer would always be positive.

Of course this result is rather obvious, since ‘equivalence’ was interpreted in its most general sense. One could prohibit certain forms of evaluation to gain decidability. Statman takes another approach: he weakens the optimality notion, while the translation process remains unconstrained.

4.2 Weakened optimality

So it stands to reason to look for the maximal length of combinatory code for a closed λ -term of specified length. For a translation a , we define $wcl_a(n)$ to be the Worst Case Length for terms of length n or less.

Definition 9 (*worst case length*)

Let a be a variable-eliminating algorithm. The *worst case length* of a is defined by

$$wcl_a(n) = \max\{|a(t)| \mid t \text{ a } \lambda\text{-term, } |t| \leq n\}$$

An algorithm is optimal if it minimizes $wcl_a(n)$ for arbitrary n .

Definition 10 (*optimal algorithm*)

Let a be a variable-eliminating algorithm, with $wcl_a(n) = \Omega(f(n))$. a is called *optimal* if for any other algorithm b with $wcl_b(n) = \Omega(g(n))$: $f \leq g$.

In the subsequent sections, it is shown that for any a ,

$$c_0 n \log(n) \leq wcl_a(n) \tag{4.1}$$

We provide an algorithm s by Rick Statman for which:

$$wcl_s(n) \leq c_1 n \log(n) \tag{4.2}$$

This establishes that an optimal algorithm does exist.

4.2.1 Lowerbound on $wcl_a(n)$

Consider the terms of length n or less. Suppose there are at least $l(n)$ that translate into *distinct* cafs. These $l(n)$ cafs have by definition a length less than or equal to $wcl_a(n)$. From corollary 1.1 we can derive an upperbound

$u(wcl_a(n))$ on the number of such cdfs. In this manner Statman arrives at the following inequality:

$$l(n) \leq (\text{number of cdfs of size } \leq wcl_a(n)) \leq u(wcl_a(n)) \quad (4.3)$$

It is not unlikely that l and u are both ever-increasing functions. Assume that we know the inverse function u^{-1} of u . Then one may conclude from 4.3 that:

$$wcl_a(n) \geq u^{-1}(l(n)) \quad (4.4)$$

This is the strategy we intend to follow. We start by deriving the function u . Recall that corollary 1.1 gave us the number t_n of cdfs of size n .

$$t_n = k^n \cdot \xi_n \quad (4.5)$$

k is the number of constants used. Here, we regard combinatorics also as ordinary constants. The number of cdfs of size m or less is equal to

$$\sum_{i=1}^m t_i \quad (4.6)$$

We seek to find an upperbound $u(m)$ on this sum, such that

$$\sum_{i=1}^m t_i \leq u(m) \quad (4.7)$$

Since k^i occurs as a factor in t_i , $u(m)$ should be at least an exponential function c_k^m in which the constant is linearly dependent on k . In lemma 6, it is proved that one may take $u(m) = (4k)^{m+1}$.

Lemma 6 (*upperbound on number of cdfs*)

Let t_i be the number of cdfs of flat size i , using k constants.

$$\sum_{i=1}^n t_i < c_k^{m+1}$$

where $c_k = 4k$.

Proof: Use corollary 1.1. The p^{th} Catalan number is given by: $\xi_p = \binom{2p-2}{p-1}$.

$$2^{(2p-2)} = \binom{2p-2}{0} + \binom{2p-2}{1} + \dots + \binom{2p-2}{p-1} + \dots + \binom{2p-2}{2p-2}$$

Hence:

$$\binom{2p-2}{p-1} < 2^{(2p-2)}$$

$$\sum_{i=1}^m t_i < \sum_{i=1}^m (4k)^i = \frac{(4k)^{m+1} - 4k}{4k - 1} < (4k)^{m+1}$$

□

It is more complicated to find a suitable function l . For all n , we must be sure $l(n)$ terms of size n or less translate into distinct cafs. Consider a term in which no subexpression may be evaluated. No reductions are possible, even no constant-defining ones. Such an expression is called *irreducible*. Assume F and G are irreducible and distinct. Furthermore, assume they translate to the same caf. Then we may conclude that $F = G$, as they yield the same value for any argument. It is a theorem from the λ -calculus that an irreducible form is unique. Hence this cannot happen. Irreducible terms translate to unique cafs.

Lemma 7 (*lowerbound on number of irreducible terms*)

There are at least $g(n)^{g(n)}$ terms of length $\leq n$ that are irreducible, where $g(n) = \lfloor (n-5)/4 \rfloor$.

Proof: Consider

$$E = (\lambda F.(\lambda x.(F(\lambda f_1.(\dots(F(\lambda f_l.(wx))))))))$$

where $l =$ largest integer $\leq \frac{n-5}{4}$, and w is any word of length l made up from f_1, \dots, f_l . There are l^l such words. Note that for all these choices of w , E is irreducible. E has length $5 + 4l$, and hence its length is less than or equal to n . □

Summarising what has been established about u and l in this subsection:

$$l(n) \leq (\text{nr cafs of size } \leq wcl_a(n)) \leq u(wcl_a(n))$$

$$u(m) = c_k^{m+1}$$

$$l(m) = 2^{g(m) \log g(m)}$$

The next theorem adds the finishing touch by taking $(u^{-1}l)$.

Theorem 6 (*lowerbound on $wcl_a(n)$*)

For any variable-eliminating algorithm a using k constants:

$$wcl_a(n) \geq d_k \cdot g(n) \log g(n) - 1$$

where $d_k = 1/(2 \log k)$ and $g(n) = \lfloor (n - 5)/4 \rfloor$.

Proof: From lemma 6, we know that there are $c_k^{wcl_a(n)+1}$ worst output cdfs at most. According to lemma 7 there are at least $2^{g(n) \log g(n)}$ terms of length n or less that will translate into distinct cdfs of size $wcl_a(n)$ or less. Hence:

$$2^{g(n) \log g(n)} \leq c_k^{wcl_a(n)+1} = 2^{(2 \log k) \cdot (wcl_a(n)+1)}$$

and one may deduce that

$$wcl_a(n) \geq (1/(2 \log k))g(n) \log g(n) - 1$$

□

4.2.2 An optimal algorithm

In Turner's algorithm, arguments are sent separately down the tree. Serious problems were to be expected with regard to the removal of large numbers of variables. Statman takes a different approach. Why not pack all arguments into one structure and send them down the tree in one sweeping move? On encountering a λ -introduction, the argument is packed into the structure. If a value x_i is needed, it is retrieved from the structure. This is much like the conventional way to implement local variables in a computer language. The structure is called an *environment*. The idea is exemplified in figure 4.1. An environment is denoted by $\langle x_1, x_2, \dots, x_n \rangle$, where the x_i are the argument-values. To facilitate reasoning, we assume that x_1, \dots, x_n are introduced in that order.

Statman's algorithm is the following: (Derived from a formulation in [Mul85].)

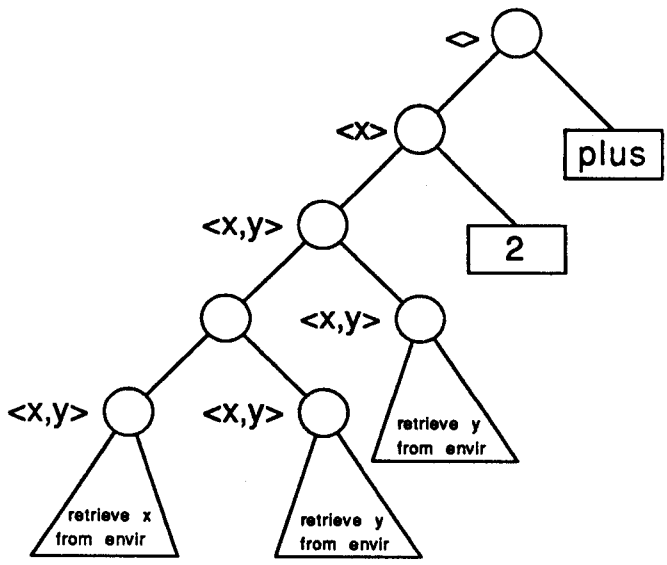
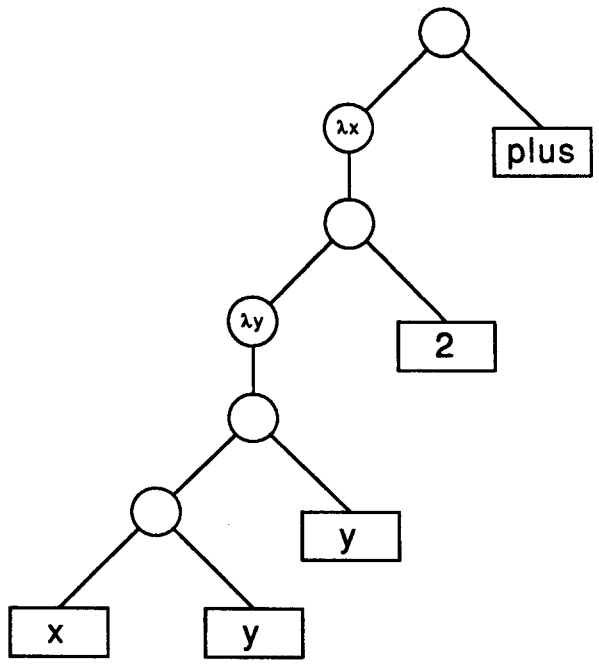


Figure 4.1: The environment solution.

Algorithm 7 (*Statman's algorithm*)

$$\begin{array}{llll}
 \textit{StatmanTrans} & x_i & n & = F_i^n \\
 \textit{StatmanTrans} & c & n & = c \\
 \textit{StatmanTrans} & (E_1 E_2) & n & = (S (\textit{StatmanTrans} E_1 n) \\
 & & & (\textit{StatmanTrans} E_2 n)) \\
 \textit{StatmanTrans} & (\lambda x_1.E) & 0 & = (\textit{StatmanTrans} E 1) \\
 \textit{StatmanTrans} & (\lambda x_{n+1}.E) & n & = (B(B(\textit{StatmanTrans} E (n+1))))A_n \\
 \textit{Statman} & E & & = \textit{StatmanTrans} E 0
 \end{array}$$

F_i^n finds x_i in the environment $\langle x_1, \dots, x_n \rangle$. A_n packs the actual value for x_{n+1} into $\langle x_1, \dots, x_n \rangle$.

End of algorithm 7

A simple example and its evaluation are depicted in figure 4.2. Obviously, the best way to organize $\langle x_1, \dots, x_n \rangle$ is a perfect binary search tree. The implementation of these trees in the λ -calculus is difficult. We provide the definitions, and conjecture without proof that they work properly, which is all that is necessary to comprehend the rest of this section.

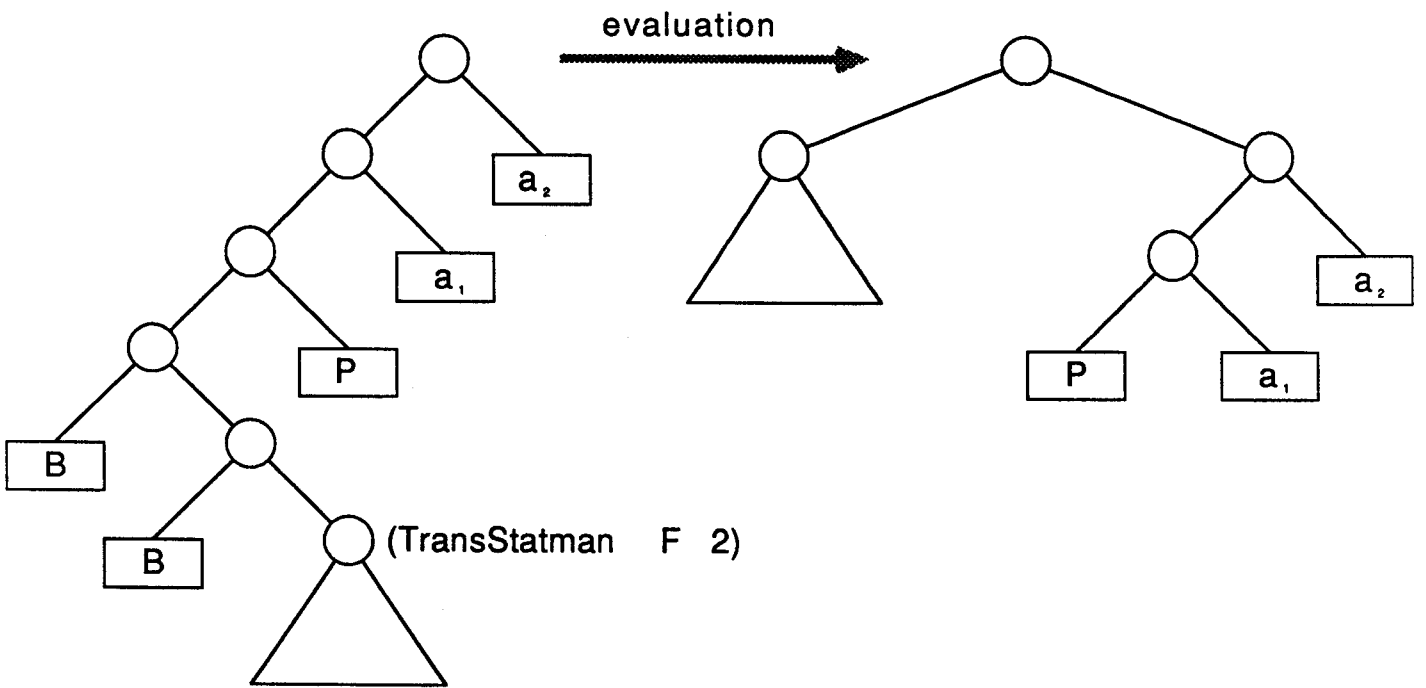


Figure 4.2: Example Statman translation.

Definition 11 (*functions in Statman's algorithm*)

1. R , L and P are predefined combinators the semantics of which are provided by:

$$\begin{aligned} L((P x)y) &= x \\ R((P x)y) &= y \\ (P(L x))(R x) &= x \end{aligned}$$

(For those familiar with functional programming: $L \equiv \text{head}$, $R \equiv \text{tail}$, $P \equiv \text{pair}$, $\cdot \equiv \cdot$.)

- 2.

$$\begin{aligned} A_{<} &= (\lambda x.(\lambda y.(\lambda z.((P((x(Ly))z))(Ry)))))) \\ A_{>} &= (\lambda x.(\lambda y.(\lambda z.((P(Ly))((x(Ry))z)))))) \end{aligned}$$

3. The *packing functions* A_n are defined as follows:

$$\begin{aligned} A_1 &= P \\ A_n &= A_{<}A_{n-2^k} \quad \text{if } 2 \cdot 2^k \leq n < 3 \cdot 2^k \\ A_n &= A_{>}A_{n-2^{k+1}} \quad \text{if } 3 \cdot 2^k \leq n < 4 \cdot 2^k \end{aligned}$$

4. The *retrieval functions* F_i^n are defined as follows:

$$\begin{aligned} F_1^1 &= I \\ F_i^n &= (B L)F_i^{n^*} \quad \text{if } i \leq n^* \\ F_i^n &= (B R)F_i^{n^*} \quad \text{if } i > n^* \end{aligned}$$

where $n^* = \text{smallest integer greater than } n/2 \text{ and } 2^{k+1}i^* = 2^k + i - 1$, for some $k \geq 0$.

As one might guess, complexity analysis is a good deal less difficult than the program itself. We make an estimate for $|A_n|$ and $|F_i^n|$. This enables us to prove that the algorithm is optimal. S , B , I , L , R and P are the primitive combinators in the *cafs*.

Lemma 8 (*size of packing functions*)

$$|A_n| \leq C_A \log n$$

where $C_A = \max(|A_{<}|, |A_{>}|) (= 13)$

Proof: In each step of the recursive definition 13 (= length of $A_{<} = \text{length}$

of $A_{>}$) is added to the total, clearly, at most $(\log n)$ such steps can be made. \square

Lemma 9 (*size of retrieval functions*)

$$|F_i^n| \leq 2 \log n$$

Proof: One easily verifies that in each step 2 combinators are added. There are $(\log n)$ steps at most. \square

Theorem 7 (*caf size for arbitrary λ -terms using Statman's algorithm*)

Let E be a λ -term of length n , with m free variables.

$$|\text{StatmanTrans } E \ m| \leq Cn \log(n + m)$$

where $C = \max(C_A/2, 2)$.

Proof: Abbreviate: $T = \text{StatmanTrans}$. By induction on E .

- E atomic, then either $E = c$ and then $|T \ E \ m| = |c| = 1$ or $E = x_i$ then $(TE) = F_i^m$. By lemma 9 $|T \ E \ m| \leq 2 \log m$. So in the atomic case, the theorem holds.
- $E = (X_1 \ X_2)$.

Hypothesis

$$\begin{aligned} |T \ X_1 \ m| &\leq C \cdot n_0 \log(n_0 + m) \\ |T \ X_2 \ m| &\leq C \cdot n_1 \log(n_1 + m) \\ \text{where } n_0 + n_1 &= n \end{aligned}$$

$$\begin{aligned} |(T \ E \ m)| &= \\ 1 + |(T \ X_1 \ m)| + |(T \ X_2 \ m)| &= \end{aligned}$$

$$\begin{aligned}
1 + C(n_0 \log(n_0 + m) + n_1 \log(n_1 + m)) &< \\
1 + C(n_0 + n_1)(\log(n_0 + m) + \log(n_1 + m)) &\leq \\
Cn(\log(n_0 + m) + \log(n_1 + m)) &\leq \\
Cn \log(n + m) &
\end{aligned}$$

- $E = (\lambda x_{m+1}.F)$.

Hypothesis

$$|(T F (m + 1))| \leq C(n - 2) \log((n - 2) + (m + 1))$$

$(T E m) = (B(B(T F (m + 1))))A_m$, hence:

$$\begin{aligned}
|(T E m)| &= \\
2 + |(T F (m + 1))| + |A_m| &\leq \\
2 + |(T F (m + 1))| + C_A \log m &\leq \\
2 + C(n - 2) \log((n - 2) + (m + 1)) + C_A \log m &\leq \\
2 + Cn \log(n + m - 1) - 2C \log(n + m - 1) + C_A \log m &\leq \\
2 + Cn \log(n + m - 1) - 2C \log(n + m - 1) + 2(C_A/2) \log(n + m - 1) &\leq \\
2 + Cn \log(n + m - 1) &< \\
Cn \log(n + m) &
\end{aligned}$$

□

Theorem 8 (*optimality of Statman's algorithm*)

Let a be Statman's algorithm. Then:

$$(g(n) \log g(n)) / 2 \log k - 1 \leq wcl_a(n) \leq Cn \log n$$

Where

$$\begin{aligned}
k &= \text{number of primitive combinators} \\
C &= \max(C_A/2, 2) \\
C_A &= \max(|A_{<}|, |A_{>}|) \\
g(n) &= \lfloor (n - 5)/4 \rfloor.
\end{aligned}$$

Proof: Immediate from the previous theorem and lemmas. □

Theorem 8 tells us that we cannot expect a spectacular improvement over Statman's worst case. Although all estimates were very rough, the algorithm is only a small factor worse than the best that could possibly be achieved.

Chapter 5

Possible improvements

In chapter 3, it was shown that Turner's algorithm produces code of a size less than or equal to n^2 , n being the length of the input. The performance of Statman's is much better for the worst case. The two bounds are depicted graphically in figure 5.1. Note that we did not prove that Statman's bound can be attained, as we did for Turner's. The average behaviour of his algorithm, however, is very poor. Some results derived from [Mul85] are summarized in figure 5.2. Recall how Turner achieved his major improvement over Curry's algorithm. Essentially, this was done by introducing more refined combinators. In this chapter of our paper the question is raised whether application of Turner's method to Statman's algorithm might lead to further improvements.

Surprisingly, there appears to exist a strong connection between the abstraction of subexpressions and the introduction of new combinators. By *abstraction of subexpressions* the following is meant: Consider an expression

$$(\lambda x.((plus\ 1)x)) \tag{5.1}$$

This is equivalent to the result of evaluating

$$((\lambda y.(\lambda x.(y\ x)))(plus\ 1)) \tag{5.2}$$

Abstraction of subexpressions may be seen as the inverse of evaluating an application.

We will study the effect of rewriting expressions by abstraction on the code size. We discuss whether there is an optimal way of performing the abstractions. Unfortunately, finding the optimal rewriting sequence is an unsolved problem, and probably even undecidable. One has to look for more practical ways of reducing code size by abstraction.

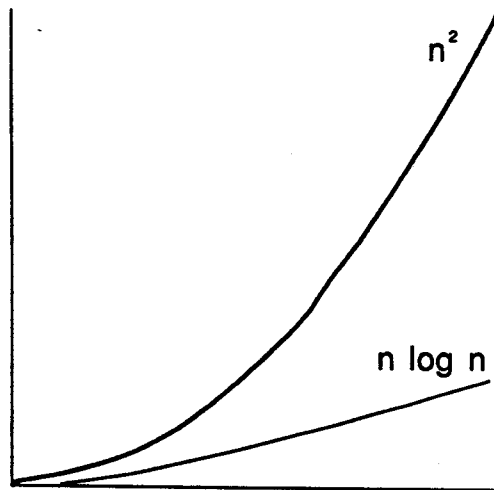


Figure 5.1: Bounds from chapter 3 and 4.

λ -- term E	$ E $	$ TurnerTrans E $	$ Statman E $
$(\lambda x.(\lambda y.(\lambda z.((xz)(yz))))))$	10	4	41
$(\lambda x.(\lambda y.(((xy)x)(\lambda z.((xz)y))))))$	12	9	46
$(\lambda x_1.(\lambda x_2.(\lambda x_3.(\lambda x_4.(\lambda x_5.x_1))))))$	11	7	75

Figure 5.2: Some figures on code-length.

For Statman's algorithm, it is possible to compute whether a single abstraction is profitable. This could serve in developing heuristics to construct a good rewriting sequence.

The effect on Turner's algorithm is more spectacular. In careful 'balancing' of the expression tree, abstractions are used to minimize spanning tree sizes. In this way one arrives at a blend of Statman's and Turner's algorithms, combining good properties of both. This algorithm is only applicable to applicative forms.

Finally, we briefly touch on translation to super-combinators, a variable elimination method devised by John Hughes [Hug82]. It is entirely based on subexpression abstraction.

5.1 Introduction of combinators

In chapter 3, substantial improvements over Curry's algorithm were achieved by introducing B , C , S' , B' and C' . One wonders whether further refinements might lead to better results in the case of Turner's algorithm. And Statman's? Could extra combinators improve its average behaviour?

5.1.1 Adding extra combinators to Turner's algorithm

Extra combinators will not improve Kennaway's worst case bounds by an order of magnitude. The proof of this conjecture is omitted here.

We might want to introduce extra combinators because it is known that some particular case occurs fairly often. How should they be inserted in Turner's algorithm? For ease of reference, the variable-removing part of Turner's algorithm is repeated here:

Algorithm 8 (Turner's algorithm)

- (1) (a) $T x x = I$
- (b) $T x E = (K E)$
- (2) When E contains no variables
 - (a) $T x ((E x)F_x) = (S E)(T x F_x)$
 - (b) $T x ((E x)F) = (C E)F$
 - (c) $T x ((E F_x)G_x) = ((S' E)(T x F_x))(T x G_x)$
 - (d) $T x ((E F)G_x) = ((B' E)F)(T x G_x)$
 - (e) $T x ((E F_x)G) = ((C' E)(T x F_x))G$
- (3) (a) $T x (E x) = E$
- (b) $T x (E_x F_x) = (S (T x E_x))(T x F_x)$
- (c) $T x (E F_x) = (B E)(T x F_x)$
- (d) $T x (E_x F) = (C (T x E_x))F$

End of algorithm 8

Clearly far reaching combinators should be given preference over others. Generally speaking, this is the case in T . Case 1 introduces combinators that deal with as much of the expression tree as possible, and thus contain no recursive application of T . Those in case 2 reach one level deeper than those in case 3. The irregular 'compile-time evaluations' 2^a , 2^b and 3^a do not fit into this scheme. We neglect them here. Define the 'range' of a combinator to be the length of the template in its introduction rule. Then an extended Turner-algorithm could look like this:

Algorithm 9 (*Turner's algorithm with extra combinators*)

- (1) (a) $T x x = I$
(b) $T x E = (K E)$
- (2)
- ...
- ...
- Special combinators in order of descending range
(But without 'irregularities' like $2^{a,b}$)
- ...
- ...
- (3) (a) $T x (E x) = E$
(b) $T x (E_x F_x) = (S (T x E_x))(T x F_x)$
(c) $T x (E F_x) = (B E)(T x F_x)$
(d) $T x (E_x F) = (C (T x E))F$

End of algorithm 9

5.1.2 Adding combinators to Statman's algorithm

Statman's algorithm uses a terse combinator set to build cafs. Yet even for this small number, the algorithm is not able to take full advantage of them. S itself translates into a constant applicative form of size 42. Two possible optimizations may be suggested.

- Splitting the environment. We could send arguments only to where they are used. This resembles Turner's way of dealing with this problem.
- Optimization of the environment structure. How could one do better than Statman's intricate tree?

Strictly speaking, one does not need new combinators for these optimizations. What is certainly required however, are new *rules* for introducing combinators.

Let us first take a closer look at the splitting of an environment. Just as in Turner's algorithm, we want to send arguments only to the corresponding

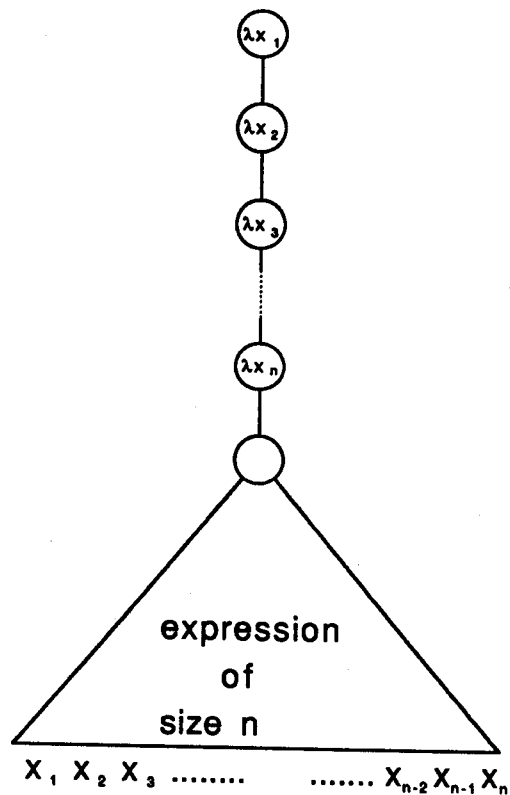


Figure 5.3: A bad case for environment splitting in Statman's algorithm.

variable occurrences. An improvement that costs (almost) nothing is blocking the environment from propagating into a constant subexpression by K . How about splitting the environment to its optimal extent? Will it help to reduce the code size?

A splitting operation could easily cost $c \log n$ extra combinators. In case like the one exemplified in figure 5.3 this will amount to $cn \log n$, the worst possible case in Statman's algorithm. We conclude that not much is to be expected from splitting the environment.

Statman's tree is an optimal storage structure in the λ -calculus. We cannot hope to improve the bounds for packing and retrieval code size by an order of magnitude. Introduction of specialized combinators might prove

combinator	Translation into S, K, I
S	S
K	K
I	I
B	$(S(K S))K$
C	$(S(S(K S)((S(K K))S)))(K K)$
S'	$(B(B S))B \quad (((S(K S))K)((S(K S))K)S)((S(K S))K)$
B'	$(B B) \quad (((S(K S))K)((S(K S))K))$
C'	$(B(B C))B \quad (((S(K S))K)((S(K S))K) \quad ((S(S(K S)((S(K K))S)))(K K)))$ $((S(K S))K)$

Figure 5.4: Translation of Turner's combinator set into S, K and I .

helpful, however. First, consider the retrieval functions F_i^n . A typical example is:

$$F_3^6 = (BL)((BR)((BL)I)) \quad (5.3)$$

By abbreviating (BL) and (BR) into predefined single combinators, the size of F_i^n is reduced by a factor 2. For the insertion functions A_n one could take $A_<$ and $A_>$ as *primitives*. This brings the constant in theorem 8 down to 2, an improvement by a factor 3.

5.2 The redundancy of new combinators

Is the introduction of new combinators essential? Might we not achieve as much using S, K and I only? The answer is yes. Use any of the foregoing algorithms to get a caf. Replace all combinators by their definitions in S, K and I . All you lose is a constant factor c in the code size. This constant is bounded by the longest definition. From figure 5.4, we may conclude that $c = 22$ for Turner's algorithm.

One may wonder whether the same result cannot be obtained without the intermediate step of introducing new combinators. Swierstra discovered that it may be done by abstraction of subexpressions [Swi86]. He shows how S' may be rendered superfluous. As an example, consider the introduction of B

$$(T x (E F_x)) = (B E)(T x F_x) \quad (5.4)$$

Using the translations from figure 5.4 outlined above, this may be rewritten to:

$$(((S(K S))K)E) (T x F_x) \quad (5.5)$$

We show how this last expression may be obtained by using two operations:

- A modification of Curry's algorithm called *CRV*, altered to apply *K* on *maximal* constant subexpressions. Also, redundant applications are removed.

$$\begin{aligned} CRV x (Ex) &= E \text{ (if } x \notin \text{freevars}(E)) \\ CRV x x &= I \\ CRV x E &= (KE) \text{ (if } x \notin \text{freevars}(E)) \\ CRV x (E_1 E_2) &= (S E_1)E_2 \text{ (if no earlier rule applies)} \end{aligned}$$

This algorithm is called (*abcf*) in [CHS72].

- Abstraction of subexpressions.

Starting with $CRV x (E F_x)$, it is possible to attain $(B E)(CRV x F_x)$. The clue lies in the choice of abstraction steps intertwined with *CRV*'s execution.

$$\begin{aligned} CRV x (E F_x) &\Rightarrow \text{introduction of } S \\ (S(CRV x E))(CRV x F_x) &\Rightarrow \text{introduction of } K \\ (S(K E))(CRV x F_x) & \end{aligned}$$

For brevity, consider the left expression:

$$\begin{aligned} (S(K E)) &\Rightarrow \text{abstraction of } E \\ ((CRV a (S(K a)))E) &\Rightarrow \text{introduction of } S \\ (((S(CRV a S))(CRV a (K a)))E) &\Rightarrow \text{introduction of } K \\ (((S(K S))(CRV a (K a)))E) &\Rightarrow \text{redundant application} \\ (((S(K S))K)E) & \end{aligned}$$

The same process may be applied to *C*, *S'*, *B'* and *C'*. Surely abstraction of subexpressions appears to be a pretty strong tool.

5.3 Abstraction of expressions

In the preceding section, we saw how new combinators may be simulated by a simple translation algorithm, combined with the abstraction of subexpressions. But how does one recognize the right abstraction? In general, one

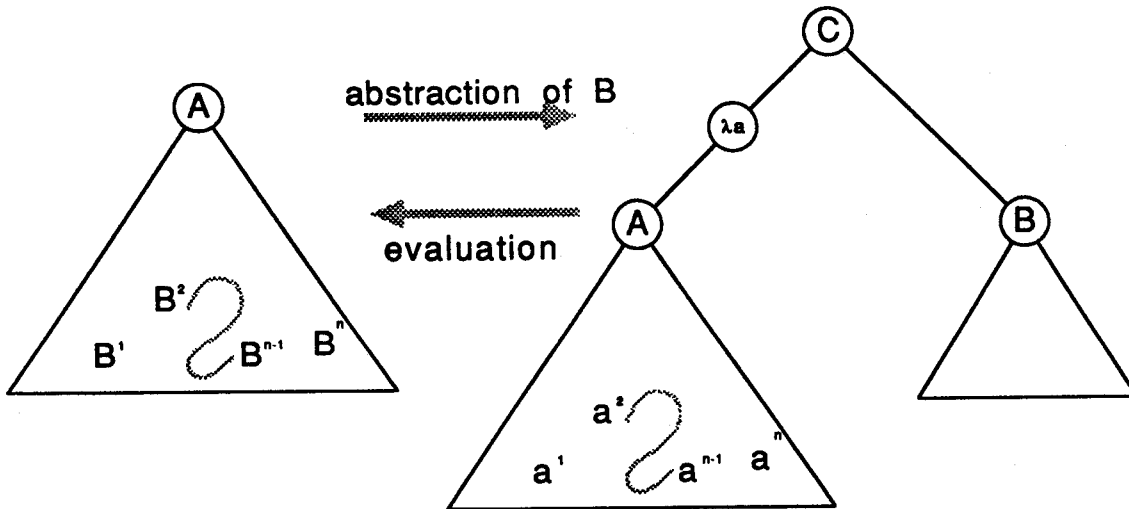


Figure 5.5: General abstraction of subexpressions.

wonders whether a sequence of abstractions resulting in a minimal code size is computable. We argue that this is probably not the case. Therefore, ways are examined to identify profitable abstractions in Statman's algorithm. In a heuristic way they may help us to find a profitable sequence of abstractions.

In Turner's algorithm, abstractions will be used to minimize the spanning tree sizes. This enables us to bring the worst case for applicative forms to the same order of magnitude as Statman's algorithm.

An interesting idea is to use abstraction of the largest possible expressions. This was done by John Hughes [Hug82]. He abstracts 'maximal free expressions'. Maximal free expressions are subterms that do not contain free occurrences of the variable that is being eliminated. Moreover, they are not contained in other terms sharing the same property.

5.3.1 Conditions on abstraction

Before we proceed to discuss the effects of abstracting expressions on the code size, agreement on the exact properties of this operation is required. In figure 5.5 it is depicted graphically. Abstraction should always be reversible by evaluation.

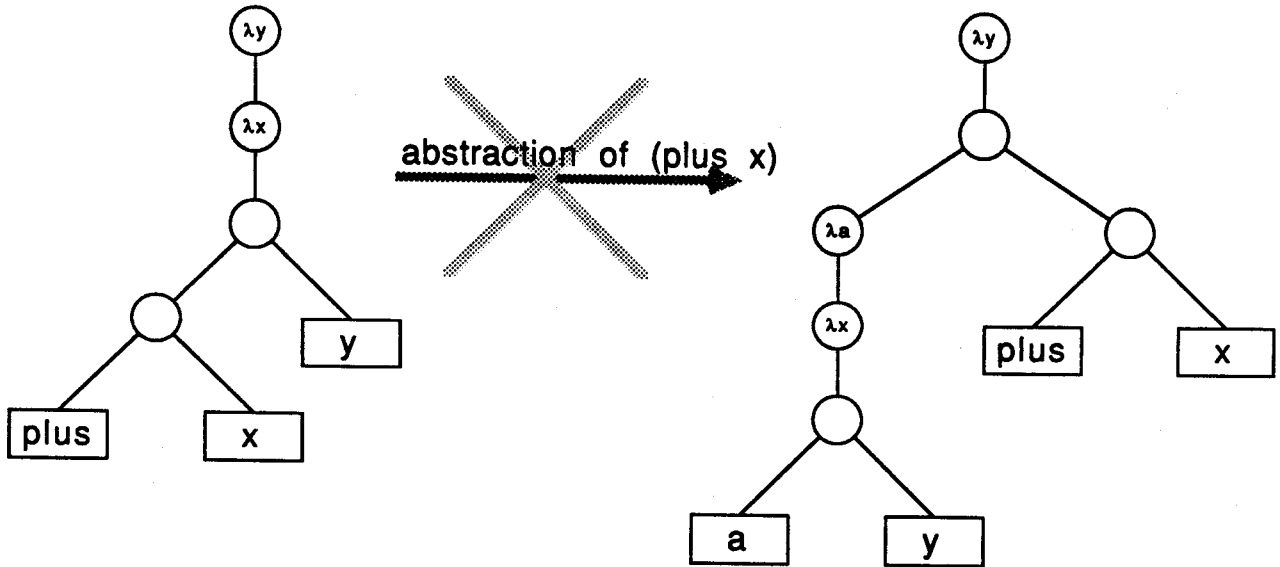


Figure 5.6: Variables should not be moved outside their scope.

Consider figure 5.6. Here, there is no way back, because x is no longer bound by its introduction-node. Evaluation of the right-hand tree does not yield the one on the left. Hence we require that no variable is moved outside its scope. (In $(\lambda x.E)$, E is the range of x , the scope of x is E minus expressions of the form $(\lambda x.F)$).

Since abstraction is to be used for practical purposes, it is worthwhile to exclude some degenerate cases. Consider the abstraction of an expression from itself:

$$E \Rightarrow (\lambda a.a)E \quad (5.6)$$

Since E is still unaltered in the result, one can hardly expect reduction of the code size in going from E to $(\lambda a.a)E$. Abstractions of true subexpressions are called *proper*.

Note that non-proper abstractions could serve for common subexpression elimination. Expressions could be made equal by them. We shall not pursue this issue. In the sequel, abstractions are assumed to be proper.

5.3.2 The optimal abstraction sequence

In the preceding chapters, three algorithms to eliminate variables were presented, Curry's, Turner's and Statman's. They all work by bumping the λ -introductions down the expression tree. Curry's and Turner's method move one λ -introduction at a time, requiring multiple sweeps to remove them all. Statman bumps them all down in one sweep. This suggests exploiting expression abstraction as follows. Each time a new combinator is introduced, we consider the whole expression tree. A (possibly empty) sequence of abstractions is performed. The algorithm proceeds transforming the *whole* tree until no λ -introductions remain. This outline is sketched (for multi-sweep algorithms) in the following pascal-like fragment:

Algorithm 10 (*outline multi-sweep algorithm with abstractions*)

```
PROCEDURE Translate(VAR WholeTree : ExprTree);
VAR Subject : ExprTree;

    FUNCTION FindInnerLambda(T : ExprTree;
                               VAR Result : ExprTree) : BOOLEAN;

        FUNCTION Do(T : ExprTree) : BOOLEAN;
        BEGIN
            IF T=(lambda x.E), E contains no lambda's
            THEN BEGIN
                Result := T;
                Do := TRUE
            END
            ELSE IF NOT Do(T^.Left)
            THEN Do(T^.Right)
        END;

    BEGIN
        Result := NIL;
        FindInnerLambda := Do(T)
    END;

PROCEDURE BumpDownLambda(VAR Subject : ExprTree);
{equivalent to CRV}
```

```

BEGIN
  ... let E and x be such that (lambda x. E)=Subject ...
  IF   match(E, (D x)) AND
      NOT (x IN FreeVars(D)) THEN Subject := D
  ELSIF match(E, x)
      THEN Subject := I
  ELSIF NOT (x IN FreeVars(E)) THEN Subject := (K E)
  ELSIF match(E, (E1 E2))}   THEN Subject := ((S E1) E2)
  ELSE {does not occur}
  FI
END;

BEGIN { Translate }
  PerformAbstractions(WholeTree);
  WHILE FindInnerLambda(WholeTree, Subject) DO
    BEGIN
      BumpDownLambda( Subject );
      PerformAbstractions( WholeTree )
    END
  END; {Translate}

```

End of algorithm 10

We are faced with the question: Can `PerformAbstractions` be defined to get the smallest code possible? In chapter 4, we saw that this cannot be done in a general way. However in the present context we are considering very, very restricted algorithms only. Is there a best one? Assume there is. Recall that abstraction is the inverse of evaluation. Given some value, we are looking for a function that will compute it. This function should translate to the shortest possible `caf` using our procedure `BumpDown`. In all likelihood this is an undecidable problem, but the author was unable to prove this conjecture. It is possible however, to attain the 'best' case as defined by Statman. We show this for applicative forms and Turner's algorithm in a subsequent section.

5.3.3 Abstractions in Statman's algorithm

In this paragraph it is investigated what might be gained by a single expression abstraction in the original expression using Statman's algorithm. We

define a ‘cost’ for every expression. Subsequently, it is proved that a larger cost means a larger translation. Hence, if an abstraction brings down the cost of an expression, one has gained something. The ‘cost’ may be more easily computed than the translation according to Statman’s algorithm. This is important, since otherwise one could simply do the translation to check whether a particular abstraction is profitable.

The length of Statman’s output is depending on the size of the environment and the length of the translation of subexpressions. This suggests the following definition:

Definition 12 (*expression cost*)

Let E be a λ -expression, and $envir$ be a set of variables. The *cost of E in $envir$* is

$$\begin{aligned} \text{cost } (\lambda x.E) \text{ } envir &= (\text{cost } E_1 \text{ } envir \cup \{x\}) + |envir| \\ \text{cost } (E_1 E_2) \text{ } envir &= (\text{cost } E_1 \text{ } envir) + (\text{cost } E_2 \text{ } envir) \\ \text{cost } x \text{ } envir &= |envir| \\ \text{cost } c \text{ } envir &= 1 \end{aligned}$$

We conjecture that this cost function may be used to judge abstractions. First, it is shown that a decrease in cost implicates that the code will not grow.

Lemma 10 (*caf size and expression cost*)

Let TR be the Statman translation *StatmanTrans*. Let E and F be λ -terms. Let $envir$ be a set of n variables, containing both $freevars(E)$ and $freevars(F)$. If $(\text{cost } E \text{ } envir) < (\text{cost } F \text{ } envir)$ then $|TR E n| \leq |TR F n|$.

Proof: Let D be a λ -term. We show by induction on D that $|TR D n|$ is monotonically increasing in $\text{cost } D \text{ } envir$.

- D atomic: Then by definition $\text{cost } D \text{ } envir = 1$ or $\text{cost } D \text{ } envir = |envir|$. $|TR D n| = |F_i^n| = 2 \log n$, where $n = |envir|$, so here the conjecture holds.
- $D = (D_1 D_2)$:

Hypothesis $TR D_i n = f_i(\text{cost } D_i \text{ } envir)$, f_i monotonically increasing. ($i = 1, 2$)

$$\begin{aligned}
|TR D n| &= \\
|S(TR D_1 n)(TR D_2 n)| &= \\
1 + f_1(cost D_1 envir) + f_2(cost D_2 envir)
\end{aligned}$$

which is of course monotonically increasing too.

- $D = (\lambda x.E)$:

$$\text{Hypothesis } |TR E (n + 1)| = f(cost E envir \cup \{x\})$$

$$\begin{aligned}
|TR D n| &= \\
3 + C \log |envir| + |TR E (n + 1)| &= \\
3 + C \log |envir| + |f(cost E envir \cup \{x\})|
\end{aligned}$$

If the cost of D increases, then so does this expression.

□

In figure 5.7 the abstraction of B from A is depicted. Assuming that the cost at each node of the tree is known, it should be possible to compute the cost difference brought about. B may occur more than once in A , its occurrences being distinguished by a superscript. At a node X , $(envir X)$ is the environment of the corresponding subterm. $(cost X)$ signifies its cost, $(cost X (envir X))$.

Lemma 11 (*abstractions and cost*)

Let E be an expression with $cost E envir = c$. Then

$$cost (\lambda x.E) envir = c + (lambdas E) + (vars E) + |envir|$$

Where

$$lambdas E = \text{number of } \lambda\text{-introductions in } E$$

$$vars E = \text{number of variable-occurrences in } E$$

Proof: For each variable and each λ -introduction we have a cost-increase of 1. At the node itself, an increase by $|envir|$ is inflicted. □

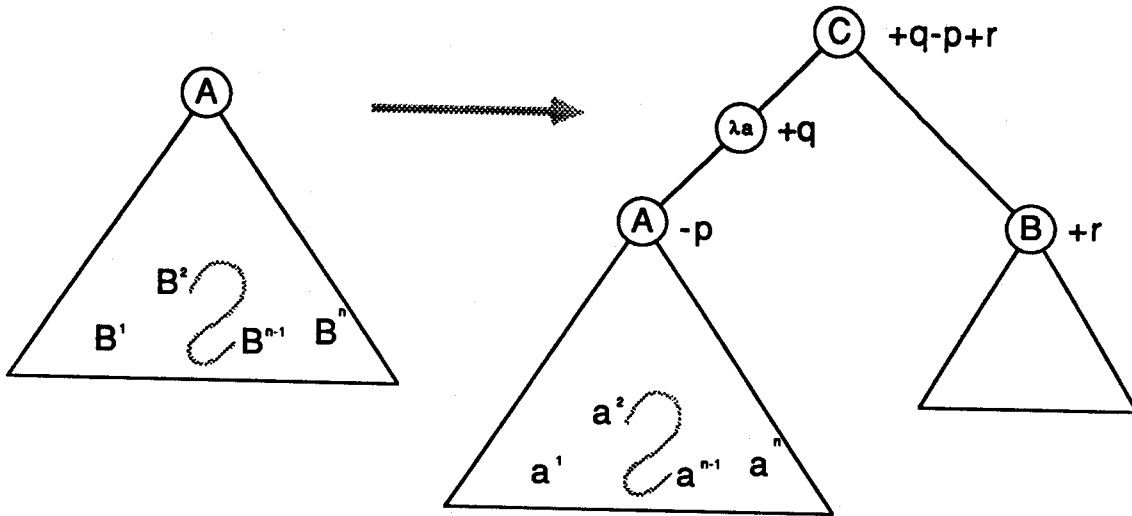


Figure 5.7: Computing the cost difference for an abstraction.

Theorem 9 (abstractions and cost difference)

Let A , B^i and C as in the left tree of figure 5.7. The B^i are occurrences of a subtree B . Let

$$p = \sum_{i=1}^n (\text{cost } B^i) - \sum_{i=1}^n |\text{envir } B^i|$$

$$q = \text{lambda } A + (\text{vars } A) + |\text{envir } A| - n(\text{vars } B + \text{lambda } B)$$

$$r = \text{cost } B^1 + (|\text{envir } C| - |\text{envir } B^1|)(\text{lambda } B + \text{vars } B)$$

Assuming all variables are distinct, the total cost increase inflicted by abstraction in figure 5.7 is:

$$q - p + r$$

Proof: p , q and r are explained.

p: In A , we remove the occurrences of B , B^i . This decreases the cost by $\sum_{i=1}^n \text{cost } B^i$. However, they are replaced by a 's, each accounting

for cost $|envir B^i|$, where $(i = 1, \dots, n)$. Therefore, the cost of A is decreased by $\sum(cost B^i) - \sum|envir B^i|$

q: Immediate from the previous lemma.

r: On the right hand side in figure 5.7 B is a ‘new’ subtree. Its cost is $cost B^1$. However the environment is reduced compared to B^1 . $(envir B^1) \supseteq (envir C)$, because all variables are distinct. So we get an *increase* of $|envir C| - |envir B^1|$ for each λ -introduction and variable occurrence in B .

□

To compute this measure efficiently, one should augment each node with the necessary information as attributes. This can be done during tree construction. Recomputation after an abstraction may be done in linear time.

5.3.4 Turner on the chopping block

Any attempt at optimizing Turner’s algorithm should be submitted to the test whether it improves on the worst-case bound as it is stated in chapter 3. Abstraction of expressions does so for applicative forms. By incorporating some simple abstractions, the output will be rendered ‘weakly optimal’, as described in chapter 4.

The main idea is straightforward enough. We should minimize the sum of the spanning tree sizes. For an applicative form, this is equivalent to minimizing the sum of all path-lengths from the root of the body to the leaves.

It is a well-known fact that the so-called ‘weight-balanced trees’ exhibit the following property: Each path from the root to a leaf has length $\mathcal{O}(\log n)$ (i.e. $\leq C \log n$ for n large enough). Consider an applicative form

$$(\lambda x_1. (\lambda x_2. (\dots (\lambda x_k. B) \dots))) \quad (5.7)$$

Assume that the body B is a weight-balanced tree. Translation will add

$$\mathcal{O}(|B| \log |B|) \quad (5.8)$$

combinators at most. The whole translation will have length

$$\mathcal{O}((|B| \log |B|) + |B|) = \mathcal{O}(|B| \log |B|). \quad (5.9)$$

We will try to convert any applicative form into such a balanced tree by abstraction of expressions.

Balanced trees are ubiquitous in computer science. We already met them in this paper: the environment structure by Statman. Here we will use a more relaxed version. The main goal is to prevent trees from listing either too far to the left or to the right. Weight-balanced trees were introduced by Nievergelt and Reingold [NR73]. We cite the definitions from [BM80].

Definition 13 (*balance of a tree*)

Let T be a binary tree. If T is a single leaf, the the *root-balance* $\beta(T)$ is $1/2$, otherwise we define $\beta(T) = |T_L|/|T|$, where $|T_L|$ is the number of leaves in the left subtree of T and $|T|$ is the number of leaves in tree $|T|$.

Definition 14 (*balanced tree*)

A binary tree T is said to be of *bounded balance* α , or in the set $BB[\alpha]$, for $0 \leq \alpha \leq 1/2$, if and only if

1. $\alpha \leq \beta(T) \leq 1 - \alpha$
2. T is a single leaf or both subtrees are of bounded balance α .

For reasons that will be explained later on, α is taken such that $\alpha \leq 1/4$. A λ -term is said to be of *bounded balance* α if each application node satisfies the conditions under definition 14.

To give the reader a more concrete idea of what we intend to do, an example is provided in figure 5.8. The expression depicted is a bad case for Turner's algorithm. The tree is listing far to the right. As a consequence, the term is *not* balanced in the sense defined above. The translation introduces 53 combinators. By abstraction of E_5 , the situation is profoundly changed. The term is balanced and requires only 44 combinators.

Lemma 12 (*balanced expressions in Turner's algorithm*)

Let E be a balanced applicative form

$$(\lambda x_1. (\lambda x_2. (\dots (\lambda x_k. D) \dots)))$$

where $|D| = n$.

$$|\text{TurnerTrans } D| = \mathcal{O}(n \log n)$$

Proof: Let p be the sum of the lengths of all distinct paths from the root

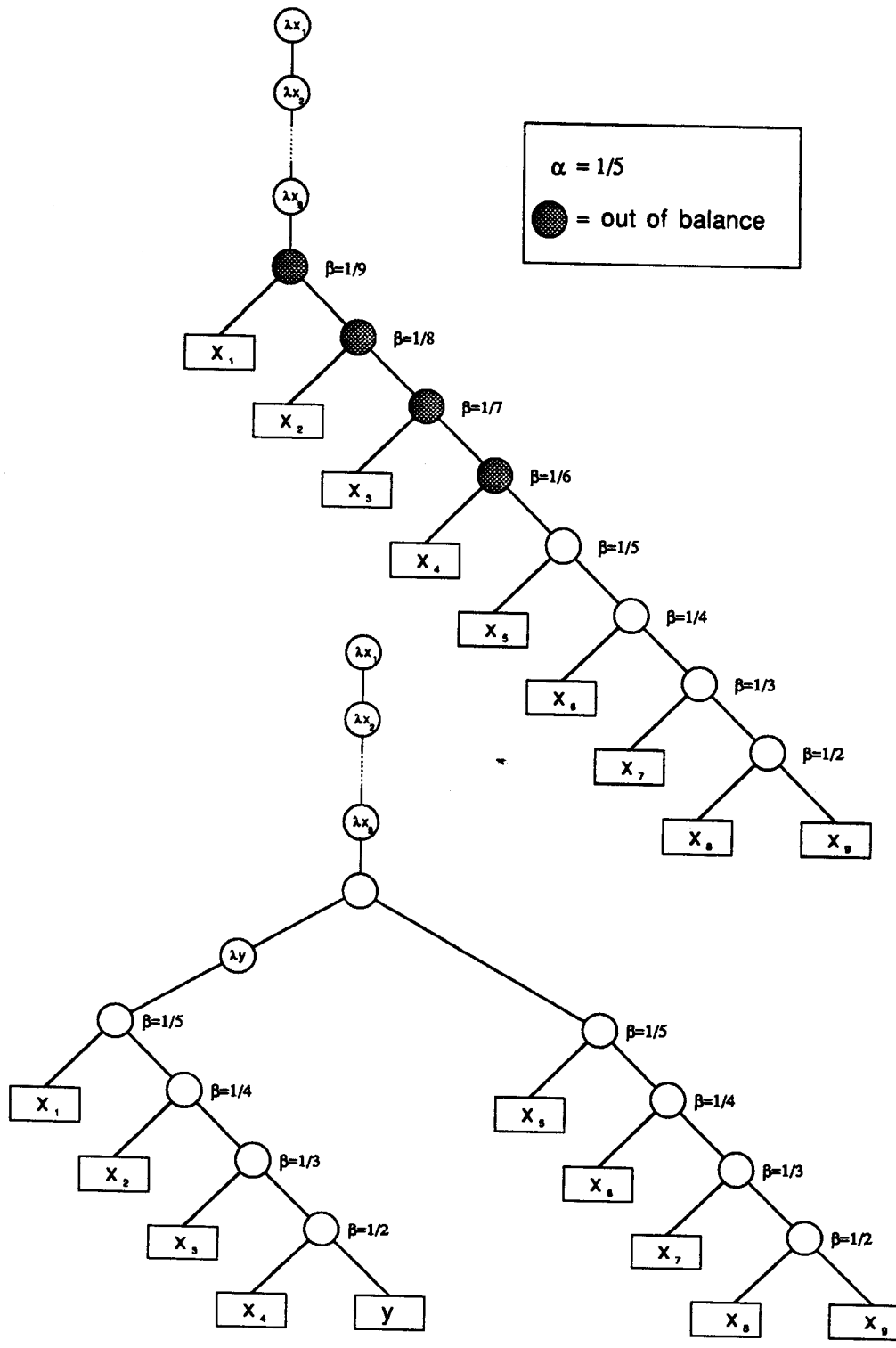


Figure 5.8: Example of expression balancing by abstraction.

of D to the leaves. Each path has length $\mathcal{O}(\log n)$ because D is a balanced binary tree. Hence $p = \mathcal{O}(n \log n)$. $F := (t D)$ From the definition of t (algorithm 4), it is clear that no path is made any longer. The sum of the number of internal nodes in the spanning trees in F of x_1, x_2, \dots, x_k is equal to or less than p .

Application of U (result: G) yields $\mathcal{O}(n \log n)$ combinator introductions, accounting for a total expression length of $|G| = \mathcal{O}(n \log n)$.

Define t^{-1} to be:

$$\begin{aligned} t^{-1} (EFG) &= (((t^{-1}E)(t^{-1}F))(t^{-1}G)) \\ t^{-1} (EF) &= ((t^{-1}E)(t^{-1}F)) \\ t^{-1} a &= a \end{aligned}$$

Translate G back to the conventional tree representation using t^{-1} . The length of G is not altered.

Summarizing, we used $t^{-1}(\text{Trans } U)t$ to get a translation of length $\mathcal{O}(n \log n)$. (Trans is algorithm 3 adapted to the new representation.) By lemma 2 $t^{-1}(\text{Trans } U)t = (\text{Trans } T')$. $\text{Trans } T'$ always performs worse than the original algorithm $\text{TurnerTrans} = \text{Trans } T$, because cases 2^a , 2^b and 3^a are dropped. \square

Theorem 10 (*caf size for balanced terms*)

Let E be a balanced λ -term, with $|E| = n$.

$$|\text{TurnerTrans } E| = \mathcal{O}(n \log n)$$

Proof: For E an applicative form, the theorem holds by the previous lemma. Observe that

$$\begin{aligned} |\text{TurnerTrans } ((\lambda x.E)D)| &= \\ |\text{TurnerTrans } (D(\lambda x.E))| &\leq \\ |\text{TurnerTrans } (\lambda x.(E D))| & \end{aligned}$$

By the previous lemma, we may conclude that the theorem holds for general λ -terms, too. \square

Consider the general form of a worst case for Turner's algorithm, as depicted in figure 5.9. We chop it up into two parts of approximately equal length, and combine them by lambda abstraction into one balanced node. Both subtrees will increase in size by an equal amount of extra nodes, so this node remains balanced throughout the rest of the process.

The algorithm is applied recursively to both subtrees. We continue in this way until a subexpression has only three leaves (or less). Note that this situation is always reached after $\log n$ steps: At each step the sizes of subtrees shrink by a factor 2.

Clearly, the number of nodes in the tree grows rapidly by this transformation. In the first step, 2 extra nodes are introduced, in the second 4, in the third 8 and so on. This amounts to:

$$\sum_{i=1}^{\log n} 2^i = 2^{(\log n)+1} - 2 = 2n - 2 \quad (5.10)$$

extra nodes. However, the resulting tree of size $3n - 2$ (excluding λ 's) translates to an expression of length $\mathcal{O}(n \log n)$ by the previous theorem. In the end a definite gain has been booked, as exemplified in figure 5.3.4.

Could this algorithm be applied to any expression tree? Here, we cannot always find a subexpression of size $n/2$ or $(n - 1)/2$. To solve this problem, a strategy expressed by the function `search` could be employed:

```

search tree size =
  IF (sizeof tree) <= size -> tree
    (sizeof tree) > size -> IF sizeof (leftSub tree) >=
                          sizeof (rightSub tree) ->
                            (search (leftSub tree) size))
                          OTHERWISE ->
                            (search (rightSub tree) size)
    FI
  FI

```

As soon as `sizeof tree <= size` applies, the search is over. By the definition of `search`, the parent to `tree` is larger than `size`, and `tree` is the larger of its sons. Hence `(sizeof tree) >= size/2 = (sizeof expr)/4`. We may readily conclude that the algorithm presented for the worst case may also be used in all other cases to get a tree of bounded balance less than $1/4$. The result remains the same, but we have to use \log to the base $4/3$.

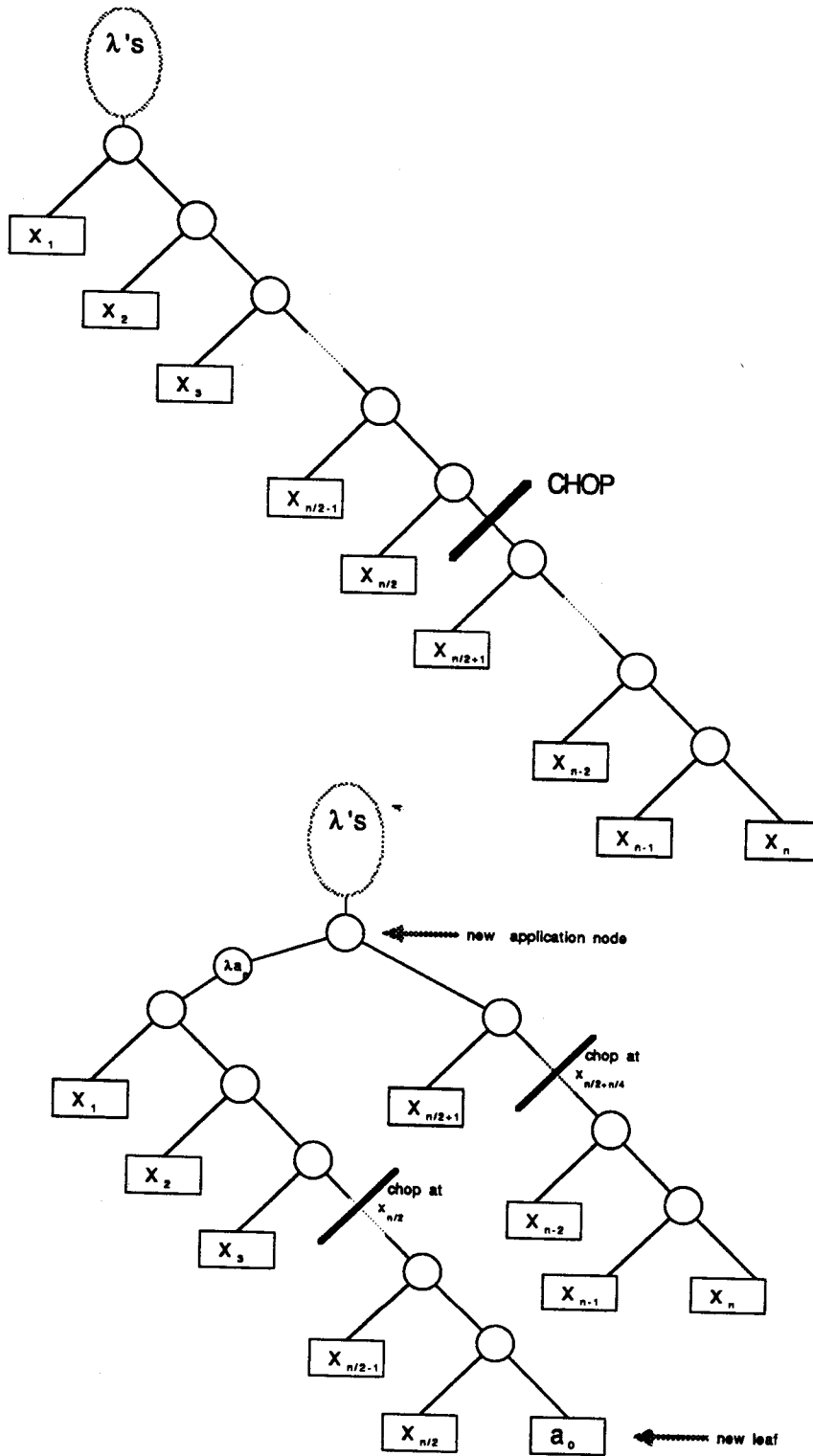


Figure 5.9: Improving on Turner's worst case behaviour.

expression length	TurnerTrans length	optimtrans ($\alpha = 0.25$) length
15	13	14
33	64	50
48	134	81

Figure 5.10: A comparison in code-length.

The algorithm, implemented in SASL ([Tur76]), is given below:

Algorithm 11 (*balancing of λ -terms*)

```

sizeof ('app",e1,e2) = (sizeof e1) + (sizeof e2)
sizeof e             = 1 || atoms

beta ('app", e1, e2) = sizeof e1/ ((sizeof e1)+
                                (sizeof e2))
beta e               = 1/2 || atoms

|| -----
||               balancing an applicative form
|| -----
top_balance alpha ('lam",y,e) = ('lam",y,
                                (top_balance alpha e))
top_balance alpha e         = balance alpha 1000 e

|| -----
||               guarantee a balanced tree
|| -----
balance alpha x e = (alpha <= beta(e)) &
                   (beta(e) <= 1-alpha)
                                -> (sub_balance
                                    alpha x e)
                                (('app",
                                ('lam",
                                x,

```

```

                (balance alpha (x+1) body)),
                (balance alpha (x+1) abstracted))
WHERE (body, abstracted) =
        (chop e ((sizeof e)/2) x))

sub_balance alpha x ('app",e1,e2) = ('app",
                                     (balance alpha x e1),
                                     (balance alpha x e2))
sub_balance alpha x e                = e      || atoms only

|| -----
||                                     chop a subtree of the right size
|| -----

chop tree size x = (sizeof tree) <= size
                  -> (('id",x), tree)
                  (chopappl tree size x)
chopappl ('app",e1,e2) size x =
          (sizeof e1) >= (sizeof e2) -> (((('app",r1,e2),r2)
                                           WHERE (r1,r2)=(chop
                                                         e1
                                                         size
                                                         x))
                                           (((('app",e1,r1),r2)
                                           WHERE (r1,r2) = (chop
                                                         e2
                                                         size
                                                         x)))

|| -----
||                                     optimized Turner translation
|| -----

optim_trans alpha e = turner_trans (top_balance alpha e)

```

End of algorithm 11

This approach is very similar to the one taken by Burton [Bur82]. In chopping, he searches for the tree having as close to $(n + 1)/2$ leaves as possible. This enables one to take $\alpha = 1/3$. An interesting extension proposed by him is the application to expressions that consist of closed subterms only, i.e. within the body of a function no variables from an enclosing term are used. By simply taking $\text{size}(\text{translatedexpr}) = 1$ the method is readily generalized to such terms. Any λ -expression can be converted to the indicated form by introducing extra parameters for global variables. A method to do this transformation is introduced in the next section.

5.3.5 Abstraction of maximal free expressions

We are investigating the effect of expression abstractions on code size. As pointed out, no variable should be moved outside its scope. All candidates are *free* expressions, they do not contain any occurrence of the variable that we intend to eliminate. What if all *maximal free* expressions are abstracted to the outside? A maximal free expressions (mfe) is a free term that is not enclosed in another with the same property. An example of repeated mfe abstraction is given in figure 5.11. Single constants are not abstracted.

Hughes [Hug82] uses this technique to remove variables. His method boils down to the introduction of specialized combinators for each program, rather than using a fixed set as we did up to now. These new combinators are called *super-combinators*. In figure 5.11 c_0 and c_1 are super-combinators.

A translation to super-combinators from [Hug85] is presented here. For clarity, applications ($E D$) are marked (*app* $E D$) to distinguish them from program fragments. Also, identifiers are denoted by (*id* x), x an integer.

Algorithm 12 (*Hughes' algorithm*)

```

Strans (id x)      = (id x)
Strans (app E1 E2) = ((Strans E1) (Strans E2))
Strans (lam x E)   = (abstract (mfes x tb) x tb)
                   WHERE tb = trans b

```

```

abstract m x b = mkapp m (mkcom (m++[x]) b)
mkapp [] f     = f
mkapp (a:m) f  = mkapp m ( app f a)

```

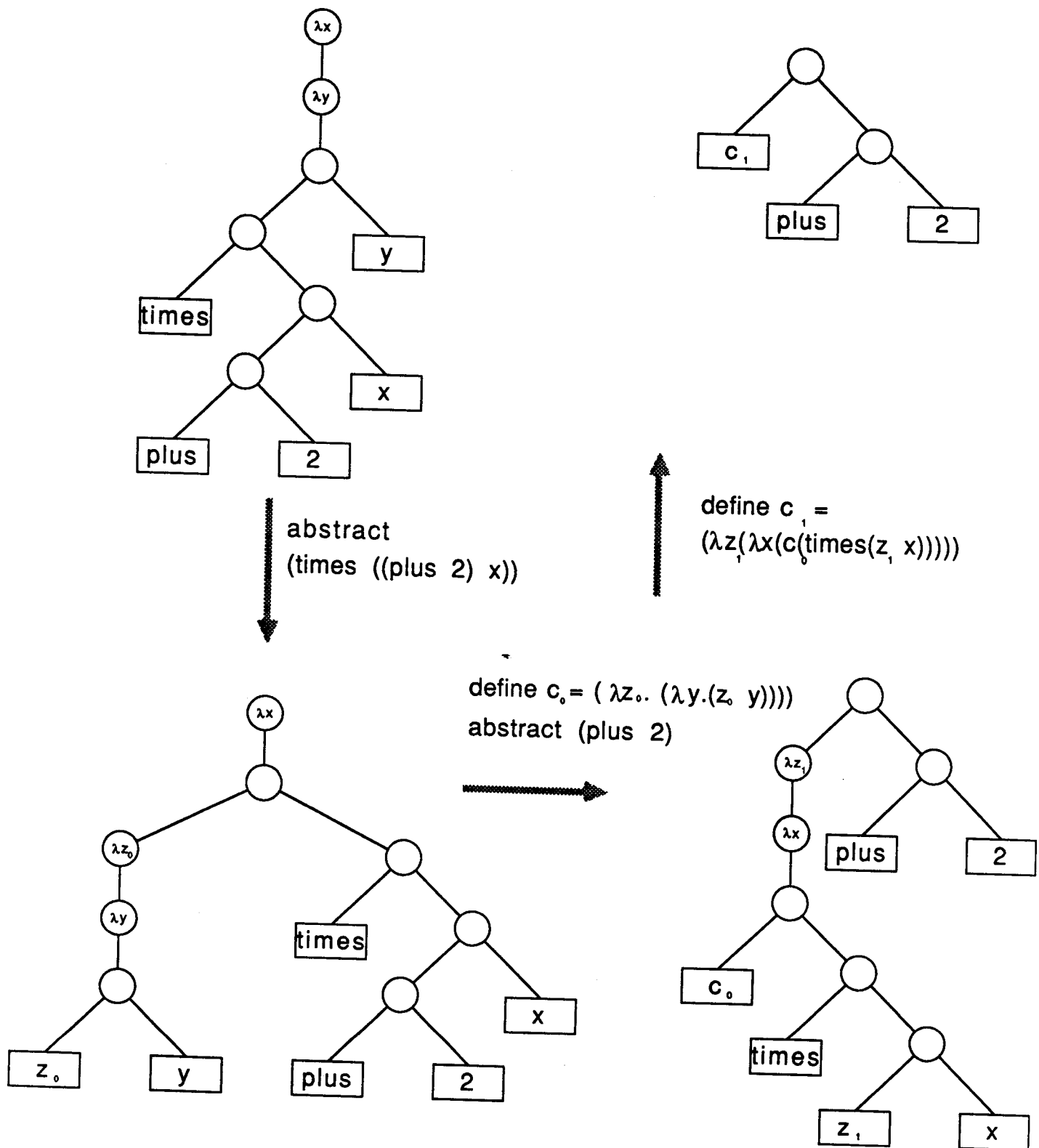


Figure 5.11: Example mfe abstractions.

`mkcom m b = (com (length n) (subst m [1..length m] b))`

End of algorithm 12

'abstract' removes the mfes that are passed as a list in its first argument from *tb*, and introduces a super-combinator for the remaining body.

A new kind of node is introduced: *com a b*. This signifies a super-combinator with *a* arguments and body *b*. The parameters in the body are represented by integers from 1 to *a*. *subst* introduces the numbers, replacing them for the mfes that are abstracted.

To find the maximal free expressions it is necessary to determine the most global identifier in each subexpression. If this is a more global one than the variable to be abstracted, we have a free expression.

One may wonder what happens if the following translation scheme is used: First translate an expression to super-combinator code. Second, translate the super-combinators using Turner's algorithm. The combinators from the fixed set $\{S, K, I, B, C, S, B', C', S'\}$ act as microcode for the super-instructions.

Intuitively, one might expect that this would yield a good translation. It is, however, not difficult to find a counter-example. Consider our familiar right-skewed case:

$$(\lambda x_1. (\lambda x_2. (\dots (\lambda x_n. (x_1 (x_2 (\dots (x_{n-1} x_n) \dots)))))) \dots)) \quad (5.11)$$

E is submitted to *Strans*. The generated super-combinator is equal to the whole expression. The translation will inevitably end up with a result that is even larger than what direct application of Turner's algorithm would have yielded.

Chapter 6

Discussion

In this section, we intend to dwell a little on the consequences of the results we obtained. First, some possible directions for further research are examined. Second, we discuss how the results might be applied in practice.

The first extension to the present work should be the implementation of all algorithms. Apart from the fact that this programming task should be a most instructive exercise, we will get empirical data on the average performance of the translations. This should serve as a basis for average case analysis. While we are happy to know that no really bad output can occur, the question whether the average behaviour of a compiler to cafs is good is much more important.

The algorithms presented in this paper only provide a snapshot of the wide spectrum available. The summary in [Mul85] is much more complete. It should be illuminating to make a more exhaustive comparison in complexity.

The properties of expression abstraction are by no means covered completely in chapter 5. The question remains how we can do better than the crude splitting of expressions. The theme of the connection to super-combinators should be investigated in more depth. Especially the optimizations from [Hug82] provide an interesting research theme.

We did not consider compile-time evaluation as a technique to improve on variable-eliminating algorithms. The main reason for not doing so is that this would demand a lengthy exposition on 'strictness analysis'. Compile-time evaluation is certainly advantageous from the viewpoint of time-efficiency, if graph representations with subexpression-sharing are used. For abstraction of subexpressions, this is not necessarily the case.

It remains to be seen how much may be gained by sharing substructures

in the graph representation. The question whether there is a trade-off between caf size and evaluation time is still open. What is needed is a sound theory on efficiency analysis of functional programs.

It was the main aim of this paper to contribute in a modest way to our insight in variable-elimination. While interesting from a theoretical point of view, its practical merits are not great. Computer memories are ever-increasing in size and falling in price, so reduction of the code size from n^2 to $n \log n$ is not that important. However, a feasible practical application might lie in the distributed execution of functional programs. Paul Hudak and Benjamin Goldberg argue in [HG85] that a variant of Hughes' super-combinators, called serial combinators, provide 'maximal grains of parallelism'. These identify subcomputations with 'minimal' communication costs. This highly resembles our minimization of the expression tree. Identifying subtrees with processes, we seek the lowest number of processes computing our function. Research on this correspondence should enhance our understanding of specialized architectures for functional languages. A recent survey of this field was made by Steven Vegdahl in [Veg84].

Acknowledgements

I am very grateful to Doaitse Swierstra who introduced me to the subject. Moreover he passed the research theme on to me and scrutinized an early version of this paper. Thanks are due to Hans Mulder, Henk Barendregt and Matthijs Kuiper for their kind help in locating the relevant literature. I am also thankful to Hans Zantema for suggesting many improvements.

Bibliography

- [Bac78] J. Backus. Can programming be liberated from the Von Neumann style? *Communications of the ACM*, 21(4):613–641, 1978.
- [Bar84] H.P. Barendregt. *The lambda calculus: Its syntax and semantics*. Volume 103 of *Studies in logic and the foundations of mathematics*, North-Holland, 1984.
- [BM80] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11:303–320, 1980.
- [Bur82] F.W. Burton. A linear space translation of functional programs to turner combinators. *Information Processing Letters*, 14(5):201–203, 1982.
- [CHS72] H.B. Curry, J.R. Hindley, and J.P. Seldin. *Combinatory logic II. Studies in logic and the foundations of mathematics*, North-Holland, 1972.
- [HG85] P. Hudak and B. Goldberg. Distributed execution of functional programs using serial combinators. *IEEE Transactions on Computers*, C-34(10):881–891, 1985.
- [Hug82] J. Hughes. Super-combinators: a new implementation method for applicative languages. In *Proc. ACM Symp. on Lisp functional programming*, pages 1–10, ACM, 1982. ACM 0-89791-082-6/82/008/0001.
- [Hug85] J. Hughes. Lazy memo-functions. In J.P. Jouannaud, editor, *Functional programming languages and computer architecture*, pages 130–146, Springer-Verlag, 1985. LCNS 201.

- [JRB85] M.S. Joy, V.J. Rayward-Smith and F.W. Burton. Efficient combinator code. *Comput. Lang.*, 10(3/4):211–224, 1985.
- [Ken84] J.R. Kennaway. *The complexity of a translation of λ -calculus to combinators*. internal report CSA/13/1984, University of East Anglia, Norwich, 1984.
- [Mul85] J.C. Mulder. *Complexity of combinatory code*. preprint 389, University of Utrecht, Department of Mathematics, 1985.
- [NR73] J. Nievergelt and E.M. Reingold. Binary search trees of bounded balance. *Siam J. Comput.*, 2(1):33–43, 1973.
- [RND77] E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.
- [Sch24] M. Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92:305–316, 1924.
- [Sta83] R. Statman. *An optimal translation of λ -terms into combinators*. internal report, Carnegie-Mellon University, Pittsburg, Pennsylvania, 1983.
- [Swi86] S.D. Swierstra. Reducing combinator complexity by abstracting expressions. 1986. personal communication.
- [Tur76] D.A. Turner. *SASL language manual*. Technical Report, University St. Andrews, 1976.
- [Tur79a] D.A. Turner. Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44:267–270, 1979.
- [Tur79b] D.A. Turner. A new implementation technique for applicative languages. *Software: Practice and Experience*, 9:31–49, 1979.
- [Tur82] D.A. Turner. Recursion equations as a programming language. In J. Darlington, P. Henderson, and D.A. Turner, editors, *Functional programming and its applications*, pages 1–28, Cambridge University Press, 1982.
- [Veg84] S.R. Vegdahl. A survey of proposed architectures for the execution of functional languages. *IEEE Transactions on Computers*, C-33(12):1050–1071, 1984.

Index

- abstraction of subexpressions 57, 64
 - cost difference 71
 - conditions 65
 - multi-sweep algorithm 67
 - optimal way 67
 - proper 66
- application 5
- applicative form 13
 - number of 15
- B* 23
- B'* 23
- balance 73
- balanced binary tree 73
- balanced λ -expression 73
 - caf size 75
- balancing, λ -expression 78
- B, C* 23
- binary tree 14
 - balance 73
 - balanced 73
 - number of 14
- body 7
- C* 23
- C'* 23
- caf 13
 - expected form 23
 - number of 14
 - size 13
 - T* 41
 - U* 40
 - Curry's algorithm 20
 - expression cost 69
 - Statman's algorithm 54
 - Turner's algorithm 43
 - Turner's algorithm on applicative forms 41
 - Turner's algorithm on balanced expression 73
 - upperbound on number of 47
- Catalan number 14
- Church's thesis 4
- Church-Rosser property 7
- combinator 13
 - dashed 23
 - deeply reaching 23
 - redundancy of new 63
- compile-time evaluation 26, 45
- compositional evaluation 7
- constant 5
- constant applicative form 13
- cost 69
 - difference from abstraction 71
- Curry's algorithm 19
 - caf size 20
- environment 49
- evaluation 6
 - compositional 7
 - lazy 7
 - variable-free expression 8
- expected caf-form 23
- flat measure 13
- flat size 13

- free variable 6
- function body 7
- general variable-eliminating algorithm 28
- higher-order function 5
- Hughes' algorithm 80
- I* 18
- Identität 18
- identifier-length 13
- irreducible λ -expression 48
 - lower bound on number 48
- K* 18
- Konstanzfunktion 18
- λ -expression 5
 - application 5
 - balanced 73
 - balancing 78
 - body 7
 - constant 5
 - cost 69
 - evaluation 6
 - λ -introduction 5
 - irreducible 48
 - reduction 7, 8
 - size 13
 - substitution of a term for a variable 6
 - tree representation 8
 - new 30
 - variable 5
- lazy evaluation 7, 8
- maximal free expression 80
- mfe 80
- multi-sweep algorithm with abstractions 67
- new tree representation 30
 - translation to 30
- optimal algorithm 46
- optimality, attainable 46
- optimality, unattainable 45
- packing function 53
 - size 53
- reduction 7, 8
- retrieval function 53
 - size 54
- S* 18
- S'* 23
- S', B', C'* 23
- Schönfinkel's Verschmelzungsfunktion 18
 - size 13
- S, K, I* 18
- spanning tree 31
 - caf size 33
- Statman's algorithm 51
 - abstractions 68
 - caf size 54
 - expression cost 69
 - extra combinators 62
 - optimality 55
 - splitting the environment 61
- substitution 6
- super-combinator 80
- T* 26
- T'* 27
- t* 30
- tree measure 13
- tree representation 8, 30
 - translation to new 30
- tree size 13
- Turner's algorithm 26
 - balanced expressions 73
 - caf size 43
 - applicative forms 41
 - easy-to-analyze simplification 31
 - expected caf-form 23
 - extra combinators 61

- simplified 27
 - caf size 33
 - on new tree representation 31
 - on old and new representation 31
- U* 31
 - caf size 33, 40
- vaf 13
- variable applicative form 13
- variable 5
- $wcl_a(n)$ 46
 - lowerbound on 49
- worst case length 46
 - lowerbound on 49

List of Figures

1.1	Formula and tree representations of λ -terms.	9
1.2	Example tree representation.	10
1.3	The evaluation process.	11
2.1	Example λ -expression.	17
2.2	Translation of figure 2.1.	17
2.3	Example of Curry's algorithm.	18
2.4	Translations of expressions in 1.2.	19
3.1	Schönfinkel's translation of application nodes.	24
3.2	Reaching deeper down the tree.	25
3.3	An outline of the complexity analysis.	28
3.4	A more natural way of representing dashed combinators in a tree.	29
3.5	Example spanning tree.	32
3.6	Code size and spanning trees.	34
3.7	Which tree produces the largest code?	35
3.8	Differences in spanning trees from figure 3.7.	36
3.9	A worst case input.	36
3.10	Expressions from lemma 5.	38
3.11	Differences in the spanning trees from figure 3.10.	39
3.12	Cases in the proof of theorem 4.	42
4.1	The environment solution.	50
4.2	Example Statman translation.	52
5.1	Bounds from chapter 3 and 4.	58
5.2	Some figures on code-length.	58
5.3	A bad case for environment splitting in Statman's algorithm.	62
5.4	Translation of Turner's combinator set into S , K and I	63

5.5	General abstraction of subexpressions.	65
5.6	Variables should not be moved outside their scope.	66
5.7	Computing the cost difference for an abstraction.	71
5.8	Example of expression balancing by abstraction.	74
5.9	Improving on Turner's worst case behaviour.	77
5.10	A comparison in code-length.	78
5.11	Example mfe abstractions.	81