Implementing SASL without garbage collection

TECHNISCHE HOGESCHOOL EINDHOVEN

ONDERAFDELING DER WISKUNDE EN

INFORMATICA

EINDHOVEN UNIVERSITY OF TECHNOLOGY

DEPARTMENT OF MATHEMATICS AND

COMPUTING SCIENCE

Implementing SASL without garbage collection

by

J.P.H. Aerts

EUT-Report 81-WSK-05

November 1981

Contents.

IMPLEMENTING SASL WITHOUT GARBAGE COLLECTION

by

J.P.H. Aerts

Abstract

In this report an implementation technique for the language SASL is described. This technique makes use of some results from combinatory logic. SASL is described in chapter 1, and combinatory logic is the subject of chapter 2. All the results of chapter 2 are well known, and can be found in [1], [2], [4], or [5].

An implementation of SASL is also described by D.A. Turner in [9]. The implementation that we give in chapter 3 is based upon the ideas proposed in [9], and uses many of them.

In chapter 4 the most important part of this report is treated: how to avoid the use of a garbage collector at the cost of only a few extra arrangements.

The subject of chapter 5 is a mild variant of combinatory logic, and also the consequences of this variant for the implementation are described. In chapter 6 some final remarks are made.

I would like to thank J.L.A. van de Snepscheut for his many contributions to this report.

1. SASL

SASL is a mathematical notation for describing certain kinds of objects. The name SASL stands for 'St. Andrews Static Language', and this language has been proposed by D.A. Turner. In this chapter we shall give an overview of SASL. For a complete description of the language the reader is referred to [8].

In SASL a program is an expression describing some data object, and the result of a program is the printing of a representation of the object. There are six types of objects:

(i) the integers,

(ii) the truthvalues false and true,

(iii) the characters,

(iv) lists: a list is a finite or infinite sequence of objects called its components,

(v) functions: a function does not describe a data object, and therefore it is impossible to print a representation of a function,

(vi) there is a unique object 'undefined', which is the result of ill formed expressions.

Any object can be named, can be the value of an expression, can be a component of a list, can be given to a function as an argument or returned by a function as a result. So among the possibilities are a list of lists, a list of functions, a function which returns a function, and a list as an argument of a function.

Every SASL expression has an object for its value, and has no other signi-

ficance than as a way of talking about this object. It can be replaced by any other expression that has the same value without affecting the value of any larger expression of which it is a part. This property of expressions is called referential transparency.

There are six kinds of expressions:

(i) simple expressions

A simple expression is a name or a constant. Also any expression (no matter how large and complicated) can be enclosed in brackets without altering its value, and then becomes a simple expression

(ii) combinations

A combination consists of two simple expressions that are juxtaposed. The juxtaposition represents the application of a function to an argument.

(iii) operator expressions

Operator expressions are built up out of simple expressions and combinations using zero or more operators. The operators of SASL are the arithmetical operators +, -, *, div, mod, the relational operators <, ≤, =, ≠, ≥, >, and the logical operators ¬, ∨, ∧. Also there are some special operators on lists: the pairing operator :, the head operator hd, and the tail operator tl.

(iv) list expressions

A list expression is a list of operator expressions. These operator expressions have to be separated by commas to denote a list. A special notation is needed for empty lists and for lists with only one component. For a list of length one, of which c is the first and only component, the notation c, is used, and the empty list is denoted by ().

(v) conditional expressions

These are of the form

condition → object1 ; object2 ,

and the value is object1 if condition = true, object2 if condition = = false, and undefined otherwise. Syntactically, the condition can be represented by an operator expression, object1 and object2 by a conditional expression, a list expression, or an operator expression.

(vi) where-expressions

After each expression the definition of the meanings of one or more names can be given in a where-part. A where-part consists of the symbol where, followed by one or more definitions separated by semi-colons. Each name defined can be used throughout the where-expression with the given meaning, unless it is redefined in an inner where-expression, in which case its use for a different purpose in the inner where-expression in no way interferes with its use in the rest of the expression.

When names are used in an expression, the meanings of those names are defined in the where-part of the expression. These definitions can be given in several ways.

The simplest form of definition sets a single name equal to an expression, as in

x = 7 .

This is a special case of the more general form

'namelist' = 'expression' ,

where 'namelist' is a construction built up out of names, commas, and the

pairing operator.

Examples: a,b,c = 1,2,(3,4)

has the same effect as

```
a = 1 ; b = 2 ; c = 3,4 .
```

```
a : b : () = x < y → -1, false ; 1, true
```

has the same effect as

```
a = x < y → -1 ; 1 ; b = x < y → false ; true .
```

Another form of definiton is

```
'function form' = 'expression' ,
```

where 'function form' consists of the name of the function being defined, followed by one or more formal parameters. Each formal parameter is a name or a 'namelist' enclosed in brackets, and has to occur in the defining 'expression'. (If the formal parameter is a namelist, at least one of the names that occur in the namelist has to occur in the defining expression.)

Examples: id x = x

defines the identity function.

If the function gcd, defined by

```
gcd(x,y) = x > y → gcd(x - y,y) ;
           x < y → gcd(x,y - x) ;
           x
```

is applied to the list (m,n), where m and n are integers, the value of this application is 0 if m = n = 0, the greatest common divisor of m and n if m > 0 and n > 0, and undefined otherwise.

In SASL definitions can be recursive, as has been shown in the last example. Mutual recursion as in

```
odd x = x ≥ 1 ∧ even (x - 1) ;

even x = x = 0 → true ; odd (x - 1)
```

is also permitted. Not only functions, but also lists can be defined recursively.

Examples: `count a b = a > b → () ; a : count (a + 1) b`

defines the list a,a + 1,...,b.

```
from n = n : from (n + 1)
```

defines the infinite list n,n + 1,n + 2,...

An important feature of SASL is the concept of lazy evaluation. This means that an argument of a function is evaluated at most once and as late as possible. So an expression is only then evaluated, when the computation of the value of the SASL program can not be continued without knowing the value of this expression.

Examples: Consider the following SASL program:

```
hd (from 6) where from n = n : from (n + 1) .
```

In this case it is necessary to evaluate the combination 'from 6'. The first step of this evaluation gives:

```
from 6 = 6 : from (6 + 1) .
```

Without knowing the value of 'from (6 + 1)' the result of this program can be computed. So 'from (6 + 1)' will not be evaluated, and the computation of the result of this program ends with:

```
hd (6 : from (6 + 1)) = 6 .
```

The example of mutual recursion given before can be replaced by the following equivalent definitions:

```
odd x = x ≥ 1 ∧ even (x - 1) ;

even x = x = 0 ∨ odd (x - 1) .
```

We shall end this chapter with an example of a SASL program. The problem is to generate in increasing order the sequence of all positive integers that are divisable by no primes other than 2, 3, or 5. So the required SASL program has to be an expression that has this sequence in increasing order for its value. A solution to this problem is the following program (due to J.L.A. van de Snepscheut):

```
x

where x = 1 : m (m (mul 2 x) (mul 3 x)) (mul 5 x) ;

      mul n (a : l) = a * n : mul n l ;

      m (a : b) (c : d) = a < c → a : m b (c : d) ;

                          c < a → c : m (a : b) d ;

                          a : m b d .
```

2. Combinatory logic

We assume that there is an infinite set of variables, and a finite or infinite set of constants, including the combinators I, K, S, B, and C. The variables and constants are called atoms.

Definition 2.1. The set of combinatory terms is defined by induction as follows:

(i) Every atom is a combinatory term.

(ii) If X and Y are combinatory terms, then (XY) is a combinatory term.

It is assumed that the set of atoms does not intersect the set of terms of form (XY), and that (XY) ≡ (UV) implies X ≡ U and Y ≡ V. Identity of terms will be denoted by '≡'. Lower case letters will denote atoms or variables, and capitals will denote arbitrary combinatory terms, unless stated otherwise. Brackets will be omitted in such way that 'XYZ' denotes the term '((XY)Z)'.

Definition 2.2. The relation X occurs in Y (X is part of Y) is defined by induction on the construction of Y as follows:

(i) X occurs in X.

(ii) If X occurs in U or in V, then X occurs in UV.

The phrase 'x occurs in M' may be written 'x ∈ M'.

Definition 2.3. For any terms N,M, and any variable x, the result [N/x]M of substituting N for every occurrence of x in M is defined as follows by induction on the construction of M:

(i) $[N/x]x \equiv N$.

(ii) $[N/x]a \equiv a$ for all atoms $a \not\equiv x$.

(iii) $[N/x](M_1M_2) \equiv ([N/x]M_1)([N/x]M_2)$.

For mutually distinct $x_1,\ldots,x_n$, and for any terms $N_1,\ldots,N_n$, we similarly define $[N_1/x_1,\ldots,N_n/x_n]M$ to be the result of simultaneously substituting N_1 for $x_1,\ldots,N_n$ for x_n, in M.

Let X be a combinatory term. The construction of X can be described as a

tree with labeled nodes. The root of the tree will correspond to X, and therefore will have the label X. We define the corresponding tree as follows by induction on the construction of X:

(i) If X is an atom, the corresponding tree will have just one node, labeled X.

(ii) If $X \equiv UV$, the tree corresponding to X will be defined by attaching the two trees corresponding to U an V to a new root with label X as follows: the node with label U of the tree corresponding to U will be the left successor of the new root and the node with label V of the tree corresponding to V will be the right successor of the new root.

Such a tree describes the construction uniquely and completely, so that constructions with the same tree will be regarded as instances of the same construction. A construction can thus be identified with its corresponding tree.

We assign to each node Y a sequence p, called <u>position index</u> (or simply <u>position</u>) of Y, of the form $p = (p_1,\ldots,p_n)$, $n \geq 0$:

(i) If Y is the root, then $p = ()$ (i.e. $n = 0$).

(ii) If Y has position (q) and $Y \equiv UV$, then U has position (q,0) and V has position (q,1).

<u>Definition 2.4</u>. The relation $X \gg Y$ (X <u>contracts to</u> Y) is defined by induction as follows:

<u>Axiom-schemes</u>:

(I) $IX \gg X$ for all X

(K) $KXY \gg X$ for all X,Y

(S) $SXYZ \gg XZ(YZ)$ for all X,Y,Z

(B) $BXYZ \equiv X(YZ)$ for all X,Y,Z

(C) $CXYZ \equiv XZY$ for all X,Y,Z

Deduction-rules:

(μ) $X \geqslant Y$ implies $ZX \geqslant ZY$ for all X,Y,Z

(ν) $X \geqslant Y$ implies $XZ \geqslant YZ$ for all X,Y,Z .

The reflexive and transitive closure of '$\geqslant$' is denoted by '$\overset{*}{\geqslant}$'. The relation '$X \overset{*}{\geqslant} Y$' will be called '$X$ reduces to Y'.

The axiom-schemes for the combinators will be called reduction rules. Each combinator b has a unique reduction rule of the form

(*) $bx_1x_2\ldots x_m \geqslant A$ $(m > 0)$,

where m depends on b, and A is a combination of $x_1,x_2,\ldots,x_m$. This uniqueness is meant in the sense that given a combinator b there is a unique m and A such that (*) is postulated as an axiom-scheme. 'A' is called the contractum and '$bx_1x_2\ldots x_m$' a b-redex.

Lemma 2.5. Given any combinatory term X there is a unique atom v and there are unique combinatory terms $V_1,\ldots,V_n$ $(n \geq 0)$, such that $X \equiv vV_1\ldots V_n$.

Proof. (Induction on the construction of X)

1° If X is an atom a, then $v \equiv a$ and $n = 0$.

2° If $X \equiv UV$, we have by the induction hypothesis: there is a unique atom u and there are unique terms $U_1,\ldots,U_m$ $(m \geq 0)$, such that $U \equiv uU_1\ldots U_m$. Choose $v \equiv u$, $n = m + 1$, $V_i \equiv U_i$ $(1 \leq i \leq m)$, and $V_n \equiv V$. Then $X \equiv vV_1\ldots V_n$. The uniqueness follows from the assumption that $XY \equiv UV$ implies $X \equiv U$ and $Y \equiv V$, and from the induction hypothesis. □

If $X \equiv vV_1\ldots V_n$, then v is called the head of X. A redex is a combinatory

term such that there is a combinator b for which it is a b-redex. This combinator b will of course be the head of the redex. The replacement of a redex R by its contractum will be called a contraction of R.

Lemma 2.6. Let R and S be redexes that occur both in the same combinatory term. Let r, s, hr, and hs be the position indices of R, S, the head of R, and the head of S, respectively. If hr = hs, then r = s.

Proof. If hr = hs, then the head of R is the same combinator, say b, as the head of S. For this combinator b there is a unique reduction rule

(*) $bx_1 \ldots x_m \gg A \quad (m > 0)$,

and thus the value of m in (*) is uniquely determined. So $R \equiv bR_1 \ldots R_m$ and $S \equiv bS_1 \ldots S_m$. By lemma 2.5, and because hr = hs, we now conclude: $R_i \equiv S_i$, $1 \leq i \leq m$. So r = s. □

Lemma 2.7. $X \overset{*}{\gg} Y$ if and only if there is a sequence $X_0, \ldots, X_n$ $(n \geq 0)$, such that $X \equiv X_0$, $Y \equiv X_n$, and $X_{i-1} \gg X_i$, $0 < i \leq n$.

Such a sequence will be called a reduction (from X to Y). The number n is the length of the reduction, and $X_{i-1} \gg X_i$ the i-th step of the reduction.

Definition 2.8. A combinatory term is said to be in normal form if and only if it contains no redexes. If U reduces to an X in normal form, this X is called a normal form of U.

Lemma 2.9. For all combinatory terms $N_1, \ldots, N_m$, and mutually distinct variables $x_1, \ldots, x_m$:
if $X \overset{*}{\gg} Y$, then $[N_1/x_1, \ldots, N_m/x_m]X \overset{*}{\gg} [N_1/x_1, \ldots, N_m/x_m]Y$.

Proof. Induction on the proof that $X \overset{*}{\gg} Y$. Every substitution instance of an axiom of definition 2.4 is an axiom, and the same is true for the rules. □

Below we list six specifications marked (a) - (f). Each of these specifies a replacement for a component of the form [x]M. Any selection of these specifications in a prescribed order is called an algorithm. We indicate the algorithm by writing the appropriate symbols 'a' - 'f' in the prescribed order between brackets. So the algorithm (f a b) is that containing the specifications (f), (a), (b) in that order. We interpret an algorithm as meaning that a term of the form [x]M is to be replaced according to the earliest of the specifications in the prescribed order that is applicable. With this understanding, the list of specifications is as follows:

(a) $[x]M \equiv KM$ if $x \notin M$

(b) $[x]x \equiv I$

(c) $[x]Ux \equiv U$ if $x \notin U$

(d) $[x]UY_2 \equiv BU([x]Y_2)$ if $x \notin U$

(e) $[x]Y_1V \equiv C([x]Y_1)V$ if $x \notin V$

(f) $[x]Y_1Y_2 \equiv S([x]Y_1)([x]Y_2)$.

Theorem 2.10. (Law of abstraction)
For all algorithms, for all combinatory terms M and N, and for each variable x:

$$([x]M)N \overset{*}{\geqslant} [N/x]M.$$

Proof. By lemma 2.9 it is enough to prove that $([x]M)x \overset{*}{\geqslant} M$. This will be done by induction on the construction of M.

1° If M is an atom, there are only two possible cases.

(i) If $M \equiv x$, only specification (b) is applicable: $([x]x)x \equiv Ix \geqslant x$ by the reduction rule (I).

(ii) If M is an atom distinct from x, then $x \notin M$, and only specification (a)

is applicable: $([x]M)x \equiv KMx \succcurlyeq M$ by the reduction rule (K).

2° If $M \equiv UV$, we have by the induction hypothesis: $([x]U)x \twoheadrightarrow U$ and $([x]V)x \twoheadrightarrow V$.

(i) Replacing $[x]M$ according to specification (a) (if $x \notin M$) gives situation 1°(ii).

(ii) Replacing $[x]M$ according to specification (c) (if $x \notin U$ and $V \equiv x$) gives: $([x]M)x \equiv ([x]Ux)x \equiv Ux \equiv M$, and thus $([x]M)x \twoheadrightarrow M$.

(iii) Replacing $[x]M$ according to specification (d) (if $x \notin U$) gives:

$([x]UV)x \equiv BU([x]V)x$

$\succcurlyeq U(([x]V)x)$ by the reduction rule (B)

$\twoheadrightarrow UV$ by the induction hypothesis and the deduction-rule (μ).

(iv) Replacing $[x]M$ according to specification (e) (if $x \notin V$) gives a situation analogous to 2°(iii).

(v) Replacing $[x]M$ according to specification (f) gives:

$([x]UV)x \equiv S([x]U)([x]V)x$

$\succcurlyeq ([x]U)x(([x]V)x)$ by the reduction rule (S)

$\twoheadrightarrow UV$ by the induction hypothesis and the deduction-rules (μ) and (ν). □

Definition 2.11. For variables $x_1,\ldots,x_m$ (not necessarily distinct) and for each term M, define

$$[x_1,\ldots,x_m]M \equiv [x_1](\ldots([x_m]M)\ldots) .$$

Theorem 2.12. For distinct variables $x_1,\ldots,x_m$ and combinatory terms $N_1,\ldots,N_m$ and M, and for all algorithms:

$$([x_1,\ldots,x_m]M)N_1\ldots N_m \twoheadrightarrow [N_1/x_1,\ldots,N_m/x_m]M .$$

Proof. By lemma 2.9 it is enough to prove that $([x_1,\ldots,x_m]M)x_1\ldots x_m \twoheadrightarrow M$, which can easily be done by induction on m, using theorem 2.10. □

Definition 2.13. A general replacement system $(V,\Rightarrow)$ is defined as a set V together with a binary relation $\Rightarrow$ on V.

We shall write '$\overset{*}{\Rightarrow}$' for the reflexive and transitive closure of '$\Rightarrow$'.
Let CT be the set of all combinatory terms. Then $(CT,\gg)$ is a general replacement system.

Definition 2.14. A general replacement system $(V,\Rightarrow)$ has the Church-Rosser property if and only if for all elements R,X,Y of V:
$R \overset{*}{\Rightarrow} X$ and $R \overset{*}{\Rightarrow} Y$ implies there exists an element Z of V such that $X \overset{*}{\Rightarrow} Z$ and $Y \overset{*}{\Rightarrow} Z$.

Theorem 2.15. Let $(V,\Rightarrow)$ be a general replacement system, and let $\underset{1}{\Rightarrow}$ be a binary relation on V such that

(1): $\overset{*}{\underset{1}{\Rightarrow}} = \overset{*}{\Rightarrow}$

(2): for all elements R,X,Y of V: $R \underset{1}{\Rightarrow} X$ and $R \underset{1}{\Rightarrow} Y$ implies there exists an element Z of V such that $X \underset{1}{\Rightarrow} Z$ and $Y \underset{1}{\Rightarrow} Z$.

In that case $(V,\Rightarrow)$ has the Church-Rosser property.

Proof. 1° First we shall prove: if $R \underset{1}{\Rightarrow} X$ and $R \overset{*}{\underset{1}{\Rightarrow}} Y$, there exists a Z such that $X \overset{*}{\underset{1}{\Rightarrow}} Z$ and $Y \underset{1}{\Rightarrow} Z$.

(i) If $R \equiv Y$, choose $Z \equiv X$.

(ii) If $R \underset{1}{\Rightarrow} Y_1$ and $Y_1 \overset{*}{\underset{1}{\Rightarrow}} Y$, there exists by (2) a Z_1 such that $Y_1 \underset{1}{\Rightarrow} Z_1$ and $X \underset{1}{\Rightarrow} Z_1$. So $Y_1 \underset{1}{\Rightarrow} Z_1$ and $Y_1 \overset{*}{\underset{1}{\Rightarrow}} Y$. By induction we conclude: there exists a Z such that $Z_1 \overset{*}{\underset{1}{\Rightarrow}} Z$ and $Y \underset{1}{\Rightarrow} Z$. $\therefore X \overset{*}{\underset{1}{\Rightarrow}} Z$ and $Y \underset{1}{\Rightarrow} Z$.

2° Now we shall prove: if $R \overset{*}{\underset{1}{\Rightarrow}} X$ and $R \overset{*}{\underset{1}{\Rightarrow}} Y$, there exists a Z such that $X \overset{*}{\underset{1}{\Rightarrow}} Z$ and $Y \overset{*}{\underset{1}{\Rightarrow}} Z$. (This is the Church-Rosser property of $(V,\underset{1}{\Rightarrow})$.)

(i) If $R \equiv X$, choose $Z \equiv Y$.

(ii) If $R \underset{\perp}{\Rightarrow} X_1$ and $X_1 \overset{*}{\underset{\perp}{\Rightarrow}} X$, there exists by 1° a Z_1 such that $X_1 \overset{*}{\underset{\perp}{\Rightarrow}} Z_1$ and $Y \underset{\perp}{\Rightarrow} Z_1$. So $X_1 \overset{*}{\underset{\perp}{\Rightarrow}} Z_1$ and $X_1 \overset{*}{\underset{\perp}{\Rightarrow}} X$. By induction we conclude: there exists a Z such that $Z_1 \overset{*}{\underset{\perp}{\Rightarrow}} Z$ and $X \overset{*}{\underset{\perp}{\Rightarrow}} Z$. $\therefore$ $X \overset{*}{\underset{\perp}{\Rightarrow}} Z$ and $Y \overset{*}{\underset{\perp}{\Rightarrow}} Z$.

3° Finally we shall prove the Church-Rosser property of $(V,\Rightarrow)$. Suppose $R \overset{*}{\Rightarrow} X$ and $R \overset{*}{\Rightarrow} Y$. Then by (1) also $R \overset{*}{\underset{\perp}{\Rightarrow}} X$ and $R \overset{*}{\underset{\perp}{\Rightarrow}} Y$. By 2° we conclude: there exists a Z such that $X \overset{*}{\underset{\perp}{\Rightarrow}} Z$ and $Y \overset{*}{\underset{\perp}{\Rightarrow}} Z$. Then by (1) also $X \overset{*}{\Rightarrow} Z$ and $Y \overset{*}{\Rightarrow} Z$. □

Definition 2.16. (i) Let R and S be two redexes that occur in the same combinatory term. Let r be the position index of R, s the position index of S, $r = (r_1,\ldots,r_n)$ $(n \geq 0)$, and $s = (s_1,\ldots,s_m)$ $(m \geq 0)$. Then R and S are called non-overlapping redexes, if there exists a k such that $r_k \neq s_k$. If $R_1,\ldots,R_t$ $(t \geq 0)$ are redexes that occur in the same combinatory term, they are called non-overlapping redexes, if for all $0 \leq i,j \leq t$: $i \neq j$ implies R_i anf R_j are non-overlapping.

(ii) A special contraction is defined as a simultaneous contraction of 0 or more non-overlapping redexes. If Y is the result of a special contraction of X, we shall write '$X \underset{s}{\geqslant} Y$'. The reflexive and transitive closure of '$\underset{s}{\geqslant}$' will be denoted by '$\overset{*}{\underset{s}{\geqslant}}$'. If $X \overset{*}{\underset{s}{\geqslant}} Y$, then X reduces to Y by a special reduction.

Theorem 2.17. $X \overset{*}{\geqslant} Y$ if and only if $X \overset{*}{\underset{s}{\geqslant}} Y$.

Proof. 1° A special contraction is a simultaneous contraction of zero or more non-overlapping redexes. So: if $U \geqslant V$, then $U \underset{s}{\geqslant} V$.

$\therefore$ If $X \overset{*}{\geqslant} Y$, then $X \overset{*}{\underset{s}{\geqslant}} Y$.

2° (i) First we shall prove: if $U \geqslant_s V$, then $U \overset{*}{\geqslant} V$.

Suppose $U \geqslant_s V$. Then U contains non-overlapping redexes $R_1,\ldots,R_k$ $(k \geq 0)$, and V is the result of contracting these k redexes simultaneously. Define $U_0 \equiv U$, and let U_i be the result of contracting R_i in U_{i-1} $(1 \leq i \leq k)$. (Note that U_i contains the redexes $R_{i+1},\ldots,R_k$.) Then $U_{i-1} \geqslant U_i$ $(1 \leq i \leq k)$, and $U_k \equiv V$. So $U \overset{*}{\geqslant} V$.

(ii) Suppose $X \overset{*}{\geqslant}_s Y$. Then there are $X_0,\ldots,X_n$ $(n \geq 0)$ such that $X \equiv X_0 \overset{*}{\geqslant} X_1 \overset{*}{\geqslant} \ldots \overset{*}{\geqslant} X_n \equiv Y$. So $X \overset{*}{\geqslant} Y$. □

Theorem 2.18. If $U \geqslant_s X$ and $U \geqslant_s Y$, then there exists a Z such that $Y \geqslant_s Z$ and $X \geqslant_s Z$.

Proof. (Induction on the construction of U)

1° If U is an atom, then $U \equiv X \equiv Y$. So choose $Z \equiv U$.

2° Let $U \equiv U_1U_2$, and suppose $U \geqslant_s X$ by the simultaneous contraction of the non-overlapping redexes $R_1,\ldots,R_k$ $(k \geq 0)$ and $U \geqslant_s Y$ by the simultaneous contraction of the non-overlapping redexes $Q_1,\ldots,Q_m$ $(m \geq 0)$. Define $RR = \{R_i \mid 1 \leq i \leq k\}$ and $QQ = \{Q_i \mid 1 \leq i \leq m\}$.

(i) Suppose $R_1 \equiv U$ and $Q_1 \equiv U$. Then $k = 1$ and $m = 1$. $\therefore R_1 \equiv Q_1$, and therefore $X \equiv Y$. So choose $Z \equiv X$.

(ii) Suppose $R_1 \equiv U$ and for all $1 \leq i \leq m$: $Q_i \not\equiv U$. Then $k = 1$.

(I) Suppose $U_1 \equiv I$. Then $U \equiv U_1U_2 \equiv IU_2 \geqslant_s U_2 \equiv X$, and for all $1 \leq i \leq m$: $Q_i \in U_2$. Define Z as the result of the simultaneous contraction of $Q_1,\ldots,Q_m$ in U_2. Then $U \equiv IU_2 \geqslant_s IZ \equiv Y$.
$\therefore Y \equiv IZ \geqslant Z$ and $X \equiv U_2 \geqslant_s Z$.

(K) Suppose $U_1 \equiv KV$. Then $U \equiv U_1U_2 \equiv KVU_2 \geqslant_s V \equiv X$. Define QV as the set of non-overlapping redexes that are element of QQ and part

of V, and QU_2 as the set of non-overlapping redexes that are element of QQ and part of U_2. Define Z as the result of the simultaneous contraction of the elements of QV in V, and define Z' as the result of the simultaneous contraction of the elements of QU_2 in U_2. Then $V \underset{s}{\geqslant} Z$ and $U_2 \underset{s}{\geqslant} Z'$. So $U \equiv U_1U_2 \equiv KVU_2 \underset{s}{\geqslant} KZZ' \equiv Y$. $\therefore X \equiv V \underset{s}{\geqslant} Z$ and $Y \equiv KZZ' \geqslant Z$.

(S) Suppose $U_1 \equiv SV_1V_2$. Then $U \equiv U_1U_2 \equiv SV_1V_2U_2 \underset{s}{\geqslant} V_1U_2(V_2U_2) \equiv X$. Define QV_i as the set of non-overlapping redexes that are element of QQ and part of V_i $(i = 1,2)$, and define QU_2 as the set of non-overlapping redexes that are element of QQ and part of U_2. Define V_i' as the result of the simultaneous contraction of the elements of QV_i in V_i $(1 = 1,2)$. Then $V_1 \underset{s}{\geqslant} V_1'$ and $V_2 \underset{s}{\geqslant} V_2'$. Define U_2' as the result of the simultaneous contraction of the elements of QU_2 in U_2. Then $U_2 \underset{s}{\geqslant} U_2'$. $\therefore U \equiv SV_1V_2U_2 \underset{s}{\geqslant} SV_1'V_2'U_2' \equiv Y$. Define $Z \equiv V_1'U_2'(V_2'U_2')$. Then $Y \geqslant Z$, and $X \underset{s}{\geqslant} Z$ by the simultaneous contraction of all the occurrences of elements of $QV_1 \cup QV_2 \cup QU_2$ in X.

(B) and (C): analogous to (S).

(iii) Suppose $Q_1 \equiv U$ and for all $1 \leq i \leq k$: $R_i \not\equiv U$. This case is analogous to situation (ii).

(iv) Suppose for all $1 \leq i \leq k$: $R_i \not\equiv U$, and for all $1 \leq i \leq m$: $Q_i \not\equiv U$. By defintion 2.16 (ii) there exist an X_1 and an X_2 such that $U_1 \underset{s}{\geqslant} X_1$ and $U_2 \underset{s}{\geqslant} X_2$ and $X \equiv X_1X_2$. Also there exist an Y_1 and an Y_2 such that $U_1 \underset{s}{\geqslant} Y_1$ and $U_2 \underset{s}{\geqslant} Y_2$ and $Y \equiv Y_1Y_2$. Then by the induction hypothesis there exist a Z_1 and a Z_2 such that $X_i \underset{s}{\geqslant} Z_i$ and $Y_i \underset{s}{\geqslant} Z_i$ $(i = 1,2)$. Choose $Z \equiv Z_1Z_2$, then $X \equiv X_1X_2 \underset{s}{\geqslant} Z_1Z_2 \equiv Z$ and $Y \equiv Y_1Y_2 \underset{s}{\geqslant} Z_1Z_2 \equiv Z$. □

Theorem 2.19. Let CT be the set of all combinatory terms. Then $(CT, \trianglerighteq)$ has the Church-Rosser property.

Proof. This theorem is an immediate consequence of the theorems 2.15, 2.17, and 2.18. □

Corollary 2.19.1. A combinatory term can have at most one normal form.

Definition 2.20. We say that $X = Y$ holds if and only if this statement can be deduced form the axiom-schemes (I), (K), (S), (B), (C), and the deduction-rules (μ), (ν) of definition 2.4 (with '=' instead of '$\trianglerighteq$'), together with the rules:

(ρ) $X = X$ for all X

(σ) $X = Y$ implies $Y = X$ for all X,Y

(τ) $X = Y$ and $Y = Z$ implies $X = Z$ for all X,Y,Z .

Lemma 2.21. The equality relation '=' defined in definiton 2.20 is the equivalence relation generated by the relation '$\trianglerighteq$'.

Theorem 2.22. If $X = Y$, then there is a Z such that $X \trianglerighteq^{*} Z$ and $Y \trianglerighteq^{*} Z$.

Proof. This is another form of the Church-Rosser property. □

Corollary 2.22.1. If $X = Y$ and Y is in normal form, then $X \trianglerighteq^{*} Y$.

Corollary 2.22.2. If $X = Y$, then either X and Y do not have normal forms, or they both have the same normal form.

Corollary 2.22.3. If X and Y are both in normal form, then $X \neq Y$ implies $X \neq Y$.

Let C be a construction of X, let R be a redex in X with contractum T, and let Y be the result of replacing R by T in X.

There is a subtree CR of C that is a construction of R. Let CT be a construction of T. Then we define the tree C' as follows:

(i) replace CR in C by CT,

(ii) replace R in the labels of every node of C by T.

Then C' is a construction of Y.

Let S be any redex in X. Let $\mathbf{r} = (r_1,\ldots,r_n)$ $(n \geq 0)$ be the position index of R and $s = (s_1,\ldots,s_m)$ $(m \geq 0)$ the position index of S.

Then we define the residuals of S with respect to R as a set of components of Y as follows:

(i) If $r = s$, S has no residuals.

(ii) If there is a k such that $r_k \neq s_k$ (R and S are non-overlapping redexes), S has a unique residual: the node in C' with position s.

(iii) If $n > m$ and $r_i = s_i$, $1 \leq i \leq m$, S has a unique residual: the node in C' with position s.

(iv) Let $m > n$ and $r_i = s_i$, $1 \leq i \leq n$. Let $R \equiv bU_1\ldots U_k$, then S is part of a U_p for certain p. This U_p takes the place of x_p in (*) (see page 10). Then T will contain an occurrence of U_p for each occurrence, if any, of x_p in A; in each such occurrence there will be an occurrence of S. These occurrences of S are the residuals of S.

The definition of residuals of S can be extended to define residuals with respect to a reduction of arbitrary length n as follows:

- If the length n is 0, every redex is its own residual.
- If $n > 0$ and the residuals of S with respect to the first $n - 1$ steps have been defined, the residuals of S with respect to the first n steps will be the residuals of those residuals with respect to the redex contracted in the n-th step.

We call a redex R senior to a redex S just when the head of R lies to the left of the head of S.

Let $X \equiv X_0 \gg \ldots \gg X_n \equiv Y$ $(n \geq 0)$ be a reduction. If the redex contracted in the k-th step of the reduction is R_k, the reduction is a standard reduction just when no R_{k+1} for $k > 0$ is a residual of a redex R_k' that is senior to R_k.

A redex R in a combinatory term U is called a head redex if and only if the head of R coincides with the head of U; any other redex in U is called an internal redex. The contraction of a head redex will be called a head contraction, and a reduction consisting exclusively of such contractions a head reduction; similarly for internal contractions and internal reductions. A reduction in which no internal contraction precedes a head contraction will be called a semistandard reduction.

Theorem 2.23. If D is a reduction from X to Y, then there is a semistandard reduction D' from X to Y.

Proof. Suppose the reduction D reduces X to Z by a reduction D_1, and then reduces Z to Y by an internal reduction D_2. If we can find a semistandard reduction D_1' from X to Z, then the reduction D_1' followed by D_2 will be a semistandard reduction from X to Y. This will always be the case if D_1' is void (that is if $Z \equiv X$).

This argument shows that it is sufficient to prove the theorem on the hypothesis that D ends with a head reduction. We shall do this by induction on the number of internal contractions preceding this final head reduction.

1° If there are no such internal contractions, D is a head reduction, and so it is itself the D' sought.

2° Suppose now that D consists of a reduction D_1, followed by a one step

reduction D_2 in which an internal redex R is contracted, followed in turn by a head reduction D_3. (So D_2 is the last internal contraction before the final head reduction.) Let D_1 reduce X to Z_0, D_2 reduce Z_0 to Z_1, and let D_3 be $Z_1 \gg Z_2 \gg \ldots \gg Z_n$ $(n \geq 2)$, where $Z_n \equiv Y$.

Let D_2' be the head reduction constituted as follows:

(i) Let it start with $Z_1' \equiv Z_0$. Then for $i = 1$ we have a Z_i' such that:

(a) the head of Z_i' is the same as the head of Z_i,

(b) the residuals of R consist of a non-overlapping set of internal replicas of R,

(c) contraction of all the residuals of R forms an internal reduction leading form Z_i' to Z_i.

(ii) Let there be a Z_i' such that (a), (b), and (c). If $i < n$, let Z_{i+1}'' be formed by contracting the head redex of Z_i'.

- If no residual of R is at the head of Z_{i+1}'', we let $Z_{i+1}' \equiv Z_{i+1}''$.
- If there is a residual R' of R at the head of Z_{i+1}'', we form Z_{i+1}' by contracting R'.

In either case Z_{i+1}' has properties (a), (b), and (c).

Thus we continue until we reach Z_n', when D_2' stops. $\therefore$ D_2' is a head reduction.

Let D_3' be the internal reduction from Z_n' to $Z_n \equiv Y$. (D_3' is the internal reduction formed by contracting all the residuals of R in Z_n'.)

By the induction hypothesis there is a semistandard reduction D'' from X to Z_n'. This reduction D'' followed by D_3' will be the D' sought. □

Theorem 2.24. (Standardization theorem)

If $X \overset{*}{\gg} Y$, there is a standard reduction from X to Y.

Proof. By theorem 2.23 we may suppose that the given reduction $X \overset{*}{\gg} Y$ is semistandard. Let Z be the last stage that is reached by head contractions alone. Then $Z \equiv U_0U_1\ldots U_p$, where U_0 is an atom (lemma 2.5). Therefore $Y \equiv U_0V_1\ldots V_p$, where $U_i \overset{*}{\gg} V_i$ $(1 \leq i \leq p)$.

Now apply the same procedure to each of the reductions $U_i \overset{*}{\gg} V_i$, and so on. This amounts to using a structural induction on Y. The basic step of this induction is the case where $Y \equiv Z \equiv U$, in which case the reduction $X \overset{*}{\gg} Y$ is a head reduction.

Each V_i has fewer atoms than Y; so eventually the induction process must come to an end. □

A special case of standard reduction is normal order reduction, in which each step is determined by contraction of the leftmost redex i.e. the (unique) redex senior to all other redexes in a term.

Theorem 2.25. If X has a normal form Y, X can always be reduced to Y by normal order reduction.

Proof. If Y is in normal form, Y contains no redexes. By theorem 2.24 there is a standard reduction $X \equiv X_0 \gg \ldots \gg X_n \equiv Y$ $(n \geq 0)$ from X to Y. Let the redex contracted in the k-th step of this standard reduction be R_k, and suppose there is an R_k for $k > 0$ that is not the leftmost redex in X_{k-1}. Let R_k' be the leftmost redex in X_{k-1}. Then the residual of R_k' with respect to the reduction $X_{k-1} \gg X_k \gg \ldots \gg X_n \equiv Y$ will be the leftmost redex in Y, because this reduction is a standard reduction. ∴ Y contains a redex.

∴ Contradiction. □

3. The implementation of SASL

We shall use the results from combinatory logic for the implementation of

SASL. In an applicative language, such as SASL, bound variables play an important part. They occur as formal parameters like the 'n' in:

fac n = n = 0 → 1 ; n * fac(n - 1) ,

or as local variables like the 'x' in:

(x + 1) + (x - 2) where x = 7 .

Each SASL program will be translated into a form from which all bound variables have been removed.
In the previous chapter we have described a method to remove variables from a combinatory term. A combinatory term is either an atom or it is of the form XY, where X and Y are combinatory terms (definiton 2.1). The juxtaposition of X and Y denotes the application of a monadic function X to its argument Y. Functions of several arguments are adapted to this uniformly monadic syntax by replacing them by higher-order functions of one argument. So for example a first-order function F(x,y) of two arguments is replaced by a second-order function f so that f x y has always the same value as F(x,y). Fully bracketed we have the combinatory term ((f x) y), where f is a monadic function that, when applied to an argument x, returns another monadic function f x. After the logician H.B. Curry this transformation is called 'currying', although the idea has been used for the first time in [6].

A SASL program is always an expression. The syntax of SASL gives:

<program> ::= <expression>

<expression> ::= <expression-part> <where-part>

<where-part> ::= <empty> | where <definitions> .

In the expression-part identifiers may occur as names of local variables,

which are defined in the wher-part, and as names of standard functions. Now we shall describe the translation of a SASL program into a form from which all local variables have been removed. This transformation consists of two steps.

Step 1. In order to remove local variables from an expression-part this expression-part has to be transformed to a combinatory term. This is done by replacing all operators that occur in the expression-part by their curried versions (prefix form). We shall use the same symbol for a SASL operator and for its curried version, unless stated otherwise.

Examples: x + y is replaced by ((+ x)y).

The defintion of the function fac given at the beginning of this chapter is transformed to:

fac n = (((→((=n)0))1)((*n)(fac((-n)1)))) .

Step 2. Now it is possible to abstract variables from the result of step 1. The algorithm (b c d e f), which has been described in the previous chapter, will be used for the abstraction operations. Specification (a) will never be used, because a variable will never be abstracted from a term in which it does not occur.

Suppose an identifier x occurs as a local variable in an expression-part E. Then there are only two possible situations:

E where 'definition of x' (1),

or:

E_1 where f = E ; 'definition of x' (2),

with E_1 an expression-part and f a variable.

The first thing to be done is to transform the definition of x to a

form:

$$x = E^* ,$$

where E* is a combinatory term from which all bound variables (local variables and formal parameters) have been removed. Then the translation of (1) is:

$$([x]E)E^* ,$$

and the translation of (2) is:

$$E_1 \text{ where } f = ([x]E)E^* .$$

This transformation has to be performed for each local variable occurring in E.

We still have to describe how the definition of x is transformed to a form

$$'x = E^{*\prime} .$$

The definition of a variable will be given in a definition of a where-part. This definition can be given in several ways (see chapter 1). First a special form will be treated, and then it will be shown how each definition can be transformed to that form.

Suppose the definition is of the form:

$$f\ x_1 \ \ldots\ x_n = E \qquad (3),$$

with $n \geq 0$ and x_i an identifier ($1 \leq i \leq n$), and suppose that this definition is not part of any pair of mutually recursive definitions. The identifiers $x_1,\ldots,x_n$ are called the formal parameters of the definition. The local variables of expression E are those variables that occur in E and that are defined in the where-part of E or in the where-part of which (3) is a part. All the other variables occurring in E are called global variables.

The expression E is translated into a form from which all local variables

have been removed by the steps 1 and 2. The result of this translation is called E', and so now we have:

$$f\ x_1 \ldots x_n = E' \ .$$

This form is translated into:

$$f = [x_1](\ldots([x_n]E')\ldots)$$

using the abstraction algorithm (b c d e f). Specification (a) is not needed, since each formal parameter x_i has to occur in E'.
If $f \notin E'$, the required form 'f = E*' is:

$$f = [x_1](\ldots([x_n]E')\ldots) \ .$$

If $f \in E'$, an extra abstraction has to be performed. The definition of f is called recursive in this case. The required form is:

$$f = (Y([f]([x_1](\ldots([x_n]E')\ldots)))) \ ,$$

where Y is the fixed-point combinator. Y is a solution of the equation

$$Yf = f(Yf) \qquad (4),$$

where '=' is the equality relation defined in definition 2.20.

Theorem 3.1. $Y \equiv WS(BWB)$, where $W \equiv CSI$, is a solution of $Yf = f(Yf)$.

Proof. 1° $Wfx = CSIfx = SFIx = fx(Ix) = fxx$.

2° $WS(BWB)f = S(BWB)(BWB)f$ by 1 .

So $WS(BWB)f = BWBf(BWBf)$ (*).

$\therefore WS(BWB)f = BWBf(BWBf) = W(Bf)(BWBf) = (Bf)(BWBf)(BWBf)$ by 1° .

So $WS(BWB)f = f(BWBf(BWBf)) = f(WS(BWB)f)$ by (*). □

There are three situations in which a definition has to be transformed to the form we have discussed before.

(i) A list occurs on the left of the equality sign '=' in a definition.

An example is:

E_1 where a : b = E_2 , with a $\in E_1 \vee$ b $\in E_1$.

In such a situation the list is given a new name, and each occurrence of the old identifiers is replaced by this new name using the operators hd and tl.

So in our example the list a : b is given a new name , say ℓ, and in E_1 and E_2 (and possibly in other definitions of the where-part) each occurrence of 'a' is replaced by '(hd ℓ)' and each occurrence of 'b' by '(tl ℓ)'.

(ii) A list occurs as a formal parameter.

An example is:

f (x,y,z) = E , with x $\in$ E $\vee$ y $\in$ E $\vee$ z $\in$ E, and E an expression-part.

In such a situation a new name for the list is introduced, and the old identifiers are defined in a where-part of E using the new name and the operators hd en tl.

The results of this tranformation performed on our example is:

f ℓ = E where x = hd ℓ ; e = tl ℓ ; y = hd e ; z = hd (tl e) .

(iii) Mutual recursion following a 'where' is handled by combining all definitions concerned into a single definition.

For example:

E where f x = E_1 ; g = E_2 , with g $\in E_1 \wedge$ f $\in E_2 \wedge$ (g $\in$ E $\vee$ f $\in$ E),

is transformed to:

E where (f,g) = ([x]E_1,E_2) .

On this form the transformations of (i) are applicable.

In this way each mutually recursive pair of definitions can be converted into a single recursive definition.

The final result of all these transformations is a combinatory term in which the only occurring identifiers are the names of standard functions. This term is reasonably compact if it is the result of a single abstraction operation. When the abstraction operation is used n times, however, the number of combinators that occur in the resulting term is $0(n^2)$. In order to avoid this growth of the resulting term a few optimizations may be introduced into the abstraction algorithm. These optimizations are fully described in [10].

Using the foregoing rules each SASL program can be transformed to a combinatory term from which all identifiers have been removed, except the names of standard functions. In this combinatory term functional application is of course the only operation. The term is stored as a binary structure representing these functional applications.
Each node of the structure consists of two fields, and each field contains either a constant or a pointer to a substructure. Of a node x the two fields are called 'x.ℓ' and 'x.r'. If the left field x.ℓ contains a constant c, we shall write 'x.ℓ = c', and if x.ℓ contains a pointer to a node y, we shall write 'x.ℓ = ↑y'. For the right field x.r we shall use of course the same notation.

The binary structure representing a combinatory term is defined as follows (by induction on the construction of a term):

(i) If the combinatory term to be represented is a constant c, the structure consists of just one node, called root, and root.ℓ = I and root.r = c.

(ii) If the combinatory term to be represented is of the form XY, the structure consists of a node, called root, and

- if X is a constant, then root.ℓ = X,
- if X is not a constant, then root.ℓ = $\uparrow$x, where x is the root of the structure that represents X.

If X is not a constant, X is the name of a standard function, or of the form $X \equiv X_1X_2$. The field root.r is defined analogously.

A few remarks must be made at this place.

1. If in a combinatory term the name of a standard function occurs n times, there will be n pointers to the representation of that function. So these substructures are included by pointing and not by copying. The same thing will be done during the abstraction process of local variables. The translation of

   E <u>where</u> x = E^* , with x ϵ E ,

   is:

   ([x]E)T ,

   where T stands for E^* if x $\notin$ E^*, and for (Y([x]E^*)) if x ϵ E^*.
   When no abstraction operations are to be performed on T during the rest of the translation of the complete SASL program, T remains unchanged (T occurs in the final result), and the structure representing T will not be copied, but pointed to.

2. Recursive definitions have been handled with the use of the combinator Y. We know by (4):

   $$Yh = h(Yh) = h(h(Yh)) = \ldots = h(h(h(\ldots))) = h^{\infty} .$$

As reduction rule for Y we now choose

$$Yh \gg h^{\infty} .$$

Since a combinatory term has been represented as a binary structure, the reduction rules performed on it will be transformations of this structure. When the reduction rule for Y is performed on a node x with $x.\ell = Y$ and $x.r = h$, the result will be $x.\ell = h$ and $x.r = \uparrow x$. Then node x represents the contractum h^{∞}.

The translation of

$$E_1 \underline{\text{where}}\ f = E_2 \text{ , with } f \in E_1 \wedge f \in E_2 ,$$

is:

$$([f]E_1)(Y([f]E_2)) .$$

The representation of the function f is a node, say reprf, with $reprf.\ell = Y$ and $reprf.r = e2$, where e2 stands for the representation of $[f]E_2$. So when the reduction rule for Y has been performed on reprf, the result is: $reprf.\ell = e2$ and $reprf.r = \uparrow reprf$. We know

$$([f]E_2)f \overset{*}{\gg} E_2 \qquad (5)$$

by theorem 2.10. The term f in $([f]E_2)f$ is represented by a pointer to the node reprf. Therefore the result of reduction (5) will be the structure representing E_2 in which for each occurrence of f a pointer to reprf has been placed.

This means that while constructing the binary structure that represents a recursive definition, the fixed-point combinator Y will not always have to be used. In those situations where the abstraction of f is the last abstraction operation performed on E_2 (that is when no abstraction operations are performed on $[f]E_2$ during the rest of the translation of

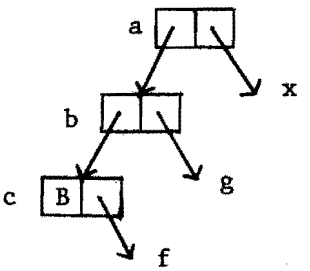
the complete SASL program), a pointer to the root of the whole structure representing the definition of f may be placed for each occurrence of f. This method has the advantage that the reduction rule for Y and all steps of reduction (5) will not be performed.

3. We have chosen as reduction rule for Y: $Yh \gg h^{\infty}$. Another possibility is to choose: $Yh \gg h(Yh)$. Then $Yh \overset{*}{\gg} h^{\infty}$ after an infinite number of steps. Our reduction rule produces the form h^{∞} directly, with a considerable gain in efficiency.

So a combinatory term, representing a SASL program, is represented by a binary directed graph. We shall now describe a reduction machine, which progressively transforms this graph by applying the reduction rules (given in appendix 1) until the graph has been reduced to a printable object, If the program has a list for its value, the components of this list will be reduced to printable objects, one after another.
The left-hand side of each rule is a redex; the right-hand side the corresponding contractum. All these reduction rules are implemented as graph-transformations. The node or field that represents the redex is overwritten with the representation of the contractum.

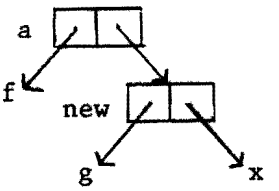
Example: Let a, b, c, x, f, and g be nodes, and $a.\ell$ = ↑b, a.r = ↑x, $b.\ell$ = ↑c, b.r = ↑g, $c.\ell$ = B, and c.r = ↑f. Then node a represents the redex Bfgx.

Figure 1: The situation before the transformation.



If the rule for B is applied on node a, the result is: a.ℓ = ↑f, a.r = ↑new, new.ℓ = ↑g, and new.r = ↑x, where new is a new node. The nodes b and c remain unchanged.

Figure 2: The situation after the transformation.



A problem arises if the contractum is an atom or an already existing expression. In the first case the problem is that a two-field node has to be overwritten with an object that can only occupy one field; in the second case, which can occur for example when applying the rule for hd, we have a problem because if the subject node is overwritten with a copy of the node that represents the result, the sharing properties, on which the efficiency of the graph-transformations depends, will be destroyed. The solution in both cases is to make the node into an indirection node with the combinator I in its left field and the result in question (an atom or a pointer to an existing expression) in its right field.

As long as the graph contains a redex, graph-transformations will be carried out. So the final result of the reduction process will not contain any redex, and therefore it is in normal form (definition 2.8). We know by theorem 2.19 that this result is independent of the order in which the reduction rules are applied. The ordering rule that is used, is that at each stage of the reduction process the contraction of the leftmost redex is carried out. This is so called 'normal order reduction' (see chapter 2). Theorem 2.25 (and the finite time in which each contraction is performed) gives that if a combinatory term has a normal form, it can always be reduced to that normal form by normal order reduction. To schedule the sequence of contractions in normal order we use the same technique as D.A. Turner in [9].

In general arguments are substituted into the body of a function in unevaluated form (lazy evaluation), because initially the redexes inside the argument are to the right of other redexes. After this substitution all occurrences of the argument in the body of the function will be pointers to a shared subgraph, and therefore the argument will be evaluated at most once.

4. Store management

In the previous chapter we have described how a SASL program is represented by a directed graph, which has one special node, called its root. In general the graphs are cyclic, because of the way we handle recursion. On such a graph transformations are being performed.

Suppose a SASL program is represented by a directed graph G = (N,P), where N is the set of nodes and P the set of pointers. If p is a pointer from node v to node w, we shall write 'p: $v \rightarrow w$'.

Definition 4.1. A path is a sequence $(v_0,a_1,v_1,a_2,\ldots,a_n,v_n)$, $n \geq 0$, such that: - $v_i \in N$, $0 \leq i \leq n$,

- $a_i \in P$ and $a_i: v_{i-1} \rightarrow v_i$, $0 \leq i \leq n$.

If q is a path from node x to node y, we shall write 'q: $x \overset{*}{\rightarrow} y$'.
Each path $(v_0,a_1,v_1,a_2,\ldots,a_n,v_n)$ with $n \geq 1$, $v_0 = v_n$, and
$\underline{A}\ i,j: 0 \leq i < n, 0 \leq j < n: i = j \vee v_i \neq v_j$ is called a cycle.
The nodes v_i, $0 \leq i \leq n$, and the pointers a_i, $0 < i \leq n$, are said to be 'part of the cycle'.

Definition 4.2. A node is called reachable if there exists at least one path from the root to that node.

It is possible that after a transformation some nodes will not be reachable any longer. These nodes will never be used again, and therefore they are considered to be 'garbage'. In order to avoid the use of a mark scan garbage collector some extra arrangements have to be made.
We assume that there is a list of free nodes. If a new node has to be created, one of the nodes of this list will be used, and if a node is no longer reachable, this node has to be added to the list. The problem is to find out which nodes are still reachable after a graph-transformation.
If the directed graph contains no cycles initially and remains acyclic during the transformation process, the problem is easy to solve. Because initially each node is reachable, it is enough to know the number of incoming pointers of each node. Then we have:

> a node is reachable if and only if it is the root or its number of incoming pointers is greater than zero (*).

Problems arise if the graph contains cycles. For nodes that are not part of any cycle (*) is valid. A node that is part of a cycle, however, is not always reachable.

The occurrence of cycles could be avoided if each form of recursion was handled by the use of the fixed-point combinator Y, and if the reduction rule for Y was $Yf \gg f(Yf)$ instead of $Yf \gg f^{\infty}$. The graphs would then be and remain acyclic. This would cause, however, a considerable loss of efficiency, especially in those cases of recursion where we have not used the fixed-point combinator.

In order to treat the root of the directed graph as an ordinary root, we shall extend the graph with an extra node. This extra node has only one field, which contains a pointer to the root. The extra node will not be

changed by any graph-transformation, and no pointer to the extra node will ever be created.
It is easy to see that a node x is reachable if and only if there is at least one path from the extra node to node x. From now on this extra node will be called the root of the directed graph. So definition 4.2 will not have to be changed.

Lemma 4.3. A node x, not the root, is reachable if and only if there is another node that is reachable, and that contains a pointer to x.

A cycle is called reachable, if each node of the cycle is reachable.

Theorem 4.4. If one node of a cycle is reachable, all nodes of the cycle are reachable.
Proof. This follows directly from lemma 4.3. □

Definition 4.5. Let G be a directed graph. Suppose that for each pair of nodes v, w in G there exist paths p: $v \overset{*}{\rightarrow} w$ and q: $w \overset{*}{\rightarrow} v$. Then G is said to be strongly connected.

Definition 4.6. Let G = (N,P) be a directed graph. We define an equivalence relation on N as follows: two nodes v and w are equivalent if there exist paths p: $v \overset{*}{\rightarrow} w$ and q: $w \overset{*}{\rightarrow} v$. Let the distinct classes under this relation be N_i, $1 \leq i \leq n$. Define for each i, $1 \leq i \leq n$, the subgraph $G_i = (N_i, P_i)$ with $P_i = \{p \in P \mid p: v \rightarrow w, v \in N_i, w \in N_i\}$.

Lemma 4.7. (i) Each G_i is strongly connected.

(ii) No G_i is a proper subgraph of a strongly connected subgraph of G.

The subgraphs G_i are called the strongly connected components of G. We shall often write 's.c.c.' instead of 'strongly connected component'.

Theorem 4.8. Let K be an s.c.c.. Then either all nodes of K are reachable or all nodes of K are not reachable.

Proof. Let x be a node of K. Then x is either reachable or not. Now let y be any other node of K. Then there are paths from x to y and from y to x. Hence, y is reachable if and only if x is reachable. □

This theorem allows us to speak about the reachability of a strongly connected component.

Definition 4.9. For each s.c.c. $G_i = (N_i, P_i)$ we define the set S_i of incoming pointers of G_i as follows:

$$S_i = \{p \in P \mid p: v \to w \text{ with } w \in N_i,\ v \notin N_i, \text{ and } v \text{ is reachable}\}.$$

We shall denote the number of elements of a set V with '$|V|$'.

Theorem 4.10. The s.c.c. G_i is reachable if and only if $|S_i| > 0$.

Proof. 1° Suppose G_i is reachable. Let $x \in N_i$, then x is reachable. So there exists a path from the root to x. This path must contain a node $y \in N_i$, a node $v \notin N_i$, and a pointer $p: v \to y$. Of course v is reachable too, and so $p \in S_i$.

2° Suppose $|S_i| > 0$. Then there exists a pointer $p: v \to w$ such that $w \in N_i$, $v \notin N_i$, and v is reachable. So there exists a path from the root to v. If we extend this path with pointer p and node w, we have a path from the root to w. So w is reachable, and by theorem 4.8 G_i is reachable. □

The following arrangements will be made.

1° Local administration.

Each node x must contain the following information:

- its number ap(x) of incoming pointers,

- its number cp(x) of incoming pointers that are element of P_i, where $G_i = (N_i, P_i)$ is the s.c.c. with $x \in N_i$.

If cp(x) > 0, the local administration of x has to contain the information to which s.c.c. x belongs.

2° Global administration.

Of each s.c.c. G_i with $P_i \neq \emptyset$ the number of incoming pointers ($=|S_i|$) has to be recorded. Note that for an s.c.c. G_j with $N_j = \{x\}$: $|S_j| = ap(x) - cp(x)$.

Now we shall describe the remove-operation, which will be used almost each time that an s.c.c. is found no longer reachable.

When an s.c.c. G_i is no longer reachable, three things have to be done:

1° Add all nodes of G_i to the list of free nodes.

2° Remove each pointer that is an element of $\{p \in P \mid p: v \to w \text{ with } v \in N_i \wedge w \notin N_i\}$.

3° If the removals of 2° have as a result that another s.c.c. is no longer reachable, perform the remove-operation on that component.

When G_i is found no longer reachable, always as least one node of G_i is known. The remove-operation will start with that node.

Each node $x \in N_i$ will be treated as follows. First x has to be added to the list of free nodes. Then the pointers that x contains have to be treated.

Let $p: x \to y$, then there are three possibilities:

(i) Suppose $y \in N_j$ and $j \neq i$. Then p has to be removed from S_j. If $|S_j| = 0$ after this removal, the remove-operation has to be performed on G_j starting with node y.

(ii) Suppose $y \in N_i$. Then y has to be removed from the graph. Therefore y

has to be treated in the same way as x.

(iii) Suppose y is an element of the list of free nodes. In that case y has been an element of N_i, but has already been removed. So in this situation nothing has to be done.

Suppose the s.c.c. K has to be removed, and v is the node of K where the remove-operation will start. Let lf be the list of free nodes. Let sr be the set of pairs (G_i,x) such that $G_i = (N_i,P_i)$ is an s.c.c. that has to be removed, and $x \in N_i$. Then we have the following algorithm for the remove-operation:

```
proc remove(G_i,x):
  var sn : set of nodes;
  proc treat(z):
    proc checkfield(t):
      if t is a constant → skip
       [] t is a pointer from z to y →
                          if y∈N_j ∧ j≠i → |S_j|:=|S_j|-1;ap(y):=ap(y)-1;
                                           if |S_j|=0 → sr:=sr∪{(G_j,y)}
                                            [] |S_j|>0 → skip
                                           fi
                           [] y∈N_i → sn:=sn∪{y}
                           [] y∈lf → skip
                          fi
      fi
    corp;
    lf:=lf∪{z};checkfield(z.ℓ);checkfield(z.r);sn:=sn\{z}
  corp;
  sn:={x};do sn≠∅ → treat(an element of sn) od;sr:=sr\{(G_i,x)}
```

corp;

sr:={(K,v)};do sr≠∅ → remove(an element of sr) od

We assume that before each graph-transformation the directed graph satisfies condition P, where

P = each node of the directed graph is reachable .

After a transformation the truth of P may be violated, and it will have to be re-established. Also the administration will have to be updated.
Each graph-transformation is a local process: the contents of only one node are changed, and at most two new nodes are created. The reduction rules, however, differ from one another, and therefore each transformation produces different changes in the graph. So for each transformation different actions will have to be performed in order to re-establish P and to update the administration.
Before these actions will be described we shall introduce some notations. The s.c.c. to which node x belongs will be denoted by '$C_x = (N_x,P_x)$'. We shall denote the set of incoming pointers of C_x by 'S_x'. A pointer from node x to node y will be denoted by 'x → y', a path from x to y by '$x \overset{*}{\rightarrow} y$'. If the left field of node x contains l, and the right field contains r, we shall write 'x: (l,r)'.
Let x be a node. If the value of $C_x = (N_x,P_x)$, S_x, ap(x), or cp(x) has changed after a transformation, the new value will be given in an assignment statement. Note that actually no assignments to C_x, N_x, P_x, and S_x are carried out, because C_x, N_x, P_x, and S_x are not incorporated in the administration. The only assignments that are carried out are those to ap(x), cp(x), and $|S_x|$. So when the number of elements of S_x is changed by an assignment to S_x, actually an assignment to $|S_x|$ will be performed (if C_x is an s.c.c. with $P_x \neq \emptyset$).

Now we shall describe the actions that will re-establish P and update the administration for each possible graph-transformation.

1° Let 0: (↑1,n) and 1: (+,m), where m and n are integers. The node 0 contains the representation of the redex (+mn). When the applicable graph-transformation has been performed on this node, the result is 0: (I,m + n) and 1: (+,m). Node 0 has become an indirection node, and of course node 1 has not changed. So the only thing that has to be done is ap(1) := ap(1) - 1. If ap(1) = 0 after this assignment, node 1 is no longer reachable and has to be added to the list of free nodes. (In situations like this, where the s.c.c. consists of only one node that contains no pointers, it is not necessary to use the remove-operation described before.)
Analogous to this case are all cases in which an arithmetical, a logical, or a relational operator is the head of the redex, except: ∨ true y ⋙ true, and: ∧ false y ⋙ false, which are treated in 7°.

2° Suppose 0: (Y,↑1). Then node 0 represents a Y-redex. Contraction of this redex leads to 0: (↑1,↑0). Of course condition P still holds after this transformation. So the only things that have to be done are ap(0) := ap(0) + 1, cp(0) := cp(0) + 1, and if cp(0) = 1 after this assignment, the global administration has to be extended with $|S_0|$:= ap(0) - 1, and node 0 has to contain the information to which s.c.c. of the global administration it belongs.

3° Suppose 1: (I,c), where c is a constant, and suppose node 0 contains a pointer to node 1. Without loss of generality we assume 0: (↑1,r), where 'r' denotes the content of the right field of node 0. When the applicable graph-transformation has been performed, the result is 0: (c,r), and node 1 remains unchanged. So ap(1) := ap(1) - 1, and if ap(1) = 0 after this assignment, node 1 is no longer reachable and has to be added to the list of free nodes.

Now assume that the right field of node 1 contains not a constant, but a pointer to node 2. In that case the result of the transformation is 0: ($\uparrow$2,r). So ap(1) := ap(1) - 1 and ap(2) := ap(2) + 1. Now there are three possibilities:

(i) Suppose that before the transformation 0, 1, and 2 belonged to the same s.c.c.. (So $C_0 = C_1 = C_2$ before the transformation.) Then 0 and 2 are still reachable after the transformation, and $C_0 = C_2$. So $P_0 := (P_0 \cup \{0 \to 2\}) \setminus \{0 \to 1\}$, cp(1) := cp(1) - 1, and cp(2) := := cp(2) + 1.

- If ap(1) = 0, then $N_0 := N_0 \setminus \{1\}$, cp(2) := cp(2) - 1, and the remove-operation has to be performed on $C_1 = (\{1\},\emptyset)$ starting with node 1.

- If ap(1) > 0 and cp(1) > 0, then $C_0 = C_1$ after the transformation as well, and node 1 remains reachable.

- If ap(1) > 0 and cp(1) = 0, node 1 no longer belongs to C_0, but $C_1 = (\{1\},\emptyset)$. So $N_0 := N_0 \setminus \{1\}$, $|S_0| := |S_0 \cup \{1 \to 2\}| - ap(1)$, $P_0 := P_0 \setminus \{1 \to 2\}$, cp(2) := cp(2) - 1, and node 1 will no longer contain the information that it belongs to the s.c.c. C_0. Node 1 remains reachable, because $|S_1| = ap(1) > 0$.

(ii) Suppose that before the transformation 0, 1, and 2 belonged to three different strongly connected components. Then after the transformation $S_2 := S_2 \cup \{0 \to 2\}$ and $S_1 := S_1 \setminus \{0 \to 1\}$. The nodes 0 and 2 remain reachable. If $|S_1| = ap(1) = 0$, the remove-operation has to be performed on $C_1 = (\{1\},\emptyset)$.

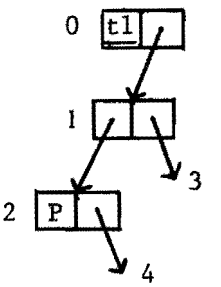
(iii) Suppose that before the transformation $C_1 = C_2$ and $C_1 \neq C_0$. Then $S_1 := (S_1 \cup \{0 \to 2\}) \setminus \{0 \to 1\}$. Before the transformation C_0 and

C_1 were reachable, and since the values of $|S_0|$ and $|S_1|$ have not changed, C_0 and C_1 are still reachable after the transformation.

4° The case that a graph-transformation is performed that establishes a contraction of the form Cfgx ≫ fxg is treated in appendix 2. Analogous situations appear when the contraction is of the form Bfgx ≫ f(gx) or Sfgx ≫ fx(gx).

5° Suppose 0: (t1,↑1), 1: (↑2,↑3), and 2: (P,↑4), where 'P' denotes the curried version of the pairing operator ':'. Then node 0 represents a redex of the form (t1(Pxy)). (If 1.r or 2.r contains a constant instead of a pointer, not all cases treated below are possible.)

Figure 3: The situation before the transformation.



Contraction of this redex leads to 0: (I,↑3). It is clear that ap(1) := ap(1) - 1, ap(3) := ap(3) + 1, and that node 0 and node 3 remain reachable.

(i) If $C_0 = (\{0\},\emptyset)$ before the transformation, then also after it. (Note that a correct SASL program never leads to the situation that node 0 and node 3 are identical.) So $S_3 := S_3 \cup \{0 \rightarrow 3\}$ and $S_1 := S_1 \setminus \{0 \rightarrow 1\}$. (Note that it is possible that two or more C_i's $(1 \leq i \leq 4)$ are identical.) If $|S_1| = 0$ after these assignments, the remove-operation has to be performed on C_1 starting with node 1.

(ii) If $|N_0| > 1$ before the transformation, then $1 \in C_0$, $C_1 = C_0$ before the transformation, and therefore $C_2 = C_0 \vee C_3 = C_0$.

- Suppose $C_3 = C_0 \wedge C_2 \neq C_0$ before the transformation. Then $C_3 = C_0$ after it, $P_0 := (P_0 \cup \{0 \to 3\}) \setminus \{0 \to 1\}$, $cp(1) := cp(1) - 1$, and $cp(3) := cp(3) + 1$.

  If $cp(1) > 1$ before the transformation, then $C_1 = C_0$ after the transformation. In that case all nodes are reachable.

  If $cp(1) = 1$ before the transformation, then $C_1 \neq C_0$ after it.

  Lemma. In this situation $C_1 = (\{1\}, \emptyset)$.

  Proof. Suppose $|N_1| > 1$. We know $3 \notin N_1$, because $C_3 = C_0 \neq C_1$. So $2 \notin N_1$. Before the transformation $C_2 \neq C_1 = C_0$, and therefore $C_2 \neq C_1$ after the transformation. $\therefore$ Contradiction. $\therefore |N_1| = 1$. $\therefore C_1 = (\{1\}, \emptyset)$. □

  So $N_0 := N_0 \setminus \{1\}$, $|S_0| := |S_0 \cup \{1 \to 3\}| - ap(1)$, $P_0 := P_0 \setminus \{1 \to 3\}$, $cp(3) := cp(3) - 1$, and node 1 will no longer contain the information that it belongs to the s.c.c. C_0. If $|S_1| = ap(1) = 0$, the remove-operation has to be performed on C_1.

- Suppose $C_2 = C_0$ before the transformation, then also $C_4 = C_0$.

  If $C_3 = C_0$ before the transformation, then also after it. In that case $P_0 := (P_0 \cup \{0 \to 3\}) \setminus \{0 \to 1\}$, $cp(1) := cp(1) - 1$, and $cp(3) := cp(3) + 1$. Otherwise $S_3 := S_3 \cup \{0 \to 3\}$, $P_0 := P_0 \setminus \{0 \to 1\}$, and $cp(1) := cp(1) - 1$.

  If $cp(1) > 1$ before the transformation, then $C_1 = C_0$ after the transformation, and all nodes remain reachable.

  If $cp(1) = 1$ before the transformation, then $C_1 \neq C_0$ after it. In this situation we have a problem, which will be treated later on.

6° Suppose 0: (hd,↑1), 1: (↑2,↑3), and 2: (P,↑4), where 'P' denotes the curried version of the pairing operator ':'. Then node 0 represents a redex of the form (hd(Pxy)). (If 1.r or 2.r contains a constant instead of

a pointer, not all cases treated below are possible.) Contraction of this redex leads to 0: (I,↑4). So ap(1) := ap(1) - 1, ap(4) := ap(4) + 1, and node 0 and node 4 are still reachable.

(i) If $C_0 = (\{0\},\emptyset)$ before the transformation, the situation is analogous to 5°(i).

(ii) If $|N_0| > 1$ before the transformation, then $1 \in N_0$, $C_1 = C_0$ before the transformation, and therefore $C_2 = C_0 \vee C_3 = C_0$.

If $C_2 = C_0$ before the transformation, then also $C_4 = C_0$. If $C_4 = C_0$ before the transformation, then also after it. In that case

$P_0 := (P_0 \cup \{0 \to 4\}) \setminus \{0 \to 1\}$, cp(1) := cp(1) - 1, and cp(4) := cp(4) + 1

If $C_4 \neq C_0$ before the transformation, then $C_4 \neq C_0$ after it,

$P_0 := P_0 \setminus \{0 \to 1\}$, cp(1) := cp(1) - 1, and $S_4 := S_4 \cup \{0 \to 4\}$.

If cp(1) > 1 before the transformation, then $C_1 = C_0$ after the transformation, and all nodes remain reachable.

If cp(1) = 1 before the transformation, then $C_1 \neq C_0$ after it. In this situation we have a problem, which will be treated later on.

7° Suppose 0: (↑1,↑2) and 1: (∨,true). (If the right field of node 0 contains a constant, not all cases treated below are possible.) Contraction of the redex represented by node 0 gives 0: (I,true). So ap(1) := ap(1) - 1, ap(2) := ap(2) - 1, and node 0 is still reachable. If ap(1) = 0, node 1 has to be added to the list of free nodes; otherwise node 1 is still reachable.

If $C_2 \neq C_0$ before the transformation, then $C_2 \neq C_0$ after it. So $S_2 := S_2 \setminus \{0 \to 2\}$. If $|S_2| = 0$ after this assignment, the remove-operation has to be performed on C_2 starting with node 2. (If node 2 contains no pointers, node 2 is simply added to the list of free nodes.)

If $C_2 = C_0$ before the transformation, then $P_0 := P_0 \setminus \{0 \to 2\}$ and

cp(2) := cp(2) - 1.

If cp(2) > 0 after this assignment, $C_2 = C_0$ after the transformation as well, and node 2 remains reachable.

If cp(2) = 0 after the assignment, then $C_2 \neq C_0$. In that case we have a problem, which will be treated later on.

Analogous to this case is the case where the redex represented by node 0 is of the form (∧ false y).

8° Suppose 0: (↑1,↑3), 1: (↑2,↑4), and 2: (→,false). (If 0.r or 1.r contains a constant instead of a pointer, not all cases treated below are possible.) Contraction of the redex represented by node 0 leads to 0: (I,↑3). It is clear that ap(1) := ap(1) - 1.

(i) Suppose $C_0 = (\{0\},\emptyset)$ before the transformation. (Note that a correct SASL program never leads to the situation that node 0 is the same node as node 3.) In that case $S_1 := S_1 \setminus \{0 \rightarrow 1\}$. If $|S_1| = 0$ after this assignment, the remove-operation has to be performed on C_1 starting with node 1. (Note that it is possible that two or more C_i's ($1 \leq i \leq 4$, $i \neq 2$) are identical.)

(ii) Suppose $|N_0| > 0$ before the transformation. Then $C_3 = C_0 \vee C_1 = C_0$ before the transformation.

- Suppose $C_3 = C_0 \wedge C_1 \neq C_0$ before the transformation, then also after it. So $S_1 := S_1 \setminus \{0 \rightarrow 1\}$, and if $|S_1| = 0$ after this assignment, the remove-operation has to be performed on C_1 starting with node 1.

- Suppose $C_1 = C_0$ before the transformation, then also $C_4 = C_0$. So $P_0 := P_0 \setminus \{0 \rightarrow 1\}$ and cp(1) := cp(1) - 1.
  If cp(1) > 1 before the transformation, then $C_1 = C_0$ after the transformation as well, and all nodes remain reachable.

If $cp(1) = 1$ before the transformation, then $C_1 \neq C_0$ after it.

In that case we have a problem, which will be treated later on.

9° Suppose 0: ($\uparrow$1,$\uparrow$3), 1: ($\uparrow$2,$\uparrow$4), and 2: ($\rightarrow$,true). (If 0.r or 1.r contains a constant, not all cases treated below are possible.) Contraction of the redex represented by node 0 leads to 0: (I,$\uparrow$4). It is easy to see that $ap(1) := ap(1) - 1$, $ap(4) := ap(4) + 1$, and $ap(3) := ap(3) - 1$.

(i) Suppose $C_0 = (\{0\},\emptyset)$ before the transformation and suppose node 3 is not the same node as node 0. In that case $S_1 := S_1 \setminus \{0 \rightarrow 1\}$, $S_4 := S_4 \cup \{0 \rightarrow 4\}$, and $S_3 := S_3 \setminus \{0 \rightarrow 3\}$. (Note that it is possible that two or more C_i's ($1 \leq i \leq 4$, $i \neq 2$) are identical.) For $i = 1,3$ we have : if $|S_i| = 0$ after these assignments, the remove-operation has to be performed on C_i starting with node i.

(ii) Suppose $|N_0| > 1$ before the transformation or node 3 is the same node as node 0. Then $C_1 = C_0 \vee C_3 = C_0$ before the transformation.

- If $C_1 = C_0$ before the transformation, then also $C_4 = C_0$. In that case $C_4 = C_0$ after the transformation, $P_0 := (P_0 \cup \{0 \rightarrow 4\}) \setminus \{0 \rightarrow 1\}$, $cp(4) := cp(4) + 1$, and $cp(1) := cp(1) - 1$.

  If $cp(1) > 1$ before the transformation, then $C_1 = C_0$ after the transformation, and node 1 and node 2 are still reachable.

  If $cp(1) = 1$ before the transformation, then $C_1 \neq C_0$ after it.

  Lemma. In this situation $C_1 = (\{1\},\emptyset)$.

  Proof. Suppose $|N_1| > 1$. $C_2 = (\{2\},\emptyset)$, so $2 \notin N_1$. $\therefore 4 \in N_1$. $\therefore C_1 = C_4 = C_0$. $\therefore$ Contradiction. $\therefore N_1 = \{1\}$. $\therefore C_1 = (\{1\},\emptyset)$. □

  So $N_0 := N_0 \setminus \{1\}$, $P_0 := P_0 \setminus \{1 \rightarrow 4\}$, $cp(4) := cp(4) - 1$, $|S_0| := |S_0 \cup \{1 \rightarrow 4\}| - ap(1)$, and node 1 will no longer contain the information that it belongs to the s.c.c. C_0. If $|S_1| = ap(1) = 0$, the remove-operation has to be performed on $C_1 = (\{1\},\emptyset)$.

- If $C_1 \neq C_0$ before the transformation, then $C_1 \neq C_0$ after it. So $S_1 := S_1 \setminus \{0 \to 1\}$. If $|S_1| = 0$ after this assignment, the remove-operation has to be performed on C_1 starting with node 1. Because $C_4 \neq C_0$ in this case, $S_4 := S_4 \cup \{0 \to 4\}$.

-- If $C_3 \neq C_0$ before the transformation, then also after it. So $S_3 := S_3 \setminus \{0 \to 3\}$. If $|S_3| = 0$ after this assignment, the remove-operation has to be performed on C_3 starting with node 3.

-- If $C_3 = C_0$ before the transformation, then $P_0 := P_0 \setminus \{0 \to 3\}$ and $cp(3) := cp(3) - 1$.

If $cp(3) > 1$ before the transformation, then $C_3 = C_0$ after it, and node 3 is still reachable.

If $cp(3) = 1$ before the transformation, then $C_3 \neq C_0$ after it. In that case we have a problem, which will be treated later on.

Each time the treatment of a possible situation has been postponed, the reason for this postponement was the following.

Before the transformation two nodes, say 0 and 1, with $0 \to 1$, belonged to the same s.c.c. $K' = (N',P')$. (So $K' = C_0 = C_1$ before the transformation.) After the transformation the pointer $0 \to 1$ has been removed, and $C_0 \neq C_1$. Sometimes we have been able to prove $C_1 = (\{1\},\emptyset)$, which means that K' has been split up into two new strongly connected components: $C_1 = (\{1\},\emptyset)$ and $C_0 = (N' \setminus \{1\}, P' \setminus \{p: v \to w \mid v = 1 \vee w = 1\})$.

In those cases, however, where the treatment of such a situation has been postponed, it is impossible to find out into which strongly connected components K' has been split up without inspecting all nodes that belong to K'.

Let $K' = (N',P')$ be an s.c.c. of the directed graph G'. After the transformation G' has been transformed to the directed graph G, and K' has

been transformed to the subgraph K = (N,P) of G, where N = N' and P consists of all pointers $v \rightarrow w$, with $v \in N \wedge w \in N$, that belong to G. Now we must find the strongly connected components of K, and update our local and global administration. The algorithm that we shall use for finding the strongly connected components is from R.E. Tarjan [7]. This algorithm is an elaboration of the depth-first search technique.

The algorithm consists of two steps. The first step is a marking phase. This phase starts at node 1, and for each node x that is reached two things have to be done:

(i) mark x "new",

(ii) for each pointer $x \rightarrow y$, with $y \in N$ and y not yet marked, perform (i) and (ii).

It is easy to see that, when this process starts at node 1, all elements of N are marked "new" in this way.

The second step of the algorithm consists of choosing a node marked "new", and determining strongly connected components of K using the depth-first search technique. All nodes of these strongly connected components will be unmarked during the search. This step of the algorithm is repeated as long as there are nodes marked "new".

If the second step is started with a "new" node v, then all strongly connected components found during this step are reachable from v because of the depth-first search. 'All strongly connected components are reachable from v' means that there exists a path from v to each node of these components. Hence, if we choose v such that v is reachable and marked "new", then all strongly connected components found are reachable. So we have to know which of the nodes marked "new" are reachable. Initially the reachable nodes can be found during the marking phase: whenever a node x with $ap(x) - cp(x) > 0$ is reached, this

node x is added to the set T of reachable nodes that are marked "new". Now suppose an s.c.c. C = (NC,PC) is determined. Then, of course, all nodes that belong to C are no longer marked "new". Therefore each node of NC that is an element of T has to be removed from T. Let $x \in NC$, then x is reachable. If x contains a pointer $x \rightarrow y$, with $y \in N \backslash NC$ and y is marked "new", then y is a reachable node marked "new". So if $y \notin T$, y has to be added to T. Besides performing these operations on T some other things have to be done when an s.c.c. is determined. For each node $x \in NC$ the new value of cp(x) has to be recorded, and if cp(x) > 0, the local administration of node x must also contain the information that x belongs to C. (Note that the value of ap(x) remains the same.) If $|PC| > 0$, the global administration has to contain the number of incoming pointers of C. So, if an s.c.c. C = (NC,PC) of the graph K = (N,P) is determined, there are two possibilities:

(i) $|PC| = 0$.

That is when C consists of only one node, say x, that does not contain a pointer to itself. In that case the only things to do are:

- cp(x) := 0,
- if $x \in T$, remove x from T,
- for each pointer $x \rightarrow y$, with $y \in N \backslash NC \wedge y$ is marked "new" $\wedge\ y \notin T$, y has to be added to T.

(ii) $|PC| > 0$.

That is when C consists of two or more nodes, or of only one node that contains a pointer to itself. Then the following things have to be done:

1° Give the number SC of incoming pointers of C the initial value 0, and give for each node $x \in NC$ the number cp(x) the initial value 0.

2° For each node $x \in NC$:

- add to the local administration of x the information that x belongs to C,
- SC := SC + ap(x),
- for each pointer $x \rightarrow y$:
  - -- if $y \in NC$, then cp(y) := cp(y) + 1, SC := SC - 1, and if $y \in T$, remove y from T,
  - -- if $y \in N \backslash NC \wedge$ y is marked "new" $\wedge\ y \notin T$, then add y to T,
  - -- if $y \in N \backslash NC$ and y is already unmarked, or if $y \notin N$, then do nothing.

The second step of the algorithm for finding the strongly connected components of K is repeated until $T = \emptyset$. It is possible that, when $T = \emptyset$, there are still some nodes marked "new". These nodes are no longer reachable and have to be added to the list of free nodes. Let L be the set of these nodes.

Lemma 4.11. If $L \neq \emptyset$, then node 1 is an element of L.

Proof. Suppose $L \neq \emptyset$ and $1 \notin L$. Then there exists a node $x \in L$, $x \neq 1$. Node 1 is reachable, because $1 \notin L$. There exists a path $1 \overset{*}{\rightarrow} x$, because $x \in N$. $\therefore$ x is reachable. $\therefore$ Contradiction. □

Lemma 4.12. For all $x \in L$, there exists a path $(v_0, a_1, v_1, a_2, \ldots, a_n, v_n)$, $(n \geq 0)$, such that $v_0 = 1$, $v_n = x$, and $v_i \in L$, $0 \leq i \leq n$.

Proof. Let $x \in L$. Since $x \in N$, there exists a path $(v_0, a_1, v_1, a_2, \ldots, a_n, v_n)$, $n \geq 0$, such that $v_0 = 1$ and $v_n = x$. $v_0 = 1 \in L$ by the previous lemma. Now suppose there is an i, $1 \leq i \leq n$, such that $v_i \notin L$. Then v_i is reachable. $\therefore$ $v_n = x$ is reachable. $\therefore$ Contradiction. □

Let lf be the list of free nodes. If an element of L is added to lf, it is possible that of some strongly connected components of G the number of incoming pointers has to be decreased, and so it is possible that some components are no longer reachable. The s.c.c.'s that are no longer reachable will be added to the set sr, and each element of sr will be removed from the graph by the remove-operation described before. The set sn contains the elements of L that are already reached, but not yet added to lf. The last lemma now leads to the following algorithm for adding all elements of L to lf, in which as usual 'C_y' denotes the s.c.c. to which node y belongs, and 'S_y' denotes the set of incoming pointers of C_y.

```
proc p(x):
  proc field(z):
    if z is a constant → skip
     □ z is a pointer to a node y ∧ y∉lf →
                        if y is marked "new" → sn:snu{y}
                         □ y is unmarked → ap(y):=ap(y)-1;|S_y|:=|S_y|-1;
                                           if |S_y|=0 → sr:=sru{(C_y,y)}
                                            □ |S_y|>0 → skip
                                           fi
                        fi
    fi
  corp;
  lf:=lfu{x};field(x.ℓ);field(x.r);sn:=sn\{x}
corp;
sr:=∅;sn:={x};do sn≠∅ → p(an element of sn) od;
do sr≠∅ → remove(an element of sr) od
```

5. A mild variant of combinatory logic

We have seen how a SASL program is translated into a combinatory term. The number of combinators that occur in this term is $O(n^2)$, where n is the number of abstractions of a variable. D.A. Turner described in [10] a way to avoid this growth. Another method to do this is described by E.W. Dijkstra in [3]. His variant of combinatory logic, when used for the translation of a SASL program, gives as a result a term of size O(n). First we shall describe this mild variant of combinatory logic.

We assume that there is an infinite set of identifiers, a finite or infinite set of constants, and three combinators S, B, and C. The identifiers and constants are called atoms.

Definition 5.1. A term is defined by induction as follows:

(i) There exists an empty term Λ.

(ii) Every atom is a term.

(iii) If T1 and T2 are terms and coms is a possibly empty sequence of combinators, then (T1 coms T2) is a term.

We shall denote a possibly empty sequence of combinators by 'coms', and a non-empty sequence of combinators by 'coms+'.

Definition 5.2. A term T is healthy if it satisfies one of the following three conditions:

(i) $T = \Lambda$,

(ii) T is an atom,

(iii) T is of the form (T1 coms T2) such that

(T1 ≠ Λ or coms contains an S or a C)

and (T2 ≠ Λ or coms contains an S or a B)

and (T1 and T2 are both healthy).

Note that no healthy term contains a redundant bracket pair; either of the form '()' or of the form '((...))'.

Definition 5.3. For each non-empty term M and each term T we define the relation M occurs in T, M ∈ T, by induction on the construction of T as follows:

(i) If T = Λ, then M ∈ T = false.

(ii) If T is an atom a, then M ∈ T = M = a.

(iii) If T = (T1 coms T2), then M ∈ T = (M ∈ T1 or M ∈ T2).

We shall write 'M ∉ T' if M does not occur in T.

Definition 5.4. For each term T and identifier x, provided x ∈ T, a term [x]T is defined by induction on the construction of T as follows:

(i) If T = x, then [x]T = Λ.

(ii) If T = (T1 coms T2), then

- if x ∈ T1 and x ∈ T2, then [x]T = ([x]T1 Scoms [x]T2),
- if x ∈ T1 and x ∉ T2, then [x]T = ([x]T1 Ccoms T2),
- if x ∉ T1 and x ∈ T2, then [x]T = (T1 Bcoms [x]T2).

'[x]T' is pronounced as 'x abstracted from T'.

Theorem 5.5. If T is a healthy term, x an identifier, and x ∈ T, then [x]T is a healthy term.

Proof. (Induction on the construction of [x]T)

(i) If T = x, then [x]T = Λ and so [x]T is healthy.

(ii) If T = (T1 coms T2), there are three possibilities:

- Suppose x $\in$ T1 and x $\in$ T2. Then [x]T = ([x]T1 Scoms [x]T2), and T1 and T2 are both healthy. By induction we have: [x]T1 and [x]T2 are both healthy.

  So: (Scoms contains an S) and ([x]T1 and [x]T2 are both healthy).

  $\therefore$ [x]T is healthy.

- Suppose x $\in$ T1 and x $\notin$ T2. Then [x]T = ([x]T1 Ccoms T2), and: (T2 $\neq \Lambda$ or coms contains an S or a B) and (T1 and T2 are both healthy). By induction we have: [x]T1 is a healthy term.

  So: (Ccoms contains a C) and (T2 $\neq \Lambda$ or Ccoms contains an S or a B) and ([x]T1 and T2 are both healthy).

  $\therefore$ [x]T is healthy.

- Suppose x $\notin$ T1 and x $\in$ T2. This case is analogous to the case that x $\in$ T1 and x $\notin$ T2. □

Definition 5.6. For mutually distinct identifiers $x_1,\ldots,x_m$ and for each term T, provided $x_i \in T$ $(1 \leq i \leq m)$, we define: $[x_1,\ldots,x_m]T = T_0$, where $T_i = [x_{i+1}]T_{i+1}$, $0 \leq i < m$, and $T_m = T$.

Definition 5.7. For any terms N,M and any identifier x, the result [N/x]M of substituting N for every occurrence of x in M is defined as follows by induction on the construction of M:

(i) [N/x]Λ = Λ,

(ii) [N/x]x = N,

(iii) [N/x]a = a for all atoms a $\neq$ x,

(iv) [N/x](M1 coms M2) = ([N/x]M1 coms [N/x]M2).

For mutually distinct identifiers $x_1,\ldots,x_n$ and for any terms $N_1,\ldots,N_n$, we similarly define $[N_1/x_1,\ldots,N_n/x_n]M$ to be the result of simultaneously substituting N_1 for x_1, ..., N_n for x_n, in M.

Definition 5.8. The relation $X \gg Y$, X contracts to Y, is defined by induction as follows:

Contraction-rules:

(S): ((T1 Scoms T2) h) $\gg$ ([T1 h] coms [T2 h])

(C): ((T1 Ccoms T2) h) $\gg$ ([T1 h] coms T2)

(B): ((T1 Bcoms T2) h) $\gg$ (T1 coms [T2 h])

where [T h] is short for:

- if $T = \Lambda$, then [T h] = h

- if $T \neq \Lambda$, then [T h] = (T h)

Deduction-rules:

(μ): if $X \gg Y$, then (X coms T) $\gg$ (Y coms T)

(ν): if $X \gg Y$, then (T coms X) $\gg$ (T coms Y).

Contraction-rules are defined for terms of the form ((T1 coms+ T2) h), where T1, T2, and h are terms. In view of the rule of absence of redundant bracket pairs, this implies $h \neq \Lambda$. A term of the form ((T1 coms+ T2) h) is called a redex. If the leading combinator of coms+ is an A, we call it an A-redex. The result of a contraction is called the contractum.

Definition 5.9. A sequence $T_0,\ldots,T_n$ ($n \geq 0$) such that $T_{i-1} \gg T_i$, $0 < i \leq n$, is called a reduction (from T_0 to T_n). The number n is called the length of the reduction, and $T_{i-1} \gg T_i$ the i-th step of the reduction.
X reduces to Y if and only if there is a reduction from X to Y.

Lemma 5.10. The relation 'reduces to' is the reflexive and transitive closure of the relation '$\gg$'.

If X reduces to Y, we shall write '$X \overset{*}{\gg} Y$'.

Theorem 5.11. For healthy T and healthy, but non-empty, h, [T h] is healthy.

Proof. (i) If $T = \Lambda$, then [T h] = h is healthy.

(ii) If $T \neq \Lambda$, then [T h] = (T h). So: $(T \neq \Lambda)$ and $(h \neq \Lambda)$ and (T and h are both healthy). $\therefore$ [T h] is healthy. □

Theorem 5.12. Contraction never introduces an unhealthy term.

Proof. (S): Suppose the contraction is ((T1 Scoms T2) h) $\gg$ ([T1 h] coms [T2 h]) and let ((T1 Scoms T2) h) be a healthy term. $\therefore$ $h \neq \Lambda$, and h and (T1 Scoms T2) are both healthy terms. $\therefore$ T1 and T2 are both healthy terms. $\therefore$ [T1 h] and [T2 h] are both healthy by theorem 5.11. $h \neq \Lambda$, and therefore $[T1\ h] \neq \Lambda$ and $[T2\ h] \neq \Lambda$.
$\therefore$ ([T1 h] coms [T2 h]) is a healthy term.

(C): Suppose the contraction is ((T1 Ccoms T2) h) $\gg$ ([T1 h] coms T2) and let ((T1 Ccoms T2) h) be a healthy term. $\therefore$ $h \neq \Lambda$, and h and (T1 Ccoms T2) are both healthy terms. $\therefore$ (T1 and T2 are both healthy) and ($T2 \neq \Lambda$ or Ccoms contains an S or a B). $\therefore$ [T1 h] is healthy by theorem 5.11. $h \neq \Lambda$, and therefore $[T1\ h] \neq \Lambda$. Also we have (T2 is healthy) and ($T2 \neq \Lambda$ or coms contains an S or a B).
$\therefore$ ([T1 h] coms T2) is a healthy term.

(B): analogous to (C).

(μ): Suppose the contraction is a consequence by deduction-rule (μ). Then the contraction is of the form (X coms T) $\gg$ (Y coms T) and

$X \gg Y$. If (X coms T) is healthy and $X \gg Y$, we have:

- X and T are both healthy,

- $X \neq \Lambda$,

- $T \neq \Lambda$ or coms contains an S or a B.

By induction we have: Y is healthy. Y is a contractum, and so $Y \neq \Lambda$.

$\therefore$ (Y coms T) is a healthy term.

(ν): analogous to (μ). □

Theorem 5.13. For all terms $N_1,\ldots,N_m$, and mutually distinct identifiers $x_1,\ldots,x_m$, and each $n \geq 0$:
if $X \overset{n}{\gg} Y$, then $[N_1/x_1,\ldots,N_m/x_m]X \overset{n}{\gg} [N_1/x_1,\ldots,N_m/x_m]Y$, where $U \overset{n}{\gg} V$ means: U reduces to V and the length of this reduction is n.

Proof. Induction on n. □

Theorem 5.14. (Law of abstraction)

For all terms M and N, and for each identifier $x \in M$: $[[x]M\ N] \overset{*}{\gg} [N/x]M$.

Proof. By theorem 5.13 it is enough to prove that $[[x]M\ x] \overset{*}{\gg} M$. This will be done by induction on the construction of M.

(i) If M = x, then $[x]x = \Lambda$, and $[[x]x\ x] = [\Lambda\ x] = x$.

(ii) If M = (M1 coms M2), then:

- If $x \in M1$ and $x \in M2$, then by induction we have: $[[x]M1\ x] \overset{*}{\gg} M1$ and $[[x]M2\ x] \overset{*}{\gg} M2$.

  So: $[[x]M\ x] = [([x]M1\ Scoms\ [x]M2)\ x] = (([x]M1\ Scoms\ [x]M2)\ x) \gg$
  $\gg ([[x]M1\ x]\ coms\ [[x]M2\ x]) \overset{*}{\gg} (M1\ coms\ M2) = M$.

- The other two cases are analogous. □

Theorem 5.15. For mutually distinct identifiers $x_1,\ldots,x_m$, and terms $N_1,\ldots,N_m$, and M:

$$[[\ldots[[x_1,\ldots,x_m]M\ N_1]\ldots]\ N_m] \overset{*}{\gg} [N_1/x_1,\ldots,N_m/x_m]M\ .$$

Proof. By theorem 5.13 it is enough to prove that

$$[[\ldots[[x_1,\ldots,x_m]M\ x_1]\ldots]\ x_m] \overset{*}{\gg} M\ ,$$

which can easily be proved by induction on m using theorem 5.14. □

Definition 5.16. A term T is said to be in normal form if and only if T contains no redexes. If U reduces to an X in normal form, this X is called a normal form of U.

Definition 5.17. (i) Let R1 and R2 (R1 ≠ R2) be two redexes that occur in the same term. Then there are three possibilities:

- R1 occurs in R2,
- R2 occurs in R1,
- R1 ∉ R2 and R2 ∉ R1; in this case R1 and R2 are called non-overlapping redexes.

Let $R_1,\ldots,R_n$ ($n \geq 0$) be redexes that occur in the same term. If $\underline{A}\ i,j$: $1 \leq i \leq n$, $1 \leq j \leq n$: $i = j \vee R_i$ and R_j are non-overlapping, we say that $R_1,\ldots,R_n$ are non-overlapping redexes.

(ii) We define a special contraction as the simultaneous contraction of zero or more non-overlapping redexes. For this binary relation we shall use the notation '$\overset{}{\gg}_s$'. With '$\overset{*}{\gg}_s$' we shall denote the reflexive and transitive closure of '$\overset{}{\gg}_s$'.

Let DT be the set of all terms. Then $(DT,\gg)$ is a general replacement system.

Theorem 5.18. $X \overset{*}{\gg} Y$ if and only if $X \overset{*}{\underset{s}{\gg}} Y$.

Proof. The same as that of theorem 2.17. □

Theorem 5.19. If $T \underset{s}{\gg} X$ and $T \underset{s}{\gg} Y$, there exists a Z such that $X \underset{s}{\gg} Z$ and $Y \underset{s}{\gg} Z$.

Proof. (Induction on the construction of T)

1° If $T = \Lambda$ or T is an atom, then $T = X = Y$. So choose $Z = T$.

2° Let T be of the form T = (T1 coms T2). Suppose $T \underset{s}{\gg} X$ by the simultaneous contraction of the non-overlapping redexes $R_1,\ldots,R_k$ $(k \geq 0)$, and suppose $T \underset{s}{\gg} Y$ by the simultaneous contraction of the non-overlapping redexes $Q_1,\ldots,Q_m$ $(m \geq 0)$. Define $RR = \{R_i \mid 1 \leq i \leq k\}$ and $QQ = \{Q_i \mid 1 \leq i \leq m\}$.

(i) Suppose $R_1 = T$ and $Q_1 = T$. Then $k = 1$ and $m = 1$. $\therefore R_1 = Q_1$, and therefore $X = Y$. So choose $Z = X$.

(ii) Suppose $R_1 = T$ and $\underline{A}\ i: 1 \leq i \leq m: Q_i \neq T$. Then $k = 1$.

(S) Suppose T = ((T11 Scoms' T12) h) $\gg$ ([T11 h] coms' [T12 h]) = X. Also T = (T1 coms T2). So T1 = (T11 Scoms' T12), T2 = h, and coms is the empty sequence. Note that T1 is not a redex. Define $RQ_i = \{Q \in QQ \mid Q \in T_i\}$ $(i = 1,2)$. Let $h \underset{s}{\gg} h^*$ by the simultaneous contraction of the non-overlapping redexes of RQ_2. T1 is not a redex and $\underline{A}\ Q: Q \in RQ_1: Q$ is a redex. So $\underline{A}\ Q: Q \in RQ_1: Q \neq T1$, and therefore $\underline{A}\ Q: Q \in RQ_1: Q \in T11 \vee Q \in T12$. Let $T1i \underset{s}{\gg} T1i^*$ by the simultaneous contraction of those redexes of RQ_1 that occur in T1i $(i = 1,2)$. Then $T = ((T11 \text{ Scoms' } T12)\ h) \underset{s}{\gg} ((T11^* \text{ Scoms' } T12^*)\ h^*) = Y$.

Define $Z = ([T11^*\ h^*] \text{ coms' } [T12^*\ h^*])$. Then $Y \gg Z$, and $X \underset{s}{\gg} Z$ by the simultaneous contraction of all occurrences of elements of $RQ_1 \cup RQ_2$.

(C) and (B): analogously.

(iii) Suppose $Q_1 = T$ and $\underline{A}$ i: $1 \leq i \leq k$: $R_i \neq T$. This case is analogous to (ii).

(iv) Suppose $\underline{A}$ i: $1 \leq i \leq k$: $R_i \neq T$ and $\underline{A}$ i: $1 \leq i \leq m$: $Q_i \neq T$. By definition 5.17 (ii) there exist an X1 and an X2 such that $T1 \underset{s}{\geqslant} X1$ and $T2 \underset{s}{\geqslant} X2$ and X = (X1 coms X2). Also there exist a Y1 and a Y2 such that $T1 \underset{s}{\geqslant} Y1$ and $T2 \underset{s}{\geqslant} Y2$ and Y = (Y1 coms Y2). Then by the induction hypothesis there exist a Z1 and a Z2 such that $Xi \underset{s}{\geqslant} Zi$ and $Yi \underset{s}{\geqslant} Zi$ (i = 1,2). Choose Z = (Z1 coms Z2), then $X \underset{s}{\geqslant} Z$ and $Y \underset{s}{\geqslant} Z$. □

Theorem 5.20. Let DT be the set of all terms. Then $(DT, \geqslant)$ has the Church-Rosser property.

Proof. This theorem is an immediate consequence of the theorems 5.18, 5.19, and 2.15. □

Corollary 5.20.1. A term can have at most one normal form.

From definition 5.4 it follows directly that if n abstraction operations have been performed on a term T = (T1 coms T2), the number of combinators that occur in T has increased by n, not counting the combinators in T1 and T2.

Now we shall look at the consequences of this variant of combinatory logic for the implementation of SASL.

The translation of a SASL program into a form from which all bound variables have been removed, is almost the same process as the process described in chapter 3. There are only two differences:

1. Instead of using the abstraction algorithm (b c d e f), the abstraction operation of definition 5.4 is used.

2. Suppose a local variable f occurs in a term T1, and the definition of f is recursive. Suppose this definition has already been translated into a form f = T2, where T2 is a term from which all local variables have been removed, and f ϵ T2. Then we have:

   T1 where f = T2 .

   This form is translated into:

   ([f]T1 (Λ Y [f]T2)) .

   The contraction rule for the combinator Y is:

   (Λ Y T) $\gg$ (T (T (T (...)))) = T^{∞} for all terms T.

The final result of the translation is a term in which the only occurring identifiers are the names of standard functions. We shall store each term as a directed graph. Each node of the graph can consist of one or three fields.

The directed graph representing a term T is defined as follows (by induction on the construction of T):

(i) If T = Λ or T is a constant, the directed graph has only one node, called root, and this node has only one field containing Λ or the constant.

(ii) If T is of the form T = (T1 coms T2), the directed graph has a node, called root, with three fields, called root.ℓ, root.r, and root.c. If coms is the empty sequence, then root.c is empty; if coms is a non-empty sequence, then root.c contains this sequence of combinators. If T1 = b, with b = Λ or b is a constant, then root.ℓ = b; otherwise root.ℓ = ↑t1, where t1 is the root of the directed graph that represents T1. If T1 is not a constant and not the empty term, T1 is the name of a standard function or of the form T1 = (T11 coms' T12). The field root.r

is defined analogously.

The sharing properties are realised in the same way as in chapter 3. This means that the optimizations described on the pages 29, 30, and 31 (remarks 1., 2., and 3.) are applicable in this situation too.
The reduction machine, which transforms the graph by applying the contraction rules, is completely analogous to the reduction machine described in chapter 3.

6. Some final remarks

In this report it is shown how by using results from combinatory logic a SASL program can be translated into a form from which all bound variables have been removed. Also we have described how this form can easily be represented by a binary directed graph, and how this graph is reduced. The reduction is a normal order graph reduction: the normal order implies that arguments are not evaluated needlessly, and the sharing properties of the graph imply that arguments are evaluated at most once.
The implementation of SASL described in [9] uses a mark scan garbage collector. This means that at unforeseen moments garbage is collected. In order to avoid this a counting technique is introduced in chapter 4. This technique requires only a few extra arrangements: the local and global administration. When a graph-transformation has been performed, it is easy to update this administration, except in the cases that are treated at the end of chapter 4. In these situations all nodes of a strongly connected component have to be inspected. Note that during a garbage collection all nodes of the store are inspected.
An important thing to know is when such situations occur. They occur when

a pointer that belongs to at least one cycle is removed and the node to which this pointer points has no longer any incoming pointers from nodes of the same strongly connected component. So always a recursive definition is involved.

Suppose the definition of an identifier f is recursive. Then this definition will be represented (directly or after some contractions) by a binary graph in which for each occurrence of f a pointer is placed to the node that represents the whole definition of f. Each of these pointers introduces a cycle, and all nodes of these cycles belong to the same strongly connected component. As long as it is possible that f will be used during the computation of the value of the complete program, the whole subgraph representing the definition of f will remain reachable. So we may assume that in general the whole structure representing a recursive definition will have to be removed, and not a small part of it. Therefore we may also assume that in general a strongly connected component will not be split up into some smaller strongly connected components. So the situations treated at the end of chapter 4 will occur only rarely. In fact an upper bound can be given for the number of occurrences of such situations, for instance the number of recursive definitions that appear in the SASL program.

The mild variant of combinatory logic described in chapter 5 has two advantages. Firstly, the definition of the abstraction operation in chapter 5 is much more symmetrical than the corresponding definition in chapter 3. Secondly, the bracket pairs in the resulting term of an abstraction operation are the same bracket pairs that occurred in the original term.

References

[1] H.B. Curry and R. Feys, Combinatory logic vol.1, North Holland Publishing Company, Amsterdam 1968.

[2] H.B. Curry, J.R. Hindley, and J.P. Seldin, Combinatory logic vol.2, North Holland Publishing Company, Amsterdam 1972.

[3] E.W. Dijkstra, A mild variant of combinatory logic, EWD 735, 1980.

[4] J.R. Hindley, B. Lercher, and J.P. Seldin, Introduction to combinatory logic, University Press, Cambridge 1972.

[5] B.K. Rosen, Tree-manipulating systems and Church-Rosser theorems, J.ACM 20, 1(Jan. 1973), 160-187.

[6] M. Schönfinkel, Über die Bausteine der mathematischen Logik, Math. Annalen, 92(1924), 305-316.

[7] R.E. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Computing, Vol.1, 2(June 1972), 146-160.

[8] D.A. Turner, SASL language manual, University of St.Andrews, 1976.

[9] D.A. Turner, A new implementation technique for applicative languages, Software - Practice and Experience, Vol.9 (1979), 31-49.

[10] D.A. Turner, Another algorithm for bracket abstraction, Journal of Symbolic Logic, 44, 2, June 1979, 267-270.

Appendix 1

We shall use the same notation for SASL operators and their curried versions, except the use of 'P' for the curried version of the pairing operator ':'.

The reduction rules performed by the reduction machine are:

S f g x	⩾	f x (g x)
B f g x	⩾	f (g x)
C f g x	⩾	f x g
Y f	⩾	f^{∞}
I x	⩾	x
hd (P x y)	⩾	x
tl (P x y)	⩾	y
→ true x y	⩾	x
→ false x y	⩾	y
∨ true y	⩾	true
∨ false true	⩾	true
∨ false false	⩾	false
∧ false y	⩾	false
∧ true false	⩾	false
∧ true true	⩾	true
¬ true	⩾	false
¬ false	⩾	true
+ m n	⩾	m + n
- m n	⩾	m - n
* m n	⩾	m * n
div m n	⩾	m div n
mod m n	⩾	m mod n
- m	⩾	-m

> m n ⋙ m > n

≥ m n ⋙ m ≥ n

= m n ⋙ m = n

≠ m n ⋙ m ≠ n

≤ m n ⋙ m ≤ n

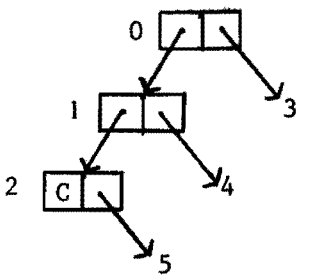
< m n ⋙ m < n

If 'm' and 'n' are used, this means that these arguments must already be reduced to integers.

Appendix 2

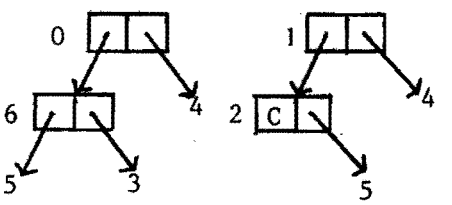
Suppose 0: (↑1,↑3), 1: (↑2,↑4), and 2: (C,↑5). Then node 0 represents a C-redex.

Figure 4: The situation before the transformation.



The result of contracting this redex is: 0: (↑6,↑4), where 6: (↑5,↑3) is a new node. (If one or more of the right fields of the nodes 0, 1, and 2 contains a constant instead of a pointer, not all cases that are discussed below will be possible.)

Figure 5: The situation after the transformation.



It is clear that ap(1) := ap(1) - 1, ap(5) := ap(5) + 1, ap(4) := ap(4) + 1, and ap(6) := 1, and it is easy to see that the nodes 0, 6, 5, 3, and 4 are still reachable after the graph-transformation.

1. Suppose $C_0 = (\{0\},\emptyset)$ before the transformation, and suppose node 3 is not the same node as node 0. In that case the following things have to be done: $S_1 := S_1 \setminus \{0 \rightarrow 1\}$, $S_5 := S_5 \cup \{6 \rightarrow 5\}$, $S_4 := S_4 \cup \{0 \rightarrow 4\}$, and $S_3 := (S_3 \cup \{6 \rightarrow 3\}) \setminus \{0 \rightarrow 3\}$. Also: $cp(6) := 0$ and $S_6 := \{0 \rightarrow 6\}$. (Note that two or more C_i's $(1 \leq i \leq 5)$ can be identical.) If $|S_1| = 0$ after these assignments, the remove-operation has to be performed on C_1 started with node 1. Otherwise node 1 is still reachable, and thus node 2 as well.

2. Suppose $|N_0| > 1$ before the transformation, or $C_0 = (\{0\},\emptyset)$ and node 3 is the same node as node 0. So: $C_1 = C_0 \vee C_3 = C_0$ before the transformation.

2.1. Suppose $C_3 = C_0 \wedge C_1 \neq C_0$ before the transformation. Then $C_3 = C_6 = C_0$ after the transformation. So the information that node 6 belongs to the s.c.c. C_0 has to be added to the local administration of node 6. Also $N_0 := N_0 \cup \{6\}$, $cp(6) := 1$, $P_0 := (P_0 \cup \{0 \rightarrow 6, 6 \rightarrow 3\}) \setminus \{0 \rightarrow 3\}$, $S_4 := S_4 \cup \{0 \rightarrow 4\}$, $S_1 := S_1 \setminus \{0 \rightarrow 1\}$, and $S_5 := S_5 \cup \{6 \rightarrow 5\}$. If $|S_1| = 0$ after these assignments, the remove-operation has to be performed on C_1 starting with node 1. (Note that two or more C_i's $(1 \leq i \leq 5, i \neq 3)$ can be identical.)

2.2. Suppose $C_1 = C_0$ before the transformation. Then $C_2 = C_0 \vee C_4 = C_0$ before the transformation.

2.2.1. Suppose $C_2 = C_0 = C_1$ before the transformation, then $C_5 = C_0$ as well. After the transformation we have $C_0 = C_5 = C_6$. So $N_0 := N_0 \cup \{6\}$, $cp(6) := 1$, $P_0 := (P_0 \cup \{0 \rightarrow 6, 6 \rightarrow 5\}) \setminus \{0 \rightarrow 1\}$, $cp(5) := cp(5) + 1$, and the local administration of node 6 has to contain the information that node 6 belongs to C_0. For $i = 3,4$ we have: $C_i = C_0$ after the transformation if and only if $C_i = C_0$ before the transformation. If

$C_3 = C_0$, then $P_0 := (P_0 \cup \{6 \to 3\}) \setminus \{0 \to 3\}$, and otherwise $S_3 := (S_3 \cup \{6 \to 3\}) \setminus \{0 \to 3\}$. If $C_4 = C_0$, then $P_0 := P_0 \cup \{0 \to 4\}$ and $cp(4) := cp(4) + 1$, and otherwise $S_4 := S_4 \cup \{0 \to 4\}$. (Note that it is possible that $C_3 = C_4$.)

Finally we have to look at C_1 and C_2. There are two possibilities:

(i) Suppose $cp(1) > 1$ before the transformation. Then after it we still have $C_1 = C_0$, and therefore also $C_2 = C_0$. So node 1 and node 2 remain reachable, and $cp(1) := cp(1) - 1$.

(ii) Suppose before the transformation $cp(1) = 1$. Then $C_1 \neq C_0$ after the transformation, since $cp(1) := cp(1) - 1$. So the information that node 1 belongs to C_0 has to be removed from the local administration of node 1, $N_0 := N_0 \setminus \{1\}$, and $|S_0| := |S_0| - ap(1)$. If $C_4 = C_0$, then also $P_0 := P_0 \setminus \{1 \to 4\}$, $S_0 := S_0 \cup \{1 \to 4\}$, and $cp(4) := cp(4) - 1$.

- If $cp(2) > 1$ before the transformation, then $C_2 = C_0$ after it. In that case node 2 remains reachable, $P_0 := P_0 \setminus \{1 \to 2\}$, $cp(2) := cp(2) - 1$, and $S_0 := S_0 \cup \{1 \to 2\}$.

  Lemma. In this situation $C_1 = (\{1\}, \emptyset)$.

  Proof. Suppose $|N_1| > 1$ after the transformation. Then $4 \in N_1$, since $C_2 = C_0 \neq C_1$. So there exists a path $4 \overset{*}{\to} 1$. Also there exist $1 \to 2$, $2 \overset{*}{\to} 0$, and $0 \to 4$. So $0 \overset{*}{\to} 1$ and $1 \overset{*}{\to} 0$. $\therefore C_0 = C_1$. $\therefore$ Contradiction. $\therefore N_1 = \{1\}$. $\therefore C_1 = (\{1\}, \emptyset)$. □

  If $|S_1| = ap(1) = 0$, the remove-operation has to be performed on C_1 starting with node 1.

- If $cp(2) = 1$ before the transformation, then $C_2 \neq C_0$ after it. So the information that node 2 belongs to the s.c.c. C_0 has to be removed from the local administration of node 2,

$N_0 := N_0 \setminus \{2\}$, $P_0 := P_0 \setminus \{2 \to 5\}$, $cp(2) := cp(2) - 1$, and $|S_0| := |S_0| - ap(2) + 1$.

Lemma. In this situation $C_2 = (\{2\},\emptyset)$ and $C_1 = (\{1\},\emptyset)$.

Proof. 1° Suppose $|N_2| > 1$. Then $5 \in N_2$, and so $C_5 = C_2$. Also $C_5 = C_0$. ∴ Contradiction. ∴ $N_2 = \{2\}$. ∴ $C_2 = (\{2\},\emptyset)$.

2° Suppose $|N_1| > 1$. Then $4 \in N_1$, because $C_1 \neq C_2$ by 1°. So there exists a path $4 \overset{*}{\to} 1$, and therefore also a path $0 \overset{*}{\to} 1$. Also $1 \to 2$, $2 \to 5$, and $5 \overset{*}{\to} 0$. ∴ $1 \overset{*}{\to} 0$. ∴ $C_0 = C_1$. ∴ Contradiction. ∴ $N_1 = \{1\}$. ∴ $C_1 = (\{1\},\emptyset)$. □

So $P_0 := P_0 \setminus \{1 \to 2\}$, $S_0 := S_0 \cup \{2 \to 5\}$, $cp(5) := cp(5) - 1$. If $|S_1| = ap(1) = 0$, the remove-operation has to be performed on C_1 starting with node 1.

2.2.2. Suppose $C_4 = C_0 \wedge C_2 \neq C_0$ before the transformation. Then there existed no path $2 \overset{*}{\to} 0$ before the transformation. So before and after the transformation $C_2 \neq C_0$ and $C_5 \neq C_0$. Of course $C_4 = C_0$ after the transformation. So $P_0 := (P_0 \cup \{0 \to 4\}) \setminus \{0 \to 1\}$, $cp(1) := cp(1) - 1$, $cp(4) := cp(4) + 1$, and $S_5 := S_5 \cup \{6 \to 5\}$.

- If $C_3 = C_0$ before and therefore also after the transformation, then $C_6 = C_0$, and so $N_0 := N_0 \cup \{6\}$, $P_0 := (P_0 \cup \{0 \to 6, 6 \to 3\}) \setminus \{0 \to 3\}$, $cp(6) := 1$, and the information that node 6 belongs to the s.c.c. C_0 has to be added to the local administration of node 6.

- If $C_3 \neq C_0$ before and therefore also after the transformation, then $C_6 := (\{6\},\emptyset)$, $S_6 := \{0 \to 6\}$, $cp(6) := 0$, and $S_3 := (S_3 \cup \{6 \to 3\}) \setminus \{0 \to 3\}$.

-- Suppose $C_2 = C_5$ before and therefore also after the transformation. Then node 2 remains reachable.

-- Suppose $C_2 \neq C_5$ before the transformation, then also after it. So $C_2 = (\{2\},\emptyset)$ before and after the transformation.

--- If $cp(1) > 1$ before the transformation, then $C_1 = C_0$ after it. In that case node 1 remains reachable.

--- If $cp(1) = 1$ before the transformation, then $C_1 \neq C_0$ after it.

Lemma. In this situation $C_1 = (\{1\},\emptyset)$.

Proof. Suppose $|N_1| > 1$. Because $C_2 \neq C_1$ before the transformation, $C_2 \neq C_1$ after it. So $4 \in N_1$, and therefore $C_1 = C_4 \wedge C_4 = C_0$.
$\therefore$ Contradiction. $\therefore N_1 = \{1\}$. $\therefore C_1 = (\{1\},\emptyset)$. □

So $cp(1) := 0$, $P_0 := P_0 \setminus \{1 \to 4\}$, $cp(4) := cp(4) - 1$, $|S_0| := |S_0 \cup \{1 \to 4\}| - ap(1)$, and the information that node 1 belongs to the s.c.c. C_0 has to be removed from the local administration of node 1. If $|S_1| = ap(1) = 0$, the remove-operation has to be performed on C_1 starting with node 1.