Implementing SASL without garbage collection

Aerts, J.P.H.

Published: 01/01/1981

Document Version
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
Implementing SASL without garbage collection

by

J.P.H. Aerts

EUT-Report 81-WSK-05

November 1981
<table>
<thead>
<tr>
<th>Contents</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>1</td>
</tr>
<tr>
<td>1. SASL</td>
<td>2</td>
</tr>
<tr>
<td>2. Combinatory logic</td>
<td>7</td>
</tr>
<tr>
<td>3. The implementation of SASL</td>
<td>22</td>
</tr>
<tr>
<td>4. Store management</td>
<td>33</td>
</tr>
<tr>
<td>5. A mild variant of combinatory logic</td>
<td>52</td>
</tr>
<tr>
<td>6. Some final remarks</td>
<td>62</td>
</tr>
<tr>
<td>References</td>
<td>64</td>
</tr>
<tr>
<td>Appendix 1</td>
<td>65</td>
</tr>
<tr>
<td>Appendix 2</td>
<td>66</td>
</tr>
</tbody>
</table>
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 ( ).
conditional expressions

These are of the form

\[ \text{condition} \times \text{object1} ; \text{object2} , \]

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.

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 \( ~ \), followed by one or more definitions separated by semicolons. 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

\[ \text{'namelist'} = \text{'expression'} , \]

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

\[
a = 1 ; b = 2 ; c = 3,4 .
\]

\[
a : b : () = x < y \rightarrow -1, \text{false} ; 1, \text{true}
\]

has the same effect as

\[
a = x < y + -1 ; 1 ; b = x < y \rightarrow \text{false} ; \text{true} .
\]

Another form of definition is

\['\text{function form}' = 'expression' ,\]

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: \(\text{id } x = x\)

defines the identity function.

If the function \(\gcd\), defined by

\[
\gcd(x,y) = x > y + \gcd(x - y,y) ;
\]

\[
x < y \rightarrow \gcd(x,y - x) ;
\]

\[x\]

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

\[
\text{odd } x = x \geq 1 \land \text{even } (x - 1) ;
\]

\[
\text{even } x = x = 0 \lor \text{true} ; \text{odd } (x - 1)
\]

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

Examples: count a b = a > b \lor () ; a : count (a + 1) b

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

\[
\text{from } n = n : \text{from} (n + 1)
\]

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:

\[
\text{hd} \ (\text{from } 6) \ \text{where} \ \text{from } n = n : \text{from} (n + 1) .
\]

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

\[
\text{from } 6 = 6 : \text{from} (6 + 1) .
\]

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

\[
\text{odd } x = x \geq 1 \land \text{even } (x - 1) ; \\
\text{even } x = x = 0 \lor \text{odd } (x - 1) .
\]

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 divisible 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 

\[
x = 1 : m (m (\text{mul} 2 x) (\text{mul} 3 x)) (\text{mul} 5 x) ; \\
\text{mul } n (a : 1) = a * n : \text{mul } n 1 ; \\
m (a : b) (c : d) = a < c + a : m b (c : d) ; \\
c < a + c : m (a : b) d ; \\
a : m b d .
\]

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 = N.

(ii) [N/x]a = a for all atoms a ≠ x.

(iii) [N/x](M_1 M_2) = ([N/x]M_1)([N/x]M_2).

For mutually distinct x_1,...,x_n, and for any terms N_1,...,N_n, we similarly define [N_1/x_1,...,N_n/x_n]M to be the result of simultaneously substituting N_1 for x_1,...,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 \( X = UV \), the tree corresponding to \( X \) will be defined by attaching the two trees corresponding to \( U \) and \( V \) 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 position index (or simply position) 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 \( Y = UV \), then \( U \) has position \( (q,0) \) and \( V \) has position \( (q,1) \).

Definition 2.4. The relation \( X \succ Y \) (\( X \) contracts to \( Y \)) is defined by induction as follows:

Axiom-schemes:

(I) \( IX \succ X \) for all \( X \)

(K) \( KXY \succ X \) for all \( X,Y \)

(S) \( SXYZ \succ XZ(YZ) \) for all \( X,Y,Z \)
(B) $BXYZ = X(YZ)$ for all $X,Y,Z$

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

**Deduction-rules:**

\[(\mu)\] $X \Rightarrow Y$ implies $ZX \Rightarrow ZY$ for all $X,Y,Z$

\[(\nu)\] $X \Rightarrow Y$ implies $XZ \Rightarrow YZ$ for all $X,Y,Z$.

The reflexive and transitive closure of '$\Rightarrow$' is denoted by '$\Rightarrow\Rightarrow$'. The relation '$X \Rightarrow Y$' 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

\[(*)\] $bx_1x_2\ldots x_m \Rightarrow A$ (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 > 0), such that $X = vV_1\ldots V_n$.

**Proof.** (Induction on the construction of $X$)

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

2° If $X = UV$, we have by the induction hypothesis: there is a unique atom $u$ and there are unique terms $U_1,\ldots,U_m$ (m > 0), such that $U = uU_1\ldots U_m$.

Choose $v = u$, $n = m + 1$, $V_i = U_i$ ($1 \leq i \leq m$), and $V_n = V$. Then $X = vV_1\ldots V_n$.

The uniqueness follows from the assumption that $XY = UV$ implies $X = U$ and $Y = V$, and from the induction hypothesis.

If $X = vV_1\ldots V_n$, 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

\[ (*) \ b x_1 \ldots x_m \rightarrow A \quad (m > 0), \]

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

Lemma 2.7. X \( \not\rightarrow \) Y if and only if there is a sequence \( X_0, \ldots, X_n \) (n \( \geq 0 \)), such that \( X = X_0, Y = X_n \), and \( X_i-1 \rightarrow X_i \), \( 0 < i \leq n \).

Such a sequence will be called a reduction (from X to Y). The number n is the length of the reduction, and \( X_{i-1} \rightarrow X_i \) 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 \( X \not\rightarrow Y \), then \([N_1/x_1, \ldots, N_m/x_m]X \not\rightarrow [N_1/x_1, \ldots, N_m/x_m]Y\).

Proof. Induction on the proof that \( X \not\rightarrow Y \). 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) \([x]M = KM\) \quad \text{if } x \notin M

(b) \([x]x = I\)

(c) \([x]u_x = u\) \quad \text{if } x \notin u

(d) \([x]u_{y_2} = Bu([x]y_2)\) \quad \text{if } x \notin u

(e) \([x]y_1 V = C([x]y_1)V\) \quad \text{if } x \notin V

(f) \([x]y_1 y_2 = S([x]y_1)([x]y_2)\).

**Theorem 2.10. (Law of abstraction)**

For all algorithms, for all combinatory terms \(M\) and \(N\), and for each variable \(x\):

\[ ([x]M)N \rightarrow^{*} [N/x]M. \]

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

\(1^o\) If \(M\) is an atom, there are only two possible cases.

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

(ii) If \(M\) is an atom distinct from \(x\), then \(x \notin M\), and only specification (a)
is applicable: \((\l{x}M)x = K\l{x}x M\) by the reduction rule (K).

2° If \(M = UV\), we have by the induction hypothesis: \((\l{x}U)x \not\in U\) and \((\l{x}V)x \not\in V\).

(i) Replacing \([x]M\) according to specification (a) (if \(x \not\in M\)) gives situation 1°(ii).

(ii) Replacing \([x]M\) according to specification (c) (if \(x \not\in U\) and \(V = x\)) gives: \((\l{x}M)x = ([x]U)x = Ux = M\), and thus \((\l{x}M)x \not\in M\).

(iii) Replacing \([x]M\) according to specification (d) (if \(x \not\in U\)) gives:
\[
([x]UV)x = BU([x]V)x
\]
\(\not\in U(([x]V)x)\) by the reduction rule (B)
\(\not\in UV\) by the induction hypothesis and the deduction-rule (\(\mu\)).

(iv) Replacing \([x]M\) according to specification (e) (if \(x \not\in V\)) gives a situation analogous to 2°(iii).

(v) Replacing \([x]M\) according to specification (f) gives:
\[
([x]UV)x = S([x]U)([x]V)x
\]
\(\not\in ([x]U)x([x]V)x\) by the reduction rule (S)
\(\not\in UV\) by the induction hypothesis and the deduction-rules (\(\mu\)) and (\(\nu\)).

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

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:
\[
([x_1, \ldots, x_m]M)N_1 \ldots N_m \not\in [N_1/x_1, \ldots, N_m/x_m]M .
\]

Proof. By lemma 2.9 it is enough to prove that \(([x_1, \ldots, x_m]M)x_1 \ldots x_m \not\in M\), 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 \((CT,\overset{*}{\rightarrow})\) 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 \(V\):

\[ R \overset{*}{\rightarrow} X \text{ and } R \overset{*}{\rightarrow} Y \text{ implies there exists an element } Z \text{ of } V \text{ such that } X \overset{*}{\rightarrow} Z \text{ and } Y \overset{*}{\rightarrow} Z. \]

Theorem 2.15. Let \((V,\rightarrow)\) be a general replacement system, and let \(\sqsupseteq\) be a binary relation on \(V\) such that

(1): \(\overset{*}{\rightarrow} = \overset{*}{\rightarrow}\)

(2): for all elements \(R,X,Y\) of \(V\): \(R \sqsupseteq X\) and \(R \sqsupseteq Y\) implies there exists an element \(Z\) of \(V\) such that \(X \sqsupseteq Z\) and \(Y \sqsupseteq Z\).

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

Proof. 1° First we shall prove: if \(R \sqsupseteq X\) and \(R \overset{*}{\rightarrow} Y\), there exists a \(Z\) such that

\(X \overset{*}{\rightarrow} Z\) and \(Y \overset{*}{\rightarrow} Z\).

(i) If \(R=X\), choose \(Z=X\).

(ii) If \(R \overset{*}{\rightarrow} Y\) and \(Y \overset{*}{\rightarrow} Y\), there exists by (2) a \(Z\) such that \(Y \overset{*}{\rightarrow} Z\) and \(X \overset{*}{\rightarrow} Z\). So \(Y \overset{*}{\rightarrow} Z\) and \(Y \overset{*}{\rightarrow} Y\). By induction we conclude:

there exists a \(Z\) such that \(Z \overset{*}{\rightarrow} Z\) and \(Y \overset{*}{\rightarrow} Z\). (This is the Church-Rosser property of \((V,\rightarrow)\).)

2° Now we shall prove: if \(R \overset{*}{\rightarrow} X\) and \(R \overset{*}{\rightarrow} Y\), there exists a \(Z\) such that

\(X \overset{*}{\rightarrow} Z\) and \(Y \overset{*}{\rightarrow} Z\). (This is the Church-Rosser property of \((V,\rightarrow)\).)

(i) If \(R=X\), choose \(Z=Y\).
(ii) If $R \xrightarrow{1} X_1$ and $X_1 \xrightarrow{1} X$, there exists by $I^0$ a $Z_1$ such that $X_1 \xrightarrow{1} Z_1$ and $Y \xrightarrow{1} Z_1$. So $X_1 \xrightarrow{1} Z_1$ and $X_1 \xrightarrow{1} X$. By induction we conclude:

there exists a $Z$ such that $Z_1 \xrightarrow{1} Z$ and $X \xrightarrow{1} Z$ and $Y \xrightarrow{1} Z$.

3° Finally we shall prove the Church-Rosser property of $(V, \Rightarrow)$.

Suppose $R \xrightarrow{1} X$ and $R \xrightarrow{1} Y$. Then by (I) also $R \xrightarrow{1} X$ and $R \xrightarrow{1} Y$. By $Z^o$ we conclude: there exists a $Z$ such that $X \xrightarrow{1} Z$ and $Y \xrightarrow{1} Z$. Then by (I) also $X \xrightarrow{1} Z$ and $Y \xrightarrow{1} Z$.

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 $R_i$ and $R_j$ 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 $X \xrightarrow{s} Y$. The reflexive and transitive closure of $\xrightarrow{s}$ will be denoted by $\xrightarrow{s}$. If $X \xrightarrow{s} Y$, then $X$ reduces to $Y$ by a special reduction.

Theorem 2.17. $X \xrightarrow{s} Y$ if and only if $X \xrightarrow{s} Y$.

Proof. $I^0$ A special contraction is a simultaneous contraction of zero or more non-overlapping redexes. So: if $U \xrightarrow{s} V$, then $U \xrightarrow{s} V$.

$\vdash$: If $X \xrightarrow{s} Y$, then $X \xrightarrow{s} Y$. 
First we shall prove: if $U \succ V$, then $U \not\succeq V$.

Suppose $U \succ V$. 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 $U_0 = U$, 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 $U_{i-1} \succ U_i$ $(1 \leq i \leq k)$, and $U_k = V$. So $U \not\succeq V$.

(ii) Suppose $X \not\succeq Y$. Then there are $X_0, \ldots, X_n$ $(n \geq 0)$ such that $X = X_0 \not\succeq X_1 \not\succeq \ldots \not\succeq X_n = Y$. So $X \not\succeq Y$.

\[ \text{Theorem 2.18.} \text{ If } U \succeq X \text{ and } U \succeq Y, \text{ then there exists a } Z \text{ such that } Y \succeq Z \text{ and } X \succeq Z. \]

\textbf{Proof.} (Induction on the construction of $U$)

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

2° Let $U = U_1 U_2$, and suppose $U \succeq X$ by the simultaneous contraction of the non-overlapping redexes $R_1, \ldots, R_k$ $(k \geq 0)$ and $U \succeq Y$ 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 $R_1 = U$ and $Q_1 = U$. Then $k = 1$ and $m = 1$. \(\therefore R_1 = Q_1\), and therefore $X = Y$. So choose $Z = X$.

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

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

(K) Suppose $U_1 = KV$. Then $U = U_1 U_2 = KV U_2 \succeq V = X$. Define $QV$ as the set of non-overlapping redexes that are element of $QQ$ and part
of \( \mathcal{V} \), and \( \mathcal{QU}_2 \) as the set of non-overlapping redexes that are element of \( \mathcal{Q} \& \mathcal{Q} \) and part of \( \mathcal{U}_2 \). Define \( \mathcal{Z} \) as the result of the simultaneous contraction of the elements of \( \mathcal{QV} \) in \( \mathcal{V} \), and define \( \mathcal{Z}' \) as the result of the simultaneous contraction of the elements of \( \mathcal{QU}_2 \) in \( \mathcal{U}_2 \). Then \( \mathcal{V} \triangleright \mathcal{Z} \) and \( \mathcal{U}_2 \triangleright \mathcal{Z}' \). So \( \mathcal{U} = \mathcal{U}_1 \mathcal{U}_2 = \mathcal{KU}_2 \triangleright \mathcal{KZZ}' \triangleright \mathcal{Y} \). 

\[ \therefore \mathcal{X} \triangleright \mathcal{Z} \text{ and } \mathcal{Y} \triangleright \mathcal{KZZ}' \triangleright \mathcal{Z}. \]

**(S)** Suppose \( \mathcal{U}_1 = \mathcal{SV}_1 \mathcal{V}_2 \). Then \( \mathcal{U} = \mathcal{U}_1 \mathcal{U}_2 = \mathcal{SV}_1 \mathcal{V}_2 \mathcal{U}_2 \triangleright \mathcal{V}_1 \mathcal{U}_2 (\mathcal{V}_2 \mathcal{U}_2) \triangleright \mathcal{X} \).

Define \( \mathcal{QV}_1 \) as the set of non-overlapping redexes that are element of \( \mathcal{Q} \) and part of \( \mathcal{V}_1 \) (i = 1,2), and define \( \mathcal{QU}_2 \) as the set of non-overlapping redexes that are element of \( \mathcal{Q} \) and part of \( \mathcal{U}_2 \). Define \( \mathcal{V}_1' \) as the result of the simultaneous contraction of the elements of \( \mathcal{QV}_1 \) in \( \mathcal{V}_1' \) (i = 1,2). Then \( \mathcal{V}_1 \triangleright \mathcal{V}_1' \) and \( \mathcal{V}_2 \triangleright \mathcal{V}_2' \). Define \( \mathcal{U}_2' \) as the result of the simultaneous contraction of the elements of \( \mathcal{QU}_2 \) in \( \mathcal{U}_2 \). Then \( \mathcal{U}_2 \triangleright \mathcal{U}_2' \). :\( \therefore \mathcal{U} = \mathcal{SV}_1 \mathcal{V}_2 \mathcal{U}_2 \triangleright \mathcal{SV}_1 \mathcal{V}_2 \mathcal{U}_2 \triangleright \mathcal{SV}_1 \mathcal{V}_2 \mathcal{U}_2 \triangleright \mathcal{Y} \). Define \( \mathcal{Z} = \mathcal{V}_1' \mathcal{U}_2' (\mathcal{V}_2 \mathcal{U}_2') \). Then \( \mathcal{Y} \triangleright \mathcal{Z} \), and \( \mathcal{X} \triangleright \mathcal{Z} \) by the simultaneous contraction of all the occurrences of elements of \( \mathcal{QV}_1 \cup \mathcal{QV}_2 \cup \mathcal{QU}_2 \)

in \( \mathcal{X} \).

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

(iii) Suppose \( \mathcal{Q}_1 = \mathcal{U} \) and for all \( 1 \leq i \leq k: R_i \neq \mathcal{U} \). This case is analogous to situation (ii).

(iv) Suppose for all \( 1 \leq i \leq k: R_i \neq \mathcal{U} \), and for all \( 1 \leq i \leq m: Q_i \neq \mathcal{U} \).

By definition 2.16 (ii) there exist an \( X_1 \) and an \( X_2 \) such that \( \mathcal{U}_1 \triangleright X_1 \) and \( \mathcal{U}_2 \triangleright X_2 \) and \( \mathcal{X} = X_1 X_2 \). Also there exist an \( Y_1 \) and an \( Y_2 \) such that \( \mathcal{U}_1 \triangleright Y_1 \) and \( \mathcal{U}_2 \triangleright Y_2 \) and \( \mathcal{Y} = Y_1 Y_2 \). Then by the induction hypothesis there exist a \( Z_1 \) and a \( Z_2 \) such that \( \mathcal{X}_i \triangleright Z_i \) and \( \mathcal{Y}_i \triangleright Z_i \) (i = 1,2).

Choose \( Z = Z_1 Z_2 \), then \( \mathcal{X} = X_1 X_2 \triangleright Z_1 Z_2 = Z \) and \( \mathcal{Y} = Y_1 Y_2 \triangleright Z_1 Z_2 = Z \). \[ \square \]
Theorem 2.19. Let $CT$ be the set of all combinatory terms. Then $(CT,\triangleright)$ 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 ($\mu$), ($\nu$) of definition 2.4 (with '=' instead of '搞好'), together with the rules:

$(\rho) x = x$ for all $x$

$(\sigma) x = y$ implies $y = x$ for all $x,y$

$(\tau) x = y$ 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 '搞好'.

Theorem 2.22. If $x = y$, then there is a $z$ such that $x \triangleright z$ and $y \triangleright z$.

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

Corollary 2.22.1. If $x = y$ and $y$ is in normal form, then $x \triangleright y$.

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 \( r = (r_1, \ldots, r_n) \) (\( n \geq 0 \)) 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 \( R = bU_1 \ldots U_k \), 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 $X = X_0 ⇒ ... ⇒ X_n = Y$ ($n ≥ 0$) 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 $Z = X$).

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 $Z_1 \Rightarrow Z_2 \Rightarrow \ldots \Rightarrow Z_n$ ($n \geq 2$), where $Z_n = Y$.
Let $D'_1$ be the head reduction constituted as follows:

(i) Let it start with $Z'_1 = Z_0$. Then for $i = 1$ we have a $Z'_1$ such that:

(a) the head of $Z'_1$ is the same as the head of $Z_1$,

(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 $Z_1'$ to $Z_1$.

(ii) Let there be a $Z'_1$ such that (a), (b), and (c). If $i < n$, let $Z''_{i+1}$ be
formed by contracting the head redex of $Z'_1$.

- If no residual of $R$ is at the head of $Z''_{i+1}$, we let $Z''_{i+1} = Z''_{i+1}$.

- 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. $D'_2$ is a head
reduction.

Let $D'_3$ be the internal reduction from $Z'_n$ to $Z_n = Y$. ($D'_3$ 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 $X \not\leq Y$, there is a standard reduction from $X$ to $Y$. 

- 21 -
Proof. By theorem 2.23 we may suppose that the given reduction $X \lessdot Y$ is semistandard. Let $Z$ be the last stage that is reached by head contractions alone. Then $Z = U_0U_1 \ldots U_p$, where $U_0$ is an atom (lemma 2.5). Therefore $Y = U_0V_1 \ldots V_p$, where $U_i \lessdot V_i$ ($1 \leq i \leq p$).

Now apply the same procedure to each of the reductions $U_i \lessdot V_i$, and so on. This amounts to using a structural induction on $Y$. The basic step of this induction is the case where $Y = Z = U$, in which case the reduction $X \lessdot Y$ 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 $X \equiv X_0 \triangleright \ldots \triangleright X_n = Y$ ($n \geq 0$) 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 $X_{k-1} \triangleright X_k \triangleright \ldots \triangleright X_n = Y$ will be the leftmost redex in $Y$, because this reduction is a standard reduction. \( \therefore Y \) contains a redex. \( \square \)

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:

\[
\text{fac } n = n = 0 + 1 ; n * \text{fac}(n - 1) ,
\]
or as local variables like the 'x' in:

\[
(x + 1) + (x - 2) \text{ where } x = 7 .
\]

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 (definition 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:

\[
\begin{align*}
\langle \text{program} \rangle & \ ::= \langle \text{expression} \rangle \\
\langle \text{expression} \rangle & \ ::= \langle \text{expression-part} \rangle \langle \text{where-part} \rangle \\
\langle \text{where-part} \rangle & \ ::= \langle \text{empty} \rangle \mid \text{where } \langle \text{definitions} \rangle .
\end{align*}
\]

In the expression-part identifiers may occur as names of local variables,
which are defined in the where-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 definition of the function fac given at the beginning of this chapter is transformed to:

$$
\text{fac } n = ((\rightarrow((=n)0))1)((\ast n)(\text{fac}((-n)1))) \\
$$

**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:

1. $E$ where 'definition of $x$'

   or:

   $E_1$ where $f = E$ ; 'definition of $x$'

   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:

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

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

\[ 'x = E^*'. \]

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:

\[ f \ x_1 \ldots \ x_n = E \quad (3), \]

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:

\[
f = [x_1]((x_n^{'}))
\]

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 \( f \not\in E' \), the required form \( 'f = E^* ' \) is:

\[
f = [x_1]((x_n^{'}))
\]

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:

\[
f = (Y([f][x_1]((x_n^{'}))))
\]

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

\[
Yf = f(Yf) \tag{4}
\]

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

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

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

\[
2° WS(BWB)f = S(BWB)(BWB)f \text{ by } 1.
\]

So \( WS(BWB)f = BWB(BWBf) \) by \( \star \).

\[
\therefore WS(BWB)f = BWB(BWBf) = W(Bf)(BWBf) = (Bf)(BWBf) \text{ by } 1^9.
\]

So \( WS(BWB)f = f(BWBf) \) by \( \star \).

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:

\[ E_1 \text{ where } a : b = E_2, \text{ with } a \in E_1 \lor b \in E_1. \]

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 \texttt{hd} and \texttt{tl}.

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

(ii) A list occurs as a formal parameter.
An example is:

\[ f(x,y,z) = E, \text{ with } x \in E \lor y \in E \lor z \in E, \text{ and } E \text{ an expression-part}. \]

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 \texttt{hd} en \texttt{tl}.

The results of this transformation performed on our example is:

\[ f \ell = E \text{ where } x = \text{hd} \ell \; ; \; e = \text{tl} \ell \; ; \; y = \text{hd} e \; ; \; z = \text{hd} (\text{tl} e). \]

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

\[ E \text{ where } f \ x = E_1 \; ; \; g = E_2, \text{ with } g \in E_1 \land f \in E_2 \land (g \in E \lor f \in E), \]

is transformed to:

\[ E \text{ where } (f,g) = ([x]E_1, E_2). \]

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 \( O(n^2) \). 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.\( \ell \)' and 'x.r'. If the left field \( x.\ell \) contains a constant \( c \), we shall write 'x.\( \ell \) = c', and if \( x.\ell \) contains a pointer to a node \( y \), we shall write 'x.\( \ell \) = \( \ast y \)'. 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.\( \ell = I \) 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 $\text{root.} \ell = X$,

- if $X$ is not a constant, then $\text{root.} \ell = \star x$, 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 $X = X_1 X_2$. 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

$$E \quad \text{where} \quad x = E^* , \text{with} \quad x \in E ,$$

is:

$$([x]E)T ,$$

where $T$ stands for $E^*$ if $x \notin E^*$, and for $(Y([x]E^*))$ if $x \in E^*$.

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

$$Yh \triangleright h^\infty.$$ 

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 $x.l = Y$ and $x.r = h$, the result will be $x.l = h$ and $x.r = x$. Then node $x$ represents the contractum $h^\infty$.

The translation of

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

is:

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

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

$$\text{reprf}.l = e2 \text{ and } \text{reprf}.r = +\text{reprf}. \text{ We know }$$

$$(f[E_2]f \triangleright E_2$$

by theorem 2.10. The term $f$ in $(f[E_2]f$ is represented by a pointer to the node $\text{reprf}$. Therefore the result of reduction (5) will be the structure representing $E_2$ in which for each occurrence of $f$ a pointer to $\text{reprf}$ 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: \( Yh \rightarrow h \). Another possibility is to choose: \( Yh \rightarrow h(Yh) \). Then \( Yh \rightarrow h \) 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 \( a.l = +b \), \( a.r = +x \), \( b.l = +c \), \( b.r = +g \), \( c.l = B \), 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: \( a.l = t_1 \), \( a.r = +\text{new} \), \( \text{new}.l = t_2 \), and \( \text{new}.r = t_3 \), where \( \text{new} \) 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 \( \text{hd} \), 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
- \( i,j: 0 \leq i < n, 0 \leq j < n: \ i = j \lor v_i \neq v_j \) 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 \( Yf \triangleright f(Yf) \) instead of \( Yf \triangleright f^\circ \). 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 $p: v \rightarrow w$ and $q: w \rightarrow v$. 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 $p: v \rightarrow w$ and $q: w \rightarrow v$. 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. \[\Box\]

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:

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

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. 1º 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 $p: v \rightarrow y$. Of course $v$ is reachable too, and so $p \in S_i$.

2º Suppose $|S_i| > 0$. Then there exists a pointer $p: v \rightarrow w$ 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. \[\Box\]

The following arrangements will be made.

1º 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 \( \{p \in P : v \rightarrow w \text{ with } v \in N_i \land w \notin N_i \} \).

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 at 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 \( p: x \rightarrow y \), 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 \( N_1 \), 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 \( \text{lf} \) be the list of free nodes. Let \( \text{sr} \) 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:

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


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

\[
P = \text{each node of the directed graph is reachable}.
\]

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 \( \mathcal{C}_x = (N_x, P_x) \). We shall denote the set of incoming pointers of \( \mathcal{C}_x \) by \( S_x \). A pointer from node \( x \) to node \( y \) will be denoted by \( x \rightarrow y \), a path from \( x \) to \( y \) by \( x \stackrel{*}{\rightarrow} y \). If the left field of node \( x \) contains 1, and the right field contains \( r \), we shall write \( x: (1,r) \).

Let \( x \) be a node. If the value of \( \mathcal{C}_x = (N_x, P_x, S_x, ap(x), cp(x)) \) has changed after a transformation, the new value will be given in an assignment statement. Note that actually no assignments to \( \mathcal{C}_x, N_x, P_x, \) and \( S_x \) are carried out, because \( \mathcal{C}_x, N_x, P_x, \) 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 \( \text{ap}(1) := \text{ap}(1) - 1 \). If \( \text{ap}(1) = 0 \) 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: \( \lor \text{true} \rightarrow \text{true} \), and: \( \land \text{false} \rightarrow \text{false} \), 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 \( \text{ap}(0) := \text{ap}(0) + 1 \), \( \text{cp}(0) := \text{cp}(0) + 1 \), and if \( \text{cp}(0) = 1 \) after this assignment, the global administration has to be extended with \( |S_0| := \text{ap}(0) - 1 \), 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 \( \text{ap}(1) := \text{ap}(1) - 1 \), and if \( \text{ap}(1) = 0 \) 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: (+2,r). So \( \text{ap}(1) := \text{ap}(1) - 1 \) and \( \text{ap}(2) := \text{ap}(2) + 1 \).

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 \( P_0 := (P_0 \cup \{0 \rightarrow 2\}) \setminus \{0 \rightarrow 1\} \), \( \text{cp}(1) := \text{cp}(1) - 1 \), and \( \text{cp}(2) := \text{cp}(2) + 1 \).

- If \( \text{ap}(1) = 0 \), then \( N_0 := N_0 \setminus \{1\} \), \( \text{cp}(2) := \text{cp}(2) - 1 \), and the remove-operation has to be performed on \( C_1 = (\{1\},\emptyset) \) starting with node 1.
- If \( \text{ap}(1) > 0 \) and \( \text{cp}(1) > 0 \), then \( C_0 = C_1 \) after the transformation as well, and node 1 remains reachable.
- If \( \text{ap}(1) > 0 \) and \( \text{cp}(1) = 0 \), node 1 no longer belongs to \( C_0 \), but \( C_1 = (\{1\},\emptyset) \). So \( N_0 := N_0 \setminus \{1\} \), \( |S_0| := |S_0 \cup \{1 \rightarrow 2\}| - \text{ap}(1) \), \( P_0 := P_0 \setminus \{1 \rightarrow 2\} \), \( \text{cp}(2) := \text{cp}(2) - 1 \), and node 1 will no longer contain the information that it belongs to the s.c.c. \( C_0 \). Node 1 remains reachable, because \( |S_1| = \text{ap}(1) > 0 \).

(ii) Suppose that before the transformation 0, 1, and 2 belonged to three different strongly connected components. Then after the transformation \( S_2 := S_2 \cup \{0 \rightarrow 2\} \) and \( S_1 := S_1 \setminus \{0 \rightarrow 1\} \). The nodes 0 and 2 remain reachable. If \( |S_1| = \text{ap}(1) = 0 \), 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 \( S_1 := (S_1 \cup \{0 \rightarrow 2\}) \setminus \{0 \rightarrow 1\} \). Before the transformation \( C_0 \) and
C₀ were reachable, and since the values of |S₀| and |S₁| have not changed, C₀ and C₁ 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: (t₂, +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 C₀ = ({0}, Ø) 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 S₀ := S₀ \ {0 → 3} and S₁ := S₁ \ {0 → 1}. (Note that it is possible that two or more Cᵢ's (1 ≤ i ≤ 4) are identical.) If |S₁| = 0 after these assignments, the remove-operation has to be performed on C₁ starting with node 1.

(ii) If |N₀| > 1 before the transformation, then 1 ∈ C₀, C₁ = C₀ before the transformation, and therefore C₂ = C₀ ∨ C₃ = C₀.
Suppose $C_3 = C_0 \wedge C_2 \neq C_0$ before the transformation. Then $C_3 = C_0$ after it, $P_0 := (P_0 \cup \{0 \rightarrow 3\}) \setminus \{0 \rightarrow 1\}$, $cp(1) := cp(1) - 1$, 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. Contradiction. $|N_1| = 1$.

So $N_0 := N_0 \setminus \{1\}$, $|S_0| := |S_0 \cup \{1 \rightarrow 3\}| - ap(1)$, $P_0 := P_0 \setminus \{1 \rightarrow 3\}$, $cp(3) := cp(3) - 1$, 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 $P_0 := (P_0 \cup \{0 \rightarrow 3\}) \setminus \{0 \rightarrow 1\}$, $cp(1) := cp(1) - 1$, and $cp(3) := cp(3) + 1$. Otherwise $S_3 := S_3 \cup \{0 \rightarrow 3\}$, $P_0 := P_0 \setminus \{0 \rightarrow 1\}$, and $cp(1) := cp(1) - 1$.

If $cp(1) > 1$ before the transformation, then $C_1 = C_0$ after the transformation, 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 $0: (I, t_4)$. 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 $5^\circ (i)$.

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

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

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

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

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

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.

$7^\circ$ Suppose $0: (t_1, t_2)$ and $1: (v, \text{true})$. (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 $0: (I, \text{true})$. 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 $S_2 := S_2 \setminus \{0 \rightarrow 2\}$. 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 \rightarrow 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 $(\land \text{false} y)$. 

8° Suppose 0: $(+1,+3)$, 1: $(+2,+4)$, and 2: $(\rightarrow,\text{false})$. (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: $(1,+3)$. 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 $S_1 := S_1 \setminus \{0 + 1\}$. 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 $C_3 = C_0 \lor C_1 = C_0$ before the transformation.

- Suppose $C_3 = C_0 \land C_1 \neq C_0$ before the transformation, then also after it. So $S_1 := S_1 \setminus \{0 + 1\}$, 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 $P_0 := P_0 \setminus \{0 + 1\}$ and $cp(1) := cp(1) - 1$.

If $cp(1) > 1$ before the transformation, then $C_4 = C_0$ after the transformation as well, and all nodes remain reachable.
If \( \text{cp}(1) = 1 \) before the transformation, then \( C_1 \neq C_0 \) after it.

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

Suppose \( 0: (t_1, t_3), 1: (t_2, t_4), \) and \( 2: (+, \text{true}) \). (If \( O.r \) or \( L.r \) contains a constant, not all cases treated below are possible.) Contraction of the redex represented by node 0 leads to \( 0: (I, t_4) \). It is easy to see that

\[
\begin{align*}
\text{ap}(1) &:= \text{ap}(1) - 1, \\
\text{ap}(4) &:= \text{ap}(4) + 1, \\
\text{ap}(3) &:= \text{ap}(3) - 0.
\end{align*}
\]

(i) Suppose \( C_0 = (\{0\}, \emptyset) \) before the transformation and suppose node 3 is not the same node as node 0. In that case

\[
\begin{align*}
S_1 &= S_1 \setminus \{0 \rightarrow 1\}, \\
S_4 &= S_4 \cup \{0 \rightarrow 4\}, \\
S_3 &= S_3 \setminus \{0 \rightarrow 3\}.
\end{align*}
\]

(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 \( C_1 = C_0 \lor C_3 = C_0 \) before the transformation.

- If \( C_1 = C_0 \) before the transformation, then also \( C_4 = C_0 \). In that case

\[
\begin{align*}
C_4 &= C_0 \text{ after the transformation}, \\
P_0 &= (P_0 \cup \{0 \rightarrow 4\}) \setminus \{0 \rightarrow 1\}, \\
\text{cp}(4) &= \text{cp}(4) + 1, \text{ and } \text{cp}(1) := \text{cp}(1) - 1.
\end{align*}
\]

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

If \( \text{cp}(1) = 1 \) 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 \( 2 \notin N_1 \). \( 4 \in N_1 \).

\( C_1 = C_4 = C_0 \). \( \therefore \) Contradiction. \( \therefore N_1 = \{1\} \). \( \therefore C_1 = (\{1\}, \emptyset) \). \( \square \)

So \( N_0 := N_0 \setminus \{1\} \), \( P_0 := P_0 \setminus \{1 \rightarrow 4\} \), \( \text{cp}(4) := \text{cp}(4) - 1 \), \( |S_0| := |S_0 \cup \{1 \rightarrow 4\}| - \text{ap}(1) \), and node 1 will no longer contain the information that it belongs to the s.c.c. \( C_0 \). If \( |S_1| = \text{ap}(1) = 0 \), 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 $S_1 := S_1 \setminus \{0 \rightarrow 1\}$. 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, $S_4 := S_4 \cup \{0 \rightarrow 4\}$.

-- If $C_3 \neq C_0$ before the transformation, then also after it. So $S_3 := S_3 \setminus \{0 \rightarrow 3\}$. 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 $P_0 := P_0 \setminus \{0 \rightarrow 3\}$ 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 $0 \rightarrow 1$, belonged to the same s.c.c. $K' = (N', P')$. (So $K' = C_0 = C_1$ before the transformation.) After the transformation the pointer $0 \rightarrow 1$ 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 $C_0 = (N' \setminus \{1\}, P' \setminus \{p: v \rightarrow w \mid v = 1 \lor w = 1\})$.

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 $v \in N \land w \in N$, 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 ∈ NC, then x is 
reachable. If x contains a pointer x → y, with y ∈ N\NC and y is marked 
"new", then y is a reachable node marked "new". So if y ∉ 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 ∈ 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 ∈ T, remove x from T,
- for each pointer x → y, with y ∈ N\NC ∧ y is marked "new" ∧ y ∉ 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 ∈ NC the number cp(x) the initial value 0.
For each node $x \in \text{NC}$:
- add to the local administration of $x$ the information that $x$ belongs to $C$,
- $\text{SC} := \text{SC} + \text{ap}(x)$,
- for each pointer $x \rightarrow y$:
  -- if $y \in \text{NC}$, then $\text{cp}(y) := \text{cp}(y) + 1$, $\text{SC} := \text{SC} - 1$, and if $y \in \text{T}$, remove $y$ from $\text{T}$,
  -- if $y \in \text{NC} \land y$ is marked "new" $\land y \notin \text{T}$, then add $y$ to $\text{T}$,
  -- if $y \in \text{NC} \land y$ is already unmarked, or if $y \notin \text{N}$, then do nothing.

The second step of the algorithm for finding the strongly connected components of $K$ is repeated until $\text{T} = \emptyset$. It is possible that, when $\text{T} = \emptyset$, 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 $1 \neq x$, because $x \in \text{N}$. 
\[ \therefore x \text{ is reachable.} \therefore \text{Contradiction.} \]

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 $x \in \text{N}$, 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. \[ \therefore v_n = x \text{ is reachable.} \therefore \text{Contradiction.} \]
Let \( \text{lf} \) be the list of free nodes. If an element of \( L \) is added to \( \text{lf} \), 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 \( \text{sr} \), and each element of \( \text{sr} \) will be removed from the graph by the remove-operation described before. The set \( \text{sn} \) contains the elements of \( L \) that are already reached, but not yet added to \( \text{lf} \). The last lemma now leads to the following algorithm for adding all elements of \( L \) to \( \text{lf} \), in which as usual \( \text{C}_y \) denotes the s.c.c. to which node \( y \) belongs, and \( \text{S}_y \) denotes the set of incoming pointers of \( \text{C}_y \).

```
proc p(x):
    proc field(z):
        if z is a constant ⇒ skip
        if z is a pointer to a node \( y \) ∧ \( y \notin \text{lf} \) ⇒
            if \( y \) is marked "new" ⇒ \( \text{sn} := \text{sn} \cup \{y\} \)
            if \( y \) is unmarked ⇒ \( \text{ap}(y) := \text{ap}(y) - 1; |S_y| := |S_y| - 1; \)
                if \( |S_y| = 0 \) ⇒ \( \text{sr} := \text{sr} \cup \{\text{C}_y, y\} \)
                if \( |S_y| > 0 \) ⇒ skip
        fi
    fi
    corp;
    \( \text{lf} := \text{lf} \cup \{x\}; \text{field}(x.l); \text{field}(x.r); \text{sn} := \text{sn} \setminus \{x\} \)
    corp;
    \( \text{sr} := \emptyset; \text{sn} := \{x\}; \text{do} \ \text{sn} \neq \emptyset ⇒ p(\text{an element of sn}) \text{ od}; \)
    \text{do} \ \text{sr} \neq \emptyset ⇒ \text{remove(\text{an element of sr}) \text{ 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 $\Lambda$.

(ii) Every atom is a term.

(iii) If $T_1$ and $T_2$ are terms and $\text{coms}$ is a possibly empty sequence of combinators, then $(T_1 \text{coms} T_2)$ 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 $(T_1 \text{coms} T_2)$ such that
(T1 ≠ A or coms contains an S or a C)
and (T2 ≠ A 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 \( T \), \( M \in T \), by induction on the construction of \( T \) as follows:

(i) If \( T = A \), then \( M \in T = \text{false} \).

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

(iii) If \( T = (T1 \text{ coms } T2) \), then \( M \in T = (M \in T1 \text{ or } M \in T2) \).

We shall write \('M \not\in T' if \( M \) does not occur in \( T \).

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

(i) If \( T = x \), then \( [x]T = A \).

(ii) If \( T = (T1 \text{ coms } T2) \), then

- if \( x \in T1 \) and \( x \in T2 \), then \( [x]T = ([x]T1 \text{ Scoms } [x]T2) \),
- if \( x \in T1 \) and \( x \notin T2 \), then \( [x]T = ([x]T1 \text{ Ccoms } T2) \),
- if \( x \notin T1 \) and \( x \in T2 \), then \( [x]T = (T1 \text{ Bcoms } [x]T2) \).

'\([x]T\)' is pronounced as '\( x \) abstracted from \( T \)'.

**Theorem 5.5.** If \( T \) is a healthy term, \( x \) an identifier, and \( x \in T \), then \( [x]T \) is a healthy term.

**Proof.** (Induction on the construction of \( [x]T \))
(i) If \( T = x \), then \( [x]T = \Lambda \) and so \([x]T\) is healthy.

(ii) If \( T = (T_1 \coms T_2) \), there are three possibilities:

- Suppose \( x \in T_1 \) and \( x \in T_2 \). Then \( [x]T = ([x]T_1 \coms [x]T_2) \), and \( T_1 \) and \( T_2 \) are both healthy. By induction we have: \( [x]T_1 \) and \( [x]T_2 \) are both healthy.

  So: (\( \coms \) contains an \( S \)) and (\( [x]T_1 \) and \( [x]T_2 \) are both healthy).

  \( \therefore [x]T \) is healthy.

- Suppose \( x \in T_1 \) and \( x \notin T_2 \). Then \( [x]T = ([x]T_1 \coms T_2) \), and: (\( T_2 \neq \Lambda \) or \( \coms \) contains an \( S \) or an \( A \)) and (\( T_1 \) and \( T_2 \) are both healthy). By induction we have: \( [x]T_1 \) is a healthy term.

  So: (\( \coms \) contains a \( C \)) and (\( T_2 \neq \Lambda \) or \( \coms \) contains an \( S \) or an \( A \)) and (\( [x]T_1 \) and \( T_2 \) are both healthy).

  \( \therefore [x]T \) is healthy.

- Suppose \( x \notin T_1 \) and \( x \in T_2 \). This case is analogous to the case that \( x \in T_1 \) and \( x \notin T_2 \).

\[ \square \]

**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
\[
T_i = [x_{i+1}]T_{i+1}, \quad 0 \leq i < m, \quad \text{and} \quad T_m = T.
\]

**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) \( [N/x]A = A \),

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

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

(iv) \( [N/x](M_1 \coms M_2) = ([N/x]M_1 \coms [N/x]M_2) \).
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 \( X \triangleright Y \), \( X \) contracts to \( Y \), is defined by induction as follows:

**Contraction-rules:**

(S): \(((T_1 \textsf{coms} T_2) \ h) \triangleright ([T_1 \ h] \textsf{coms} [T_2 \ h])\)

(C): \(((T_1 \textsf{coms} T_2) \ h) \triangleright ([T_1 \ h] \textsf{coms} T_2)\)

(B): \(((T_1 \textsf{coms} T_2) \ h) \triangleright (T_1 \textsf{coms} [T_2 \ h])\)

where \([T \ h]\) is short for:

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

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

**Deduction-rules:**

(\(\mu\)): if \( X \triangleright Y \), then \((X \textsf{coms} T) \triangleright (Y \textsf{coms} T)\)

(\(\nu\)): if \( X \triangleright Y \), then \((T \textsf{coms} X) \triangleright (T \textsf{coms} Y)\).

Contraction-rules are defined for terms of the form \(((T_1 \textsf{coms}^+ T_2) \ h)\), where \( T_1 \), \( T_2 \), 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 \(((T_1 \textsf{coms}^+ T_2) \ h)\) is called a redex. If the leading combinator of \( \textsf{coms}^+ \) is an \( \Lambda \), we call it an \( \Lambda \)-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 \( T_{i-1} \triangleright T_i \), \( 0 < i \leq n \), is called a reduction (from \( T_0 \) to \( T_n \)). The number \( n \) is called the length of the reduction, and \( T_{i-1} \triangleright T_i \) 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 '≽'.

If X reduces to Y, we shall write 'X ≽ Y'.

Theorem 5.11. For healthy T and healthy, but non-empty, h, [T h] is healthy.
Proof. (i) If T = A, then [T h] = h is healthy.

(ii) If T ≠ A, then [T h] = (T h). So: (T ≠ A) and (h ≠ A) and (T and
h are both healthy). ∴ [T h] is healthy. □

Theorem 5.12. Contraction never introduces an unhealthy term.
Proof. (S): Suppose the contraction is ((T1 Scoms T2) h) ≻ ([T1 h] coms [T2 h])
and let ((T1 Scoms T2) h) be a healthy term. ∴ h ≠ A, and h and
(T1 Scoms T2) are both healthy terms. ∴ T1 and T2 are both healthy
terms. ∴ [T1 h] and [T2 h] are both healthy by theorem 5.11.
h ≠ A, and therefore [T1 h] ≠ A and [T2 h] ≠ A.
∴ ([T1 h] coms [T2 h]) is a healthy term.

(C): Suppose the contraction is ((T1 Ccoms T2) h) ≻ ([T1 h] coms T2)
and let ((T1 Ccoms T2) h) be a healthy term. ∴ h ≠ A, and h and
(T1 Ccoms T2) are both healthy terms. ∴ (T1 and T2 are both
healthy) and (T2 ≠ A or Ccoms contains an S or a B). ∴ [T1 h]
is healthy by theorem 5.11. h ≠ A, and therefore [T1 h] ≠ A.
Also we have (T2 is healthy) and (T2 ≠ A or coms contains an S
or a B).
∴ ([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) ≻ (Y coms T) and
X \gg Y. If (X \text{ coms } T) is healthy and X \gg Y, we have:

- X and T are both healthy,
- X \not\in \Lambda,
- T \not\in \Lambda or \text{ coms contains an S or a B}.

By induction we have: Y is healthy. Y is a contractum, and so

Y \not\in \Lambda.

\therefore (Y \text{ coms } T) is a healthy term.

(\nu): analogous to (\mu). \Box

\textbf{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 \gg Y, then \([N_1/x_1, \ldots, N_m/x_m]X \gg [N_1/x_1, \ldots, N_m/x_m]Y\), where U \gg V means:

U reduces to V and the length of this reduction is n.

\textbf{Proof.} Induction on n. \Box

\textbf{Theorem 5.14. (Law of abstraction)}

For all terms M and N, and for each identifier x \in M: \([[x]M N] \gg [N/x]M].

\textbf{Proof.} By theorem 5.13 it is enough to prove that \([[x]M x] \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 = (M_1 \text{ coms } M_2), then:

- If x \in M_1 and x \in M_2, then by induction we have: \([[x]M_1 x] \gg M_1 and

[[x]M_2 x] \gg M_2.

So: [[x]M x] = [[[x]M_1 \text{ Scoms } [x]M_2) x] = (((x)M_1 \text{ Scoms } [x]M_2) x) \gg

\gg ([[x]M_1 x] \text{ coms } [[x]M_2 x]) \gg (M_1 \text{ coms } M_2) = M.

- The other two cases are analogous. \Box
Theorem 5.15. For mutually distinct identifiers $x_1, \ldots, x_m$, and terms $N_1, \ldots, N_m$, and $M$:

$$[[\ldots[[x_1, \ldots, x_m]M \ldots] N_1]\ldots] N_m] \not\to [N_1/x_1, \ldots, N_m/x_m]M.$$  

Proof. By theorem 5.13 it is enough to prove that

$$[[\ldots[[x_1, \ldots, x_m]M x_1]\ldots] x_m] \not\to M,$$

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 $R_1$ and $R_2$ ($R_1 \neq R_2$) be two redexes that occur in the same term. Then there are three possibilities:

- $R_1$ occurs in $R_2$,
- $R_2$ occurs in $R_1$,
- $R_1 \not\in R_2$ and $R_2 \not\in R_1$; in this case $R_1$ and $R_2$ are called non-overlapping redexes.

Let $R_1, \ldots, R_n$ ($n \geq 0$) be redexes that occur in the same term. If $\forall i, j: 1 \leq i \leq n, 1 \leq j \leq n: i = j \lor R_i$ 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 '$\Rightarrow$'.

With '$\Rightarrow$' we shall denote the reflexive and transitive closure of '$\Rightarrow$'.

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

**Theorem 5.18.** If $X \triangleright Y$ if and only if $X \triangleright S Y$.

**Proof.** The same as that of theorem 2.17. \hfill \Box

**Theorem 5.19.** If $T \triangleright S X$ and $T \triangleright S Y$, there exists a $Z$ such that $X \triangleright S Z$ and $Y \triangleright S Z$.

**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 $T = (T_1 \text{ combos} T_2)$. Suppose $T \triangleright S X$ by the simultaneous contraction of the non-overlapping redexes $R_1, \ldots, R_k$ ($k \geq 0$), and suppose $T \triangleright S Y$ 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 $R_1 = T$ and $Q_1 = T$. Then $k = 1$ and $m = 1$. \because $R_1 = Q_1$, and therefore $X = Y$. So choose $Z = X$.

(ii) Suppose $R_1 = T$ and $A_i: 1 \leq i \leq m; Q_i \neq T$. Then $k = 1$.

(S) Suppose $T = ((T_1 \text{ Scoms} T_12) h) \triangleright S ([T_11 h] \text{ Scoms} [T_12 h]) = X$.

Also $T = (T_1 \text{ combo} T_2)$. So $T_1 = (T_11 \text{ Scoms} T_12), T_2 = h$, and coms is the empty sequence. Note that $T_1$ is not a redex. Define $R_{Q_1} = \{Q \in QQ \mid Q \in T_1\} (i = 1, 2)$. Let $h \triangleright S h^*$ by the simultaneous contraction of the non-overlapping redexes of $R_{Q_2}$. $T_1$ is not a redex and $A: Q: Q \in R_{Q_1}: Q$ is a redex. So $A: Q: Q \in R_{Q_1}: Q \neq T_1$, and therefore $A: Q: Q \in R_{Q_1}: Q \in T_11 \lor Q \in T_12$. Let $T_11 \triangleright S T_11^*$ by the simultaneous contraction of those redexes of $R_{Q_1}$ that occur in $T_11 (i = 1, 2)$. Then $T = ((T_1 \text{ Scoms} T_12) h) \triangleright S ((T_11^* \text{ Scoms} T_12^*) h^*) = Y$.

Define $Z = ([T_11^* h^*] \text{ Scoms} [T_12^* h^*])$. Then $Y \triangleright S Z$, and $X \triangleright S Z$ by the simultaneous contraction of all occurrences of elements of $R_{Q_1} \cup R_{Q_2}$. 
(C) and (B): analogously.

(iii) Suppose $Q_i = T$ and $A_i: 1 \leq i \leq k: R_i \neq T$. This case is analogous to (ii).

(iv) Suppose $A_i: 1 \leq i \leq k: R_i \neq T$ and $A_i: 1 \leq i \leq m: Q_i \neq T$. By definition 5.17 (ii) there exist an $X_1$ and an $X_2$ such that $T_1 \supset X_1$ and $T_2 \supset X_2$ and $X = (X_1 \text{ cons } X_2)$. Also there exist a $Y_1$ and a $Y_2$ such that $T_1 \supset Y_1$ and $T_2 \supset Y_2$ and $Y = (Y_1 \text{ cons } Y_2)$.

Then by the induction hypothesis there exist a $Z_1$ and a $Z_2$ such that $X_i \supset Z_i$ and $Y_i \supset Z_i$ ($i = 1, 2$). Choose $Z = (Z_1 \text{ cons } Z_2)$, then $X \supset Z$ and $Y \supset Z$.

Theorem 5.20. Let $DT$ be the set of all terms. Then $(DT, \supset)$ 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 $T = (T_1 \text{ cons } T_2)$, the number of combinators that occur in $T$ has increased by $n$, not counting the combinators in $T_1$ and $T_2$.

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 \( T_1 \), and the definition of \( f \) is recursive. Suppose this definition has already been translated into a form \( f = T_2 \), where \( T_2 \) is a term from which all local variables have been removed, and \( f \in T_2 \). Then we have:

\[
T_1 \text{ where } f = T_2.
\]

This form is translated into:

\[
([f]T_1 \ (A \ Y \ [f]T_2))
\]

The contraction rule for the combinator \( Y \) is:

\[
(A \ Y \ T) \Rightarrow (T \ (T \ (T \ (...) \ )) = T^\infty \text{ for all terms } T.
\]

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

(ii) If \( T \) is of the form \( T = (T_1 \ coms \ T_2) \), the directed graph has a node, called root, with three fields, called root.\( \ell \), 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 \( T_1 = b \), with \( b = \Lambda \) or \( b \) is a constant, then root.\( \ell \) = \( b \); otherwise root.\( \ell \) = +t1, where t1 is the root of the directed graph that represents \( T_1 \). If \( T_1 \) is not a constant and not the empty term, \( T_1 \) is the name of a standard function or of the form \( T_1 = (T_11 \ coms' \ T_12) \). 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


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:

\[ S f g x \rightarrow f x (g x) \]
\[ B f g x \rightarrow f (g x) \]
\[ C f g x \rightarrow f x g \]
\[ Y f \rightarrow f^\infty \]
\[ I x \rightarrow x \]
\[ \text{hd} (P x y) \rightarrow x \]
\[ \text{tl} (P x y) \rightarrow y \]
\[ \rightarrow \text{true} x y \rightarrow x \]
\[ \rightarrow \text{false} x y \rightarrow y \]
\[ \lor \text{true} y \rightarrow \text{true} \]
\[ \lor \text{false} \text{true} \rightarrow \text{true} \]
\[ \lor \text{false} \text{false} \rightarrow \text{false} \]
\[ \land \text{false} y \rightarrow \text{false} \]
\[ \land \text{true} \text{false} \rightarrow \text{false} \]
\[ \land \text{true} \text{true} \rightarrow \text{true} \]
\[ \rightarrow \text{true} \rightarrow \text{false} \]
\[ \rightarrow \text{false} \rightarrow \text{true} \]
\[ + m n \rightarrow m + n \]
\[ - m n \rightarrow m - n \]
\[ * m n \rightarrow m * n \]
\[ \text{div} m n \rightarrow m \text{ div } n \]
\[ \text{mod} m n \rightarrow m \text{ mod } n \]
\[ - m \rightarrow -m \]
\[ mn \mapsto m \geq n \]
\[ \geq mn \mapsto m \geq n \]
\[ = mn \mapsto m = n \]
\[ \neq mn \mapsto m \neq n \]
\[ \leq mn \mapsto m \leq n \]
\[ < mn \mapsto m < n \]

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: \((6, 5)\). 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: $S_1 := S_1 \setminus \{0 + 1\}$, $S_5 := S_5 \cup \{6 + 5\}$, $S_4 := S_4 \cup \{0 + 4\}$, and $S_3 := (S_3 \cup \{6 + 3\}) \setminus \{0 + 3\}$. Also: $cp(6) := 0$ and $S_6 := \{0 + 6\}$. (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: $C_1 = C_0 \lor C_3 = C_0$ before the transformation.

2.1. Suppose $C_3 = C_0 \land C_1 \neq C_0$ 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 $N_0 := N_0 \cup \{6\}$, $cp(6) := 1$, $P_0 := (P_0 \cup \{0 + 6, 6 + 3\}) \setminus \{0 + 3\}$, $S_4 := S_4 \cup \{0 + 4\}$, $S_1 := S_1 \setminus \{0 + 1\}$, and $S_5 := S_5 \cup \{6 + 5\}$. 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 $C_2 = C_0 \lor C_4 = C_0$ 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 $N_0 := N_0 \cup \{6\}$, $cp(6) := 1$, $P_0 := (P_0 \cup \{0 + 6, 6 + 5\}) \setminus \{0 + 1\}$, $cp(5) := cp(5) + 1$, 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 $S_0 := S_0 \cup \{1 \to 2\}$.

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

\textbf{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 $4 \not\in 1$. Also there exist $1 \to 2$, $2 \not\in 0$, and $0 \to 4$. So $0 \not\in 1$ and $1 \not\in 0$. $C_0 = C_1$. $\vdash$ Contradiction. $\vdash N_1 = \{1\}, \vdash C_1 = (\{1\}, \emptyset)$. $\Box$

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

**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 \( C_5 = C_0 \). \( \therefore \) Contradiction. \( \therefore N_2 = \{2\}. \)

\( \therefore C_2 = ((2),\emptyset). \)

2° Suppose \( |N_1| > 1 \). Then \( 4 \in N_1 \), because \( C_1 \neq C_2 \) by 1°. So there exists a path \( 4 \xrightarrow{*} 1 \), and therefore also a path \( 0 \xrightarrow{*} 1 \). Also \( 1 \to 2, 2 \to 5, \) and \( 5 \xrightarrow{*} 0. \)

\( \therefore 1 \xrightarrow{*} 0. \) \( \therefore C_0 = C_1 \). \( \therefore \) Contradiction. \( \therefore N_1 = \{1\}. \)

\( \therefore C_1 = ((1),\emptyset). \) \( \square \)

So \( P_0 := P_0 \setminus \{1 \to 2\}, S_0 := S_0 \cup \{2 \to 5\}, \text{cp}(5) := \text{cp}(5) - 1. \)

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

2.2.2. Suppose \( C_4 = C_0 \land C_2 \neq C_0 \) before the transformation. Then there existed no path \( 2 \xrightarrow{*} 0 \) 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 \( P_0 := (P_0 \cup \{0 \to 4\}) \setminus \{0 \to 1\}, \text{cp}(1) := \text{cp}(1) - 1, \)
\( \text{cp}(4) := \text{cp}(4) + 1, \) 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 \( N_0 := N_0 \cup \{6\}, P_0 := (P_0 \cup \{0 \to 6, 6 \to 3\}) \setminus \{0 \to 3\}, \)
\( \text{cp}(6) := 1, \) 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 \( C_6 := ((6),\emptyset), S_6 := \{0 \to 6\}, \text{cp}(6) := 0, \)
\( \text{and S_3 := (S_3 \cup \{6 \to 3\}) \setminus \{0 \to 3\}.} \)
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 $C_1 = C_4 \land C_4 = C_0$.

$\therefore$ Contradiction. $\therefore N_1 = \{1\}$, $\therefore C_1 = (\{1\}, \emptyset)$.

So $cp(1) := 0$, $P_0 := P_0 \setminus \{1 \rightarrow 4\}$, $cp(4) := cp(4) - 1$, $|S_0| := |S_0 \cup \{1 \rightarrow 4\}| - ap(1)$, 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.