Some algorithms to decide the equivalence of recursive types

Eindhoven University of Technology

Department of Mathematics and Computing Science

Some algorithms to decide the equivalence of recursive types

by

H.M.M. ten Eikelder

91/31

Computing Science Note 91/31
Eindhoven, December 1991

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

SOME ALGORITHMS TO DECIDE THE EQUIVALENCE OF RECURSIVE TYPES

H.M.M. ten Eikelder

Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
The Netherlands

Abstract:
The paper gives a formal specification and a correctness proof of some more or less well-known algorithms for deciding the equivalence of recursive types. It turns out that these algorithms are based upon algorithms for computing the set of nodes reachable from a given node in a graph.

CONTENTS

1 Introduction

2 Equivalence of Types

3 Relation with Finite Automata

4 Reformulation as Properties of Reachable States

5 Algorithms to Compute Reachable Nodes

6 Algorithms to Compute Predicates on Reachable Nodes I

7 Algorithms to Compute Predicates on Reachable Nodes II

8 Algorithms to Decide the Equivalence of Recursive Types

9 Appendix 1

10 Appendix 2

11 Appendix 3

1. INTRODUCTION

It is well known that the possibility of recursive types is a very useful property of a programming language. If we associate a tree with each type, then recursive types give (in general) rise to infinite trees. Two types are called equivalent if the corresponding trees are identical. The equivalence problem for recursive types has extensively been studied, see for instance Coppo [Co], Cardone and Coppo [CC] or Cardelli [Ca]. The more general problem of the equivalence of solutions of systems of equations has been studied by Courcelle et al. [CKV]. That paper contains an actual algorithm that can be used for deciding type equivalence. In this paper we discuss some other algorithms used for that purpose. These algorithms appeared already in connection with the programming language Algol 68. The first one is used in the defining report of Algol 68 (§7.3 of [Wij]); it is a formalization of an algorithm given by by Koster [Ko]. A similar algorithm has more recently been described by Cardelli [Ca]. After some experimenting with these algorithms one gets the strong impression that they are indeed correct. However, as far as we know, a formal specification and a simple correctness proof have never been given.

In Section 2 we describe recursive types and we define the equivalence of recursive types. In fact the type-syntax used in Section 2 is only an example, various other type constructors can easily be added. In Section 3 we show that the equivalence of two types corresponds to the equivalence of two states in a finite automaton. This relation has already, in a less formal way, been described by Krǎl [Kr]. In Section 4 the problem is rewritten as the problem of determining whether all reachable states (from a given initial state) of a finite automaton satisfy a certain property. This leads to the reachability problem in directed graphs. In Sections 5, which is in fact the main section of this note, we discuss some (new?) algorithms for the reachability problem in a directed graph. In fact these algorithms are based upon recursive relations for the set of nodes reachable from a given node without passing through the nodes of some set. In Sections 6 and 7 these algorithms are adapted such that they can be used to check whether a predicate holds on all reachable nodes (from a given node). The application to type equivalence is given in Section 8. This ultimately results in algorithms which strongly resemble the ones used in [Wij, §7.3], [Ca] and [Ko]. Finally in appendix 1 we give some definitions concerning trees and in the appendices 2 and 3 we prove some technical theorems.

2. EQUIVALENCE OF TYPES

We shall illustrate the problem of type equivalence using the following type syntax. Let V be a set of type variables and C be a set of type constants not containing $\uparrow, \times, +$ and $\amalg$. The set of *type expressions* Texp is generated by the following rules.

$$
\begin{array}{ll}
\tau ::= u & (u \in V \cup C), \\
\tau ::= \uparrow\tau, & \\
\tau ::= \tau \times \tau, & \\
\tau ::= \tau + \tau, & \\
\tau ::= \mu(\lambda s\cdot\tau) & (s \in V).
\end{array}
$$

This syntax is rather arbitrary, other type constructors like $\rightarrow$ may also be added. Type expressions which only differ in the names of their bound variables will be identified. The set of free variables of a type expression τ will be denoted by $FV(\tau)$.

With every element of Texp a, possibly infinite, tree is associated in the following way. Let the functions d: Texp $\rightarrow \mathbb{N}$ and δ: Texp $\times\ \mathbb{N} \overset{p}{\rightarrow}$ Texp be defined by

$$
\begin{array}{llll}
d(u) = 0 & (u \in V \cup C), & & \\
d(\uparrow\tau) = 1 & \delta(\uparrow\tau, 0) = \tau, & & \\
d(\tau_0 \times \tau_1) = 2 & \delta(\tau_0 \times \tau_1, i) = \tau_i, & & (0 \leqslant i < 2) \\
d(\tau_0 + \tau_1) = 2 & \delta(\tau_0 + \tau_1, i) = \tau_i, & & (0 \leqslant i < 2) \\
d(\mu(\lambda s\cdot\tau)) = d(\tau) & \delta(\mu(\lambda s\cdot\tau), i) = \left(\delta(\tau, i) \right)^{s}_{\mu(\lambda s.\tau)}. & & (0 \leqslant i < d(\tau))
\end{array}
$$

So $\delta(\tau, i)$ is defined for $0 \leqslant i < d(\tau)$. The function δ is extended way to a partial function δ: Texp $\times\ \mathbb{N}^* \overset{p}{\rightarrow}$ Texp by

$$\delta(\tau, \varepsilon) = \tau,$$

$$\delta(\tau, i\alpha) = \delta(\delta(\tau, i), \alpha)$$ for all $i \in \mathbb{N}$ and $\alpha \in \mathbb{N}^*$ such that the right hand side is defined.

Further we introduce a function L: Texp $\rightarrow V \cup C \cup \{\uparrow, \times, +, \amalg\}$ by

$$L(u) = u \qquad (u \in V \cup C),$$

$$L(\uparrow\tau) = \uparrow,$$

$$L(\tau_0 \times \tau_1) = \times,$$

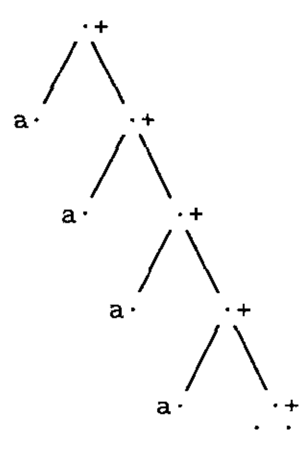
$$L(\tau_0 + \tau_1) = +,$$

$$L(\mu(\lambda s.\tau)) = \begin{cases} L(\tau) & \text{if } L(\tau) \neq s \\ \perp\!\!\!\perp & \text{if } L(\tau) = s \end{cases} .$$

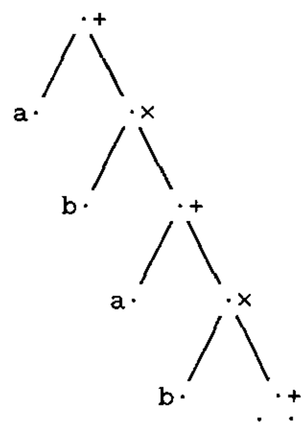
The *tree* $T(\tau)$ *corresponding to the type expression* τ is defined by

- $\mathrm{dom}(T(\tau)) = \{ \alpha \in \mathbb{N}^* \mid \delta(\tau,\alpha) \text{ is defined } \}$,
- for all $\alpha \in \mathrm{dom}(T(\tau))$: $T(\tau)(\alpha) = L(\delta(\tau,\alpha))$.

Some general definitions concerning trees are given in appendix 1. Note that if $\alpha \in \mathrm{dom}(T(\tau))$, then the type expression $\tau' = \delta(\tau,\alpha)$ describes the subtree of $T(\tau)$ in α and $L(\tau')$ is the tree label in α. The trees defined in this way are ranked trees: nodes with label in $V \cup C \cup \{\perp\!\!\!\perp\}$ have no subtrees, nodes with label $\uparrow$ have one subtree and nodes with label $\times$ or $+$ have two subtrees. For instance the trees corresponding to $\mu(\lambda s.\, a+s)$ and its unfolding $a + \mu(\lambda s.\, a+s)$ can both be depicted as



Also type expressions containing different recursive types can generate the same tree. For instance $\mu(\lambda s \cdot a+(b\times s))$ and $a + \mu(\lambda s \cdot b\times(a+s))$ both generate the following tree.



Also the type expressions $\mu(\lambda s.s)$ and $\mu(\lambda s.\mu(\lambda t.s))$ both yield the one node tree

$$\cdot \perp$$

Two type expressions will be called *equivalent* ($\cong$) if the corresponding trees are equal, formally:

$$\tau_0 \cong \tau_1 \quad \equiv \quad (T(\tau_0) = T(\tau_1)).$$

3. RELATION WITH FINITE AUTOMATA.

Since tree domains can be infinite, the equivalence of two type expressions cannot be computed by simply verifying whether the labels in all points of the corresponding tree domains are equal. Fortunately the trees turn out to be *regular trees*, i.e. the number of different subtrees is finite. For $\tau \in \text{Texp}$ let $\bar{R}(\tau) = \{\ \delta(\tau,\alpha) \mid \alpha \in \mathbb{N}^*,\ \delta(\tau,\alpha) \text{ is defined }\}$.

Theorem 3.1

For all type expressions τ the set $\bar{R}(\tau)$ is finite.

Proof:

see appendix 2.

□

In fact every type expression τ can be seen as an encoding of its tree $T(\tau)$. This theorem states that the number of encodings $\delta(\tau,\alpha)$ of the subtrees of the tree $T(\tau)$ is finite. This implies that the number of different subtrees is finite, so $T(\tau)$ is a regular tree.

The equivalence of type expressions can now be formulated in terms of the equivalence of states of a (slightly generalized) finite automaton. A *finite automaton* M is a tuple $(Q, \Sigma, \delta, F, L)$ where

- Q is a finite set of states,
- Σ is a (finite) alphabet,
- δ: $Q \times \Sigma \overset{p}{\rightarrow} Q$ is the (partial) transition function,
- F is a set of labels,
- L: $Q \rightarrow F$ is a function,

In fact a finite automata of this type can also be seen as a directed graph with nodes labeled by elements of F and edges labeled by elements of Σ. The classical finite automaton with accepting and non-accepting states can easily be simulated by this type of automaton. For $q \in Q$ we define

$$D(q) = \mathrm{dom}(\delta(q, \cdot)) = \{ a \in \Sigma \mid \delta(q,a) \text{ is defined } \}.$$

Again δ is extended to a partial function (also denoted by) δ: $Q \times \Sigma^* \overset{p}{\rightarrow} Q$ in the usual way. Furthermore for $q \in Q$ we define

$$D^*(q) = \{ \alpha \in \Sigma^* \mid \delta(q,\alpha) \text{ is defined } \}.$$

So $D^*(q)$ is the set of strings which can be "fed" to the automaton, starting from state q. Two states in a finite automaton are called *equivalent* ($\simeq$) if starting in both of them we can "feed" the same strings and encounter the same labels. Formally:

$$\begin{aligned} q_1 \simeq q_2 \equiv \big(& D^*(q_1) = D^*(q_2) \\ (3.2) \qquad & \wedge (\forall\alpha \in D^*(q_1) \cap D^*(q_2){:}{:}\ L(\delta(q_1,\alpha)) = L(\delta(q_2,\alpha)) \ )\big). \end{aligned}$$

A finite automaton will be called *label ranked* if $D(q_1) = D(q_2)$ for all states q_1, q_2 with $L(q_1) = L(q_2)$. In words, in a label ranked automaton states with the same label have the same possible transitions. It is shown in appendix 3 that for a label ranked automaton the equivalence of two states can be written as

$$(3.3) \qquad q_1 \simeq q_2 \equiv (\forall\alpha \in D^*(q_1) \cap D^*(q_2){:}{:}\ L(\delta(q_1,\alpha)) = L(\delta(q_2,\alpha)) \ ).$$

To study the equivalence of types τ_o and τ_1 we take the finite automaton $M_1 = (Q, \Sigma, \delta, L, F)$ with

- $Q = \bar{R}(\tau_0) \cup \bar{R}(\tau_1)$,
- $\Sigma = \{0,1\}$,
- $F = V \cup C \cup \{\uparrow, \times, +, \amalg\}$,
- δ and L are the same functions as in Section 2.

Then the type expressions τ_0 and τ_1 are equivalent ($\cong$) if and only if they are equivalent ($\simeq$) as states of the automaton M_1.

The automaton M_1 given above is label ranked, i.e. if $L(q_1) = L(q_2)$ then $\delta(q_1,a)$ is defined iff $\delta(q_2,a)$ is defined ($a \in \Sigma$). This amounts to the fact that the trees corresponding to type expressions are ranked trees, i.e. nodes with the same labels have the same number of subtrees. Hence to decide the equivalece q_1 and q_2 it is sufficient to compute the right hand side of (3.3).

Note that due to theorem 3.1 the set Q is indeed a finite set. Further $\Sigma = \{0,1\}$ since every node in the tree has at most two subtrees, or equivalently, every type constructor has at most two arguments. Of course this can easily be adapted for type constructors taking more (but finitely many) arguments.

4. REFORMULATION AS PROPERTIES OF REACHABLE STATES

We show that the equivalence of two states in a finite automaton M can be rewritten as a property of reachable states in a product automaton derived from M. Let $M = (Q, \Sigma, \delta, F, L)$ be a label ranked, finite automaton as described in Section 3. Define the product automaton $M_2 = (Q_2, \Sigma, \delta_2, F_2, L_2)$ by

- $Q_2 = Q \times Q$,
- $\delta_2((q_1,q_2),a) = \begin{cases} (\delta(q_1,a),\ \delta(q_2,a)) & \text{if both terms are defined} \\ \text{undefined otherwise} & \end{cases}$,
- $F_2 = \mathbb{B}$, the set of booleans with the usual operations,
- $L_2((q_1,q_2)) = (\ L(q_1) = L(q_2)\ )$.

Let $q_1, q_2 \in Q$ and $q = (q_1,q_2) \in Q_2$. Note that $D^*(q) = D^*(q_1) \cap D^*(q_2)$. Then, since M is label ranked

$q_1 \simeq q_2 \quad =$

$(\forall\alpha \in D^*(q_1) \cap D^*(q_2)\colon\ L(\delta(q_1,\alpha)) = L(\delta(q_2,\alpha))\ ) \quad =$

$(\forall\alpha \in D^*(q_1) \cap D^*(q_2)\colon\ L_2(\delta(q_1,\alpha),\delta(q_2,\alpha))\ ) \quad =$

$(\forall\alpha \in D^*(q)\colon\ L_2(\delta_2(q,\alpha))\ ) \quad =$

$(\forall q' \in \bar{R}(q)\colon\ L_2(q')\ )$,

where in the last step we used that the set of reachable states $\bar{R}(q)$ equals $\{\ \delta(q,\alpha) \mid \alpha \in D^*(q)\ \}$. Hence the states q_1 and q_2 of M are equivalent if all points reachable from q in the product automaton M2 satisfy the predicate L_2.

5. ALGORITHMS TO COMPUTE REACHABLE NODES

In the previous section we have seen that the equivalence problem of two states in a label ranked, finite automaton can be solved by determining whether all states reachable in a (product) automaton satisfy a certain condition. In fact from this latter automaton we only need its underlying graph structure. Hence we first discuss the reachability problem for directed graphs. For this problem several algorithms are known, see for instance Rem [Re]. Here we discuss some algorithms written in terms of recursive functions or procedures since they form the basis for the type equivalencing algorithms to be discussed in Section 8.

We shall use a definition of directed graph which lies closely to the definition of finite automaton given before. A directed graph is a tuple (Q,d,δ) where

- Q is a finite set of nodes,
- $d\colon Q \to \mathbb{N}$ is a function yielding the number of successors of a node,
- $\delta\colon Q\times\mathbb{N} \overset{p}{\to} Q$ is the successor function, i.e. for $0 \leqslant i < d(q)$ the nodes $\delta(q,i)$ are the successors of q.

Note that, following this definition of a graph, loops and multiple edges between two nodes are allowed. Similarly to the case of finite automata we use elements of $\mathbb{N}^*$ to describe walks through a graph and we extend δ to a partial function $\delta\colon Q\times\mathbb{N}^* \overset{p}{\to} Q$ such that $\delta(q,\alpha)$ is the node reached from q after a walk described by α. We also use again

$D^*(q) = \{\ \alpha \in \mathbb{N}^* \mid \delta(q,\alpha) \text{ is defined }\}$.

Consider a graph (Q,r,δ). The set of states reachable from a state q can be written as

$$\overline{R}(q) = \{q' \in Q \mid (\exists\alpha \in D^*(q):\ q' = \delta(q,\alpha)\ )\}.$$

Furthermore for $q \in Q$, $\beta \in D^*(q)$ and $V \in \mathcal{P}(Q)$ we define

$$B(q,\beta,V) = (\forall\alpha \in \Sigma^*:\ \alpha \leqslant \beta :\ \delta(q,\alpha) \notin V),$$

$$R(q,V) = \{q' \in Q \mid (\exists\alpha \in D^*(q)::\ q' = \delta(q,\alpha) \wedge B(q,\alpha,V)\ )\},$$

where $\leqslant$ denotes the prefix order on Σ^*. So $B(q,\beta,V)$ means that during the walk, which starts in q and is described by β, nodes from V are not met. Also $R(q,V)$ is the set of nodes which can be reached from q without "passing through a node of V". Clearly $\overline{R}(q) = R(q,\emptyset)$.

We now describe two recursive relations for the function R. Each of these relations gives rise to an algorithm to compute the function R. The properties of R given in theorems 5.2 and 5.3 give rise to algorithm 5.4. The properties of R given in theorems 5.3 and 5.6 give rise to the more efficient algorithms 5.7 and 5.9.

Theorem 5.1

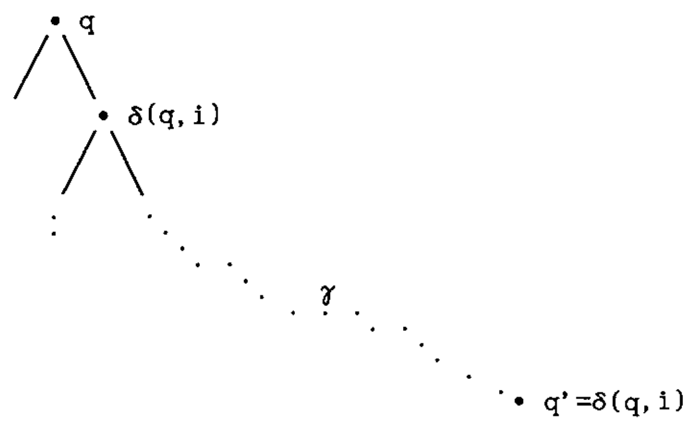
Let $q,q' \in Q$, $V \in \mathcal{P}(Q)$ with $q \neq q'$ and $q' \in R(q,V)$. Then there exists an i with $0 \leqslant i < d(q)$ such that $q' \in R(\ \delta(q,i),\ V \cup \{q\}\ )$.

Proof:

Let β be a row in $\mathbb{N}^*$ with minimal length such that $q' = \delta(q,\beta)$ and $B(q,\beta,V)$ holds. Since $q \neq q'$ the row $\beta \neq \varepsilon$, hence there exist an i with $0 \leqslant i < d(q)$ and a $\gamma \in \mathbb{N}^*$ such that $\beta = i\gamma$. Then trivially $B(\delta(q,i),\gamma,V)$. Furthermore the minimality of $|\beta|$ implies that $B(\delta(q,i),\gamma,\{q\})$ also holds. Hence $B(\delta(q,i),\gamma,V \cup \{q\})$, which implies that $q' \in R(\ \delta(q,i),\ V \cup \{q\}\ )$.

□

The situation in this proof may be elucidated by the following figure.



<u>Theorem 5.2</u>

Let $q \in Q$, $V \in \mathcal{P}(Q)$ with $q \notin V$. Then

$$R(q,V) = \{q\} \cup (\bigcup i:\ 0 \leq i < d(q):\ R(\delta(q,i),\ V \cup \{q\}))\ .$$

<u>Proof</u>:

The "$\subseteq$" part follows immediately from theorem 5.1. Next we prove "$\supseteq$". First, from $q \notin V$ we conclude $q \in R(q,V)$. Further if $q' \in R(\delta(q,i),\ V \cup \{q\})$ then, because R is antimonotonic in its second argument, also $q' \in R(\delta(q,i),\ V)$. Since $q \notin V$ this implies that $q' \in R(q,V)$.

□

<u>Theorem 5.3</u>

Let $q \in Q$, $V \in \mathcal{P}(Q)$ with $q \in V$. Then

$$R(q,V) = \emptyset.$$

<u>Proof</u>:

In this case $B(q,\beta,V)$ is false for all $\beta \in D^*(q)$.

□

The theorems 5.2 and 5.3 now yield the following algorithm to compute $R(q,V)$.

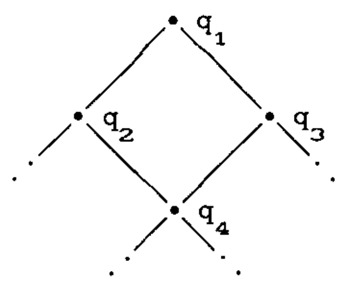
<u>Algorithm 5.4</u>

$$R(q,V) = \ \textbf{if}\ q \in V \rightarrow \emptyset$$
$$\quad\quad\quad\quad\ \square\ \ q \notin V \rightarrow \{q\} \cup (\bigcup i:\ 0 \leq i < d(q):\ R(\delta(q,i),\ V \cup \{q\}))$$
$$\quad\quad\quad\quad\ \textbf{fi}$$

□

Note that since V is always a subset of the finite set Q, this algorithm must terminate. The set of nodes reachable from a given state q can now be found by computing $R(q,\emptyset)$.

The algorithm given above is not very efficient. In fact the call of $R(q,V)$ leads to a kind of depth first search, where the nodes encountered on the path from q to the present node are collected in the set V. The investigation of a branch terminates if a node is met which is already in the set V. In this form it can happen that parts of a graph may be visited several times. Consider for instance the following situation.



Here, in computing $R(q_1,\emptyset)$, the part of the graph reachable from q_4 will be investigated at least twice. We now give stronger versions of the theorems 5.1 and 5.2, which lead to a more efficient algorithm to compute the reachable nodes. For $q \in Q$, $V \in \mathcal{P}(Q)$ and $0 \leq i \leq d(q)$ define the sets W_i by

$$(5.5) \qquad \begin{aligned} W_0 &= \emptyset, \\ W_{i+1} &= W_i \cup R\big(\delta(q,i),\ V \cup \{q\} \cup W_i\big). \end{aligned}$$

Then W_1 is the set of points reachable from $\delta(q,0)$ without passing through points from $V \cup \{q\}$. Next W_2 is the extension of W_1 with the points reachable from $\delta(q,1)$ without passing through points from $V \cup \{q\} \cup W_1$. In general W_{i+1} consists of the points reachable from some $\delta(q,j)$ with $0 \leq j \leq i$ without passing through a point from $V \cup \{q\} \cup W_j$. So the sets W_i correspond to a left to right search proces in which the investigation of a branch is stopped if a point from $V \cup \{q\}$ *or an earlier found point* is met.

<u>Theorem 5.6</u>

Let $q, q' \in Q$, $V \in \mathcal{P}(Q)$ with $q \neq q'$ and $q' \in R(q,V)$. Suppose that the sets W_i are defined by as in (5.5). Then there exists an i with $0 \leq i < d(q)$ such that

$$q' \in R\big(\delta(q,i),\ V \cup \{q\} \cup W_i\big).$$

Proof:

Let β be a row with minimal length such that $q' = \delta(q,\beta)$ and $B(q,\beta,V)$ holds. Since $q \neq q'$ the row $\beta \neq \langle\rangle$, hence there exist an i with $0 \leqslant i < d(q)$ and a $\gamma \in \mathbb{N}^*$ such that $\beta = i\gamma$. Then trivially $B(\delta(q,i),\gamma,V)$. Furthermore the minimality of $|\beta|$ implies $B(\delta(q,i),\gamma,\{q\})$. Hence we have

$$B(\delta(q,i),\gamma,V \cup \{q\}). \qquad (*)$$

We now consider two cases.

i) $B(\delta(q,i),\gamma,W_i)$ holds. This means that on the walk from $\delta(q,i)$ to q', described by γ, no nodes from W_i are encountered. Then $B(\delta(q,i),\gamma,V \cup \{q\} \cup W_i)$ holds and hence $q' \in R\big(\delta(q,i),\ V \cup \{q\} \cup W_i\big)$.

ii) $B(\delta(q,i),\gamma,W_i)$ does not hold. This means that in going from $\delta(q,i)$ to q', following the walk described by γ, a node from W_i is encountered. Let γ' be the longest prefix of γ where this happens, so

$$\delta(q,i\gamma') \in W_i\ , \qquad (**)$$

$$(\forall\alpha\colon\ \gamma' < \alpha \leqslant \gamma\colon\ \delta(q,i\alpha) \notin W_i). \qquad (***)$$

From (**) and the definition of the set W_i we conclude that there exists a j: $0 \leqslant j < i$ such that

$$\delta(q,i\gamma') \in R\big(\ \delta(q,j),\ V \cup \{q\} \cup W_j\big). \qquad (****)$$

Furthermore from (*), (***) and $W_j \subseteq W_i$ we conclude that

$$(\forall\alpha\colon\ \gamma' < \alpha \leqslant \gamma\colon\ \delta(q,i\alpha) \notin V \cup \{q\} \cup W_j).$$

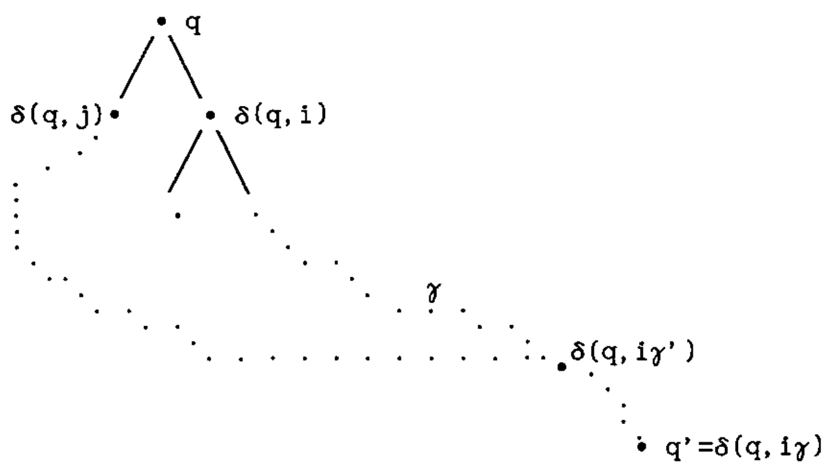
Together with (****) this implies that

$$q' = \delta(q,i\gamma) \in R\big(\ \delta(q,j),\ V \cup \{q\} \cup W_j\big)\ ,$$

which ends case ii).

□

The situation in case ii) of this proof may be elucidated by the following figure.



Theorem 5.7

Let $q \in Q$, $V \in \mathcal{P}(Q)$ with $q \notin V$. Suppose that the sets W_i are defined as in (5.5). Then

$$R(q,V) = \{q\} \cup W_{d(q)}.$$

Proof:

The "$\subseteq$" part follows immediately from theorem 5.6. Next we prove "$\supseteq$". First, from $q \notin V$ we conclude $q \in R(q,V)$. Further if $q' \in W_{d(q)}$, then trivially $q' \in R(\,\delta(q,i),\ V \cup \{q\} \cup W_i\,)$ for some i with $0 \leq i < d(q)$ and, since R is antimonotonic in its second argument, also $q' \in R(\,\delta(q,i),\ V\,)$. Since $q \notin V$ this implies that $q' \in R(q,V)$.

□

The theorems 5.7 and 5.3 now give rise to the following more efficient algorithm to compute $R(q,V)$.

Algorithm 5.8

$$\begin{aligned} R(q,V) = \ & \textbf{if } q \in V \rightarrow \emptyset \\ & \square\ \ q \notin V \rightarrow \tilde{R}(q,d(q),V) \\ & \textbf{fi} \end{aligned}$$

where the function $\tilde{R}: Q \times \mathbb{N} \times \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$ is given by

$$\tilde{R}(q,k,V) = \mathbf{if}\ k = 0 \rightarrow \{q\}$$
$$\square\ \ k > 0 \rightarrow \tilde{R}(q,k-1,V) \cup R\big(\ \delta(q,k-1),\ V \cup \tilde{R}(q,k-1,V)\ \big)$$
$$\mathbf{fi}$$

□

Clearly, in the context of (5.5), $\tilde{R}(q,i,V) = W_i \cup \{q\}$. An imperative version of this algorithm is given by a procedure p1 with specification

(5.9) $\quad$ $(*\ \ V = V_o\ \ *)$ $\quad$ $p1(\downarrow q\!:\ Q;\ \updownarrow V\!:\ \mathcal{P}(Q))$ $\quad$ $(*\ \ V = V_o \cup R(q,V_o)\ \ *)$

Here value parameters are preceded by a $\downarrow$, result parameters are preceded by a $\uparrow$ and value-result parameters are preceded by a $\updownarrow$. The annotated code of procedure p is given below.

Algorithm 5.10

$$
\begin{array}{l}
\mathbf{proc}\ p1 = \\
\quad (\downarrow q\!:\ Q;\ \updownarrow V\!:\ \mathcal{P}(Q)\ \mid \\
\quad \mathbf{if}\ q \in V \rightarrow\ (*\ R(q,V) = \emptyset\ *)\ \mathrm{skip} \\
\quad \square\ \ q \notin V \rightarrow \\
\qquad [\![\ \mathbf{var}\ i\!:\ \mathbb{N} \mid \\
\qquad\ V{:=}\ V \cup \{q\};\ \ i{:=}\ 0 \\
\qquad\ (*\ \mathrm{invariant}\!:\ V = V_o \cup \tilde{R}(q,i,V_o)\ *) \\
\qquad\ ;\mathbf{do}\ i \neq d(q) \rightarrow \\
\qquad\qquad p1(\delta(q,i),V) \\
\qquad\qquad (*\ V = V_o \cup \tilde{R}(q,i,V_o) \cup R\big(\delta(q,i),V_o \cup \tilde{R}(q,i,V_o)\big)\ , \\
\qquad\qquad\quad \mathrm{so}\ V = V_o \cup \tilde{R}(q,i+1,V_o)\ *) \\
\qquad\qquad ;i{:=}\ i + 1 \\
\qquad\ \mathbf{od} \\
\qquad\ (*\ \mathrm{invariant} \wedge i = d(q)\ ,\ \mathrm{so}\ V = V_o \cup R(q,V_o)\ *) \\
\qquad]\!] \\
\quad \mathbf{fi} \\
\quad)
\end{array}
$$

□

6. ALGORITHMS TO COMPUTE PREDICATES ON REACHABLE NODES I

Let (Q, d, δ) be a directed graph and let $L: Q \to \mathbb{B}$, where $\mathbb{B}$ denotes the set of the booleans with the usual operations. So L is a predicate on the nodes of the graph. We extend L to a function LS: $\mathcal{P}(Q) \to \mathbb{B}$ by

(6.1) $\quad LS(V) = (\forall q: q \in V : L(q))$.

In this section and the next one we shall discuss several algorithms to compute, for a given node q, the value of $LS(\bar{R}(q))$. In other words we consider algorithms which compute whether a predicate L holds on all nodes reachable from a given node q.

A simple approach is to compute first the set $\bar{R}(q)$ using one of the algorithms given in Section 5 and then to verify whether L holds for all elements of $\bar{R}(q)$. The correctness of this type of algorithm is of course trivial. Of course we can also compute the predicate L "on the fly", i.e. as soon as a new reachable point q' is found, the value of L(q') is computed. At first instance this leads to a function $g: Q \times \mathcal{P}(Q) \to \mathcal{P}(Q) \times \mathbb{B}$ with specification

(6.2) $\quad g(q,V) = < R(q,V) \; , \; LS(R(q,V)) >$.

Algorithms for the function g can be obtained by extending the algorithms for the function R given in Section 5 with a "second component".
If R is computed with the (inefficient) algorithm 5.4 this leads to

<u>Algorithm 6.3</u>

$g(q,V) =$

if $q \in V \quad \to <\emptyset, \text{ true}>$
$\Box\ q \notin V \wedge \neg L(q) \to <\{q\} \cup (\cup i: 0 \le i < d(q): \pi_1(g_i)), \text{ false}>$
$\Box\ q \notin V \wedge L(q) \quad \to <\{q\} \cup (\cup i: 0 \le i < d(q): \pi_1(g_i)) \; , (\forall i: 0 \le i < d(q): \pi_2(g_i))>$
fi

where $g_i = g(\delta(q,i), V \cup \{q\})$.

□

If in fact one is only interested in the question whether L holds on all nodes from a set R(q,V) and not in the set R(q,V) itself, only the second component of g(q,V) is needed. In algorithm 6.3 the computation of this second component is done without inspecting the first component. Hence we can construct a function $f\colon Q \times \mathcal{P}(Q) \to \mathbb{B}$ with specification

(6.4) $\quad f(q,V) = LS(R(q,V))$.

From algorithm 6.3 we then obtain

<u>Algorithm 6.5</u>

$$
\begin{aligned}
f(q,V) = \;& \textbf{if } q \in V \to \text{true} \\
& \square \;\; q \notin V \wedge \neg L(q) \to \text{false} \\
& \square \;\; q \notin V \wedge L(q) \to (\forall i\colon 0 \leqslant i < d(q)\colon f(\,\delta(q,i),\; V \cup \{q\}\,)) \\
& \textbf{fi}
\end{aligned}
$$

□

The correctness of this algorithm follows immediately from the correctness of algorithm 6.3.

7. ALGORITHMS TO COMPUTE PREDICATES ON REACHABLE NODES II

The algorithms given in the previous section were based on the (inefficient) reachable points algorithm given in 5.4. More efficient algorithms to compute the function g satisfying 6.2, can be obtained by computing R with algorithms 5.8 or the imperative version 5.10. Starting from 5.8 leads to

<u>Algorithm 7.1</u>

$$
\begin{aligned}
g(q,V) = \;& \textbf{if } q \in V \to \langle \emptyset,\ \text{true} \rangle \\
& \square \;\; q \notin V \to \tilde{g}(q,d(q),V) \\
& \textbf{fi}
\end{aligned}
$$

where the function $\tilde{g}\colon Q \times \mathbb{N} \times \mathcal{P}(Q) \to \mathcal{P}(Q) \times \mathbb{B}$ is given by

$$
\begin{aligned}
\tilde{g}(q,k,V) = \;& \textbf{if } k = 0 \to \langle \{q\}, L(q) \rangle \\
& \square \;\; k > 0 \to \langle \pi_1(\tilde{g}1) \cup \pi_1(g1),\ \pi_2(\tilde{g}1) \wedge \pi_2(g1) \rangle \\
& \textbf{fi}
\end{aligned}
$$

with $\tilde{g}1 = \tilde{g}(q,k-1,V)$ and $g1 = g\big(\delta(q,k-1),\ V \cup \pi_1(\tilde{g}1)\,\big)$.

□

An imperative version of this algorithm is given by a procedure p2 with specification

$$(* \quad V = V_o \quad *)$$

(7.2) $\quad p2(\downarrow q\!: Q;\ \updownarrow V\!: \mathcal{P}(Q);\ \uparrow b\!: \mathbb{B})$

$$(* \quad (V = V_o \cup R(q,V_o)) \wedge (b \equiv LS(R(q,V_o))) \quad *)$$

The annotated code of procedure p is given below.

Algorithm 7.3

```
proc p2 =
     (↓q: Q; ↕V: 𝒫(Q); ↑b:𝔹 |
     if q ∈ V → (* R(q,V) = ∅ *) b:= true
     ▯ q ∉ V →
          ⟦ var i: ℕ, b1:𝔹|
           V:= V ∪ {q}; i:= 0; b:= L(q);
           (* invariant: (V = Vo ∪ R̃(q,i,Vo)) ∧ (b ≡ LS(R̃(q,i,Vo))) *)
           ;do i ≠ d(q) →
               p2(δ(q,i),V,b1)
               (* V = Vo ∪ R̃(q,i,Vo) ∪ R(δ(q,i),Vo ∪ R̃(q,i,Vo)) ,
                 so V = Vo ∪ R̃(q,i+1,Vo) ,
                 b1 ≡ LS(R(δ(q,i), Vo ∪ R̃(q,i,Vo)))
               *)
               ;b:= b ∧ b1
               (* b ≡ LS(R̃(q,i+1,Vo)) *)
               ;i:= i + 1
               (* invariant *)
           od
          ⟧
     fi
     )
```

□

Finally, similarly to Section 6, we consider again the case that one is only interested in $LS(R(q,V))$ and not in the set $R(q,V)$ itself. In algorithm 6.3 the two components of g were computed independently of each other. This observation resulted in algorithm 6.5, where only the the second component of g was computed. Unfortunately in the algorithms 7.1 and 7.3 the computation of the second component of g depends essentially on its first component. Hence we cannot extract from 7.1 and 7.3 algorithms to compute only the second component. However, some improvement can be obtained by replacing g by the function h: $Q \times \mathcal{P}(Q) \to \mathcal{P}(Q) \times \mathbb{B}$ with specification

$$
(7.4)\qquad
\begin{aligned}
&LS(R(q,V)) \Rightarrow \pi_1(h(q,V)) = R(q,V)\\
&\pi_2(h(q,V)) = LS(R(q,V))
\end{aligned}
$$

So the second component of h yields again the desired result. Furthermore, if the second component equals true, the first component of h yields again the set R(q,V). If the second component of h equals false, the value of the first component is unspecified. For h we can use the following algorithm.

<u>Algorithm 7.5</u>

$$
\begin{aligned}
h(q,V) = \;&\textbf{if } q \in V \to \langle \emptyset,\ \text{true}\rangle\\
&\square\ \ q \notin V \to \tilde{h}(q,d(q),V)\\
&\textbf{fi}
\end{aligned}
$$

where the function $\tilde{h}: Q \times \mathbb{N} \times \mathcal{P}(Q) \to \mathcal{P}(Q) \times \mathbb{B}$ is given by

$$
\begin{aligned}
\tilde{h}(q,k,V) = \;&\textbf{if } k = 0 \to \langle \{q\}, L(q)\rangle\\
&\square\ \ k > 0 \to \textbf{if } \neg\, \pi_2(\tilde{h}1) \to \langle \emptyset,\ \text{false}\rangle\\
&\qquad\qquad\quad \square\quad \pi_2(\tilde{h}1) \to \langle \pi_1(\tilde{h}1) \cup \pi_1(h1),\ \pi_2(\tilde{h}1) \wedge \pi_2(h1)\rangle\\
&\qquad\qquad\quad \textbf{fi}\\
&\textbf{fi}
\end{aligned}
$$

with $\tilde{h}1 = \tilde{h}(q,k-1,V)$ and $h1 = h\big(\delta(q,k-1),\ V \cup \pi_1(\tilde{h}1)\ \big)$.

□

The corresponding imperative version of this algorithm is the procedure p3 with the following specification.

$$
(7.6)\qquad
\begin{aligned}
&(*\ \ V = V_0\ \ *)\\
&p3(\downarrow q\text{: } Q;\ \updownarrow V\text{: } \mathcal{P}(Q);\ \uparrow b\text{: }\mathbb{B})\\
&(*\ \ (b \equiv LS(R(q,V_0)))\ \ \wedge\ \ (b \Rightarrow (V = V_0 \cup R(q,V_0)))\ \ *)
\end{aligned}
$$

The code of procedure p3 is simply derived from algorithm 7.3. Now however, as soon as a node has been reached where L does not hold, the traversal of the graph is terminated.

Algorithm 7.7

```
proc p3 =
    (↓q: Q; ↕V: 𝒫(Q); ↑b: 𝔹 |
    if q ∈ V → (* R(q,V) = ∅ *) b:= true
    ▯  q ∉ V  →
         ⟦ var i: ℕ|
          V:= V ∪ {q}; i:= 0; b:= L(q);
          (* inv.: (b ≡ LS(R̃(q,i,V₀))) ∧ (b ⇒ (V = V₀ ∪ R̃(q,i,V₀))) *)
          ;do b ∧ i ≠ d(q) →
              (* LS(R̃(q,i,V₀)) ∧ (V = V₀ ∪ R̃(q,i,V₀))  *)
              p3(δ(q,i),V,b)
              (* b ≡ LS(R(δ(q,i),V₀ ∪ R̃(q,i,V₀)) ,
                so b ≡ LS(R̃(q,i+1,V₀)),
                b ⇒ (V = V₀ ∪ R̃(q,i,V₀) ∪ R(δ(q,i),V₀ ∪ R̃(q,i,V₀)) ),
                so b ⇒ (V = V₀ ∪ R̃(q,i+1,V₀)) ,
              *)
              ;i:= i + 1
              (* invariant *)
          od
         ⟧
    fi
    ) .
```

□

8. ALGORITHMS TO DECIDE THE EQUIVALENCE OF RECURSIVE TYPES

We now combine the results of the previous sections to obtain algorithms to determine the equivalence of (recursive) types. Let τ_0 and τ_1 be two type expressions. Then τ_0 and τ_1 are equivalent as types iff they are equivalent as states of the automaton M_1 as given in Section 3. Let M_2 be the product automaton of M_1, as described in Section 4. Then type equivalence of τ_0 and τ_1 means that L_2 holds for all states reachable from (τ_0,τ_1) in M_2 (seen as directed graph). Using the algorithm given in 6.5 this can be computed with the recursive function f: $\mathrm{Texp} \times \mathrm{Texp} \times \mathcal{P}(\mathrm{Texp} \times \mathrm{Texp}) \to \mathbb{B}$ given by

$$
f(\rho,\sigma,V) = \quad \textbf{if } (\rho,\sigma) \in V \rightarrow \text{true}
$$

$$
\square \ (\rho,\sigma) \notin V \wedge L(\rho) \neq L(\sigma) \rightarrow \text{false}
$$

$$
\square \ (\rho,\sigma) \notin V \wedge L(\rho) = L(\sigma) \rightarrow \tag{8.1}
$$

$$
\big(\forall i:\ 0 \leq i < d(\rho):\ f(\delta(\rho,i),\delta(\sigma,i),V \cup \{(\rho,\sigma)\})\ \big)
$$

$$
\textbf{fi}
$$

The equivalence of two types ρ and σ can now be checked by the function call $f(\rho,\sigma,\emptyset)$, i.e. $f(\rho,\sigma,\emptyset) \equiv (\rho \cong \sigma)$. This type equivalence algorithm strongly resembles the ones given in [Wij, §7.3] and in [Ca]. Note that the necessary unfoldings of recursive types are hidden in the function δ.

A more efficient algorithm for type equivalence can be obtained by starting with the procedure p3 described in 7.7. This leads to

```
    proc p3 =
         (↓ρ,σ :Texpq; ↕V: 𝒫(Texp×Texp); ↑b: 𝔹 |
         if (ρ,σ) ∈ V → b:= true
         □  (ρ,σ) ∉ V  →
              ⟦ var i: ℕ|
               V:= V ∪ {(ρ,σ)}; i:= 0; b:= L(ρ)=L(σ);
(8.2)          ;do b ∧ i ≠ d(ρ) →
                   p3(δ(ρ,i),δ(σ,i),V,b)
                   ;i:= i + 1
               od
              ⟧
         fi
         ) .
```

Now the equivalence of the types ρ and σ can be found be calling the procedure p3 with $V = \emptyset$, i.e.

$$
(*\ \ V = \emptyset\ \ *)\ \ p3(\rho,\sigma,\emptyset,b)\ \ (*\ \ b \equiv (\rho \cong \sigma)\ \ *)
$$

Note that in procedure p3 we can replace V by a global variable. We then obtain a type equivalence algorithm similar to the one given in [Ko].

APPENDIX 1

Here we define trees and some related notions. See also Barendregt[Ba]. Let $\mathbb{N}^*$ be the set of rows of natural numbers. We shall not make a difference between a natural number and a row of length 1 containing that number. The concatenation of elements of $\mathbb{N}^*$ will be denoted by juxtaposition. Elements of $\mathbb{N}^*$ will usually be denoted by Greek letters. The empty row will always be denoted with the letter ε. On $\mathbb{N}^*$ we define the prefix order $\leqslant$, i.e. $(\alpha \leqslant \beta) \equiv (\exists\, \gamma \in \mathbb{N}^*\colon\ \alpha\gamma = \beta)$. As usual we define $(\alpha < \beta) \equiv (\alpha \leqslant \beta) \wedge \neg(\alpha = \beta)$. The length of the row $\alpha \in \mathbb{N}^*$ will be denoted as $|\alpha|$. Next we consider tree domains. A subset A of $\mathbb{N}^*$ will be called a *tree domain* if

i) $A \neq \phi$,

ii) $\alpha \in A \wedge \beta \leqslant \alpha \Rightarrow \beta \in A$,

iii) $\alpha(n+1) \in A \Rightarrow \alpha n \in A$.

Let F be some set and $d\colon F \rightarrow \mathbb{N}$. The pair (F,d) is called a *graded alphabet*. A (*ranked*) *tree* over (F,d) is a partial function $T\colon\ \mathbb{N}^* \overset{p}{\Rightarrow} F$ such that

i) dom(T) is a tree domain,

ii) if $\alpha \in \mathrm{dom}(T)$, then $\alpha i \in \mathrm{dom}(T)$ for all i: $0 \leqslant i < d(T(\alpha))$.

So if $\alpha \in \mathrm{dom}(T)$, then $T(\alpha)$ is the label in α and $d(T(\alpha))$ is the number of subtrees emerging from α. Note that, since a tree domain is not empty, ε is an element of dom(T) for every tree T. A tree T will be called *finite* if dom(T) is a finite set.

APPENDIX 2

We prove that for all types τ the set $\tilde{R}(\tau)$ is finite. If this does not hold the set of states of the automata M_1 described in Section 3 may not be finite and the algorithms given in Section 7 do not necessarily terminate (since the set V of pairs of types and hence the recursion depth are not bounded). If τ does not contain a recursive type, $R(\tau)$ consist of all sub expressions of τ and is trivially bounded. However if τ is a recursive type then (in the computation of $\delta(\tau,i)$ for suitable i) an unfolding takes place thus generating possibly new and longer type expressions.

Similar to the case of automata we introduce for every type expression τ

$D(\tau) = \mathrm{dom}(\delta(\tau,\cdot)) = \{i \mid 0 \leq i < d(\tau)\}$,

$D^*(\tau) = \{\alpha \in \mathbb{N}^* \mid \delta(\tau,\alpha) \text{ is defined}\}$.

Then

$\bar{R}(\tau) = \{\delta(\tau,\alpha) \mid \alpha \in D^*(\tau)\}$.

In the following theorems we investigate the behaviour of $D^*(\tau)$ and $\delta(\tau,\cdot)$ under substitutions in τ.

<u>Theorem 10.1</u>

Let $\tau,\sigma \in \mathrm{Texp}$, $t \in V$ and $i \in \mathbb{N}$. If $i \in D(\tau)$ then $i \in D(\tau^t_\sigma)$ and

$\delta(\tau^t_\sigma, i) = (\ \delta(\tau,i)\ )^t_\sigma$.

<u>Proof</u>:

Induction with respect to τ. The other cases being trivial, we only consider the case that $\tau = \mu(\lambda s \cdot \rho)$. If $t = s$ then τ does not contain the free variable t and the result becomes trivial. Next consider the case $t \neq s$. Without loss of generality we may assume that $s \notin FV(\sigma)$. Then

$i \in D(\tau)$	$\Rightarrow$	[def. of δ]
$i \in D(\rho)$	$\Rightarrow$	[induction hypothesis]
$i \in D(\rho^t_\sigma)$	$\Rightarrow$	[def. of δ]
$i \in D(\ \mu(\lambda s \cdot \rho^t_\sigma)\ )$	$\Rightarrow$	[$s \notin FV(\sigma)$, $s \neq t$]
$i \in D(\tau^t_\sigma)$.		

Furthermore for $i \in D(\tau)$

$\delta(\tau^t_\sigma, i)$	$=$	[$s \notin FV(\sigma)$, $s \neq t$]
$\delta(\mu(\lambda s \cdot \rho^t_\sigma), i)$	$=$	[def. of δ]
$(\ \delta(\rho^t_\sigma, i)\ )^s_{\mu(\lambda s \cdot \rho^t_\sigma)}$	$=$	[$i \in D(\rho)$, induction hypothesis]
$(\ \delta(\rho,i)^t_\sigma\ )^s_{\mu(\lambda s \cdot \rho^t_\sigma)}$	$=$	[$s \notin FV(\sigma)$, $s \neq t$]
$(\ \delta(\rho,i)^t_\sigma\ )^s_{\mu(\lambda s \cdot \rho)^t_\sigma}$	$=$	[prop. of subst., $s \notin FV(\sigma)$, $s \neq t$]

$(\delta(\rho, i)^{s}_{\mu(\lambda s \cdot \rho)})^{t}_{\sigma}$ = [def. of δ]

$(\delta(\mu(\lambda s \cdot \rho), i))^{t}_{\sigma}$ =

$\delta(\tau, i)^{t}_{\sigma}$.

□

Theorem 10.2

Let $\alpha \in \mathbb{N}^*$, $\tau, \sigma \in$ Texp and $t \in V$. If $\alpha \in D^*(\tau)$ then $\alpha \in D^*(\tau^{t}_{\sigma})$ and

$$\delta(\tau^{t}_{\sigma}, \alpha) = (\delta(\tau, \alpha))^{t}_{\sigma} .$$

Proof:

Induction with respect to $|\alpha|$ using the previous theorem.

□

So $D^*(\tau)$ is always a subset of $D^*(\tau^{t}_{\sigma})$ and for α in $D^*(\tau)$ the operations "compute expression that describes the subtree at α" and "substitution" commute. In general $D^*(\tau)$ is a proper subset of $D^*(\tau^{t}_{\sigma})$. Next we study $\delta(\tau^{t}_{\sigma}, \alpha)$ for rows α in $D^*(\tau^{t}_{\sigma}) \setminus D^*(\tau)$.

Theorem 10.3

Let $\tau \in$ Texp with $D^*(\tau) = \{\varepsilon\}$. Then there exist an $u \in V \cup C$ and $k \in \mathbb{N}$ mutually different variables $s_1, \ldots, s_k$ such that

$$\tau = \mu(\lambda s_1 \cdot \ldots \ \mu(\lambda s_k \cdot u) \ldots).$$

Proof:

Induction with respect to τ.

□

In terms of trees this theorem describes the general form of a type expression that describes a single node tree (labeled u or ⫴ if $u = s_i$ for some i).

Theorem 10.4

Let $\tau, \sigma \in$ Texp, $t \in V$ and $\alpha \in \mathbb{N}^*$. If $\alpha \in D^*(\tau^{t}_{\sigma}) \setminus D^*(\tau)$, then there exists a $\gamma \in D^*(\sigma)$, $0 < |\gamma| \leq |\alpha|$, such that

$$\delta(\tau^{t}_{\sigma}, \alpha) = \delta(\sigma, \gamma).$$

Proof:

Let $\alpha = \beta\gamma$ where β is the longest prefix of α with $\beta \in D^*(\tau)$. Then $D^*(\delta(\tau,\beta)) = \{\varepsilon\}$ and $0 < |\gamma| \leq |\alpha|$. Further

$$\begin{array}{lll} \delta(\tau^t_\sigma,\alpha) & = & \\ \delta(\ \delta(\tau^t_\sigma,\beta),\ \gamma\ ) & = & [\beta \in D^*(\tau),\ \text{theorem } 10.2] \\ \delta(\ \delta(\tau,\beta)^t_\sigma,\ \gamma\ ). & & \end{array}$$

So $\gamma \in D^*(\delta(\tau,\beta)^t_\sigma)$. Now since $D^*(\delta(\tau,\beta)) = \{\varepsilon\}$, the previous theorem implies the existence of a $k \in \mathbb{N}$, variables $s_1,\ldots,s_k$ and an $u \in V \cup C$ such that

$$\delta(\tau,\beta) = \mu(\lambda s_1 \cdot \ldots\ \mu(\lambda s_k \cdot u)\ldots).$$

Of course the variables $s_1,\ldots,s_k$ can be chosen such that they are no elements of $FV(\sigma)$. Now $t \in FV(\delta(\tau,\beta))$ otherwise $D^*(\delta(\tau,\beta)^t_\sigma) = D^*(\delta(\tau,\beta)) = \{\varepsilon\}$ which cannot contain $\gamma \neq \langle\rangle$. So $u = t$ and $s_i \neq t$ for $i = 1,\ldots,k$. Then, with $\gamma = j\xi$,

$$\begin{array}{lll} \delta(\ \delta(\tau,\beta)^t_\sigma,\ \gamma\ ) & = & \\ \delta\big(\ (\mu(\lambda s_1 \cdot \ldots\ \mu(\lambda s_k \cdot t)\ldots)\ )^t_\sigma,\ \gamma\ \big) & = & [s_i \notin FV(\sigma) \cup \{t\} \text{ for } i = 1..k] \\ \delta\big(\ \mu(\lambda s_1 \cdot \ldots\ \mu(\lambda s_k \cdot \sigma)\ldots)\ ,\ \gamma\ \big) & = & [\gamma = j\xi,\ \text{def. } \delta] \\ \delta\big(\ \delta\big(\mu(\lambda s_1 \cdot \ldots\ \mu(\lambda s_k \cdot \sigma)\ldots),j\ \big),\ \xi\ \big) & = & [\text{def. of } \delta,\ s_i \notin FV(\sigma) \text{ for } i = 1..k] \\ \delta(\delta(\sigma,j),\xi) & = & \\ \delta(\sigma,\gamma). & & \end{array}$$

□

Now we are able to give a relation for $\bar{R}(\tau)$ if τ is a recursive type.

Theorem 10.5

Let $\rho \in$ Texp and $s \in V$. Then

$$\bar{R}(\mu(\lambda s \cdot \rho)) \subseteq \{\mu(\lambda s \cdot \rho)\} \cup \{\ \sigma^s_{\mu(\lambda s \cdot \rho)} \mid \sigma \in \bar{R}(\rho)\ \}.$$

Proof:

Let $A = \{\mu(\lambda s\cdot\rho)\} \cup \{ \sigma^{s}_{\mu(\lambda s\cdot\rho)} \mid \sigma \in \bar{R}(\rho) \}$. We prove with induction to $|\beta|$ that for all $\beta \in D^{*}(\mu(\lambda s\cdot\rho))$

$$\delta(\mu(\lambda s\cdot\rho),\beta) \in A. \qquad (*)$$

The induction basis $|\beta| = 0$ is trivial. Next suppose $n \in \mathbb{N}$ and assume as induction hypothesis that $(*)$ holds for all β with $|\beta| \leqslant n$. Let $i\alpha \in D^{*}(\mu(\lambda s\cdot\rho))$ with $|\alpha| = n$. Then, with $\tau = \delta(\rho, i)$,

$$\begin{array}{lll} \delta(\mu(\lambda s\cdot\rho), i\alpha) & = & [\text{def. } \delta] \\ \delta(\ \delta(\mu(\lambda s\cdot\rho), i)\ ,\alpha) & = & [\text{def. } \delta] \\ \delta(\ \delta(\rho, i)^{s}_{\mu(\lambda s\cdot\rho)}\ ,\alpha) & = & [\text{def } \tau] \\ \delta(\ \tau^{s}_{\mu(\lambda s\cdot\rho)}\ ,\alpha) & & (**) \end{array}$$

So $\alpha \in D^{*}(\tau^{s}_{\mu(\lambda s\cdot\rho)})$. To show that $(**)$ is an element of A we consider two cases.

i) $\alpha \in D^{*}(\tau)$. Then

$$\begin{array}{lll} \delta(\ \tau^{s}_{\mu(\lambda s\cdot\rho)}\ ,\alpha) & = & [\text{theorem } 10.2] \\ \delta(\tau,\alpha)^{s}_{\mu(\lambda s\cdot\rho)} & = & [\text{def. of } \tau] \\ \delta(\rho, i\alpha)^{s}_{\mu(\lambda s\cdot\rho)} & \in A & \end{array}$$

ii) $\alpha \notin D^{*}(\tau)$. Then theorem 10.4 yields the existence of $\gamma \in D^{*}(\mu(\lambda s\cdot\rho))$ with $|\gamma| \leqslant |\alpha|$ such that

$$\delta(\ \tau^{s}_{\mu(\lambda s\cdot\rho)}\ ,\alpha) \quad = \quad \delta(\mu(\lambda s\cdot\rho),\gamma).$$

Since $|\gamma| \leqslant |\alpha| = n$, the induction hypothesis now implies that the right hand side is an element of A.

□

Finally we can prove the desired result.

Theorem 10.6

For all $t \in$ Texp the set $\bar{R}(\tau)$ is finite.

Proof:

Induction with respect to τ. Assume as induction hypothesis that $\bar{R}(\rho)$ is finite for all sub expressions of τ. We consider the following cases.

i) $\tau \in V \cup C$. Then $\bar{R}(\tau) = \{\tau\}$.

ii) $\tau = \uparrow\rho$. Then $\bar{R}(\tau) = \{\tau\} \cup \bar{R}(\rho)$, which is finite by the induction hypothesis.

iii) $\tau = \rho_0 \times \rho_0$ or $\tau = \rho_0 + \rho_1$. Then $\bar{R}(\tau) = \{\tau\} \cup \bar{R}(\rho_0) \cup \bar{R}(\rho_1)$, which is finite by the induction hypothesis.

iv) $\tau = \mu(\lambda s \cdot \rho)$. Theorem 10.5 yields

$$\bar{R}(\tau) \subseteq \{\tau\} \cup \{\, {\sigma_\tau}^{s} | \sigma \in \bar{R}(\rho)\},$$

which is finite since by the induction hypothesis $\bar{R}(\rho)$ is finite.

□

APPENDIX 3

Let $M = (Q,\Sigma,\delta,L,F)$ be a finite automaton which is label ranked, i.e. if $L(q_1) = L(q_2)$ then $D(q_1) = D(q_2)$. We prove that for all states q_1 and q_2

$$\big(\forall\alpha \in D^*(q_1) \cap D^*(q_2){:}\ L(\delta(q_1,\alpha)) = L(\delta(q_2,\alpha))\ \big) \qquad (*)$$

$$\Rightarrow$$

$$D^*(q_1) = D^*(q_2).$$

Assume that $(*)$ holds and suppose that for instance $D^*(q_1) \not\subseteq D^*(q_2)$. Let $\alpha \in D^*(q_1) \setminus D^*(q_2)$ with $|\alpha|$ minimal. From $\varepsilon \in D^*(q_1)$ and $\varepsilon \in D^*(q_2)$ we conclude $|\alpha| > 0$. So there exist $\beta \in D^*(q_1)$ and $a \in \Sigma$ such that $\alpha = \beta a$. Since $|\alpha|$ is minimal, $\beta \in D^*(q_2)$. Then $(*)$ implies that $L(\delta(q_1,\beta)) = L(\delta(q_2,\beta))$, hence $\delta(q_1,\beta)$ and $\delta(q_2,\beta)$ have the same transitions. Now $\delta(\delta(q_1,\beta),a) = \delta(q_1,\alpha)$, so $\delta(q_1,\beta)$ has a transition under a. Then also $\delta(q_2,\beta)$ has a transition under a which yields a contradiction with $\alpha = \beta a \notin D^*(q_2)$. So $D^*(q_1) \subseteq D^*(q_2)$. Similarly we can prove $D^*(q_2) \subseteq D^*(q_1)$.

□

REFERENCES

[Ba] Barendregt, H.P., *The Lambda calculus, its syntax and semantics*, revised edition, North Holland, Amsterdam (1984).

[Ca] Cardelli, L., Typechecking Dependent Types and Subtypes, in: *Foundations of Logic and Functional Programming*, 45-57, LNCS 306 (1986).

[CC] Cardone, F. and Coppo, M., Type inference with recursive types: Syntax and Semantics, to appear in: *Information and Computation.*

[Cp] Coppo, M., A completeness theorem for recursively defined types, in: *Automata, Languages and Programming*, 120-129, LNCS 194 (1985).

[CKV] Courcelle, B., Kahn, G. and Vuillemin, J., Algorithmes d'equivalence et de reduction a des expressions minimales dans un classe d'equations recursives simples, in: *International Conference on Automata, Languages and Programming*, 200-213, LNCS 14 (1974).

[Ko] Koster, On infinite modes, *Algol Bulletin* **30**, 86-89 (1969).

[Kr] Krăl, The Equivalence of Modes and the Equivalence of Finite Automata, *Algol Bulletin* **35**, 34-35 (1973).

[Re] Rem, M., Small programming exercises5, *Sci Comput. Programming* **4**, 323-333 (1984).

[Wij] Van Wijngaarden, *Revised report on the algorithmic language Algol 68*, Mathematical Centre Tracts 50, Amsterdam (1976).

In this series appeared:

89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a parallel program for Dirichlet convolution.
89/8	E.H.L.Aarts A.E.Eiben K.M. van Hee	A general theory of genetic algorithms.
89/9	K.M. van Hee P.M.P. Rambags	Discrete event systems: Dynamic versus static topology.
89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.
89/12	A.T.M.Aerts K.M. van Hee	A concise formal framework for data modeling.
89/13	A.T.M.Aerts K.M. van Hee M.W.H. Hesen	A program generator for simulated annealing problems.
89/14	H.C.Haesen	ELDA, data manipulatie taal.
89/15	J.S.C.P. van der Woude	Optimal segmentations.
89/16	A.T.M.Aerts K.M. van Hee	Towards a framework for comparing data models.
89/17	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra.

90/1	W.P.de Roever- H.Barringer- C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, p. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.
90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.

90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.

91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.

91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.