Correctness of acceptor schemes for regular languages

Citation for published version (APA):

Document status and date:
Published: 01/01/1991

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 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

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.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at: openaccess@tue.nl providing details and we will investigate your claim.
Eindhoven University of Technology
Department of Mathematics and Computing Science

Correctness of Acceptor Schemes for Regular Languages

by

H. ten Eikelder R. van Geldrop

Computing Science Note 91/29
Eindhoven, November 1991
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

All rights reserved
editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.
Correctness of Acceptor Schemes for Regular Languages

H. ten Eikelder       R. van Geldrop
Eindhoven University of Technology

28 November 1991

Abstract

We study the correctness of a simple mapping yielding acceptor programs for languages described by regular expressions. The proofs are given using the formalism of dynamic logic.
1. INTRODUCTION

An important part of the front end of a compiler is the lexical scanner. For a given set of regular expressions, the main task of a lexical scanner is to segment an input string into successive parts, each of them corresponding to one of those regular expressions. Clearly to construct a scanner, we need acceptors for given regular expressions. A usual approach to construct a scanner/acceptor is via finite automata, see for instance the well-known LEX algorithms in [AhSeUIJ]. A different approach is to use a mapping, directly from regular expressions to acceptor programs. An acceptor generated by such a mapping has a structure similar to that of the original regular expression. This approach is simpler than the usual one, but it is only applicable to a special class of regular expressions. The method has been described in [Le], [Wi] and [tEH]. However, as far as we know a correctness proof of this method has never been published. In this note we give such a proof. To illustrate the method, consider the regular expression

\[(a|b^*)c\]

The derived acceptor program (acting on a string \(r \in \{a, b, c\}^\ast\)) is

```plaintext
if head(r) = a -> r := tail(r)
[] head(r) \in \{b, c\} do head(r) = b -> r := tail(r) od
fi
```

Our acceptor will be written in a programming language which consists of regular programs in the style of Harel [Ha], built from some suitable atomic statements. Ultimately we shall arrive at deterministic regular programs, which correspond to the usual Guarded Command Language (GCL) programs. The correctness proofs will be given in a corresponding dynamic logic. Hence this note can also be seen as a try-out in giving correctness proofs using dynamic logic.

The structure of this note is as follows. In section 2 we shall introduce our programming language together with some mathematical preliminaries. A first version of our "acceptor mapping" will be treated in section 3. However, since lookahead is lacking, it is very limited in its usability. In the next version lookahead is added, thus yielding the acceptor mapping in section 4. In section 5 we introduce conditions which ensure that the resulting programs are deterministic. We also give the corresponding GCL version. Finally in section 6, we describe a modified acceptor mapping, yielding more elegant GCL code. Some conclusions are given in section 7.

2. PRELIMINARIES

In the first part of this section we recapitulate some basic notions from language theory, regular expressions and the corresponding languages and some related notions. Then we introduce a programming language in the style of Harel [Ha] and a
related calculus. In the sequel V will be an alphabet, i.e. some finite (non empty) set of symbols. \( V^* \) is the set of all strings over V. \( \varepsilon \) will always be used for the empty string. The concatenation of two strings is denoted by juxtaposition.

2.1 Languages and regular expressions

A language over V is a subset of \( V^* \), i.e. an element of \( \mathcal{P}(V^*) \). Let \( X, Y \in \mathcal{P}(V^*) \). Their concatenation \( XY \) is the language defined by

\[
XY = \{ xy | x \in X \land y \in Y \}
\]

Clearly \( \{\varepsilon\} \) is the unit of (language) concatenation. Moreover, concatenation is associative and distributes over arbitrary unions. Also, if \( F, G \subseteq V \), then

\[
(2.1.1) \quad FX \cap GY = (F \cap G)(X \cap Y)
\]

Furthermore we introduce

\[
(2.1.2) \quad X^0 = \{\varepsilon\},
(2.1.3) \quad X^{k+1} = XX^k \text{ for all } k \in \mathbb{N},
(2.1.4) \quad X^* = \bigcup_{k \in \mathbb{N}} X^k,
\]

In the sequel we frequently use the following definition

2.1.5 Definition \{\text{FIRST}\}

The function FIRST : \( \mathcal{P}(V^*) \) \( \rightarrow \) \( \mathcal{P}(V) \) is defined by

\[
\text{FIRST}(X) = \{ a \in V | \exists x \in V^* : ax \in X \}
\]

\( \square \)

So the FIRST of a language consists of the set of initial symbols of the strings of that language. Conversely, if we have a subset \( F \) of \( V \) we can construct the language of all strings with initial elements in \( F \).

2.1.6 Definition \{\text{\_}\}

The function \( \_ : \mathcal{P}(V) \rightarrow \mathcal{P}(V+) \) is defined by

\[
F = FV^*
\]

\( \square \)

For \( a \in V \) we shall use the abbreviation \( a \) instead of \( \{a\} \).

The following elementary properties hold for all \( F, G \in \mathcal{P}(V), X \in \mathcal{P}(V^*) \)

\[
(2.1.7) \quad F \subseteq G \Rightarrow F \subseteq G
(2.1.8) \quad \text{If } \varepsilon \notin X \text{ then } X \subseteq \text{FIRST}(X)
\]

In the sequel we shall sometimes use that for \( X \in \mathcal{P}(V^*), F \in \mathcal{P}(V) \):

\[
(2.1.9) \quad \text{FIRST}(X) \subseteq F = (F \cap X = X)
\]

"\( \Leftarrow \)" follows immediately from FIRST(\( F \)) = \( F \).

Next we prove "\( \Rightarrow \)"
\[ \text{FIRST}(X) \subseteq F \]
\[ = \{ (2.1.7) \} \]
\[ \text{FIRST}(X) \subseteq F \]
\[ \Rightarrow \{ (2.1.8) \}, X \in \mathcal{P}(V^+) \} \]
\[ X \subseteq F \]
\[ = \]
\[ F \cap X = X \]

Next we summarize regular expressions, corresponding languages and some related notions.

2.1.10 Definition \{ RE, regular expressions over V \}
The set \( \mathcal{R}e \) of regular expressions over \( V \) is the smallest set \( X \) satisfying the following rules. For all \( a \in V, e, f \in X \):
\[ X \subseteq X \]
\[ = \]
\[ a X \]
\[ \{ e.f \} X \]
\[ \{ e|f \} X \]
\[ e^* X \]

In order to save brackets we assign priorities to the operators \( | \) and \( . \) and \( * \) such that \( \text{prio}(*) > \text{prio}(.) > \text{prio}(|) \). Also outermost brackets will be omitted.

For regular expressions we define the corresponding languages as follows.

2.1.11 Definition \{ L, language defined by regular expression \}
The mapping \( L: \mathcal{R}e \rightarrow \mathcal{P}(V^*) \) is defined recursively as follows.
For all \( a \in V, e, f \in \mathcal{R}e \):
\[ L(\epsilon) = \{ \epsilon \} \]
\[ L(a) = \{ a \} \]
\[ L(e.f) = L(e) \cup L(f) \]
\[ L(e|f) = L(e) \cap L(f) \]
\[ L(e^*) = (L(e))^* \]

Some elementary properties of languages defined by regular expressions are described by the functions given in the following definitions.

2.1.12 Definition \{ Empty \}
The predicate \( \text{Empty} \) on \( \mathcal{R}e \) is defined recursively as follows.
For all \( a \in V, e, f \in \mathcal{R}e \):
\[ \text{Empty}(\epsilon) = \text{true} \]
- Empty(a) = false
- Empty(e.f) = Empty(e) ∧ Empty(f)
- Empty(e|f) = Empty(e) ∨ Empty(f)
- Empty(e*) = true

Clearly for all regular expressions e, Empty(e) = ε ∈ L(e)

2.1.13 Definition { First }
The mapping First : REV → P(V) is defined recursively as follows.
For all a ∈ V, e, f ∈ REV:
- First(e) = 0
- First(a) = {a}
- First(e.f) = {First(e), First(f)}, if ¬Empty(e)
- First(e|f) = First(e) ∪ First(f)
- First(e*) = First(e)

Trivially for all e ∈ REV, First(e) = FIRST(L(e)), i.e.
First(e) is the set of initial symbols of the strings of the language generated by e.

In the sequel we use the following name conventions
- k, m, n ∈ N
- a, b, c ∈ V
- x, y, z ∈ V*
- X, Y, Z ∈ P(V*)
- E, F, G ∈ P(V)
- e, f, g ∈ REV

Furthermore we shall use ~ for the set complement in V, i.e. ~F = V\F.

2.2 Regular programs
Starting from atomic programs we define regular programs, their semantics and a related calculus. The state space of our programs is V*, i.e. we have one (anonymous) variable of type V*.

2.2.1 Definition { AP, atomic programs }
AP = {skip, next} ∪ {F? | F ∈ P(V)}

2.2.2 Definition { RP, regular programs }
The set RP of regular programs is the smallest set R such that
- $\text{AP} \subseteq R$
- for all $\alpha, \beta \in R$
  \[(\alpha; \beta) \in R \quad (\alpha \circlearrowright \beta) \in R \quad \alpha^* \in R\]

As abbreviation we shall use

\[(2.2.3) \quad \alpha^0 = \text{skip} \]
\[(2.2.4) \quad \alpha^{k+1} = \alpha; \alpha^k\]

Moreover for $\alpha \in V$ we shall write $\alpha^?$ instead of $\{\alpha\}?$. Additional name conventions shall be

$\alpha, \beta, \gamma \in \text{RP}$

If, for the moment, the element of the state space is called $r$, then the programs "next" and "F?" correspond in a Pascal like language to "$r := \text{tail}(r)$" and "if head(r) $\notin F$ then abort". Furthermore ; and $\&$ and $\ast$ denote sequencing, nondeterministic selection and repetition of programs. The semantics of a regular program will be a (binary) relation on $V^*$. Informally this relation describes the possible initial and final states of the program. To abbreviate notation we introduce the relation

$I_X = \{ (x,x) \mid x \in X \}$

Relations can be composed in the following way. Let $R_1$ and $R_2$ be relations, then the composition $R_1; R_2$ is the relation given by

$R_1; R_2 = \{ (x,y) \mid \exists z \in V^* : (x,z) \in R_1 \land (z,y) \in R_2 \}$

Relations form a monoid with $;$ as operator and $I_{V^*}$ as unit. As subsets of $V^* \times V^*$ the union of relations exists. Furthermore given a relation $S \in \mathcal{P}(V^* \times V^*)$ its reflexive transitive closure $S^*$ is defined by

\[(2.2.5) \quad S^* = \bigcup_{k \in \mathbb{N}} S^k, \text{ where} \quad S^0 = I_{V^*}\]
\[(2.2.6) \quad S^{k+1} = S; S^k \quad \text{for } k \in \mathbb{N} \cdot\]

2.2.8 Definition \{ semantics of regular programs \}

The semantics of an RP program is described by a mapping $[.] : \text{RP} \rightarrow \mathcal{P}(V^* \times V^*)$, inductively defined by

- $[[\text{skip}]] = I_{V^*}$
- $[[\text{next}]] = \{ (ax, x) \mid a \in V, x \in V^* \}$
- $[[F?]] = I_{V^*} \quad (= I_E)$
- $[[\alpha; \beta]] = [[\alpha]] \circlearrowright [[\beta]]$
- $[[\alpha \circlearrowright \beta]] = [[\alpha]] \cup [[\beta]]$
- $[[\alpha^*]] = [[\alpha]]^*$

6
Using induction one can easily prove that
\[(\alpha^n) = [\alpha]^n\]

The semantics of a regular program \(\alpha\) should be interpreted as follows. If \((x, y) \in [\alpha]\), then a computation under control of \(\alpha\) may lead from initial state \(x\) to final state \(y\). For a regular program \(\alpha \in R\) and a set of final states \(Y \in P(V^*)\) the set of corresponding initial states \(<\alpha > Y\) is defined by
\[<\alpha > Y = \{ x \in V^* | \exists y \in V^*: (x, y) \in [\alpha]^\wedge y \in Y \}\]

Informally \(<\alpha > Y\) consists of those initial states for which it is possible that some computation under control of \(\alpha\) results in a final state in \(Y\). Of course, since \(\alpha\) is not deterministic, an initial state in \(<\alpha > Y\) may also lead to abortion, nontermination or termination in a final state not in \(Y\). We shall assume that \(<.>\) binds to the right, i.e.
\[<\alpha > <\beta > Y = <\alpha > (<\beta > Y)\]

Using set calculus, the following rules are easily verified
\[
\begin{align*}
(2.2.10) \quad \text{skip} > Y &= Y \\
(2.2.11) \quad \text{F?>Y} &= \emptyset \cap Y \\
(2.2.12) \quad \text{next} > Y &= VY \\
(2.2.13) \quad <\alpha; \beta > Y &= <\alpha > <\beta > Y \\
(2.2.14) \quad <\alpha \parallel \beta > Y &= <\alpha > Y \cup <\beta > Y \\
(2.2.15) \quad <\alpha^* > Y &= \bigcup n \in \mathbb{N} : (<\alpha^n > Y)
\end{align*}
\]

In the sequel, we sometimes use the program \(a?; \text{next}\). For a given set of final states \(Y \subseteq V^*\), the corresponding initial states can be calculated as follows
\[
\begin{align*}
(2.2.16) \quad \text{a?; next} > Y &= \{ (2.2.13) \} \\
\text{a?} <\text{next} > Y &= \{ (2.2.11), (2.2.12) \} \\
\{a\} \cap VY &= \{ \text{def} \{a\} \} \\
\{a\} V^* \cap VY &= \{ (2.1.1), \{a\} \subseteq V \} \\
\{a\} Y
\end{align*}
\]
2.3 Deterministic regular programs

In general an RP program is not deterministic, i.e. for a given initial state there may be several possible final states. Since our atomic programs are deterministic, the source of nondeterminism lies in constructions of the form $\alpha \parallel \beta$ or $\alpha^\ast$. By imposing some syntactic restrictions on the use of these constructions, we obtain a subset of RP, which consists of deterministic programs (see also [Ha], p.21).

2.3.1 Definition { DRP, deterministic regular programs }

The set DRP of deterministic regular programs is the smallest set $R$ such that

- $AP \subseteq R$
- for all $\alpha, \beta \in R, F, G \in \mathcal{P}(V)$ with $F \cap G = \emptyset$
  $$(\alpha; \beta) \in R$$
  $$((F?; \alpha) \parallel (G?; \beta)) \in R$$
  $$F?; \alpha^\ast; G? \in R$$

So in DRP, the selection and repetition are based on "guards". The two disjunctive guards $F?$ and $G?$ in a selection statement of the form $(F?; \alpha) \parallel (G?; \beta)$ allow an execution in which a deterministic choice between $\alpha$ or $\beta$ is made. Also the disjunctive guards $F?$ and $G?$ in a construct of the form $(F?; \alpha)^*; G?$ allow an execution with a suitable number of repetitions.

Clearly $DRP \subseteq RP$, therefore all previously given results for RP also hold for DRP.

2.3.2 Theorem

For all $\alpha \in DRP$, if $x \in V^*$ then there is at most one $y \in V^*$ such that $(x, y) \in [[\alpha]]$.

Proof. Induction with respect to $\alpha$.

Recall that for an RP program $\alpha$ the set $<\alpha>Y$ consists of the initial states for which it is possible that some computation under control of $\alpha$ results in a final state in $Y$. For a DRP program the set $<\alpha>Y$ consists of the initial states for which the (!) computation under control of $\alpha$ results in a (unique) final state in $Y$.

DRP programs can be transformed into equivalent guarded command language (GCL) programs in terms of a variable $r : V^*$ by a mapping $S : DRP \rightarrow GCL$ defined by

\[
\begin{align*}
S(\text{skip}) &= \text{skip} \\
S(\text{next}) &= r := \text{tail}(r) \\
S(\text{F?}) &= \text{if head}(r) \in F \rightarrow \text{skip} \text{ fi} \\
S(\alpha; \beta) &= S(\alpha); S(\beta) \\
S(F?; \alpha \parallel G?; \beta) &= \text{if head}(r) \in F \rightarrow S(\alpha) \\
&\quad \text{fi head}(r) \in G \rightarrow S(\beta) \\
&\quad \text{fi}
\end{align*}
\]
Recall that \( a? \) is an abbreviation for \( \{a\}? \). Consequently the program \( S(\{a\}? \) will be abbreviated to

\[
\text{if } \text{head}(r) = a \rightarrow \text{skip fi}
\]

Next we consider the relation between the semantics of a DRP program \( \alpha \) and the wp semantics of the corresponding GCL program (see for instance [Dij], [Gr]). If we identify predicates on \( V^* \) with subsets of \( V^* \), this relation is given in the following theorem

2.3.4 Theorem
For all \( \alpha \in \text{DRP} \) and \( Y \in \mathcal{P}(V^*) \)

\[
< \alpha > Y = \text{wp}( S(\alpha), Y )
\]

Proof. Induction with respect to \( \alpha \). See Appendix.

\( \Box \)

3. AN RP ACCEPTOR PROGRAM

Given a regular expression \( e \), we define a regular program \( T(e) \) that can act as an acceptor for the language \( L(e) \).

3.1 Definition \{ T \}

The function \( T : \mathcal{RE} \rightarrow \text{RP} \) is defined by

\[
\begin{align*}
T(\varepsilon) &= \text{skip} \\
T(a) &= a? \; ; \text{next} \\
T(e.f) &= T(e) \; ; T(f) \\
T(e|f) &= T(e) \| T(f) \\
T(e^*) &= T(e)^*
\end{align*}
\]

\( \Box \)

Next we try to relate the regular program \( T(e) \) to the regular language \( L(e) \). That will be done in theorem 3.5. There we shall use induction over regular expressions. To smooth the case of an expression of the form \( f^* \) we first give lemma 3.4. In theorem 3.5 as well as in lemma 3.4 we shall use the following abbreviations

\[
\begin{align*}
H(e) &= \forall Y \in \mathcal{P}(V^*) : < T(e) > Y = L(e) \; Y \\
H_n(e) &= \forall Y \in \mathcal{P}(V^*) : < T(e)^n > Y = L(e)^n \; Y
\end{align*}
\]

Trivially

\[
\begin{align*}
(3.2) & \quad H(\varepsilon) = \text{true} \\
(3.3) & \quad H_0(e) = \text{true} \quad \text{for all } e
\end{align*}
\]
Then we have the following lemma

3.4 Lemma
For all $e \in R:V$:

$$H(e) \Rightarrow (\forall n \in \mathbb{N} : H_n(e))$$

Proof.
Assume $H(e)$ holds. We prove the right-hand side of the implication with induction.

$n = 0$  Trivial, (3.3)

$n = m + 1$
IH:  $H_m(e)$

Let $Y \in \mathcal{P}(V^*)$

$$< T(e)^{m+1} > Y$$

$$= \{ (2.2.4), (2.2.13) \}$$

$$< T(e) < T(e)^m > Y$$

$$= \{ IH \}$$

$$< T(e) > (L(e)^m Y)$$

$$= \{ H(e) \}$$

$$L(e) L(e)^m Y$$

$$= \{ \text{associativity, (2.1.3)} \}$$

$$L(e)^{m+1} Y$$

$\square$

The relation between $T(e)$ and $L(e)$ is given in the following theorem

3.5 Theorem

$H(e)$ holds for all $e \in R:V$

Proof. { By induction on the structure of $e$ }

$e :: \epsilon$  Trivial, (3.2)

$e :: a$

Let $Y \in \mathcal{P}(V^*)$

$$< T(a) > Y$$

$$= \{ (3.1) \}$$

$$< a? \cdot \text{next} > Y$$

$$= \{ (2.2.16) \}$$

{a}Y
\[\begin{align*}
\text{IH1:} & \quad H(f) \\
\text{IH2:} & \quad H(g)
\end{align*}\]

Let \( Y \in \mathcal{P}(V^*) \)

\[< T(f,g) > Y \]

\[= \quad \{ (3.1) \} \]

\[< T(f); T(g) > Y \]

\[= \quad \{ (2.2.13) \} \]

\[< T(f) > < T(g) > Y \]

\[= \quad \{ \text{IH2} \} \]

\[< T(f) > (\mathcal{L}(g) Y) \]

\[= \quad \{ \text{IH1} \} \]

\[\mathcal{L}(f) \mathcal{L}(g) Y \]

\[= \quad \{ \text{associativity, (2.1.11)} \} \]

\[\mathcal{L}(f.g) Y \]

---

\[\begin{align*}
\text{IH1:} & \quad H(f) \\
\text{IH2:} & \quad H(g)
\end{align*}\]

Let \( Y \in \mathcal{P}(V^*) \)

\[< T(f|g) > Y \]

\[= \quad \{ (3.1) \} \]

\[< T(f) \parallel T(g) > Y \]

\[= \quad \{ (2.2.14) \} \]

\[< T(f) > Y \cup < T(g) > Y \]

\[= \quad \{ \text{IH1, IH2} \} \]

\[\mathcal{L}(f) Y \cup \mathcal{L}(g) Y \]

\[= \quad \{ \text{distributivity} \} \]

\[(\mathcal{L}(f) \cup \mathcal{L}(g)) Y \]

\[= \quad \{ (2.1.11) \} \]

\[\mathcal{L}(f|g) Y \]

---

\[\begin{align*}
\text{IH1:} & \quad H(f) \\
\text{IH2:} & \quad H(g)
\end{align*}\]

Let \( Y \in \mathcal{P}(V^*) \)

\[< T(f^*) > Y \]

\[= \quad \{ (2.1.11) \} \]

\[\mathcal{L}(a) Y \]
IH: \( H(f) \)

Let \( Y \in \mathcal{P}(V^*) \)

\[
< T(f^*), Y > = \{ (3.1) \} \quad < T(f)^*, Y > = \{ (2.2.15) \}
\]

\[
\bigcup n \in \mathbb{N} : (< T(f)^* >, Y)
\]

\[
= \{ \text{IH, Lemma 3.4} \} \quad \bigcup n \in \mathbb{N} : (L(f^*) Y)
\]

\[
= \{ \text{distributivity} \} \quad (\bigcup n \in \mathbb{N} : L(f)^* Y)
\]

\[
= \{ (2.1.4) \} \quad L(f)^* Y
\]

\[
= \{ (2.1.11) \} \quad L(f^*) Y
\]

Hence we have shown that for all \( e \) and \( Y \)

(3.6) \quad < T(e), Y > = L(e)Y

Informally, this result can be interpreted as follows. Let the anonymous element of the state space again be denoted by \( r \). Then, if initially \( r \in L(e)Y \), it is possible that the program \( T(e) \) terminates with \( r \in Y \). Also, if initially \( r \notin L(e)Y \), then it is not possible that the program \( T(e) \) terminates with \( r \in Y \), i.e. it does not terminate or it terminates with \( r \notin Y \). More specific results can be obtained by making several choices for \( Y \):

i. Suppose \( r \) contains an \( L(e) \) prefix, i.e. \( r = xy \) with \( x \in L(e) \). By taking \( Y = \{ y \} \) we obtain from (3.6) that \( T(e) \) can terminate with \( r = y \), i.e. the \( L(e) \) prefix \( x \) has been removed.

ii. Suppose \( r \) does not contain an \( L(e) \) prefix, i.e. \( r \notin L(e)V^* \). Then by taking \( Y = V^* \) we obtain from (3.6) that the program \( T(e) \) cannot terminate with \( r \in V^* \), i.e. \( T(e) \) does not terminate.

Note that this means that from i. and ii. we can infer that if \( T(e) \) terminates, it always has removed an \( L(e) \) prefix from \( r \).

As an example of the theorem, consider the regular expression \( e = (a|e).b \). Then
\[ T(e) = T(a;\varepsilon); T(b) \\
\quad (T(a) \mid T(\varepsilon)); T(b) \\
\quad ((a?; \text{next}) \mid \text{skip}); b?; \text{next} \]

In this case \( \mathcal{L}(e) = \{ab, b\} \) and taking \( Y = \{\varepsilon\} \) leads to
\[ <(a?; \text{next}) \mid \text{skip}; b?; \text{next} > \{\varepsilon\} = \{ab, b\} \]

So starting the program with \( r = ab \) or \( r = b \) may lead to \( r = \varepsilon \) and starting with \( r \neq ab \) and \( r \neq b \) never leads to a final state with \( r = \varepsilon \). In particular for \( r = aa \) the program always aborts and for \( r = aba \) the program may terminate with \( r = a \) which is not in \( Y \).

Next we show that \( T(e) \) behaves like an acceptor.

**3.7 Definition**  \{ Language accepted by program \}
The language accepted by an RP program \( \alpha \) is defined as
\[ L(\alpha) = \{ x \in \mathbb{V}^* | (\forall y \in \mathbb{V}^* : xy \in <\alpha>{\{y\}}) \} \]
\( \square \)

Informally this means that the program \( \alpha \) can remove an \( L(\alpha) \) prefix from its input.

**3.8 Theorem**
For all \( e \in \mathbf{Rev} \):
\[ L(T(e)) = \mathcal{L}(e) \]
Proof.
\[ L(T(e)) = \{ \text{def 3.7} \} \]
\[ \{ x \in \mathbb{V}^* | (\forall y \in \mathbb{V}^* : xy \in <T(e){\{y\}}) \} \]
\[ = \{ (3.6) \} \]
\[ \{ x \in \mathbb{V}^* | (\forall y \in \mathbb{V}^* : xy \in \mathcal{L}(e){\{y\}}) \} \]
\[ = \mathcal{L}(e) \]
\( \square \)

In general the programs generated by the mapping \( T \) given in 3.1 are not in DRP. Hence in most cases those programs are not deterministic. The construction of DRP acceptor programs will be discussed in the next section.

**4. AN RP ACCEPTOR PROGRAM WITH LOOKAHEAD**
Given a regular expression \( e \), we describe the construction of an RP acceptor for \( \mathcal{L}(e) \) using lookahead. Consider a regular expression \( ef \). The mapping \( T \) as given
in (3.1) leads to the acceptor program

\[ T(e) \sqcup T(f) \]

which is generally not deterministic. As a first step towards a DRP version of such an acceptor, we can take

\[(E?; T(e)) \sqcup (F?; T(f))\]

for suitable \(E, F \in \mathcal{P}(V)\) with \(E \cap F = \emptyset\). A possible choice for \(E, F\) could be \(E = \text{First}(e)\) and \(F = \text{First}(f)\). However, this may result in incorrect programs. Consider expressions of the form \((e|f).g\). The suggestion above leads to

\[(4.2) (\text{First}(e)?; T(e)) \sqcup (\text{First}(f)?; T(f)); T(g)\]

If, for instance, \(\varepsilon \in \mathcal{L}(f)\) then the program \(T(f)\) can accept the empty string, i.e. it can act as a skip. The acceptance of a string of \(\mathcal{L}(g) \subseteq \mathcal{L}((e|f).g)\) should proceed as follows: first \(T(f)\) is chosen, which performs a skip, then \(T(g)\) actually processes the string. However, \(T(f)\) is guarded by \(\text{First}(f)?\), which is not related with the initial symbols of a string from \(\mathcal{L}(g)\). To be more specific, take for instance \((a|e).b\), then the suggestion above yields the program

\[ ((\{a\}?; T(a)) \sqcup (\emptyset?; T(e))); T(b) \]

This program cannot accept the string \(b \in \mathcal{L}(b) \subseteq \mathcal{L}((a|e).b)\). Another example is given by \((a|e^*).b\). This would yield the program

\[ ((\{a\}?; T(a)) \sqcup (\{e\}?; T(e^*))); T(b) \]

Again the string \(b \in \mathcal{L}(b) \subseteq \mathcal{L}((a|e^*).b)\) is not accepted by this program. Hence the guards in (4.1) should be chosen more delicately than in (4.2). In general, for an expression of the form \((e|f).g\) with \(\varepsilon \in \mathcal{L}(f)\), the guard \(F?\) in the corresponding DRP program

\[ ((E?; T(e)) \sqcup (F?; T(f))); T(g) \]

should also allow execution of \(T(f)\) if a string of \(\mathcal{L}(g)\) has to be accepted. Hence in such case the guard \(F?\) should be \((\text{First}(f) \cup \text{First}(g))?\). Note that this means that in the construction of the acceptor corresponding to \((e|f)\) we also need as additional information the set \(\text{First}(g)\). Of course, the same situation appears if \(\varepsilon \notin \mathcal{L}(e)\).

In general in the construction of an acceptor for \(\mathcal{L}(e)\), we also need the set of symbols which can follow on a string of \(\mathcal{L}(e)\) (in some context). By adding an additional parameter to \(T\) we obtain the mapping \(T\). To shorten notation we give the following definition.

4.3 Definition \(\{h\}\)

The mapping \(h : R \times E \times \mathcal{P}(V) \to \mathcal{P}(V)\) is defined by

\[ h(e, F) = \begin{cases} \text{First}(e), & \text{if } \neg \text{EMPTY}(e) \\ \text{First}(e) \cup F, & \text{otherwise} \end{cases} \]
In the sequel we shall use that, for all $m \geq 1$

\[(4.4) \quad \text{if } \text{FIRST}(X) \subseteq F, \text{ then } \text{FIRST}(L(e)^mX) \subseteq h(e, F)\]

The mapping $T$ is described in the following definition.

4.5 Definition \{ $T$ \}

The function $T$: $\mathcal{R}E_v \times \mathcal{P}(V) \to \mathcal{R}P$ is defined by

\[
\begin{align*}
T(e, F) &= \text{skip} \\
T(a, F) &= a?; \text{ next} \\
T(e.f, F) &= T(e, h(f, F)); T(f, F) \\
T(e|f, F) &= (h(e, F)?; T(e, F)) \parallel (h(f, F)?; T(f, F)) \\
T(e*, F) &= (h(e, F)?; T(e, F \cup \text{First}(e))^*; F)
\end{align*}
\]

Two tasks remain. First we prove a theorem similar to 3.5, i.e. we give a specification of $T(e, F)$. That results in theorem 4.9. Next, in section 5 we shall give conditions which imply that $T$ yields a DRP program.

Achieving our first task proceeds in the same way as in the previous section. Here we introduce the abbreviations

\[
\begin{align*}
\mathcal{H}(e) &= \forall F \in \mathcal{P}(V), Y \in \mathcal{P}(V^+): \\
&\quad \text{FIRST}(Y) \subseteq F \Rightarrow < T(e, F) > Y = L(e) Y \\
\mathcal{H}_n(e) &= \forall F \in \mathcal{P}(V), Y \in \mathcal{P}(V^+): \\
&\quad \text{FIRST}(Y) \subseteq F \Rightarrow < (h(e, F)?; T(e, F \cup \text{First}(e)))^n > Y = L(e)^n Y
\end{align*}
\]

Trivially

\[(4.6) \quad \mathcal{H}(e) = \text{true} \]
\[(4.7) \quad \mathcal{H}_0(e) = \text{true} \quad \text{for all } e\]

Then we have the following lemma

4.8 Lemma.

For all $e \in \mathcal{R}E_v$:

\[
\mathcal{H}(e) \Rightarrow ( \forall n \in \mathbb{N} : \mathcal{H}_n(e) )
\]

Proof.

Assume $\mathcal{H}(e)$ holds. We prove the RHS of the implication by induction. We use the abbreviation $\alpha$ for the program $(h(e, F)?; T(e, F \cup \text{First}(e)))$.

\[
\begin{align*}
n = 0 & \quad \text{Trivial, (4.7)} \\
n = m + 1 & \\
\text{IH: } & \quad \mathcal{H}_m(e)
\end{align*}
\]
Let $F \in \mathcal{P}(V)$, $Y \in \mathcal{P}(V^+)$ such that $\text{FIRST}(Y) \subseteq F$. Then

$< \alpha^{m+1} > Y$

$= \{ \text{(2.2.13)} \}$

$< \alpha > < \alpha^m > Y$

$= \{ \text{FIRST}(Y) \subseteq F, \text{IH} \}$

$< \alpha > (\mathcal{L}(e)^m Y)$

$= \{ \text{def } \alpha, \text{(2.2.13)}, \text{(2.2.11)} \}$

$h(e, F) \cap < T(e, F \cup \text{First}(e)) > (\mathcal{L}(e)^m Y)$

$= \{ \mathcal{H}(e), \text{for all } m \geq 0 : \text{FIRST}(\mathcal{L}(e)^m Y) \subseteq F \cup \text{First}(e) \}$

$h(e, F) \cap \mathcal{L}(e) \mathcal{L}(e)^m Y$

$= \{ \text{(4.4)}, \text{(2.1.9)} \}$

$\mathcal{L}(e)^{m+1} Y$

\[ \square \]

This lemma can be compared with lemma 3.4 in the case without lookahead. After these preparations we can formulate the main theorem.

4.9 **Theorem**

$\mathcal{H}(e)$ holds for all $e \in \text{REV}$

**Proof.** { By induction on the structure of $e$ }

\begin{array}{ll}
e :: e & \text{Trivial, (4.6)} \\
e :: a & \text{Follows immediately from (2.2.16)} \\
e :: f.g & \mathcal{H}(f) \\
\text{IH1:} & \mathcal{H}(g) \\
\text{IH2:} & \mathcal{H}(g)
\end{array}

Let $F \in \mathcal{P}(V)$, $Y \in \mathcal{P}(V^+)$ such that $\text{FIRST}(Y) \subseteq F$. Then

$< T(f.g, F) > Y$

$= \{ \text{(4.5)}, \text{(2.2.13)} \}$

$< T(f, h(g, F)) > < T(g, F) > Y$

$= \{ \text{IH2, FIRST}(Y) \subseteq F \}$

$< T(f, h(g, F)) > (\mathcal{L}(g) Y)$

$= \{ \text{IH1, (4.4)} \}$

$\mathcal{L}(f) \mathcal{L}(g) Y$

$= \{ \text{associativity, (2.1.11)} \}$
\(\mathcal{L}(f,g) Y\)

e :: flg

IH1: \(\mathcal{H}(f)\)

IH2: \(\mathcal{H}(g)\)

Let \(F \in \mathcal{P}(V)\), \(Y \in \mathcal{P}(V^+))\) such that \(\text{FIRST}(Y) \subseteq F\). Then

\[< T(f\|g, F) > Y\]

\[= \{ (4.5) \} \]

\[< (h(f, F)\|; T(f, F)) \| (h(g, F)\|; T(g, F)) > Y\]

\[= \{ (2.2.14), (2.2.13) \} \]

\([< (h(f, F)\| > T(f, F) > Y) \cup ( < h(g, F)\| > T(g, F) > Y )\]

\[= \{ \text{IH1, IH2, FIRST}(Y) \subseteq F \} \]

\([< (h(f, F)\| > (\mathcal{L}(f) Y)) \cup ( < h(g, F)\| > (\mathcal{L}(g) Y))\]

\[= \{ (2.2.11) \} \]

\([h(f, F) \cap \mathcal{L}(f) Y) \cup (h(g, F) \cap \mathcal{L}(g) Y)\]

\[= \{ \text{FIRST}(Y) \subseteq F, (4.4), (2.1.9) \} \]

\(\mathcal{L}(f) Y \cup \mathcal{L}(g) Y\)

\[= \{ \text{distributivity, (2.1.11)} \} \]

\(\mathcal{L}(f\|g) Y\)

e :: f*

IH: \(\mathcal{H}(f)\)

Let \(F \in \mathcal{P}(V)\), \(Y \in \mathcal{P}(V^+)\) such that \(\text{FIRST}(Y) \subseteq F\). Then

\[< T(f^*, F) > Y\]

\[= \{ (4.5) \} \]

\[< (h(f, F)\|; T(f, F \cup \text{First}(f))^*; F) > Y\]

\[= \{ (2.2.13), (2.2.11) \} \]

\[< (h(f, F)\|; T(f, F \cup \text{First}(f))^* > (F \cap Y)\]

\[= \{ \text{FIRST}(Y) \subseteq F, (2.1.9) \} \]

\[< (h(f, F)\|; T(f, F \cup \text{First}(f))^* > Y\]

\[= \{ (2.2.15) \} \]

\[\cup n \in \mathbb{N} : (< (h(f, F)\|; T(f, F \cup \text{First}(f))^n > Y)\]

\[= \{ (4.8), \text{IH, FIRST}(Y) \subseteq F \} \]

\[\cup n \in \mathbb{N} : (\mathcal{L}(f)^n Y)\]

\[= \{ \text{distributivity, (2.1.4)} \} \]
\[ \mathcal{L}(f)^* Y \]
\[ = \quad \{ (2.1.11) \} \]
\[ \mathcal{L}(f^*) Y \]

So, if \( \text{FIRST}(Y) \subseteq F \) then
\[ (4.10) \quad <T(e, F)>Y = \mathcal{L}(e)Y \quad \text{for all } e, F \text{ and } Y \]

Note that the set \( F \) must contain all symbols which can possibly follow on the strings to be removed.

Compare these results to theorem 3.5 and (3.6) in the case without lookahead.

Again we give an informal interpretation of the theorem. Let the anonymous variable of the state space be denoted by \( r \). Then, if \( Y \in \mathcal{P}(V^+) \) and \( F \in \mathcal{P}(V) \) with \( \text{FIRST}(Y) \subseteq F \) and initially \( r \in \mathcal{L}(e)Y \), the program \( T(e, F) \) may terminate with \( r \in Y \). Also, if \( r \notin \mathcal{L}(e)Y \) the program \( T(e, F) \) cannot terminate with \( r \in Y \). More specific results can be obtained by making several choices for \( Y \),

i. Suppose \( r \) contains an \( \mathcal{L}(e)F \) prefix, i.e. \( r = xy \) with \( x \in \mathcal{L}(e) \) and \( y \in FV^* \). By taking \( Y = \{ y \} \) we obtain from (4.10) that \( T(e, F) \) can terminate with \( r = y \), i.e. the \( \mathcal{L}(e) \) prefix \( x \) has been removed.

ii. Suppose \( r \) does not contain an \( \mathcal{L}(e)F \) prefix, i.e. \( r \notin \mathcal{L}(e)FV^* \). Then, by taking \( Y = FV^* \), we obtain from (4.10) that \( T(e, F) \) cannot terminate with \( r \in FV^* \).

This can happen in two ways:

a. \( T(e, F) \) does not terminate, or

b. \( T(e, F) \) terminates with \( r \notin FV^* \), i.e. \( r \in (\neg F)V^* \).

In case b. we can conclude that, although \( r \) did not have an \( \mathcal{L}(e)F \) prefix, it did have an \( \mathcal{L}(e) \) prefix. This can be argued in the following way. The semantics of \( T(e, F) \) is a subset of the semantics of \( T(e) \); this is easily proved using induction with respect to \( e \). Hence if, for a given initial state, \( T(e, F) \) terminates, then also \( T(e) \) can terminate with the same final state. Hence (see ii. in the informal description of theorem 3.5) \( T(e) \), and so \( T(e, F) \) have removed an \( \mathcal{L}(e) \) prefix from \( r \).

The next example may illustrate this theorem. Take \( e = (a|e).b \) and \( Y = \{ \bot \} \). Choosing \( F = \{ \bot \} \), we get

\[ (4.11) \quad T(e, \{ \bot \}) = T(a|e, h(b, \{ \bot \})); T(b, \{ \bot \}) \]
\[ = T(a|e, \{ b \}); T(b, \{ \bot \}) \]
\[ = ((h(a, \{ b \})) \cup (h(e, \{ b \}))). \]

18
Then with \( \mathcal{L}(e) = \{ab, b\} \) we have
\[
< T(e, \{a\}) > \{a\} = \{ab\}
\]
So starting the program with \( r = ab\) and \( r = b\) may lead to \( r = \perp\). Starting with \( r \neq ab\) or \( r \neq b\) never leads to a final state with \( r = \perp\), in particular \( r = aab\) leads to abortion and \( r = aba\) leads to a final state \( r = a \notin Y\).

Next we consider again the relation between \( T(e, F) \) and the accepted language.

4.12 Definition \{ Language accepted by a program with lookahead \}
The language accepted by an RP program \( \alpha \) with lookahead set \( F \) is defined by
\[
\mathcal{L}(\alpha, F) = \{ x \in V^* \mid (\forall y \in FV^* : xy \in <\alpha>\{y\}) \}
\]

Similar to (3.8) we now have

4.13 Theorem
For all \( e \in \mathcal{R}(V) \), \( F \in \mathcal{P}(V) \) with \( F \neq \emptyset 
\[
\mathcal{L}(T(e, F), F) = \mathcal{L}(e)
\]
Proof.
\[
\begin{align*}
\mathcal{L}(T(e, F), F) &= \{ x \in V^* \mid (\forall y \in FV^* : xy \in <T(e, F)>\{y\}) \} \\
&= \{ x \in V^* \mid (\forall y \in FV^* : xy \in L(e)\{y\}) \} \\
&= \{ F \neq \emptyset \} \\
&\subseteq \mathcal{L}(e)
\end{align*}
\]
\[ \square \]

5. A DRP ACCEPTOR PROGRAM
The purpose of this section is the construction of a DRP acceptor. The mapping \( T \) defined in (4.5) generally does not yield DRP programs. For instance, the regular expression \( e = aa.b \) yields the acceptor
\[
T(e, F) = \{(a)?; T(a, F)\} \{T(a)\}\{a\}\{T(a.b, F)\}
\]
which is clearly not in DRP form. Recall that an RP program is in DRP form if selection and repetition are deterministic, see (2.3.1). From the definition of the
mapping $T$ in (4.5) and the definition of DRP programs, we infer that we have to put conditions on the guards in the selections and repetitions involved in (4.5). To formulate these conditions we give the following two definitions.

5.2 Definition \{ $D_s$ \}
The predicate $D_s : REV \times P(V) \rightarrow \mathbb{B}$ is defined recursively by:

$$
\begin{align*}
D_s(\varepsilon, F) &= \text{true} \\
D_s(\alpha, F) &= \text{true} \\
D_s(\alpha.f, F) &= D_s(\alpha, h(f, F)) \land D_s(f, F) \\
D_s(\alpha.f, F) &= (h(\alpha, F) \cap h(f, F)) = \emptyset \land D_s(\alpha, F) \land D_s(f, F) \\
D_s(\alpha^*, F) &= D_s(\alpha, \text{First}(\alpha) \cup F)
\end{align*}
$$

5.3 Definition \{ $D_r$ \}
The predicate $D_r : REV \times P(V) \rightarrow \mathbb{B}$ is defined recursively by:

$$
\begin{align*}
D_r(\varepsilon, F) &= \text{true} \\
D_r(\alpha, F) &= \text{true} \\
D_r(\alpha.f, F) &= D_r(\alpha, h(f, F)) \land D_r(f, F) \\
D_r(\alpha.f, F) &= (h(\alpha, F) \cap F = \emptyset) \land D_r(\alpha, \text{First}(\alpha) \cup F)
\end{align*}
$$

Informally, the predicates $D_s(\alpha, F)$ and $D_r(\alpha, F)$ imply the determinism of all selections respectively repetitions in the program $T(\alpha, F)$. Using induction on the structure of $\alpha$ we can prove theorem 5.4.

5.4 Theorem
For all $\alpha \in REV$, $F \in P(V)$ with $D_s(\alpha, F) \land D_r(\alpha, F)$:

$$
T(\alpha, F) \in \text{DRP}
$$

Using theorem 2.3.2 we conclude that if $D_r(\alpha, F)$ and $D_s(\alpha, F)$ hold, then $< T(\alpha, F) > X$ consists of the initial states for which the program $T(\alpha, F)$ terminates in a unique final state in $X$. If $\text{FIRST}(X) \subseteq F$, then theorem 4.9 yields $< T(\alpha, F) > X = L(\alpha)X$, i.e. $L(\alpha)X$ is the set of those initial states.

As an example take again $\alpha = (a|\varepsilon).b$ and $F = \{ \bot \}$. Then

$$
\begin{align*}
D_s(\alpha, \{ \bot \}) &= D_s(a|\varepsilon, h(b, \{ \bot \})) \land D_s(b, \{ \bot \}) \\
&= D_s(a|\varepsilon, \{ b \}) \land \text{true} \\
&= (h(a, \{ b \}) \cap h(\varepsilon, \{ b \}) = \emptyset) \\
&\phantom{=} \land D_s(a, \{ b \}) \land D_s(\varepsilon, \{ b \})
\end{align*}
$$
\[
\{(a) \cap \{b\} = \emptyset\} \land true \land true
\]

and
\[
D_r(e, \{\perp\}) = D_r(a \varepsilon, h(b, \{\perp\})) \land D_r(b, \{\perp\})
\]
\[
= D_r(a \varepsilon, \{b\}) \land true
\]
\[
= D_r(a, \{b\}) \land D_r(\varepsilon, \{b\})
\]
\[
= true \land true
\]
\[
= true
\]

Hence the program \(T(e, \{\perp\})\) is in DRP form, which is easily verified from (4.11). On the other hand, consider the example \(e = a[(a,b)]\) and \(F\) arbitrary.

\[
D_s(e, F) = (h(a, F) \cap h(a,b, F) = \emptyset)
\]
\[
\land D_s(a, F) \land D_s(a,b, F)
\]
\[
= (\{a\} \cap \{a\} = \emptyset)
\]
\[
\land true \land D_s(a, h(b, F)) \land D_s(b, F)
\]
\[
= false \land true \land true \land true
\]
\[
= false
\]

Indeed the program \(T(e, F)\), as given in (5.1) is not in DRP form.

Also the choice \(e = (a.b)*.a\) and \(F = \{\perp\}\) leads to a non DRP program (since \(D_r(e, F)\) does not hold). However, the equivalent regular expression \(e' = a.(b.a)*\) leads to a DRP program \(T(e', F)\).

Consider a (DRP) program \(T(e, F)\) with \(D_s(e, F)\) and \(D_r(e, F)\) that with a given initial state, does not terminate. This may happen due to
i) abortion in the atomic programs next (due to end of the input), \(a? , F? \)
ii) abortion in a selection statement of the form \((F? ;\alpha) \notin (G? ; \beta)\) since none of the guards holds
iii) nontermination in a repetition of the form \((F? ;\alpha)\ast ; G? \) since \(F\) always holds (and \(G\) does not).

Next we show that the situation in case iii. never occurs. The only repetitions in our programs appear due to regular expressions of the form \(f^*\), viz.

\[
T(f^*, F) = (h(f, F)? ; T(f, First(f) \cup F) )^* ; F?
\]

If \(D_s(f^*, F) \land D_r(f^*, F)\) holds, then \(h(f, F) \cap F = \emptyset\), which implies that \(-EMPTY(f)\) holds. Then the program \(T(f, First(f) \cup F)\) in (5.5) always removes a non empty prefix of \(L(f)\) from the input. This can happen only finitely many times.
If one is interested in acceptors written in GeL, we can compose the mapping \( T \) with the mapping \( S : \text{DRP} \to \text{GCL} \) as given in (2.3.3). This leads to the mapping \( R = S \circ T : \text{REv} \times \mathcal{P}(V) \to \text{GCL} \).

\[
(5.6) \quad R(\epsilon, F) = \text{skip} \\
R(a, F) = \text{if } \text{head}(r) = a \rightarrow \text{skip}; r := \text{tail}(r) \\
R(e.f, F) = R(e, h(f, F)); R(f, F) \\
R(e|f, F) = \text{if } \text{head}(r) \in h(e, F) \rightarrow R(e, F) \\
\text{fi} \\
R(e^{*}, F) = \text{do } \text{head}(r) \in h(e, F) \rightarrow R(e, F \cup \text{First}(e)) \text{ od} \\
\text{if } \text{head}(r) \in F \rightarrow \text{skip} \text{ fi}
\]

From the theorems 2.3.4, 4.9 and 5.4 we now obtain that, if FIRST(Y) \( \subseteq F \) and \( D_{s}(e, F) \wedge D_{s}(e, F) \) holds, then

\[
wp(R(e, F), Y) = L(e)Y
\]

6. A MODIFIED DRP ACCEPTOR

The correctness of the program \( T(e, F) \), as specified in (4.10) requires only the condition FIRST(Y) \( \subseteq F \). Then to obtain determinism the additional conditions \( D_{s}(e, F) \) and \( D_{s}(e, F) \) have to be imposed. However the resulting GCL program, as given in (5.6), contains, in the case \( e^{*} \), the statement

"if head(r) \in F \rightarrow \text{skip fi}".

This statement comes from the last case in the definition of \( S \), see (2.3.4), i.e.

\[
S((F?; \alpha)^{*}; G?) = \text{do } \text{head}(r) \in h(e, F) \rightarrow S(\alpha) \text{ od} \\
\text{if } \text{head}(r) \in G \rightarrow \text{skip fi}
\]

Note that if \( G = \neg F \) the right-hand side is equivalent to

\[
\text{do } \text{head}(r) \in h(e, F) \rightarrow S(\alpha) \text{ od}
\]

In this section we modify the mapping \( T \) in order to use this observation. The modified version of \( T \) leads to some adaptions of the results given in sections 4 and 5. We summarize them in this section. It will turn out that the predicate \( D_{s}(e, F) \) is already needed in the correctness proofs and for determinism we only need the additional condition \( D_{s}(e, F) \).

6.1 Definition \{ modified mapping \( \tilde{T} \) \}

The function \( \tilde{T} : \text{REv} \times \mathcal{P}(V) \to \text{RP} \) is defined by

\[
\tilde{T}(\epsilon, F) = \text{skip} \\
\tilde{T}(a, F) = a?; \text{ next}
\]
\[ \bar{T}(e.f, F) = \bar{T}(e, h(f, F)); \bar{T}(f, F) \]
\[ \bar{T}(e.f', F) = (h(e, F)?; \bar{T}(e, F)) \cup (h(f, F)?; T(f, F)) \]
\[ \bar{T}(e^*, F) = (h(e, F)?; \bar{T}(e, F \cup \text{First}(e)))^*; (\neg h(e, F))? \]

\[ \Box \]

Again we abbreviate notation by introducing
\[ \bar{H}(e) = \forall F \in \mathcal{P}(V), Y \in \mathcal{P}(V^+): \]
\[ D_r(e, F) \land \text{FIRST}(Y) \subseteq F \Rightarrow <\bar{T}(e, F)> Y = \mathcal{L}(e)Y \]
\[ \bar{H}_n(e) = \forall F \in \mathcal{P}(V), Y \in \mathcal{P}(V^+): \]
\[ D_r(e, F \cup \text{First}(e)) \land \text{FIRST}(Y) \subseteq F \Rightarrow \]
\[ <(h(e, F)?; \bar{T}(e, F)) > Y = \mathcal{L}(e)^nY \]

Then again
\[ \bar{H}(e) = \text{true} \]
\[ \bar{H}_0(e) = \text{true} \quad \text{for all } e \]

Instead of lemma 4.8 and theorem 4.9 now we have the modified versions

6.2 Lemma
\[ \bar{H}(e) \Rightarrow (\forall n \geq 0 : \bar{H}_n(e)) \]

Proof. See lemma 4.8 with \( \mathcal{H} \) and \( \mathcal{H}_n \) replaced by \( \bar{H} \) and \( \bar{H}_n \).

6.3 Theorem
\( \bar{H}(e) \) holds for all \( e \in \text{REV} \)

Proof.
The theorem is again proved using induction with respect to \( e \). The only case which differs from theorem 4.9 is \( e = f^* \) which we treat here.

Suppose \( e = f^* \) and assume as induction hypothesis \( \bar{H}(f) \).
Let \( F \in \mathcal{P}(V), Y \in \mathcal{P}(V^+) \) with \( D_r(e, F) \) and \( \text{FIRST}(Y) \subseteq F \). From \( D_r(e, F) \) we conclude \( D_r(f, \text{First}(f) \cup F) \) and \( h(f, F) \cap F = \emptyset \). Hence also \( \text{FIRST}(Y) \subseteq \neg h(f, F) \).

Then
\[ <\bar{T}(f^*, F)> Y \]
\[ = \{ (6.1) \} \]
\[ <(h(f, F)?; \bar{T}(f, F \cup \text{First}(f)))^*; (\neg h(f, F))?> Y \]
\[ = \{ (2.2.13), (2.2.11) \} \]
\[ <(h(f, F)?; \bar{T}(f, F \cup \text{First}(f)))^* > (\neg h(f, F) \cap Y) \]
\[ = \{ (2.1.9) \} \]
\[ <(h(f, F)?; \bar{T}(f, F \cup \text{First}(f)))^* > Y \]

23
\[ \{ (2.2.15) \} \]
\[ \{ \text{lemma 6.2, IH, FIRST}(Y) \subseteq F \} \]
\[ \{ (2.1.4), \text{distribution} \} \]
\[ \{ (2.1.11) \} \]
\[ \mathcal{L}(f) \equiv Y \]

\[ \mathcal{L}(f^*) \equiv Y \]

So if FIRST(Y) \( \subseteq F \) and \( D_r(e,F) \), then
\[ \langle \tilde{T}(e,F) \rangle Y = \mathcal{L}(e)Y \] for all \( e, Y \) and \( F \).

Instead of theorems 4.13 and 5.4 we now get

6.4 \textbf{Theorem}
For all \( e \in \text{REV}, F \in \mathcal{P}(V) \) with \( F \neq \emptyset \) such that \( D_r(e,F) \)
\[ L(\tilde{T}(e,F), F) = \mathcal{L}(e) \]

6.5 \textbf{Theorem}
For all \( e \in \text{REV}, F \in \mathcal{P}(V) \) with \( D_s(e,F) \):
\[ \tilde{T}(e,F) \in \text{DRP} \]

Note that to conclude \( \mathcal{L}(e)Y = \langle \tilde{T}(e,F) \rangle Y \) from theorem 6.3, we now need FIRST(Y) \( \subseteq F \) and the additional condition \( D_r(e,F) \). Furthermore, to obtain a DRP program we need \( D_s(e,F) \) (see theorem 6.5). Hence for a deterministic acceptor we again need \( D_r \) and \( D_s \).

To obtain GCL acceptors we now can use the mapping \( \tilde{\mathcal{R}}: \text{REV} \times \mathcal{P}(V) \rightarrow \text{GCL} \)

\[ \tilde{\mathcal{R}}(e,F) = \text{skip} \]
\[ \tilde{\mathcal{R}}(a,F) = \text{if head}(r) = a \rightarrow \text{skip} \text{ fi; } r := \text{tail}(r) \]
\[ \tilde{\mathcal{R}}(e.f,F) = \tilde{\mathcal{R}}(e,h(f,F)); \tilde{\mathcal{R}}(f,F) \]
\[ \tilde{\mathcal{R}}(e|f,F) = \text{if head}(r) \in h(e,F) \rightarrow \tilde{\mathcal{R}}(e,F) \]
\[ \text{fi} \]
\[ \tilde{\mathcal{R}}(e^*,F) = \text{do head}(r) \in h(e,F) \rightarrow \tilde{\mathcal{R}}(e,F \cup \text{First}(e)) \text{ od} \]

Again, if FIRST(Y) \( \subseteq F \) and \( D_r(e,F) \land D_s(e,F) \)

24
The first acceptor mapping \( T \) was given in section 3. Although this is a correct acceptor, see theorem 3.8, its usability is very limited because in general it is a nondeterministic RP program. As a first step towards a deterministic acceptor, we defined in section 4 a second acceptor mapping \( \tilde{T} \), which uses one symbol lookahead. The correctness of this mapping is proved in the theorems 4.9/4.13. In section 5 we introduced the predicates \( D_T(e, F) \) and \( D_{\tilde{T}}(e, F) \), which imply that \( T(e, F) \) is a deterministic RP (DRP) program. We also gave a GCL version. To obtain a more elegant GCL acceptor, we treated in section 6 a somewhat modified mapping \( \tilde{T} \). The correctness of programs generated by \( \tilde{T} \) was proved in the theorems 6.3/6.4. There we needed already the condition \( D_{\tilde{T}}(e, F) \), see theorem 6.3. So with respect to correctness, the mapping \( T \) is somewhat more flexible than \( \tilde{T} \). However, with respect to the corresponding GCL version, the mapping \( \tilde{T} \) is preferable to \( T \). Note that in both cases we only obtain practically usable acceptors for \( L(e) \) if the predicates \( D_T(e, F) \) and \( D_{\tilde{T}}(e, F) \) hold. Clearly this is a limitation of this method. On the other hand, the method described here is much simpler than the usual construction via finite automata.

Deterministic acceptors for strings from \( L(e) \) followed by a symbol from \( F \), are obtained if \( D_T(e, F) \land D_{\tilde{T}}(e, F) \) holds. In these cases the semantics of \( T(e, F) \) (or \( \tilde{T}(e, F) \)) is in fact a partial function on the statespace, i.e. with every initial state corresponds at most one final state. If the initial state has a prefix in \( L(e) \), followed by an element of \( F \), this prefix is removed by the program \( T(e, F) \). Note that the determinism of \( T(e, F) \) implies that each string can have at most one prefix from \( L(e) \). (This result can also be proved straightforward using induction on the regular expression.)

The mappings \( T \) and \( \tilde{T} \) yield deterministic acceptors, if the conditions \( D_T(e, F) \) and \( D_{\tilde{T}}(e, F) \) hold. If these conditions do not hold, sometimes the expression \( e \) can be transformed into an equivalent expression \( e' \) for which these conditions do hold. For instance instead of \( e = (a.b)(a.c) \) we can use \( e' = a.(b|c) \) and instead of \( e = (a.b)^*a \) we might use \( e' = a.(b.a)^* \). Unfortunately this rewriting of \( e \) to satisfy the conditions \( D_T \) and \( D_{\tilde{T}} \) is not always possible. Consider for instance

\[ e = (a|ab|bc)^* \]
\[ F = \{ c, \perp \} \]

Clearly \( D_T(e, F) \) does not hold. Suppose \( e' \) is an equivalent expression such that \( D_T(e', F) \land D_{\tilde{T}}(e', F) \) holds. Then \( T(e', F) \) is a deterministic program. Hence starting with initial state abc\perp it removes the \( L(e) \) prefix ab, followed by an element of \( F \) and also it removes the \( L(e) \) prefix abc, followed by an element of \( F \). Clearly this is impossible, so the expression \( e' \) does not exist. (In fact we used the result mentioned
above that $D_2(e, F) \land D_3(e, F)$ implies that each string can have at most one prefix in $L(e)F)$. This situation can be compared with the case of recursive descent parsers for context free grammars. There the grammar has to satisfy the so-called LL(1) condition to ensure the correctness of the corresponding recursive descent parser. If a grammar does not satisfy the LL(1) condition, it is sometimes possible to transform it into an equivalent grammar which does satisfy this condition. However, there also exist grammars for which an equivalent LL(1) grammar does not exist.

Finally we remark that similarly to the theory of LL(k) grammars the acceptor mappings can be generalised to a lookahead of k symbols, i.e. $T$ becomes a mapping $T : REv \times \mathcal{P}(V^k) \rightarrow DRP$. 

26
REFERENCES


8. APPENDIX

DL SEMANTICS AND WP SEMANTICS OF DRP PROGRAMS

The GCL programs that we consider have $V^*$ as their statespace. In order to relate the semantics of DL and WP, we connect predicates (in the variable $r$) and sets as follows

$$Q^* = \{ \dot{r} \in V^* \mid Q(r) \}$$

In particular for $head(r) \in F$ we have

(8.1)  $(head(r)) \in F)^* = \emptyset$

Moreover, for $X \in P(V^*)$ its complement in $V^*$ will be denoted by $\overline{X}$

8.2 Definition {wp-semantics [Gr], set version}

• $wp(skip, Y) = Y$
• $wp(r := E, Y) = \{ r \in \text{dom}(E) \mid E(r) \in Y \}$
• $wp(S0; S1, Y) = wp(S0, wp(S1, Y))$
• $wp(if B \rightarrow S ft, Y) = B^* \cap wp(S, Y)$
• $wp(if B0 \rightarrow S0 \land B1 \rightarrow S1 ft, Y) = (B0^* \cup B1^*) \cap (B0^* \cup wp(S0, Y)) \cap (B1^* \cup wp(S1, Y))$
• $wp(do B \rightarrow S od, Y) = (\cup k \in \mathbb{N} : H_k(Y))$
  where
  $H_0(Y) = B^* \cap Y$
  $H_k(Y) = H_{k-1}(Y) \cup wp(if B \rightarrow S ft, H_{k-1}(Y))$
  for all $k \in \mathbb{N}$

\[ \square \]

8.3 Lemma 0.  $wp(r := tail(r), Y) = VY$

Proof.  $wp(r := tail(r), Y)$

$= \{ (8.2) \}$

$\{ r \in \text{dom}(E) \mid tail(r) \in Y \}$

$= V^+ \cap VY$

$= VV^* \cap VY$

$= \{ (2.1.1) \}$

$VY$

\[ \square \]
8.4 Lemma Let $\alpha \in \text{DRP}$ such that $\text{wp}(S(\alpha), X) = < \alpha > X$ for all $X \in \mathcal{P}(V^*)$. Let $F \in \mathcal{P}(V)$ and $Y \in V^*$ such that $F \cap Y = \emptyset$. Then

$$\text{wp}(\text{do} \ head(t) \in F \rightarrow S(\alpha) \ \text{od}, Y) = < (F?; \alpha)^* > Y$$

Proof. Define the sequence $L_k$ by

$$L_0 = \mathcal{E} \cap Y \quad (= Y)
L_k = L_0 \cup \text{wp}(\text{if} \ head(t) \in F \rightarrow S(\alpha) \ \text{fi}, L_{k-1})$$

First we derive a recurrency relation for $L_k$.

$$L_k = L_0 \cup \text{wp}(\text{if} \ head(t) \in F \rightarrow S(\alpha) \ \text{fi}, L_{k-1})$$

$$= \{ (8.2), (8.1) \}$$

$$L_0 \cup (\mathcal{E} \cap \text{wp}(S(\alpha), L_{k-1}))$$

$$= \{ \text{wp}(S(\alpha), X) = < \alpha > X \}$$

$$L_0 \cup (\mathcal{E} \cap < \alpha > L_{k-1})$$

$$= \{ \text{def } L_0, (2.2.11) \text{ and } (2.2.13) \}$$

$$Y \cup < F?; \alpha > L_{k-1}$$

From this relation it follows with induction that

$$L_k = (\bigcup_{i \in \{0, \ldots, k\}}: < (F?; \alpha)^i > Y)$$

Now, we calculate,

$$\text{wp}(\text{do} \ head(t) \in F \rightarrow S(\alpha) \ \text{od}, Y)$$

$$= \{ (8.2), L_k = H_k(Y) \}$$

$$= \{ (8.5) \}$$

$$= \{ \text{set calculus} \}$$

$$= \{ (2.2.15) \}$$

$$< (F?; \alpha)^* > Y$$

\square

8.6 Theorem

For all $\gamma \in \text{DRP}$

$$\text{wp}(S(\gamma), Y) = < \gamma > Y$$

Proof { By induction on DRP programs }

29
\( \gamma \in AP \)

- \( \text{wp}(S(\text{skip}), Y) = \text{wp}(\text{skip}, Y) = Y = <\text{skip}>Y \)
- \( \text{wp}(S(\text{next}), Y) = \text{wp}(r := \text{tail}(r), Y) = \{ (8.3) \} = VY = <\text{next}>Y \)
- \( \text{wp}(S(F?), Y) \)
  \[
  = \quad \{ (2.3.3) \} \\
  \text{wp}(\text{if head}(r) \in F \rightarrow \text{skip} \; r, Y)
  \]
  \[
  = \quad \{ (8.2), (8.1) \} \\
  F \cap \text{wp}(\text{skip}, Y)
  \]
  \[
  = \quad \{ (8.2) \} \\
  E \cap Y
  \]
  \[
  = \quad \{ (2.2.11) \}
  <F?>Y
  \]

\( \gamma :: \alpha; \beta \)

IH: \( \forall X : \text{wp}(S(\alpha), X) = <\alpha>X \land \text{wp}(S(\beta), X) = <\beta>X \)

\[
\text{wp}(S(\alpha; \beta), Y)
\]
\[
= \quad \{ (2.3.3) \} \\
\text{wp}(S(\alpha); S(\beta), Y)
\]
\[
= \quad \{ (8.2) \} \\
\text{wp}(S(\alpha), \text{wp}(S(\beta), Y))
\]
\[
= \quad \{ \text{IH, 2 times} \}
<\alpha > <\beta > Y
\]
\[
= \quad \{ (2.2.13) \}
<\alpha; \beta > Y
\]

\( \gamma :: (F?; \alpha) [\| (G?; \beta)), \text{ where } F \cap G = \emptyset \)

IH: \( \forall X : \text{wp}(S(\alpha), X) = <\alpha>X \land \text{wp}(S(\beta), X) = <\beta>X \)

\[
\text{wp}(S((F?; \alpha) [\| (G?; \beta))
\]
\[
= \quad \{ (2.3.3) \} \\
\text{wp}(\text{if head}(r) \in F \rightarrow S(\alpha) \\
\| \text{head}(r) \in G \rightarrow S(\beta) \\
\; r, Y)
\]
\[
= \quad \{ (8.2), (8.1) \} \)
\[(E \cup G) \cap (E \cup wp(S(\alpha), Y)) \cap (G \cup wp(S(\beta), Y))\]
\[= \{ \text{III, 2 times} \}\]
\[(E \cup G) \cap (E < \alpha > Y) \cap (G < \beta > Y)\]
\[= \{ \text{set calculus} \}\]
\[(E \cap G < \alpha > Y)\]
\[\cup\]
\[(E \cap < \alpha > Y \cap < \beta > Y)\]
\[\cup\]
\[(G \cap F < \beta > Y)\]
\[\cup\]
\[(G \cap < \alpha > Y \cap < \beta > Y)\]
\[= \{ F \cap G = \emptyset, \text{set calculus} \}\]
\[(E < \alpha > Y) \cup (G < \alpha > Y)\]
\[= \{ (2.2.11), (2.2.13) \}\]
\[<F?; \alpha > Y \cup <G?; \beta > Y\]
\[= \{ (2.2.14) \}\]
\[<\{F?; \alpha \} \{G?; \beta \}> Y\]

\[\gamma ::= ((F?; \alpha)^*; G?) \], where \(F \cap G = \emptyset\)

**III:** \[\forall X : wp(S(\alpha), X) = < \alpha > X\]
\[wp(S((F?; \alpha)^*; G?), Y)\]
\[= \{ (2.3.3) \}\]
\[wp(\text{do head(r) \in F \rightarrow S(\alpha) od} \]
\[\quad ; \text{if head(r) \in G \rightarrow skip f1, Y)} \]
\[= \{ (8.2), (8.1) \}\]
\[wp(\text{do head(r) \in F \rightarrow S(\alpha) od, G \cap Y})\]
\[= \{ F \cap G = \emptyset, (8.4) \}\]
\[<(F?; \alpha)^*; (G \cap Y)\]
\[= \{ (2.2.11), (2.2.13) \}\]
\[<(F?; \alpha)^*; G? > Y\]

\[\square\]
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

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
A systematic design of a parallel program for Dirichlet convolution.

89/7 P.Struik
A general theory of genetic algorithms.

89/8 E.H.L.Aarts
A.E.Eiben
K.M. van Hee
Characterizations of Delay-Insensitive Communication Protocols.

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.
Formal methods and tools for the development of distributed and real-time systems, p. 17.

Dynamic process creation in high-level Petri nets, pp. 19.

Foundations of Compositional Program Refinement - safety properties -, p. 38.

Decomposition of delay-insensitive circuits, p. 25.

On the delay-sensitivity of gate networks, p. 23.


A logic for one-pass, one-attributed grammars, p. 14.

Receptive Process Theory, p. 16.

Combining the functional and the relational model, p. 15.

A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).

A proof system for process creation, p. 84.

A proof theory for a sequential version of POOL, p. 110.

Proving termination of Parallel Programs, p. 7.

A proof system for the language POOL, p. 70.

Compositionality in the temporal logic of concurrent systems, p. 17.

A fully abstract model for concurrent logic languages, p. 23.

On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.
<table>
<thead>
<tr>
<th>Year/No</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>91/02</td>
<td>R.P. Nederpelt, H.C.M. de Swart</td>
<td>Implication. A survey of the different logical analyses &quot;if...then...&quot;, p. 26.</td>
</tr>
<tr>
<td>91/03</td>
<td>J.P. Katoen, L.A.M. Schoenmakers</td>
<td>Parallel Programs for the Recognition of P-invariant Segments, p. 16.</td>
</tr>
<tr>
<td>91/05</td>
<td>D. de Reus</td>
<td>An Implementation Model for GOOD, p. 18.</td>
</tr>
<tr>
<td>91/06</td>
<td>K.M. van Hee</td>
<td>SPECIFICATIEMETHODEN, een overzicht, p. 20.</td>
</tr>
<tr>
<td>91/07</td>
<td>E. Poll</td>
<td>CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.</td>
</tr>
<tr>
<td>91/12</td>
<td>E. van der Sluis</td>
<td>A parallel local search algorithm for the travelling salesman problem, p. 12.</td>
</tr>
<tr>
<td>91/14</td>
<td>P. Lemmens</td>
<td>The PDB Hypermedia Package. Why and how it was built, p. 63.</td>
</tr>
<tr>
<td>Paper ID</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>91/16</td>
<td>A.J.J.M. Marcelis</td>
<td>An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.</td>
</tr>
<tr>
<td>91/18</td>
<td>Rik van Geldrop</td>
<td>Transformational Query Solving, p. 35.</td>
</tr>
<tr>
<td>91/19</td>
<td>Erik Poll</td>
<td>Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.</td>
</tr>
<tr>
<td>91/23</td>
<td>K.M. van Hee, L.J. Somers, M. Voorhoeve</td>
<td>Z and high level Petri nets, p. 16.</td>
</tr>
<tr>
<td>91/24</td>
<td>A.T.M. Aerts, D. de Reus</td>
<td>Formal semantics for BRM with examples, p. 25.</td>
</tr>
<tr>
<td>91/25</td>
<td>P. Zhou, J. Hooman, R. Kuiper</td>
<td>A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, p. 52.</td>
</tr>
<tr>
<td>91/27</td>
<td>F. de Boer, C. Palamidessi</td>
<td>Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.</td>
</tr>
<tr>
<td>91/28</td>
<td>F. de Boer</td>
<td>A compositional proof system for dynamic process creation, p. 24.</td>
</tr>
<tr>
<td>91/30</td>
<td>J.C.M. Baeten, F.W. Vaandrager</td>
<td>An Algebra for Process Creation, p. 29.</td>
</tr>
</tbody>
</table>