Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods

Citation for published version (APA):

Document status and date:
Published: 01/01/1993

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.

Download date: 15. Oct. 2019
Eindhoven University of Technology
Department of Mathematics and Computing Science

Deriving the Aho-Corasick Algorithms:
A Case Study into the
Synergy of Programming Methods

by

Rik van Geldrop

Computing Science Note 93/01
Eindhoven, January 1993
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.
Deriving the Aho-Corasick Algorithms: A Case Study into the Synergy of Programming Methods

Rik van Geldrop

15 January 1993

Abstract

Imperative programs can be derived using methods like stepwise refinement, but they do not lend themselves very well to transformational programming. The Bird-Meertens approach offers a powerful transformation calculus for functional programs, but it pays little or no attention to imperative algorithms. In this paper we will show how the two approaches can be combined by giving a transformational derivation of some efficient and practical imperative programs, viz. the Aho-Corasick string pattern matching algorithms
1. Introduction

In Hoare's "An axiomatic basis for computer programming", [13], and Dijkstra's "A Discipline of Programming", [9], the basis is laid for a dominant method in imperative programming, see also [10, 14]. This method - with a strong heuristic component based on Hoare logic and stepwise refinement - is directed towards the derivation of efficient non-deterministic programs over an arbitrary datatype. For a transformational approach to programming however, the method is less appropriate because the programs are not (yet) equipped with a sufficiently rich algebraic structure fit for manipulation.

The Bird-Meertens formalism (BMF) is a method for functional programming [4, 3, 17]. In this method - based on equational reasoning within manipulatively attractive algebraic structures - functional expressions are transformed into efficient deterministic programs over a restricted class of datatypes. The main objective of BMF is to develop theorems about solutions of several problem classes.

Imperative programming and BMF are engaged in different machine levels. Imperative programs are more efficiently executable on present-day hardware than functional ones, but for transformational algorithm derivation however, BMF is preferable because of its algebraic laws. The goal of this paper is to have the best of both worlds: use BMF in favour of the derivation of efficient imperative programs. This combination of methods will be achieved as follows. We start from a trivially correct functional specification and transform this via BMF laws to an efficient functional program reflecting the structure of the solution. If an appropriate functional program is obtained, we have arrived at a point where implementation details are inevitable and we will continue the derivation in an imperative style. To aid the understanding of this swap, we developed some conversions from functional to imperative level. We will illustrate the cooperative action of methods in a derivation for the pattern matching algorithms of Aho-Corasick. More precisely, we will derive two algorithms which occur in [1]: Algorithm 1 and its adaption as outlined there in chapter 6. (In the sequel, we shall refer to these Aho-Corasick algorithms as AC-FAIL and AC-OPT respectively.)

The paper is organized as follows. In section 2, we start with some essential ingredients of BMF, their use and conversions to imperative components. Afterwards, in section 3, the pattern matching problem is specified and some necessary formal language properties are given. The derivation in view is subject of section 4. Finally, in section 5, we summarize our results and give some conclusions. Appendix A contains a summary of BMF and its problem specific extensions, while several other details and proofs are added in Appendix B. For the sake of convenience we mention that the laws and the lemmas to which is referred to in this paper can be found in Appendix A and Appendix B respectively.

2. The Method

In the forthcoming derivation we combined two methodologies: BMF and imperative pro-
gramming. Imperative programming is supposed to be well-known, so we only highlight the essential BMF notions and their relationship with imperative programming. The reader who is familiar with these subjects may skip this section, the laws and the lemmas used in the derivation are summarized in the appendices.

The BMF method is built around data types and their data structure preserving functions (homomorphisms). If $\alpha$ is an initial datatype, then there is a unique homomorphism from $\alpha$ to each datatype of the same kind. These homomorphisms (called catamorphisms in BMF) can be defined via a unique extension property (UEP). UEP provides BMF with important laws, fusion being one of them. Fusion is an example of a structure depending law. Structure independent laws are supplied by operators on functions, e.g. composition, function tupling and projection. Many BMF laws state the equality of functional expressions (so they are perfect for equational reasoning) and in some of them well-known programming techniques are made explicit (divide and conquer is represented by functional composition, e.g. in rewriting $f$ as $g \circ h$, if $f = g \circ h$, and promotion and tupling are captured in law 2 and law 7 respectively).

To apply BMF to a problem, one has to fix a suitable datatype and model the problem components in functions in order to find a specification in the form of a functional expression. Using the laws, this expression can be transformed yielding a BMF derivation. How such a derivation can be directed towards an efficient functional program, will be explained now.

Typically, catamorphisms are functions which may occur in (the definition part of) functional programs. One speaks of: catamorphisms are programs. Not every function over an initial datatype is a catamorphism, but there is a uniform way to extend a function to a catamorphic form: tupling with the identity function, [18]. In this sense one can speak of: each (primitive recursive) function over an initial datatype gives rise to a functional program via catamorphisms. The relative costs of functional programs can be calculated, if the existence of some set of atomic operators for the type is assumed. BMF [16] takes this view on efficiency by stating that the efficiency of a catamorphism is determined by the operators involved in the constructions of its definition. (Evidently, replacing operators by cheaper ones is one of their improving strategies.)

From the previous it will be clear that many (deterministic) problems over an initial datatype can be specified by catamorphisms and that derivations with a catamorphism as starting point may be directed towards an efficient functional program using the common efficiency strategies as divide and conquer, tupling and promotion.

After this global description of BMF, we will illustrate some of its components for a datatype which is used in the forthcoming derivation: the free monoid over (alphabet) $V$. Its definition and a list of relevant properties and notions is given in Appendix A. The free monoid over $V$, denoted by $(V^*, \cdot \cdot, [[]])$, is an initial datatype and its data structure preserving functions ("join-list" catamorphisms) are monoid morphisms. Consider the following functions over $V^*$ (for the definitions, see Appendix A)
map : If \( f : V \rightarrow \alpha \), then \( f^* : V^* \rightarrow \alpha^* \) is such that
\[
f^*([a_1, \ldots, a_n]) = [f(a_1), \ldots, f(a_n)]
\]

filter : If \( p : V \rightarrow \mathbb{B} \), then \( p<l : V^* \rightarrow V^* \) is such that
\[
p<l([]) = []
\]
\[
p<l(x \# [a]) = \begin{cases} p<l(x) \# [a] & \text{if } p(a) \\ p<l(x) & \text{otherwise} \end{cases}
\]

reduce : If \( \oplus \) an associative binary operator on \( \alpha \) with unit \( 1_\oplus \), then
\( \oplus/ : \alpha^* \rightarrow \alpha \) is such that
\[
\oplus/([a_1, a_2, \ldots, a_n]) = a_1 \oplus a_2 \ldots \oplus a_n
\]

left-reduce : If \( \oplus : \alpha \times V \rightarrow \alpha \) and \( e : \alpha \), then \( (\oplus \rightarrow e) : V^* \rightarrow \alpha \) such that
\[
(\oplus \rightarrow e)([ ]) = e
\]
\[
(\oplus \rightarrow e)([a_1, a_2, \ldots, a_n]) = (((e \oplus a_1) \oplus a_2) \ldots ) \oplus a_n
\]

Map, filter and reduce are "join-list" catamorphisms, while left-reduce is not. V* can also be equipped with a "cons-list" and a "snoc-list" structure. The difference between a "cons-list" (respectively "snoc-list") and a monoid structure on V* concerns their binary operators and can informally be explained as follows: in the monoid structure on V* the operator \( \# \) is defined for each pair \( (x_1, x_2) \) of finite lists over V, while in a "cons-list" (respectively "snoc-list") structure this operator is only defined if \( x_1 \) (respectively \( x_2 \)) is a singleton list. Clearly, the monoid structure is richer than that of "cons-list" and "snoc-list". If we look at the functions above in a "snoc-list" structure on V*, then we see that map, filter, reduce and left-reduce are "snoc-list" catamorphisms.

The UEP for "join-list" catamorphisms (law 0) states that each \( h : (V^*, \# , [ ]) \rightarrow (\beta, \oplus, 1_\oplus) \) is the unique extension of some function \( f \) to V*. Although it is not stated explicitly, it is not difficult to infer that
\[
f: V \rightarrow \beta \text{ is such that } f(a) = h([a])
\]

A UEP definition for map, filter and reduce is then
\[
f^* = \# / g^*
\]
\[
p<l = \# / f_p^*
\]
\[
\oplus/ = \oplus/ I^*
\]

\text{i.e.}
\[
f^* \text{ is the unique extension of } g \text{ to } V^*
\]
\[
p<l \text{ is the unique extension of } f_p \text{ to } V^*
\]
\[
\oplus/ \text{ is the unique extension of } I \text{ to } V^*
\]
The only concern of a functional programmer is to design the definition part of a program. E.g. a functional program for catamorphism h is designed if a defining form for h is established. But h can be defined in several ways, for instance, law 1 (each “join-list” catamorphism can be written as a left-reduce) offers an alternative. Finding an efficient defining form is the goal of most BMF derivations.

To enable an imperative continuation of a BMF derivation we need conversions from functional to imperative programs. All conversions needed in the derivation of the Aho-Corasick algorithms are given in Appendix B. To illustrate the several design decisions and to make it easier to read section 4.1, we will develop the conversion of “join-list” catamorphisms here.

Suppose we have to convert the functional program h(x), where h is defined by h = l/f*. For this conversion, the choice for the underlying imperative environment is free and can be adjusted to the problem under consideration. In our case we prefer a “cons-list” environment (i.e. an environment which supports a type list with the operations first and tail) because the Aho-Corasick algorithms scan a string from left to right. On cons-lists, h can be defined by the following recurrence relation

\[ \oplus/ f^*(\text{[]}) = 1_{\oplus} \quad \text{if} \; x = \text{[]} \]
\[ \oplus/ f^*([a] \parallel y) = f(a) \oplus \oplus/ f^*(y) \quad \text{if} \; x = [a] \parallel y \]

In a program derivation for h(x), this leads us to propose a tail-invariant. The program development is standard. It follows that S is a correct implementation of w = h(x)

S:  
  \begin{align*}
    & w := 1_{\oplus}; \; r := x \\
    & \text{do} \; r \neq [ ] \\
    & \quad \rightarrow a := \text{first}(r); \; r := \text{tail}(r) \\
    & \quad ; \; w := w \oplus f(a) \\
    & \text{od}
  \end{align*}

  \text{Invariant:} \; \oplus/ f^*(x) = w \oplus \oplus/ f^*(r) \quad \land \; r \in \text{suff}(x)

Remark To facilitate further conversions from the functional to the imperative level, we take a closer look to the BMF expression \( \oplus/ f^* \) and the program S. In S we have to add variables in order to save intermediate results, e.g. for the intended program result we used variable w. f appears in S in order to compute the differential with the previously (partly computed) result w. (For a good separation of concerns, one may introduce an fresh variable for the differential f(a). In view of efficiency, it is even needed to do so, if there is an appropriate recurrence relation for f.) The operator \( \oplus \) “adds” this differential to w (the invariant is re-established).

End of Remark

Using this conversion for “join-list” catamorphisms, we will find that \( S_m, S_f \) and \( S_r \) are correct implementations for \( f^*(x), p<l(x) \) and \( \oplus/(x) \) respectively, where
The conversion of a “join-list” catamorphism as given above in not included in the appendix, because it is an instance of the conversion scheme for a left-reduce which is given in Lemma 1. Since several functions which are not “join-list” catamorphisms can be written as a left-reduce, this lemma is more generally applicable and extends the class of functional programs which can be converted.

Thusfar, we converted very simple BMF expressions: one single function. For composite expressions we have the following rules

- **case f ◦ g**: Composition is converted to concatenation.
  E.g. an implementation of \( w = (f \circ g)(x) \) is \( S_1; S_2 \) where \( S_1 \) and \( S_2 \) are the conversions of \( w' = g(x) \) and \( w = f(w') \) respectively.
  Sometimes successive conversions can be interwoven, functionally this is expressed as \( f \circ g = k \). In our derivation, we will illustrate this for loop-fusion which is a special instance of law 6: fusion for left-reduces.

- **case f △ g**: Tupling is converted to a simultaneous computation.
  If \( w = (f \triangle g)(x) \), then \( w = (w_1, w_2) \) where \( w_1 = f(x) \) and \( w_2 = g(x) \). Imperatively, we have to introduce two variables to save the value of \( w \) and, consequently, the invariant needs two conjuncts to state the required relation for \( w \). In the imperative
style of programming, tupling is applied when an invariant is strengthened with a
conjunct involving a variable for a derived new expression.

Previously, we outlined the working method in BMF and gave some tools to trace out
a BMF derivation on the imperative level. As mentioned before, this derivation will be
continued imperatively in order to arrive at efficient imperative algorithms. We assumed
that the imperative method needs no further explanation.

3. The Problem

We aim at finding a derivation for the Aho-Corasick algorithms AC-OPT and AC-FAIL.
These algorithms compute solutions to the pattern matching problem which can be speci-
fied informally by: "Given text $S \in V^*$ and pattern set $P \in \mathcal{P}(V^*)$ compute all occurrences
of substrings of $S$ which are in $P$.

In order to serve as an example derivation for other imperative programs, we will distin-
guish between implementation independent (problem specific) and implementation depen-
dent (solution specific) aspects in its design. The problem specific characteristics comprise
the algebraic structure of the problem domain and it is obvious that they will be used in
the BMF part of the derivation. For the pattern matching problem, this (mathematical)
side of the derivation is considered in section 3.1. The solution specific characteristics will
be used to motivate the several choices which have to be made in derivations. One of
these choices is the problem modelling to obtain a formal specification. In section 3.2, the
Aho-Corasick algorithms lead us to a formal specification of the pattern matching prob-
lem. This trivially correct specification will be the starting point of the derivation given
in section 4.

3.1 Concepts and properties involved in pattern matching

Languages over alphabet $V$ (languages for short) are defined as subsets of $V^*$, where $V^*$
is the Kleene closure of $V$. Languages have a rather rich structure. Although it is not
a model for the spec algebra of Backhouse et all, [2], a main part of the spec axioms is
satisfied by languages and a reader familiar with the spec calculus may recognize several
of its notions, properties and calculation techniques in the sequel. The essential part of
the language structure is recapitulated now, more can be found in [15].

Being sets over a fixed universe, $V^*$, languages form a complete distributive and comple-
mented lattice under the inclusion order $\subseteq$. Moreover, languages can be concatenated
(denoted by $\cdot$):

$$ L \cdot M = \{ lm \mid l \in M \land m \in M \} $$

(as usual in language terminology, word concatenation is denoted by juxtaposition). This
concatenation is associative and has $\{\varepsilon\} - \varepsilon$ is the empty string - as its unit. Furthermore,
language concatenation is universally distributive over union, implying monotonicity w.r.t. \( \subseteq \) in both arguments. In some cases, concatenation distributes over intersection:

**Property 3.1.1**

Let \( n \in \mathbb{N} \). If \( R \in \mathcal{P}(V^n) \) or \( P \cup Q \subseteq \mathcal{P}(V^n) \), then

\[
(P \cap Q).R = P.R \cap Q.R
\]

\( \Box \)

From now on we will identify singleton languages with their unique word. A useful concept is the left factor, see also [8], which can be defined elegantly via Galois connections.

**Definition 3.1.2** {Left factor \( L/M \)}

Let \( L, M \in \mathcal{P}(V^*) \), then the left factor \( L/M \) is defined by

\[
X.M \subseteq L \iff X \subseteq L/M
\]

for each \( X \in \mathcal{P}(V^*) \)

\( \Box \)

Since \( .M \) distributes universally over \( \cup \), \( L/M \) is uniquely determined. See also [2]. The advantage of definitions via Galois connections is that they encapsulate a lot of properties which can easily picked out by instantiations. E.g.

**Properties 3.1.3**

\( a. \) \( L/M \subseteq \text{prefs}(L) \)

\( b. \) \( z.M \subseteq L \iff z \in L/M \)

\( c. \) \( zm \in L \iff z \in L/m \)

Proof. All properties follow from substitutions in definition 3.1.2.

For \( a. \) take \( X := L/M \), take \( X := \{z\} \) for \( b. \) and \( X, M := \{z\}, \{m\} \) for \( c. \).

\( \Box \)

Left factors enable us to formulate an interesting property of languages in a neat way

**Property 3.1.4** For all \( X, Y \in \mathcal{P}(V^*) \) and \( z \in V^* \)

\[
X.z \cap Y = (X \cap Y/z).z
\]

Proof. See Appendix B.

Prefix closed languages, i.e. languages \( L \) such that \( \text{pref}(l) \subseteq L \) for each \( l \in L \), form a proper subclass with additional properties.

**Properties 3.1.5** Let \( L \in \mathcal{P}(V^*) \) be prefix closed, then

\( a. \) \( \text{prefs}(L) = L \)

\( b. \) \( X.z \cap L \subseteq L.z \)

for each \( X \in \mathcal{P}(V^*) \) and \( z \in V^* \)

Proof. See Appendix B.
In the forthcoming derivation we will apply BMF laws on the problem domain of languages. Language theory and BMF have their own terminology. Although both use the same algebra, they differ in the notation of their operations. We could have decided to use only one of both terminologies, but it turned out that BMF was appropriate for the first part of the derivation while language terminology was preferred for the second part. Of course, we like to avoid notational conversions during the derivation, therefore we conclude with some links which smoothen switching between both terminologies. Firstly, we recall that each binary operator gives rise to two unary ones via sectioning, e.g. for fixed $z \in V^*$, $(\| z) : V^* \to V^*$ is a left section of $\|$ (concatenation) and is defined by $(\| z)(x) = x \| z$ and $(z \in) : \mathcal{P}(V^*) \to \mathbb{I}\mathbb{B}$ is a right section of $\in$ (membership relation) and is defined by $(z \in)(X) = z \in X$. The left section of the membership relation is frequently used in the sequel and we decided to introduce a special name for it.

Definition 3.1.6 \{ L_\epsilon \}
Let $L \in \mathcal{P}(V^*)$. Then $L_\epsilon : V^* \to \mathbb{I}\mathbb{B}$ is defined by

$$L_\epsilon(z) = z \in L$$

With this definition, we can formulate 3.1.3c on the functional level as

$$L_\epsilon \circ (\| z) = (L/z_\epsilon)$$

Simple verifications suffice for the proofs of the following.

Conversions 3.1.7

a. $(\| z)^*(L) = L.z$

b. $L_\epsilon \triangle(X) = X \cap L$

3.2 Problem specification

To apply BMF on the pattern matching problem we have to fix a suitable datatype and model its composing parts in functions. Say we model occurrences of substrings via the function $\text{subs}$ and "matches $P$" via the predicate $p$, then pattern matching is specified by

$$p \triangle \text{subs}(S)$$

(If no ambiguities will occur, functional composition is denoted by juxtaposition.) The definitions of $\text{ subs}$ and $p$ have to be given, but they depend on the choices of the underlying datatype and the specific modelling of substrings. We will focus on them now.

In the choice of the datatype only the problem description is involved. For the pattern matching problem, its domain is the set of words over alphabet $V$. Since no specific structure is required in this problem, we choose for the most general list-type: the free monoid over $V$, denoted by $(V^*, \|, \[])$. The problem component $\text{ subs}$ can be modelled in several ways, e.g. an occurrence of substring $S$ can be uniquely determined by
\((l, v, r) \in V^* \times V^* \times V^* \text{ such that } lvr = S\)
\((l, v) \in V^* \times V^* \text{ such that } \exists r \in V^*: lvr = S\)
\((v, lv) \in V^* \times V^* \text{ such that } \exists r \in V^*: lvr = S\)
e.tc

(Recall that in language terminology juxtaposition denotes word concatenation.) To obtain an appropriate modelling for substrings, we examine the Aho-Corasick algorithms. These algorithms analyze the text \(S\) from left to right and matches are detected at endpoints. Each match is uniquely identified by a two-tuple consisting of the pattern and the initial substring of \(S\) where the matching is detected. Since these matches form a subset of all possible occurrences of substrings of \(S\), we choose to model occurrences of substrings by

\[
\text{subs}(S) = \{ (v, lv) \in V^* \times V^* \mid \exists r \in V^*: lvr = S \}
\]

The matching occurrences of substrings of \(S\) have to satisfy the predicate \(p\) such that

\[
p(v, lv) \equiv v \in P
\]

(\(p\) reflects "matches \(P\")). Using the \(P_e\) notation introduced in section 3.1, we obtain the following functional expression for \(p\)

\[
p = P_e \pi_1
\]

With these choices for \(\text{subs}\) and \(p\), our pattern matching problem is specified by

\[
(P_e \pi_1) \triangleleft \text{subs}(S)
\]

This specification will be the starting point of a (purely calculational) derivation of the Aho-Corasick algorithms which is given in the next section.

4. The Derivation

In derivations for programming problems, there is no predescribed order for the various design decisions which can be made. However, we prefer to postpone typical implementation decisions (e.g. representation choices) as long as possible in order to obtain a derivation of which the first part is concerned with laying bare the architecture of the solution and its second part is engaged in the organization and realization of the solution. In this kind of derivations, there is a separation of concerns over two machine levels. For both levels a suitable programming method can be chosen to achieve their task in the derivation if conversions from the higher to the lower level are available. On the one side, these conversions have to be attuned to the components on the higher level and on the other side to the specific implementation environment of the solution. For our application we developed two conversions, one for set-homomorphisms (Appendix B, Lemma 2) and one for functions with a left-reduce form (Appendix B, Lemma 1). Since the Aho-Corasick algorithms scan the text from left to right, their implementation environment can be viewed as a cons-list one and in this environment left-reduces can be efficiently implemented by iterative programs (see [12]). To emphasize the fact that the derivation is divided over two different levels we construct its first part on the functional level in section 4.1. When implementa-
tion details are inevitable, we continue the derivation on the imperative level in section 4.2.

4.1 Derivation on the functional level

In this part of the derivation, we shall use the following conventions.
- Most steps comprise of three parts. First we outline the transformation globally. Then, its correctness proof is given in part two and, finally, the transformation result is expressed in a functional and in an imperative version.
- The imperative programs are generated by the conversion lemmas of Appendix B. In order to use these lemmas, we have to show that the function(al program) under consideration, say h, is a set-homomorphism or a left-reduce. I.e. we have to exhibit a specific form for h. To facilitate these conversions, we decided to supply each functional program with a fact containing the justification for the conversion.
- During the derivation, we will put some requirements on several components. For sake of completeness, we repeat these assumptions in framed form before each transformation result.

1. { Specification }

In section 3.2 we designed the following formal specification for the pattern matching problem

\[(P \bowtie \pi_1) \downarrow \mathrm{subs}(S)\]

An imperative specification can be obtained by the standard conversion. Therefore, we recall that filters are homomorphisms. In this case, we have that \((P \bowtie \pi_1) \downarrow = U/ f_p^*,\) where \(f_p(x, y) = (x \in P \rightarrow \{(x,y)\}, \emptyset).\) Since \(U\) is an associative, commutative and idempotent operator with unit \(\emptyset, (P \bowtie \pi_1) \downarrow\) is a set-homomorphism. We instantiate Lemma 2 with this set-homomorphism and the argument \(\mathrm{subs}(S).\)

\[(P \bowtie \pi_1) \downarrow (\mathrm{subs}(S))\]

\[O := \emptyset; R := \mathrm{subs}(S)\]

\[; \text{ do } R \neq \emptyset\]

\[\rightarrow (v, lv) \in R; R := R - \{(v, lv)\}\]

\[; \text{ if } v \in P \rightarrow O := O \cup \{(v, lv)\}\]

\[\text{fi}\]

\[\text{od}\]

Fact:
\[(P \bowtie \pi_1) \downarrow = U/f_p^*\]

\[f_p(x, y) = (x \in P \rightarrow \{(x,y)\}, \emptyset )\]

Invariant:
\[O = (P \bowtie \pi_1) \downarrow (\mathrm{subs}(S) - R)\]

2. { Divide and conquer }

Our first step towards efficiency is "filter-promotion" (fusion for filters), i.e. we aim at filtering before (a total) computation of \(\mathrm{subs}(S).\) To perform fusion a factorization of \(\mathrm{subs}(S) - R\)
is needed. We developed such a factorization and stated it in law 10. Using this law, the specification is transformed by a typical BMF calculation.

\[(P\cdot\pi_1)\succeq\text{subs} = \{ \text{law 10} \}\]
\[(P\cdot\pi_1)\succeq\text{U/ } (\forall (\text{suff } A I))\ast \text{pref} = \{ \text{fusion, law 2} \}\]
\[\text{U/ } ((P\cdot\pi_1)\succeq \ast (\forall (\text{suff } A I))\ast \text{pref} = \{ \text{map distributivity, law 3} \}\]
\[\text{U/ } ((P\cdot\pi_1)\succeq (\forall (\text{suff } A I))\ast \text{pref} = \{ \text{law 11} \}\]
\[\text{U/ } (\forall (P\cdot\Delta \times I)(\text{suff } A I))\ast \text{pref} = \{ \times - \Delta \text{ fusion, law 9} \}\]
\[\text{U/ } (\forall (P\cdot\Delta \text{ suff } A I))\ast \text{pref} = \{ \text{intro h} \}\]

\[h \text{ pref}\]

From its form, we infer that \(h\) is a set-homomorphism (law 0). The standard conversion in Lemma 2 yields an imperative program which computes the application of \(h\) on argument \(\text{pref}(S)\).

\[
h(\text{pref}(S))\quad O := \emptyset ; \quad R := \text{pref}(S) \]
\[
; \quad \textbf{do} \quad R \neq \emptyset \quad \rightarrow \quad u \in R ; \quad R := R - \{u\} \quad \cdot \quad O := O \cup g(u) \quad \textbf{od} \]

\[
\text{Invariant: } \quad O = h(\text{pref}(S) - R), \quad \text{h = U/ g}\ast \]

\[
g = \forall (P\cdot\Delta \text{ suff } A I)\]

3. { Add construction for \text{pref} }\]

Now we aim at an over-all program for the specification, i.e. at a functional application on argument \(S\). One way to obtain such a program is exploring the existence of a left-reduce form for \(h\) \(\text{pref}\). We renounce this approach \(^1\) for two reasons:

\(^1\)This approach is typical for programming methods where efficiency steps are implicit in the tools, e.g.
a We are working on the functional level. Given a construction for h in the definition part of a functional program, one does not modify this into a construction for h pref, but one adds a construction for pref.

b In the BMF method efficiency steps are explicit, e.g. in laws. We like to illustrate the fusion law for left-reduces, law 6, and to show how this shapes up in imperative programming. Linking up with our functional approach we will achieve an over-all program by adding a construction for pref. To that end we explore the existence of a left-reduce form for pref. It turns out that a tupling with the identity function is needed:

\[(\text{pref } \Delta \text{ I}) = (\odot \mapsto (\{\varepsilon\}, \varepsilon))\]  

(4.1.1)

where \(\odot\) is defined by \((X, s) \odot a = (X \cup \{sa\}, sa)\)

This construction for pref gives rise to the following specification transformation

\[
\begin{align*}
\text{h pref} \\
= & \{ \text{ def } \Delta \} \\
\text{h } \pi_1 (\text{pref } \Delta \text{ I}) \\
= & \{ (4.1.1) \} \\
\text{h } \pi_1 (\odot \mapsto (\{\varepsilon\}, \varepsilon))
\end{align*}
\]

An imperative program for the transformed specification follows from Lemma 1 and Lemma 2.

\[
(h \pi_1 (\text{pref } \Delta \text{ I}))(S)
\]

\[
\begin{align*}
\text{prefS, } s & := \{\varepsilon\}, \varepsilon; r := S \\
\{ P_0 \} \\
; \text{do } r \neq \varepsilon \\
& \rightarrow a := \text{first}(r); r := \text{tail}(r) \\
& \quad; \text{prefS, } s := \text{prefS} \cup \{sa\}, sa \\
\text{od} \\
\{ P_0 \land r = \varepsilon \} \\
\; \text{O := } \emptyset; R := \text{prefS} \\
\{ P_1 \} \\
; \text{do } R \neq \emptyset \\
& \rightarrow u \in R; R := R - \{u\} \\
& \quad; O := O \cup g(u) \\
\text{od} \\
\{ P_1 \land R = \emptyset \}
\end{align*}
\]

Fact:

\[(\text{pref } \Delta \text{ I}) = (\odot \mapsto (\{\varepsilon\}, \varepsilon))\]

imperative programming.
(X, s) \odot a = (X \cup \{sa\}, sa) \quad \text{where yr = S} \\
\text{with} \\
h = \cup/ g^* \\
g = \wp (P_e \triangleleft \text{suff} \triangle I)

Remark We assumed that the occurrence of a projection function needs no further explanation with respect to the conversion to imperative programs.
End of Remark

4. { Fusion }
The previous program consists of a composition of functions in which the right-most one is a left-reduce. For this class of programs a left-reduce form is ensured, if some condition is satisfied (fusion law for left-reduces). A transformation of the (functional) program in 3 to a left-reduce form comes down to applying loop-fusion on its imperative conversion. (I.e. loop-fusion is the imperative counterpart of an instantiation of the fusion law for left-reduces.) Obviously, such transformation benefits efficiency, so our next goal is its justification.
As might be expected, the condition that has to be satisfied is the existence of a suitable operator. We prefer to construct this operator, $\oplus$, from the condition as we indicated in Appendix A.
The following calculations transform the specification in the way sketched before.

\[
\begin{align*}
  h \pi_1 (\odot \rightarrow (\{\varepsilon\}, \varepsilon)) \\
  = & \quad \{ \text{law 8} \} \\
  \pi_1 (h \times 1) (\odot \rightarrow (\{\varepsilon\}, \varepsilon)) \\
  = & \quad \{ \text{claim below} \} \\
  \pi_1 (\oplus \rightarrow (h(\{\varepsilon\}), \varepsilon)) \\
  = & \quad \{ h(\{\varepsilon\}) = g(\varepsilon) \} \\
  \pi_1 (\oplus \rightarrow (g(\varepsilon), \varepsilon)) \\
\end{align*}
\]

Claim \quad (h \times 1)(\odot \rightarrow (\{\varepsilon\}, \varepsilon)) = (\oplus \rightarrow (h(\{\varepsilon\}), \varepsilon))

Proof \{ by construction of $\oplus$ \}
\[
\begin{align*}
  (h \times 1)(\odot \rightarrow (\{\varepsilon\}, \varepsilon)) = (\oplus \rightarrow (h(\{\varepsilon\}), \varepsilon)) \\
  \leq \quad \{ \text{law 6} \} \\
  (h \times 1)((X, s) \odot a) = (h \times 1)((X, s)) \oplus a \\
  = \quad \{ \text{def $\odot$, $\times$} \} \\
  (h \times 1)((X \cup \{sa\}, sa) = (h(X), s) \oplus a
\end{align*}
\]

for all X, s and a

for all X, s and a
\[
\text{def } x \\
(h(X \cup \{sa\}), sa) = (h(X), s) \oplus a \\
\text{for all } X, s \text{ and } a \\
\]
\[
\text{h is set-homomorphism} \\
(h(X) \cup h(\{sa\}), sa) = (h(X), s) \oplus a \\
\text{for all } X, s \text{ and } a \\
\]
\[
\text{def } h \\
(h(X) \cup g(sa), sa) = (h(X), s) \oplus a \\
\text{for all } X, s \text{ and } a \\
\]
\[
\text{instantiation} \\
(U \cup g(sa), sa) = (U, s) \oplus a \\
\text{for all } U, s \text{ and } a \\
\]

So the claim is proved if we choose the operator \( \oplus \) defined by:

\[
(U, s) \oplus a = (U \cup g(sa), sa) \\
\]

We conclude this fusion step by noting that

\[
(h \times I)(\cdot \rightarrow ((\varepsilon), \varepsilon)) \\
= \quad \{ (4.1.1) \} \\
(h \times I)(\text{pref } \triangle I) \\
= \quad \{ \times - \triangle \text{ fusion, law 9} \} \\
\]

\[
h \text{ pref } \triangle I \\
\]

Consequently, the specification is transformed into \( \pi_1 \) (h pref \( \triangle \) I). Since (h pref \( \triangle \) I) has a left-reduce form, claim above, the imperative program can be obtained by instantiating Lemma 1.

\[
\pi_1 (h \text{ pref } \triangle I)(S) \\
O, s := g(\varepsilon), \varepsilon; r := S \\\n; \textbf{do } r \neq \varepsilon \\\n\quad \rightarrow a := \text{first}(r); r := \text{tail}(r) \\\n; O, s := O \cup g(sa), sa \\\n\textbf{od} \\
\]

**Fact:**

\[
(h \text{ pref } \triangle I) = (\oplus \rightarrow (g(\varepsilon), \varepsilon)) \\
(O, s) \oplus a = (O \cup g(sa), sa) \\
\]

---

5. \{ Generalization \}

In 4, a linear program is achieved with a nice global structure. Next we concentrate on
local computations. Only the application on \( g \) needs further attention, hence, our next goal is an efficient program for \( g(sa) \). By the definition of \( g \) and 3.1.7b, we have that

\[
g(sa) = (\text{suff}(sa) \cap P) \times \{sa\}
\]

Clearly \( \in P \) operations are needed to compute \( g(sa) \). However, \( P \) is a given set and generally the operation \( \in P \) will be time consuming. To obtain some flexibility in the implementation of the \( P_e \) predicate, we introduce a set \( Q \) of candidates for \( P_e \), i.e. \( P \subseteq Q \). In the sequel we will put some suitable requirements on \( Q \). The introduction of \( Q \) modifies the specification slightly: instead of the function \( g \), the function \( \forall (P_e \triangleleft Q_e \triangleleft \text{suff} \triangleleft I) \) is involved in it. But, both functions coincide because

\[
P_e \triangleleft = \{ P_e \Rightarrow Q_e, \text{law 5} \}
\]

Therefore we do not introduce a new function name but adapt the definition of \( g \). This generalization transforms the expression in 4 to

\[
\begin{align*}
\text{P} \subseteq \text{Q} \\
\pi_1 (h \text{ pref } \triangleleft I)(S) & \quad \text{O, } s := g(e), e; r := S \\
; \text{do } r \neq e & \quad \rightarrow a := \text{first}(r); r := \text{tail}(r) \\
; \text{O, } s := O \cup g(sa), sa & \quad \text{od}
\end{align*}
\]

Fact:

\[
(h \text{ pref } \triangleleft I) = (\oplus \rightarrow (g(e), e))
\]

\[
(O, s) \oplus a = (O \cup g(sa), sa)
\]

\[
\text{Invariant:} \quad O = (h \text{ pref})(y) \land s = I(y),
\]

where \( yr = S \)

6. \{ Refinement, typical Aho-Corasick detail \}

We continue with examining \( g(sa) \), now defined as \( P_e \triangleleft (\text{suff}(sa) \cap Q) \times \{sa\} \). The set \( \text{suff}(sa) \cap Q \) has an interesting structure and we intend to exploit this in a further step to refinement. To be more precise: the suffix ordering \( \leq \), is a linear ordering on \( \text{suff}(u) \cap Q \), for each \( u \in V^* \). If this set is nonempty, then its maximum \( q \), denoted by \( \text{q} = \triangleleft (\text{suff}(u) \cap Q) \), has the property that

\[
\text{suff}(u) \cap Q = \text{suff}(q) \cap Q
\]

(4.1.2)

It even holds that, Lemma 4, \( q \) is the \( \leq_s \)-least solution of the equation (in \( x \in V^* \))

\[
x : \text{suff}(u) \cap Q = \text{suff}(x) \cap Q.
\]

(4.1.2) implies that computations over nonempty \( \text{suff}(u) \cap Q \) can be reduced to computations over its subset \( \text{suff}(k(u)) \cap Q \), where \( k = \triangleleft_{Q_e \triangleleft \text{suff}} \). It is
this reduction that we hinted at at the beginning of this step and which is the quintessence of the Aho-Corasick algorithms. To guarantee that such a reduction is valid generally, we will require that \( \varepsilon \in Q \).

Functionally, this reduction is expressed by a factorization of \( g: g = f \ (k \triangle I) \). By law 9, we have that \( f = \forall (P_{\varepsilon < Q_{\varepsilon < \text{suff}}} \times I) \) and we adapt our specification in that sense.

\[ P \subseteq Q \land \varepsilon \in Q \]

\[ \pi_1 (h \text{ pref } \triangle I)(S) \]

\[ \begin{align*}
&O, s := g(\varepsilon), \varepsilon; r := S \\
&\text{do } r \neq \varepsilon \\
&\quad a := \text{first}(r); r := \text{tail}(r) \\
&\quad q := k(sa) \\
&\quad O_1 := f(q, sa) \\
&\quad O, s := O \cup O_1, sa \\
&\text{od}
\end{align*} \]

**Fact:**
\[ (h \text{ pref } \triangle I) = (\oplus \rightarrow (g(\varepsilon), \varepsilon)) \]
\[ (O, s) \oplus a = (O \cup f(k(sa), sa), sa) \]

\[ h = \cup/ g^* \]
\[ g = f (k \triangle I) \]
\[ f = \forall (P_{\varepsilon \triangle Q_{\varepsilon \triangle \text{suff}}} \times I) \]
\[ k = \uparrow_s/ Q_{\varepsilon \triangle \text{suff}} \]

7. \{ tupling \}

The refinement in step 6 modifies our goal from an efficient program for \( g(sa) \) into an efficient program for \( k(sa) \). The same strategy as before is applied: explore the existence of a left-reduce form for \( k \). We will prove that \( k(sa) = k(k(s)a) \), if we put an additional requirement on \( Q \). Under the assumption that \( Q \) satisfies this requirement, the specification is extended with \( k \).

From the definition of \( k \), we infer that \( k \) has a left-reduce form if \( Q_{\varepsilon \triangle \text{suff}} \) has. In the following calculation we force a left-reduce form for \( Q_{\varepsilon \triangle \text{suff}} \) by imposing a restriction on \( Q \).

\[ \text{suff}(sa) \cap Q \]
\[ = \{ \text{ def suff } \} \]
\[ (\text{suff}(s).a \cup \{\varepsilon\}) \cap Q \]
\[ = \{ \text{ distributivity } \} \]
\[ (\text{suff}(s).a \cap Q) \cup (\{\varepsilon\} \cap Q) \]
\[ = \{ \varepsilon \in Q \} \]
Now we require that $Q$ is prefix closed. Then we have by (3.1.5) that $\text{suff}(s).a \cap Q = \text{suff}(s).a \cap Q.a \cap Q$ and we can continue our calculation

\begin{equation}
(suff(s).a \cap Q) \cup \{\varepsilon\}
\end{equation}

(4.1.3)

\[
= \{ \bullet Q \text{ is prefix closed} \}
\]

\[
(suff(s).a \cap Q.a \cap Q) \cup \{\varepsilon\}
\]

\[
= \{ (3.1.1) \}
\]

\[
((suff(s) \cap Q).a \cap Q) \cup \{\varepsilon\}
\]

\[
= \{ (4.1.2), q = k(s) \}
\]

\[
((suff(q) \cap Q).a \cap Q) \cup \{\varepsilon\}
\]

\[
= \{ \text{calculation backward} \}
\]

\[
(suff(q).a \cap Q) \cup \{\varepsilon\}
\]

\[
= \{ \text{calculation backward} \}
\]

\[
suff(qa) \cap Q
\]

Consequently, if $Q$ is prefix closed, then

\[
k(sa) = k(qa) = \{ q = k(s) \} \ k(k(s)a)
\]

From now on we will assume that $Q$ is prefix closed. This may improve the computation of $k(sa)$ if we tuple with $k$. It is not difficult to prove that

\[
(h \text{ pref } \triangle I \triangle k) = (\otimes \Rightarrow ((g(\varepsilon), \varepsilon, \varepsilon))
\]

where $(O, s, q) \otimes a = (O \cup f(k(qa), sa), sa, k(qa))$. The transformed specification is then given by $\pi_1 (h \text{ pref } \triangle I \triangle k)$. Due to its left-reduce form its imperative counterpart is yielded by Lemma 1.

\textbf{Fact:}

\[
(h \text{ pref } \triangle I \triangle k) = (\otimes \Rightarrow (g(\varepsilon), \varepsilon, \varepsilon))
\]

\[
(O, s, q) \otimes a = (O \cup f(q_1, sa), sa, q_1)
\]

where $q_1 = k(qa)$

\textbf{Invariant:}

\[
O = (h \text{ pref})(y) \land s = I(y) \land q = k(y)
\]

where $yr = S$

with
\[ h = \cup \, g^* \]
\[ g = f \, (k \, \Delta \, I) \]
\[ f = \emptyset \left( P_e < Q_e < \text{suff} \times I \right) \]
\[ k = \uparrow_s / Q_e < \text{suff} \]

8. \{ Computation for \( k(qa) \) \}

The algorithms in 7 leave the computation of \( k(qa) \) open. Given \( q \) and \( a \), \( k(qa) \) can be computed via the definition of \( k \), i.e. by finding the greatest suffix of \( qa \) which is in \( Q \). In this section, we will define \( k(qa) \) with cheaper operators by exploiting the properties of \( k \).

\[
k(qa)
= \{ \text{def } k, (4.1.4) \}
\]
\[
\uparrow_s / ((\text{suff}(q).a \cap Q) \cup \{ \varepsilon \})
= \{ \uparrow_s / \text{is set homomorphism} \}
\]
\[
\uparrow_s / (\text{suff}(q).a \cap Q) \uparrow_s \varepsilon
= \{ (3.1.4) \}
\]
\[
\uparrow_s / ((\text{suff}(q) \cap Q/a).a) \uparrow_s \varepsilon
= \{ \text{case } (\text{suff}(q) \cap Q/a) \neq \emptyset \}
\]
\[
(\uparrow_s / (\text{suff}(q) \cap Q/a))a
\]
\[
(4.1.5)
= \{ \text{case } (\text{suff}(q) \cap Q/a) = \emptyset \}
\]
\[
\varepsilon
\]

The case analysis will be circumvented by the introduction a fictitious element \( \bot_Q \) of \( Q \) such that \( \bot_Q \) is a unit of \( \uparrow_s \) and a left zero of \( \uparrow \) (string concatenation), i.e. \( \bot_Q \uparrow_s z = z \) and \( \bot_Q \uparrow z = \bot_Q \). By definition of \( \uparrow_s / \) it holds that \( \uparrow_s / \emptyset = \bot_Q \). Then \( k(qa) \) can be expressed by

\[ k(qa) = d(q \ominus a, \varepsilon) \]

where \( x \ominus a = \uparrow_s / (\text{suff}(x) \cap Q/a) \) and \( d(x, a) = xa \uparrow_s \varepsilon \)

For a computation of \( k(qa) \) via this definition we have to do a simple comparison after finding the greatest suffix of \( q \) which is in \( Q/a \). To see that this computation is more efficient than the previous one, we note that \( Q/a \subseteq Q : \)

\[
Q/a
\subseteq \{ (3.1.3a) \}
\]
The specification is adapted to these new definitions

\[ \text{P \subseteq Q \land Q \text{ is prefix closed}} \]

\[ \bot_Q \text{ is a fictitious element of } Q \text{ st } \bot_Q \text{ is a unit for } \uparrow_s \text{ and a left zero for } \parallel \]

\[
\pi_1 \left( h \operatorname{pref} \triangle I \triangle k \right) (S) = \begin{cases} O, s, q := g(\varepsilon), \varepsilon, \varepsilon; r := S & \text{do } r \neq \varepsilon \\ \rightarrow a := \text{first}(r); r := \text{tail}(r) \\ q' := q \triangle a \\ q := d(q', a) \\ O_1 := f(q, sa) \\ O, s := O \cup O_1, sa \end{cases}
\]

\[ \text{Fact:} \]

\[ (h \operatorname{pref} \triangle I \triangle k) = (\otimes \rightarrow (g(\varepsilon), \varepsilon, \varepsilon)) \]

\[ (O, s, q) \otimes a = (O \cup f(q_1, sa), sa, q_1) \]

where \( q_1 = k(qa) = d(q \triangle a, a) \)

with \( q \triangle a = \uparrow_s / (Q/a)_{Q \subseteq \text{suff}(q)} \)

\[ h = \cup / g^* \]

\[ g = f (k \triangle I) \]

\[ f = \varnothing (P_{\varepsilon} \triangle Q_{\varepsilon} \triangle \text{suff} \times I) \]

\[ k = \uparrow_s / Q_{\varepsilon} \triangle \text{suff} \]

\[ d(x,a) = xa \uparrow_s \varepsilon \]

It turned out that this completes the derivation on the functional level, because the Aho­Corasick algorithms suppose some precomputations. The global structure of AC-OPT and AC-FAIL is given in scheme 7 and 8 respectively when \( Q \) is instantiated with \( \text{prefs}(P) \), the least prefix closed set containing \( P \). In the next section we continue the derivation with implementation dependent steps.

4.2 Derivation on the imperative level.

In this section, we will add the imperative characteristic that functions over a finite domain can be implemented efficiently via arrays and, consequently, that retrieving such a function is an \( O(1) \) operation. The starting points for the remaining derivations are the imperative programs of section 4.1 where \( Q \) is instantiated with \( \text{prefs}(P) \). We claim that AC-OPT and
AC-FAIL follow from scheme 7 and scheme 8 respectively. In order to prove these claims efficiently, we consider their common unimplemented statements firstly. Obviously, for \( q \in Q, s \in V^* \) and \( a \in V \), the statements

\[
O := g(\varepsilon) \quad \text{and} \quad O_1 := f(q, sa)
\]

where

\[
g(\varepsilon) = (\{\varepsilon\} \cap P) \times \{\varepsilon\}
\]

\[
f(q, sa) = (\text{suff}(q) \cap P) \times \{sa\}
\]

are easily implemented, if we assume that

\[
\text{for each} \ q \in Q: \text{suff}(q) \cap P \text{ is precomputed}
\]  \hspace{1cm} (4.2.1)

We will not bother about these statements furthermore, assumption (4.2.1) is efficacious.

**Claim 1 :** AC-OPT follows from scheme 7

**Proof.**

Starting from scheme 7 and assumption (4.2.1), the only unimplemented statement is \( q := k(qa) \) where \( k(qa) = \uparrow_s (\text{suff}(qa) \cap Q) \) with \( q \in Q \) and \( a \in V \). We decide to precompute this value too. Hence we require precomputations for

\[
\uparrow_s (\text{suff}(qa) \cap Q)
\]

\[
\text{suff}(q) \cap P
\]

for each \( q \in Q \) and \( a \in V \).

The next step is: choosing a representation for these precomputations, because the way in which this information will be available might be diverse. Heading for AC-OPT, we decide for a representation via arrays, i.e.

\[
\delta : Q \times V \to Q \text{ is such that } \delta(q,a) = \uparrow_s (\text{suff}(qa) \cap Q)
\]

\[
\text{output : } Q \to \mathcal{P}(V^*) \text{ is such that } \text{output}(q) = \text{suff}(q) \cap P
\]

With this representation choice for the precomputation, AC-OPT, the adapted algorithm 1 of [1], is obtained from scheme 7:

\[
O, s, q := (\{\varepsilon\} \cap P) \times \{\varepsilon\}, \varepsilon, \varepsilon; r := S
\]

; \ \textbf{do } r \neq \varepsilon

\[
\rightarrow a := \text{first}(r); r := \text{tail}(r)
\]

; \ q := \delta(q, a)

; \ O_1 := \text{output}(q) \times \{sa\}

; \ O, s := O \cup O_1, sa

\textbf{od}

\[\square\]

**Remarks**

1. (4.2.2) is a linear algorithm while the specification was a quadratic one.
2. In [1], $\delta$ is considered as the transition function on a deterministic finite automaton.

3. It is not quite true that (4.2.2) is AC-OPT, there is one small difference. Aho-Corasick algorithms only produce output for non-empty prefixes of $S$, or explained in the program variables of (4.2.2), in the Aho-Corasick algorithms the variable $0$ is initialized with $\emptyset$. The empty pattern, i.e. $\varepsilon \in P$, is useless for practical applications, in theory there is no need to exclude it. (Is $\varepsilon \not\in P$ an implicit assumption of Aho-Corasick?) End of Remarks

Claim 2 : AC-FAIL follows from scheme 8.

Proof. Starting from scheme 8 and assumption (4.2.1), the unimplemented statements are

$$q' := q \Theta a \quad \text{and} \quad q := d(q', a)$$

where

$$q \Theta a = \uparrow_s/(Q/a)_{<\varepsilon} \text{suff}(q)$$
$$d(q', a) = q'a \uparrow_s \varepsilon$$

with $q \in Q$, $q' \in Q \cup \{\bot_Q\}$ and $a \in V$. Taking into account the requirements on $\bot_Q$, the statement $d(q', a)$ can be implemented by

$$d(q', a) := \text{if } q' = \bot_Q \rightarrow \varepsilon \; \text{[} q' \neq \bot_Q \; q'a \text{ ] fi} \quad \text{(4.2.3)}$$

More attention must be paid to the implementation of $\Theta$. $q \Theta a$ is defined as the $\leq_s$ greatest suffix of $q$ satisfying the predicate $(Q/a)_{<\varepsilon}$. Moreover, suff($q$) is linearly ordered w.r.t. $\leq_s$. Trivially $q \Theta a$ can be implemented by linear search over suff($q$). However, we can do better: the search space may be restricted to suff($q$) $\cap$ $Q$, because $Q/a \subseteq Q$, (4.1.6). $\Theta$ can be implemented by instantiating the Linear Search Theorem in Lemma 3. To that end, we mention that: suff($q$) $\cap$ $Q$ is the domain of discourse, $\leq_s$ is the linear ordering, $q$ is the maximal and $\varepsilon$ the minimal element, $F(q'') = \uparrow_s/(\text{suff(tail(q'))}) \cap Q$ is the predecessor of (non-empty) $q'$, $q' \in Q / a \equiv q'a \in Q$, $\bot_Q$ is a unit for $\uparrow_s$. An implementation for the statement $q' := q \Theta a$ is then given by

$$q' := q$$
$$; \text{do } q' \neq \varepsilon \wedge q'a \not\in Q$$
$$\rightarrow q' := F(q')$$
$$\text{od}$$
$$; \text{if } q'a \in Q \rightarrow \text{skip}$$
$$\text{[} q'a \not\in Q \rightarrow q' := \bot_Q \text{ ] fi}$$

Substituting this result in scheme 8 yields

$\bot_Q$ is a fictitious element of $Q$ st $\bot_Q$ is a unit for $\uparrow_s$ and a left zero for $\bot$
\begin{align*}
O, s, q &:= g(\epsilon), \epsilon, \epsilon; r := S \\
; \textbf{do} \; r \neq \epsilon & \\
\rightarrow a := \text{first}(r); r := \text{tail}(r) & \\
; \textbf{do} \; q' \neq \epsilon \land q'a \notin Q & \\
\rightarrow q' := F(q') & \\
\textbf{od} & \\
; \textbf{if} \; q'a \in Q & \rightarrow \text{skip} \\
\quad \textbf{if} \; q'a \notin Q & \rightarrow q' := \bot_Q & \\
\textbf{fi} & \\
; q := d(q', a) & \\
; O_1 := f(q, sa) & \\
; O, s := O \cup O_1, sa & \\
\textbf{od} & \\
\textbf{Invariant:} & \\
O = h \text{ pref}(y) \land s = I(y) \land q = k(y) & \\
\text{where } yr = S & \\
\end{align*}

We decide to precompute \( F(q') \) for \( q' \in Q - \{\epsilon\} \). Hence we require precomputations for
\begin{align*}
\uparrow_s/( \text{suff}(\text{tail}(q')) \cap Q) \\
\text{suff}(q) \cap P
\end{align*}
for each \( q \in Q \) and \( q' \in Q - \{\epsilon\} \).

The next step is: choosing a representation for these precomputations. Heading for AC-FAIL, we decide for a representation via arrays, i.e.
\begin{align*}
f : (Q - \{\epsilon\}) \rightarrow Q \quad \text{such that } f(q') = \uparrow_s/( \text{suff}(\text{tail}(q')) \cap Q) \\
\text{output} : Q \rightarrow \mathcal{P}(V^\#) \quad \text{such that } \text{output}(q) = \text{suff}(q) \cap P
\end{align*}

With these representation choice for the precomputation, the following program fragment is obtained from (4.2.4)

\begin{align*}
\bot_Q \text{ is a fictitious element of } Q \quad \text{st } \bot_Q \text{ is a unit for } \uparrow_s \text{ and a left zero for } \mp & \\
O, s, q &:= g(\epsilon), \epsilon, \epsilon; r := S \\
; \textbf{do} \; r \neq \epsilon & \\
\rightarrow a := \text{first}(r); r := \text{tail}(r) & \\
\end{align*}
In main lines, this is the algorithm AC-FAIL. To see this we take some additional transformation steps. First we contract the alternative statement and the assignment to $q$, see Lemma 5.

$\bot_Q$ is a fictitious element of $Q$ st $\bot_Q$ is a unit for $\uparrow_s$ and a left zero for $\uparrow$.

Next we substitute (4.2.3) and remove superfluous statements/variables. Consequently, the requirement on $\bot_Q$ can be released.

Following the lines of Aho-Corasick, the several guards and some corresponding actions are integrated in the definition of a function $\delta: Q \times V \rightarrow (Q \cup \{\bot\})$ such that
\[
\delta(q, a) = \begin{cases} 
q a & \text{if } qa \in Q \\
\varepsilon & \text{if } q = \varepsilon \land qa \not\in Q \\
\bot & \text{otherwise}
\end{cases}
\]

Consequently, \((\delta(q, a) = \bot) \equiv (q \neq \varepsilon \land qa \not\in Q)\). Adding the realization of \(\delta\) to the precomputation, we arrive at AC-FAIL (algorithm 1 in [1]).

\[
\begin{array}{c}
O, s, q := g(\varepsilon), \varepsilon, \varepsilon; r := S \\
\text{do } r \neq \varepsilon \\
\quad \rightarrow a := \text{first}(r); r := \text{tail}(r) \\
\quad ; \text{do } \delta(q, a) = \bot \rightarrow q := f(q) \text{ od} \\
\quad ; q := \delta(q, a) \\
\quad ; O_1 := \text{output}(q) \times \{sa\} \\
\quad ; O, s := O \cup Q_1, sa \\
\text{od}
\end{array}
\]

\(\square\)

Remark The function \(f\) which occurs in these last programs is the so-called failure function. We saw that this failure function follows quite naturally from Linear Search.

End of Remark

5. Conclusions

We exemplified that in one program derivation there can be synergy of two programming methods: BMF and imperative programming. Each method has its own strong points, e.g. - BMF exploits the algebraic structure of the problem and reasons equationally about programs. - Imperative programming permits efficient encodings of operations and is suited for stepwise refinement. We combined the advantages of both methods in our derivation.

The main structure of the derivation is a sequence of correctness-preserving transformation steps in which an initial correct but inefficient solution is transformed into a more efficient one. For this kind of derivation BMF is well-suited. Due to the application of a set of standard conversions at each step an equivalent imperative program is obtained. When the possibilities for improvement offered by BMF were exhausted, the derivation could be continued in imperative style. The advantages of this combined approach are well illustrated by step 4 (section 4.1) and claim 2 (section 4.2) respectively. Step 4 concerns the fusion of two computations with similar structure. In BMF, this is a simple application of a fusion law, whereas in imperative programming this involves a rather complex merging of two loops. On the other hand, the derivation steps following claim 2 are more easily carried out using imperative programming techniques such as linear search and tabulation.

Although it is risky to try to draw general conclusions from a specific example, the following
principles seem to emerge:
- try to exploit the algebraic structure of the problem as long as possible.
- postpone implementation dependent steps as long as possible.
- if a concrete solution of the problem is available, abstracting from implementation details may yield a suitable algebraic modelling of the problem.
- the BMF derivation could have been carried out to a further point if the method was extended with tools to reason about implementation details (a possibility which is suggested in [7]). A first start for such an extension can be found in [5].
- imperative programming can profit from the BMF laws in correctness proofs for program transformations.

With respect to the Aho-Corasick derivation, we may conclude that
- it provides insight in the structure of algorithms.
- relations between algorithms are exposed. Consider the solutions of AC-OPT and AC-FAIL as given in scheme 7 and 8 respectively. Except for their behaviour on the function k, these programs are the same. In AC-OPT, k(q,a) is computed for each q ∈ prefs(P) and a ∈ V before the text S is scanned, while in AC-FAIL k(q,a) is computed “on the fly” using the failure function f (hence only k computations are made for (q,a) combinations which arise while scanning S). - The failure function which occurs in AC-FAIL follows quite naturally from applying the standard programming technique of linear search.

Finally, we like to remark that
- In our method, a derivation is broken up in two parts: firstly, the global structure of the solution is achieved and afterwards, its implementation is managed. Since there are various alternatives for the decisions in both parts, it is conceivable that new solutions will be suggested. For instance, a new solution to the pattern matching problem results from this Aho-Corasick derivation. In AC-OPT and AC-FAIL, two extreme implementation techniques are applied on the function k. Every programmer knows that there is a way in between these two extremes by using k as a memo function. Or expressed imperatively, while scanning S, save all previously computed k values in a table and execute “on the fly” computations only if table lookup fails.
- The Knuth-Morris-Pratt (KMP) algorithm can also be derived from scheme 8. It is omitted here because of its length: a state-space transformation is needed to translate the imperative environment over lists into an environment over naturals. In the KMP derivation in [6], this domain transformation is just a simple bijection.
- One could object that the derivation given in section 4 is incomplete because precomputations are not elaborated. That is true, indeed, but we prefer to consider the precomputations as independent problems because their elaboration obscures the global structure of the solution.
- The functional derivation in section 4 agrees with the tendency in BMF to develop solutions for problem classes (the first five steps can be made for each solution of this pattern matching problem which scans the text from left to right). This agreement is not accidentally, our approach arose during the research project “A Taxonomy of Scanner Algorithms”.
The intention of this project is to clarify the relationship between solutions to the scanning problem by giving a generic derivation, see [19].

Acknowledgements
Much of the material presented above arose from the imperative approach of the scanner-club of Eindhoven University in which Bruce Watson and Gerard Zwaan were very active. I am grateful to Jaap van der Woude for his support on the transformational side of the material. The valuable criticism of Frans Kruseman Aretz and Kees Hemerik improved the presentation. In particular I would like to thank Kees Hemerik for encouraging me with a list of quotations.
References


28


APPENDIX A

Summary of definitions and Laws in BMF

Let \((\alpha, \oplus, 1_\oplus)\) be a monoid, i.e. \(\alpha\) is a set and \(\oplus\) is an associative operator on \(\alpha\) with unit \(1_\oplus\). For monoids \((\alpha, \oplus, 1_\oplus)\) and \((\beta, \otimes, 1_\otimes)\), \(h : \alpha \to \beta\) is a monoid (homo)morphism if and only if

\[
\begin{align*}
    h(1_\oplus) &= 1_\otimes \\
    h(x \oplus y) &= h(x) \otimes h(y)
\end{align*}
\]

for all \(x, y \in \alpha\).

Such monoid morphism will be denoted by \(h : (\alpha, \oplus, 1_\oplus) \to (\beta, \otimes, 1_\otimes)\). It is well-known that composition (denoted by \(\circ\)) preserves homomorphisms.

Let \(V\) be a set. The set \(V^*\) of finite lists over \(V\) is the set part of the free monoid generated by \(V\). I.e. \((V^*, \|, [\ ]\)) is a monoid, and \([\ ] : V \to V^*\) is an embedding such that for every monoid \((M, \oplus, 1_M)\), and for every \(f : V \to M\), there is a unique monoid morphism \(h : (V^*, \|, [\ ]) \to (M, \oplus, 1_M)\) such that

\[f = h \circ [\ ]\]

\(h\) is called the extension of \(f\) to \(V^*\) and is denoted by \(h = ([ f ])\), \(h\) is the "join-list catamorphism" of \(f\). Note that \(h\) is completely determined by its behaviour on singletons. This unique extension property (UEP) plays an important role in BMF.

If we had restricted ourselves to the class of monoids with commutative and idempotent operator, then we could have defined the "free set" generated by \(V\) as: \((\mathcal{P}(V), \cup, \emptyset)\) is a monoid, and \(\{\} : V \to \mathcal{P}(V)\) is an embedding such that for every monoid \((M, \oplus, 1_M)\) with commutative and idempotent operator \(\oplus\), and for every \(f : V \to M\), there is a unique monoid morphism \(h : (\mathcal{P}(V), \cup, \emptyset) \to (M, \oplus, 1_M)\) such that

\[f = h \circ \{\}\]

Here \(\mathcal{P}(V)\) is the set of finite sets over \(V\).

The monoid homomorphisms involved in this case will be called set-homomorphisms.

In BMF, the language of functions is used and mostly \(\circ\) is omitted.

The language is extended with some (standard) functions, such as

- **map**
  
  Let \(f : \alpha \to \beta\), then
  \(f^* : \alpha^* \to \beta^*\) is the unique extension of \([\ ] : f : \alpha \to \beta^*\)

- **reduce**

  Let \(\odot\) be an associative, operator on \(\alpha\) with unit \(1_\odot\),
  then \(\odot/ : \alpha^* \to \alpha\) is the unique extension of \(I : \alpha \to \alpha\)

- **filter**

  Let \(p : \alpha \to \mathbb{B}\), then
  \(p< \alpha^* \to \alpha^*\) is the unique extension of \(f_p\)
where \( f_p(a) = (p(a) \to [a], [ ]) \)

Note that map, reduce and filter can also be defined in a set-homomorphic variant, if we require that \( \odot \) is commutative and idempotent and \( f_p(a) = (p(a) \to \{a\}, \emptyset ) \)

**Left reduce**

Let \( \oplus : \beta \times \alpha \to \beta \) and \( e \in \beta \)

Then \((\oplus \to e) : \alpha^* \to \beta \) is such that

\[
\begin{align*}
(\oplus \to e)([ ]) &= e \\
(\oplus \to e)(x \vdash [a]) &= (\oplus \to e)(x) \oplus a
\end{align*}
\]

or, alternatively

\[
\begin{align*}
(\oplus \to e)([ ]) &= e \\
(\oplus \to e)([a] \vdash x) &= (\oplus \to (e \oplus a))(x)
\end{align*}
\]

**Remark**

Left reduce can also be defined via UEP, since left reduce is a "snoc-list catamorphism", i.e. \((\oplus \to e)\) satisfies the scheme

\[
\begin{align*}
f([ ]) &= e \\
f(x \vdash [a]) &= f(x) \oplus a
\end{align*}
\]

**End of Remark**

Some laws, algebraic transformations, are

0. **Uniqueness property for morphisms on the free structure:**

\[ h : (V^*, \vdash, [ ] ) \to (\beta, \oplus, 1_{\oplus}) \text{ is a monoid morphism } \equiv \exists f : h = \oplus/ f^* \]

(Note that \( \oplus \) is an associative operator with unit.)

\[ h : (P(V), \cup, \emptyset ) \to (\beta, \oplus, 1_{\oplus}) \text{ is a set-homomorphism } \equiv \exists f : h = \oplus/ f^* \]

(Note that \( \oplus \) is an associative, commutative and idempotent operator with unit.)

1. Each "join-list" catamorphism can be written as a left-reduce

If \( h = \oplus/ f^* \), then \( h = (\oplus \to 1_{\oplus}) \), where

\[ u \oplus a = u \oplus f(a) \]

2. Fusion for "join-list" catamorphisms (promotion):

If \( h : (\alpha, \odot, 1_{\odot}) \to (\beta, \odot, 1_{\odot}) \) is a monoid morphism, then

\[ h \odot/ = \odot/ h^* \]

In particular,

If \( h : (\alpha^*, \vdash, [ ]) \to (\beta, \odot, 1_{\odot}) \) is a monoid morphism, i.e. \( h = \odot/ f^* \), then

\[ h \vdash/ = \odot/ h^* \]

If \( h : (\mathcal{P}(\alpha), \cup, \emptyset ) \to (\beta, \odot, 1_{\odot}) \) is a set-homomorphism, i.e \( h = \odot/ f^* \), then
\[ h \cup/ = \otimes/ h^* \]

3. **Map distributivity:**
   \[(f \cdot g)^* = f^* \otimes g^*\]

4. **Map-filter rule:**
   \[ p \land f^* = f^* (p \land f) \land \]

5. **Filter commutativity distributivity:**
   Let \( p \) and \( q \) be total predicates, then
   \[(p \land q) \land = p \land q \land = q \land p \land \]

6. **Fusion for left-reduces (Formal differentiation):**
   \[ h (\oplus \rightarrow e) = (\otimes \rightarrow h(e)) \iff \forall (y, a :: h(y \oplus a) = h(y) \otimes a) \]

*Remark* Law 6 could have been expressed in a way similar to that of law 2, because the antecedent states that \( h \) is a suitable homomorphism. Formulating fusion as we did in law 6 may illustrate a way in which constructions come into the method: If \( h \) and \( \oplus \) are given, then the antecedent of law 6 is a system of equations in \( \otimes \). Each solution to this system may yield a definition for \( \otimes \), i.e. \( \otimes \) is found by construction.

*End of Remark*

We also use some more common functions and laws

**Definitions**
\( \pi_1, \pi_2 \) projection function on the 1st and 2nd coordinate respectively

\[ \cdot \]
Let \( f : \alpha \rightarrow \beta \) and \( g : \alpha \rightarrow \gamma \),
then \((f \cdot g) : \alpha \rightarrow (\beta \times \gamma)\) is defined by
\[(f \cdot g)(a) = (f(a), g(a)) \]

\[ \times \]
Let \( f : \alpha \rightarrow \beta \) and \( g : \gamma \rightarrow \delta \), then
\[ f \times g = (f \pi_1) \cdot (g \pi_2) \]

We assume that \( \times \) associates to the left

**Laws**

7. **Computation rule for split:**
   \[
   \begin{align*}
   \pi_1(f \cdot g) &= f \\
   \pi_2(f \cdot g) &= g
   \end{align*}
   \]

8. **Computation rule for product::**
\[ f \pi_1 = \pi_1(f \times g) \quad g \pi_2 = \pi_2(f \times g) \]

9. \( \times \cdot \Delta \) fusion:
\[ (h \times g)(f \cdot \Delta k) = h \cdot \Delta g \cdot k \]

For the specific problem of pattern matching, we have some additional definitions and laws. For the proofs of the laws we refer to [12]

Definitions

\( \emptyset \)

The function \( \emptyset : \mathcal{P}(\alpha) \times \beta \rightarrow \mathcal{P}(\alpha \times \beta) \) is defined by
\[ \emptyset(X, b) = X \times \{ b \} \]
or equivalently
\[ \emptyset(\emptyset, b) = \emptyset \]
\[ \emptyset(\{ a \}, b) = \{(a, b)\} \]
\[ \emptyset(X \cup Y, b) = \emptyset(X,b) \cup \emptyset(Y,b) \]

\( \text{pref} \)

pref : \( \alpha^* \rightarrow \mathcal{P}(\alpha^*) \) such that
\[ \text{pref}([ ]) = \{ [ ] \} \]
\[ \text{pref}(x \cdot \Delta [ a]) = \text{pref}(x) \cup \{ x \cdot \Delta [ a] \} \]

\( \text{suff} \)

suff : \( \alpha^* \rightarrow \mathcal{P}(\alpha^*) \) such that
\[ \text{suff}([ ]) = \{ [ ] \} \]
\[ \text{suff}(x \cdot \Delta [ a]) = (\cdot \Delta [ a])^* \text{suff}(x) \cup \{ [ ] \} \]

\( \text{subs} \)

The function \( \text{subs} : \alpha^* \rightarrow \mathcal{P}(\alpha^* \times \alpha^*) \) is defined by
\[ \text{subs}(S) = \{ (v, 1 \cdot \Delta v) \in \alpha^* \times \alpha^* \mid \exists r \in \alpha^* : 1 \cdot \Delta v \cdot \Delta r = S \} \]

\( \text{prefs} \)

prefs : \( \mathcal{P}(\alpha^*) \rightarrow \mathcal{P}(\alpha^*) \) such that
\[ \text{prefs} = U/ \text{pref}^* \]

Laws

10 subs = U/ (\( \emptyset \) (suff \( \Delta \) I))\(^*\) pref

11 If \( p : \alpha \rightarrow \mathcal{B} \), then
\[ (p \pi_1) \cdot \Delta \emptyset = \emptyset (p \cdot \Delta \times I) \]
APPENDIX B

Standard conversions

Lemma 1. \{ implementation of a left-reduce \}
Let $\Theta : \beta \times \alpha \rightarrow \beta$ and $e \in \beta$. Then $S$ is a correct program fragment for the computation of $w = (\Theta \,
\longrightarrow
e) x$.

\[
S: \quad \begin{array}{l}
w := e; r := x \\
\text{do } r \neq [] \\
\quad \rightarrow a := \text{first}(r); r := \text{tail}(r) \\
\quad ; w := w \Theta a \\
\text{od}
\end{array}
\]

Invariant: $(\Theta \,
\longrightarrow
e) x = (\Theta \,
\longrightarrow
e) r \land r \in \text{tails}(x)$

Proof. Follows immediately from the definition of left-reduce \hfill \Box

Remark If $\Theta$ is associative, then the invariant can also be expressed as

$(\Theta \,
\longrightarrow
e) x = w \Theta (\Theta \,
\longrightarrow
e) r \land r \in \text{tails}(x)$

End of Remark

Lemma 2. \{ implementation of a set-homomorphism \}
Let $\oplus$ be an associative, commutative and idempotent operator with unit $1_{\oplus}$. Then $S$ is a correct program for the computation of $O = \oplus/ f^*(X)$.

\[
S: \quad \begin{array}{l}
O := 1_{\oplus}; R := X \\
\text{do } R \neq \emptyset \\
\quad \rightarrow a \in R; R := R - \{a\} \\
\quad O := O \oplus f(a) \\
\text{od}
\end{array}
\]

Invariant $\oplus/ f^*(X) = O \oplus \oplus/ f^*(R)$

Remark An alternative expression for the invariant is $O = \oplus/ f^*(X - R)$

End of Remark

Lemma 3. \{ (Bounded) Linear Search \}
Let $W \in \mathcal{P}(\alpha)$ be a finite set which is linearly ordered w.r.t. $\leq_r$ and $M$ is the maximum in $W$ w.r.t. $\leq_r$ and $m$ is the minimum. Let $f : (W - \{m\}) \rightarrow W$ be the predecessor function in $W$ and $B$ a predicate on $\alpha$.

Let $\perp$ be a fictitious element of $\alpha$ such that $\perp$ is a unit for $\uparrow_r$. Then $S$ is a correct implementation for $m' = \uparrow_r/ B \downarrow (W)$, where
S: \[ m' := M \]
\[ \; \text{do} \; m' \neq m \land \neg B(m') \longrightarrow m' := f(m') \]
\[ \quad \text{od} \]
\[ \{ B(m') \lor m' = m \} \]
\[ \; \text{if} \; B(m') \rightarrow \text{skip} \]
\[ \| \neg B(m') \rightarrow m' := \bot \]
\[ \sqcup \]

Lemma 4. {least solution}
Let \( u \in V^* \) and \( Q \subseteq V^* \) such that \( \text{suff}(u) \cap Q \neq \emptyset \). Then
\[ \uparrow_s / (\text{suff}(u) \cap Q) \text{ is the } \leq_s \text{ least solution of the equation (in } X \in V^*) \]
\[ \begin{equation}
(0) \quad X : \text{suff}(u) \cap Q = \text{suff}(X) \cap Q
\end{equation} \]
Proof.
We introduce \( q \) for \( \uparrow_s / (\text{suff}(u) \cap Q) \) and \( Z \) for the \( \leq_s \) least solution of \( (0) \).
Both quantities exist, because \( \text{suff}(u) \cap Q \neq \emptyset \), \( \text{suff}(u) \) is finite and linearly ordered w.r.t. \( \leq_s \) and \( u \) solves \( (0) \). It follows that \( Z \in \text{suff}(u) \). The lemma will be proved via the following claims
(a) \( q \) solves \( (0) \)
(b) If \( Y \) solves \( (0) \), then \( q \leq_s Y \)

Proof of claim (a)
\[ \text{suff}(u) \cap Q \]
\[ = \quad \{ u = rq \} \]
\[ (\text{suff}(r).q \cup \text{suff}(q)) \cap Q \]
\[ = \quad \{ \text{suff}(r).q \cap Q = \{q\} \} \]
\[ \text{suff}(q) \cap Q \]
Hence, \( q \) solves \( (0) \)

Proof of claim (b)
Let \( Y \) solve \( (0) \), then
\[ \text{suff}(u) \cap Q = \text{suff}(Y) \cap Q \]
\[ \Rightarrow \quad \{ q \in \text{suff}(u) \cap Q \} \]
\[ q \in \text{suff}(Y) \]
\[ \Rightarrow \quad q \leq_s Y \]
\[ \square \]
Lemma 5 \{ code transformation \}
\[ \text{if } B \rightarrow S_1 \quad \lnot B \rightarrow S_2 \text{ } \text{fi}; S_3 \]
\[ = \]
\[ \text{if } B \rightarrow S_1; S_3 \lnot B \rightarrow S_2; S_3 \text{ } \text{fi} \]

Property 3.1.4
For all \( X, Y \in \mathcal{P}(V^*) \) and \( z \in V^* \)
\[ X.z \cap Y = (X \cap Y/z).z \]
Proof.
\[ X.z \cap Y \]
\[ = \{ (3.1.7) \} \]
\[ Y_e \triangleleft o (\# z)^*(X) \]
\[ = \{ \text{law 4} \} \]
\[ (\# z)^* \circ (Y_e \circ (\# z)) \triangleleft (X) \]
\[ = \{ (3.1.3.c') \} \]
\[ (\# z)^* \circ (Y/z)_e \triangleleft (X) \]
\[ = \{ (3.1.7) \} \]
\[ (X \cap Y/z).z \]
\[ \square \]

As may be clear from the previous, (3.1.4) is only a rephrasing of (a special instance of) the map-filter law.

Property 3.1.5.b
Let \( L \in \mathcal{P}(V^*) \) be prefix closed. Then
\[ V^*.z \cap L \subseteq L.z \]
Proof
\[ V^*.z \cap L \]
\[ = \{ (3.1.4) \text{ with } X := V^* \} \]
\[ L/z . z \]
\[ \subseteq \{ (3.1.3.a, \text{monotonicity}) \} \]
\[ \text{prefs}(L).z \]
\[ = \{ (3.1.5.a) \} \]
\[ L.z \]
\[ \square \]
In this series appeared:

<table>
<thead>
<tr>
<th>Volume</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>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>
</tbody>
</table>
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

91/21 J. Coenen  
Assertional Data Reification Proofs: Survey and Perspective, p. 18.
W.-P. de Roever
J.Zwiers

91/22 G. Wolf  

91/23 K.M. van Hee  
Z and high level Petri nets, p. 16.
L.J. Somers
M. Voorhoeve

91/24 A.T.M. Aerts  
Formal semantics for BRM with examples, p. 25.
D. de Reus

91/25 P. Zhou  
A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, p. 52.
J. Hooman
R. Kuiper

91/26 P. de Bra  
The GOOD based hypertext reference model, p. 12.
G.J. Houben
J. Paredaens

91/27 F. de Boer  
Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
C. Palamidessi

91/28 F. de Boer  
A compositional proof system for dynamic process creation, p. 24.

91/29 H. Ten Eikelder  
Correctness of Acceptor Schemes for Regular Languages, p. 31.
R. van Geldrop

91/30 J.C.M. Baeten  
An Algebra for Process Creation, p. 29.
F.W. Vaandrager

91/31 H. ten Eikelder  
Some algorithms to decide the equivalence of recursive types, p. 26.

91/32 P. Struik  

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  
Asynchronous communication in process algebra, p. 20.
J.W. Klop
C. Palamidessi
A note on compositional refinement, p. 27.
A compositional semantics for fault tolerant real-time systems, p. 18.
Real space process algebra, p. 42.
Program derivation in acyclic graphs and related problems, p. 90.
Conservative fixpoint functions on a graph, p. 25.
Discrete time process algebra, p. 45.
The fine-structure of lambda calculus, p. 110.
On stepwise explicit substitution, p. 30.
Composition and decomposition in a CPN model, p. 55.
Demonic operators and monotype factors, p. 29.
Set theory and nominalisation, Part II, p. 22.
The total order assumption, p. 10.
A system at the cross-roads of functional and logic programming, p. 36.
Integrity checking in deductive databases; an exposition, p. 32.
Interval timed coloured Petri nets and their analysis, p. 20.
A unified approach to Type Theory through a refined lambda-calculus, p. 30.
Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
Are Types for Natural Language? P. 32.
Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.
92/22  R. Nederpelt  
F. Kamareddine  
A useful lambda notation, p. 17.

92/23  F. Kamareddine  
E. Klein  
Nominalization, Predication and Type Containment, p. 40.

92/24  M. Codish  
D. Dams  
Eyal Yardeni  
Bottom-up Abstract Interpretation of Logic Programs, p. 33.

92/25  E. Poll  
A Programming Logic for Fo, p. 15.

92/26  T. H. W. Beelen  
W. J. J. Stut  
P. A. C. Verkoulen  
A modelling method using MOVIE and SimCon/ExSpect, p. 15.

92/27  B. Watson  
G. Zwaan  
A taxonomy of keyword pattern matching algorithms, p. 50.

93/01  R. van Geldrop  
Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.

93/02  T. Verhoeff  
A continuous version of the Prisoner’s Dilemma, p. 17

93/03  T. Verhoeff  
Quicksort for linked lists, p. 8.

93/04  E. H. L. Aarts  
J. H. M. Korst  
P. J. Zwietering  
Deterministic and randomized local search, p. 78.