Program derivation in acyclic graphs and related problems

Citation for published version (APA):

Document status and date:
Published: 01/01/1992

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.
Program Derivation in Acyclic Graphs
and Related Problems

by

J.P.H.W. van den Eijnde 92/04
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.
PROGRAM DERIVATION IN ACYCLIC GRAPHS AND RELATED PROBLEMS

J.P.H.W. van den Eijnde
Department of Computing Science, Hg. 7.16
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
(31)40-473922
wsinjvde@win.tue.nl

Abstract

In this paper several derivations of algorithms on acyclic graphs are presented. Acyclic graphs are shown to be important for the solution of a general form of recursive equations, and as such appear everywhere in the practice of programming.

In order to facilitate the construction of compact and elegant derivations and to achieve the necessary level of abstraction, some new notation is introduced, as well as an extension of set and function calculus. Furthermore, a number of additional programming constructs, such as element selection from a set, a Pascal-like if-statement and a for-statement for sets, are introduced, which greatly reduce the length of the programs and derivations; the for-statement even more so when used in combination with theorems for the refinement of expression or function assignments, stated and proved in this paper.

Reference to acyclic graphs requires knowledge of what exactly cyclicity means. Various definitions are proved equivalent and their usefulness with respect to program derivation is analyzed. Then the most suitable definition, equivalent to the fixed-point characterization in [1], is used to derive an algorithm for the well-known problem of checking whether a given graph is cyclic. The set of end points of unbounded paths turns out to play an essential role here, and in all other problems in this paper.

The central problem of the paper concerns the computation of a general class of functions on the vertex set of an acyclic graph, defined by a particular form of recursive equation, which is shown to have a unique solution. Solving this problem gives rise to two interesting subproblems: graph inversion [2] and topological sorting [3]. For these problems, compact derivations of algorithms are presented as well.

The central problem is also solved for equations additionally satisfying a certain split property, leading to a more elegant algorithm. Finally, the practical applicability of the theory is demonstrated by means of a number of typical examples, satisfying this split property.

It is stressed that neither the programming problems, nor their solutions are the key issue here: most of them are not new. However, the abstract, compact and calculational style of deriving algorithms from the problem specification, developed in this paper, is believed to simplify the solution of a wide class of practical programming problems.
1. Introduction

Many programming problems involve the solution of recursive equations over an indexed collection of mathematical entities $f_v$, with $v \in V$, for some set $V$. These recursive equations consist of basic and step equations. The basic equations specify, for a non-empty subset of indices, the $f$-values explicitly. For the remaining indices, the $f$-value is expressed in terms of $f$-values of other indices, by means of the step equations. If some $f$-value depends, either directly or indirectly, on itself via the step equations, the recursive equations are said to be cyclic, otherwise they are called acyclic.

In the step equations each entity may depend on one other (trivial example: the faculty function $f_n = n*f_{n-1}$), two others (example: the fibonacci series with $f_n = f_{n-1}+f_{n-2}$), or on any fixed positive number of others. These are the commonly known versions of recursive equations. However, in general the $f$-value for each index $v$ may depend on the $f$-values of an arbitrary set of other indices, say $P \cdot v$. Thus $<V,P>$ may be interpreted as a directed graph, with vertices set $V$ and predecessor function $P$. This explains the relevance of graphs, from a fundamental point of view: they arise in a natural way as carrier of generalized recursive equations. Furthermore, it is obvious that the cyclicity of the recursive equations is reflected by the cyclicity of the corresponding graph, i.e. the existence of $v \in V$ with $v \in P^+ \cdot v$, where $P^+$ is the set-valued function corresponding to the transitive closure of the relation $\rightarrow$, defined by $x \rightarrow y \iff x \in P \cdot y$.

Problems based on general recursive equations, relating to a graph, arise in many different fields. For instance, in digital networks or electronic circuits, a collection of components with connections between them, the behaviour of each component is in some way related to that of its direct neighbours. In topography, locations are connected with roads, and, for instance, certain reachability properties can be expressed in terms of neighbour locations. In compiler theory, attribute grammars are defined as properties (attributes) of strings in a language, which may be inherited from ancestor strings, or synthesized from their off-spring strings. And, finally, in the theory of process scheduling, properties of processes may depend on those of other processes that must be carried out before or after them. With some effort, many more problems can be modelled using graphs.

The methodology used in this paper to solve graph programming problems is based on three cornerstones: heuristics, abstraction and calculation. The use of proper heuristics was already advocated by Dijkstra [4] in 1976. For a
systematic treatment the reader is referred to [5] (although it may be noticed from this paper that I disagree with some of the ideas). However, in the "Eindhoven School", founded by Dijkstra, the general opinion seemed to be that these heuristic rules could only be acquired by working out many examples. Without wanting to question the need for practice, I have tried to formulate, more explicitly than in [5], a number of "rules of thumb", appearing in the paper wherever appropriate. For convenience's sake the rules are grouped together again in the concluding remarks (section 9) of this paper.

The need for abstraction is more widely accepted [1,6], although in some modern textbooks [7] graph algorithms are treated in terms of low level data structures. The way to abstract is also subject of discussion. In this paper we use Dijkstra's formalism, supplemented with sets and functions as data structures, some of it based on ideas in the "Small Programming Exercises" by Rem [8]. The details of the data abstraction we propose are explained in section 2.

In section 4 a touch of control abstraction is introduced, by means of some convenient programming structures. These are: element selection from a set, an if statement (as) where the remaining alternative leads to a skip instead of an abort, and a for-statement that repeats a statement list for all elements of a fixed set, considerably shortening the resulting algorithms. In order to at the same time reduce the derivation lengths three theorems are proved, all turning a special kind of assignment statement directly into a for-statement, without having to calculate with pre- or postcondition predicates all the time. They are called expression accumulation theorem, function modification theorem and lift accumulation theorem.

The third cornerstone of our methodology, a calculational style, is widely believed to be a prerogative of transformational programming, be it imperative [9], recursive [10], functional [11] or even relational [12]. One of the goals of this paper is to show that also programming with conventional Hoare triples can be done purely calculational. The necessary set and function calculus is treated in section 3. The resulting style is, admittedly, hybrid, with program notation on the one hand, and derivations using predicates on the other hand. On some occasions, e.g. in the expression accumulation theorem or the function lift rules, the influence of functional programming can be distinguished, and used to advantage.

Before we apply the theory to the central problem of solving the recursive equations on an acyclic graph, we have to take a closer look at cyclicity. In section 5 cyclicity is defined by directly formalizing the intuitive notion of
it, instead of at once choosing an opportune starting point for a derivation, like in [13], or leaving it undefined [14], and just using some presumed properties. Our formal definition, in terms of the set of vertices on a cycle, is judged unsuited for program derivation, on account of general heuristic principles, and transformed systematically into a fixpoint definition in terms of the set of end points of unbounded graph paths, using the central cyclicity theorem, which is also important in the rest of the paper. The dual of the latter set turns out to be precisely the greatest fixed point solution of a defining equation for cyclicity by Cai and Paige in [1].

An algorithm that computes this fixpoint, derived in section 6, not only solves the interesting question whether a given graph is cyclic, but also turns out to serve as a blueprint for all acyclic graph algorithms in the sequel. The "cyclicitytest" algorithm is a classical one [15], but the ways to obtain it are often verbal [14], or operational [16], and not constructive. Our fixed-point approach roughly corresponds to that of Cai and Paige in [1], but their derivations are more global, and algorithms are not described in detail. The development of a full programming methodology with heuristics does not seem to be their objective, but instead automatic algorithm generation.

In section 7 we address the central problem of computing a function on the vertices of a directed acyclic graph, first specifying it by expressing the function value for each vertex in terms of the values of its predecessors, and subsequently proving that the resulting fixed point equation has a unique solution (uniqueness theorem). A general solution is obtained by dividing the problem into three subproblems: a trivial part yielding a simple repetition, the graph inversion problem [2], and topological sorting [3]. All subproblems are shown to allow a compact and elegant derivation, using the results of previous sections. The final solution then comes down to picking up the pieces and streamlining.

Textbooks on programming are abundant, and most of them devote one or more sections to graph problems [2,3,6,7,15], probably because of the reasons mentioned in the first paragraphs of this section. However, a general treatment of the class of programming problems on acyclic graphs like the one in section 8, is often missing. Generally, only a solution to the problem of topological sorting is presented [3,7,15] (accompanied by a verbal proof), and its relation to generic solutions of acyclic-graph problems is hinted upon. Even the, surprisingly few, textbooks specialized in graph algorithms [16,17], do not contain a general treatment of the problem class. McHugh [16] only treats a number of examples of problems on acyclic graphs or trees in an
operational way, which do not appear to be strongly related. Carré [17], though developing a promising formal approach to graph algorithms, only gives a global description of a tree-traversing algorithm. Formal correctness proofs for algorithms, if present in publications, tend to be given afterwards.

Finally, in section 8, we try to solve fixpoint equations with a special property, called the split property. It turns out that in this case more elegant and efficient algorithms are feasible. An application of this program scheme to several typical examples concludes the paper.

2. Conventions, types and variables

In our program notation, termed SAL (Set Algorithmic Language), all variables are assigned a - constant - type at the beginning of the block with respect to which they are local, using a declaration statement of the form \( v : T \) (\( v \) a variable and \( T \) a type). Declarations are separated from the other statements by a bar (|). By convention ordinary program variables and dummies are denoted by small letters.

A (data) type is thought to consist of a value set and a collection of operators on elements of the value set, sometimes supplemented with a list of properties. Although strictly speaking a type is distinct from its value set, they will often be identified in the sequel, as this rarely causes problems. The types used in this paper are simple types or structured types. In addition to the standard GCL (Guarded Command Language [2]) simple types integer and boolean, in SAL denoted by \( \mathbb{Z} \) and \( \mathbb{B} \), respectively, we will sometimes use natural numbers \( \mathbb{N} \), and subrange types \([m,n)\), with \( m,n \in \mathbb{Z} \) (\( m \leq n \)). These types support the usual integer operators +, -, *, /, div and mod, as well as the infix operators \( \downarrow \) (minimum) and \( \uparrow \) (maximum), relational operators \( \leq, >, <, \geq, =, \neq \), and the boolean connectives \&, \|, \land, \lor, \Rightarrow, \Leftarrow, \equiv, \not\equiv, \not\Rightarrow, \not\Leftarrow\). Finally, we use a quantifier notation with \( \exists \) and \( \forall \) the existential and universal quantifiers, \( \# \) and \( \Sigma \) the numerical and summation quantifiers, and \( \downarrow \) and \( \uparrow \) the minimum and maximum quantifiers (cf. [2]). For instance,

\[
(\downarrow i : P \cdot i ; \Sigma \cdot i)
\]

denotes the minimum of values \( P \cdot i \) for all \( i \) that satisfy the predicate \( P \cdot i \). Note that predicates are written in script capitals. The same holds for other
functions and expressions in the metalanguage (assertions and derivations).

Although it is, in principle, possible to identify vertices of a finite graph with integers, it is preferred to introduce a new simple type for these. This type, termed $\mathbb{U}$, is not structured: no operations are defined on elements of $\mathbb{U}$, i.e. the vertices. If no confusion arises $u, v, w, x, y, \text{ and } z$ are understood to be vertices.

In this paper we will make use of three type constructors, with which structured types can be composed from simple types, in addition to the array type constructor in GCL. The first type constructor is the cartesian product (pairing constructor), denoted by $\times$. In various computer languages it corresponds to the record structure. If $S$ and $T$ are given types, then the product type $S \times T$ is defined by

$$\langle s, t \rangle \in S \times T = s \in S \land t \in T,$$

for all $s$ and $t$, where $s \in S$ means "$s$ is in the value set of $S$". For any $p \in S \times T$, the elements of the composing types can be retrieved by the $\cdot$ operator. Thus we have $p \cdot 0 \in S$, $p \cdot 1 \in T$ and $p = \langle p \cdot 0, p \cdot 1 \rangle$.

Secondly, the power set type constructor $\mathcal{P}$ is defined by

$$X \in \mathcal{P} \cdot T \equiv X \subseteq T,$$

for any type $T$, where $\subseteq$ denotes ordinary set inclusion (with respect to the value sets). The elements of the power set type are called sets. They will be used very frequently in the derivation of graph algorithms. In the spirit of the identification of types with their value sets, the powerset type constructor may also be used to construct a type from an ordinary set, so type and set arguments of $\mathcal{P}$ are used promiscuously. In the following all sets used are understood to be of type $\mathcal{P} \cdot V$, with $V \in \mathcal{P} \cdot \mathbb{U}$ a finite, fixed set of vertices, and all vertices used are elements of $V$, unless stated otherwise. By convention sets (and set-valued functions) are denoted by capital letters.

Small sets $A$ can be made explicit by listing their elements, using common mathematical notation, for example $A = \{-1, 7, 3, 159\}$. Thus a singleton set containing $t$ is denoted accordingly by $\{t\}$, and the empty set by $\emptyset$, but for the latter the more usual symbol $\emptyset$ will be used. For larger sets a quantifier (the collectifier) can be employed. So the expression

$$\{i : \mathcal{P} \cdot i : \emptyset \cdot i\}$$
denotes the set of values $\mathcal{E} \cdot i$ for those $i$ that satisfy $\mathcal{P} \cdot i$, and is as such the counterpart of the usual mathematical notation $\{\mathcal{E} \cdot i | \mathcal{P} \cdot i\}$. However, the former is preferred here, in order to conform with the quantifier notation in the Dijkstra tradition, making the dummy explicit.

In order to avoid confusion between set notation on the one hand, and the notation of hints in chain proofs and assertions in algorithms on the other hand, we introduce the convention of replacing the hint and assertion brackets $\{\}$ by the special brackets $\langle\rangle$.

For the operators corresponding to the power set data structures in general, we adopt the usual mathematical set operators $\cap$ (intersection), $\cup$ (union) and \ (difference). The reader is assumed to be familiar with their definitions and properties. Being associative and commutative operators, $\cap$ and $\cup$ can also be used as quantifiers. Also we have the cardinality function # and complement (with respect to the value set of the corresponding type) function $\sim$ on sets, to be used as unary prefix operators. Of course, the relational operators from set theory $\subseteq, \subset, \supset, \neq$ and $\not\in$ are adopted, too, as well as $\in$ and $\exists$, expressing the (non-)membership relation between an element and a set. Note that the operators belonging to some type $\mathcal{T}$ are entirely independent of the set operators for type $\mathcal{P} \cdot \mathcal{T}$ mentioned above!

In addition to the aforementioned operators two new binary infix operators, $+$ and $-$, are introduced, with definitions

$$A + B = A \cup B, \quad \text{provided} \quad A \cap B = \emptyset, \quad \text{and} \quad (3)$$
$$A - B = A \setminus B, \quad \text{provided} \quad A \cap B = B \quad \text{(or, equivalently} \ A \supseteq B). \quad (4)$$

The proof obligation that these partial operators satisfy their domain restrictions rests upon the user. Operators $+$ and $-$ are sometimes preferred to their counterparts $\cup$ and \ because of some pleasant representation properties and calculus rules (section 3).

Finally, the function type constructor $\rightarrow$ is defined, using (1) and (2):

$$f \in \mathcal{F} \rightarrow \mathcal{T} \equiv f \in \mathcal{P} \cdot (\mathcal{F} \times \mathcal{T}) \land (\forall p: p \in \mathcal{F}: (\# q: q \in \mathcal{T}: \langle p, q \rangle \in f) = 1). \quad (5)$$

Evidently, (5) defines total functions $f$, being relations, where every element $p$ of the domain $\mathcal{F}$ corresponds to a unique element $q$ in the range $\mathcal{T}$. This $q$ is said to be the value of $f$ in $p$, and we write $f \cdot p = q$. In the remainder of this section $f$ is assumed to be of type $\mathcal{P} \rightarrow \mathcal{T}$. For such a function $f$, the
domain $F$ can be retrieved using mapping $d$, according to

$$d \cdot f = F.$$  \hfill (6)

In the same way as for sets instances of functions can be made explicit. Interpreting a function as a relation, according to the first conjunct in (5), it may be denoted as a set of pairs, e.g. $\{(0,x),(1,y),(2,z)\}$ (with domain $(0,1,2)$ and range $\{x,y,z\}$), or even $\emptyset$ (with empty domain and arbitrary range). The constant function with domain $S$ and value $c$ ($c \in T$) for all $s \in S$ may be denoted by $(S; c)$. If $S$ is a singleton domain, say $(s)$, and the distinction is clear from the context, this will be abbreviated to $(s; c)$. For a constant function $(V; c)$, with standard domain $V$, we will write $c$.

The lambda quantifier, offering a more general notation for functions, may be defined as a shorthand for the corresponding set expression:

$$f = (\lambda i : \mathcal{P} : i \in \mathcal{E} : i) = \{i : \mathcal{P} : \langle i \in \mathcal{E} \rangle\},$$  \hfill (7)

where for all values of the dummy $i$ that satisfy $\mathcal{P} : i$, $f \cdot i$ takes the value $\mathcal{E} : i$. The symbol $\lambda$ is the very quantifier from lambda-calculus, with a slight adaptation of notation. Note that, by definition, the domain (6) of $f$ in (7) is fixed by

$$d \cdot f = \{i : \mathcal{P} : i\}.$$  \hfill (8)

Often $\mathcal{P} : i$ takes the form $i \in \mathcal{F}$, making the domain more explicit. Note that the range $\mathcal{T}$ of $f$ cannot be inferred from (7), although it should contain the set $(i : \mathcal{P} : i \in \mathcal{E} : i)$. This information can only follow from the type of $f$, explicitly declared in advance. In practice this ambiguity seldom causes problems.

Using (7) it is now possible to define constant functions in a formal way. For a type $S$ (or alternatively, an arbitrary set), and a value $c \in T$, we have

$$(S; c) = (\lambda i : i \in S : c) \in S \rightarrow T.$$  \hfill (9)

In the spirit of (9) we also define a domain restricted function by

$$(S \upharpoonright f) = (\lambda i : i \in S \cap \mathcal{F} : f \cdot i) \in S \cap \mathcal{F} \rightarrow T$$  \hfill (10)

(pronounce "$S \upharpoonright f"$). Together with the so-called function modifier, for
a function \( g \in \mathcal{G} \) defined by

\[
(f \mid g) = \langle \lambda i : i \in \mathcal{G}; \text{ if } i \notin \mathcal{G} \rightarrow f \cdot i \text{ } \text{ } \text{ if } i \in \mathcal{G} \rightarrow g \cdot i \rangle \in \mathcal{F} \rightarrow \mathcal{T} \tag{11}
\]

("\( f \) unless \( g \)"), it constitutes an important tool for function manipulation, building new functions out of existing ones, and changing functions in just a limited part of their domain. For the latter purpose it is sometimes useful to combine notation (11) with (9) or (10). Adopting the convention that operator \( \mid \) binds weaker than \( \text{ if } \) and \( \rightarrow \), one pair of brackets may be omitted, and we write

\[
(f \mid S; c) = \langle \lambda i : i \in \mathcal{G}; \text{ if } i \notin S \rightarrow f \cdot i \text{ } \text{ } \text{ if } i \in S \rightarrow c \rangle , \tag{12}
\]

\[
(f \mid S \sim g) = \langle \lambda i : i \in \mathcal{G}; \text{ if } i \notin S \cup \mathcal{G} \rightarrow f \cdot i \text{ } \text{ } \text{ if } i \in S \cup \mathcal{G} \rightarrow g \cdot i \rangle , \tag{13}
\]

both of type \( \mathcal{F} \rightarrow \mathcal{T} \), and call these the constant function modifier and the function replacement modifier, respectively.

Finally, two additional notational conventions are worth mentioning. Logical constants in program assertions are denoted in greek letters, and abbreviations for statement lists are always in italics.

3. Calculus of sets, functions and graphs

3.1 Set calculus

The reader is assumed to be familiar with ordinary set calculus. In this section only a few special rules are given.

First we mention the so-called shunting rule, which is somewhat less familiar in set calculus, but nonetheless very useful:

\[
A \cap B \subseteq C \equiv A \subseteq C \cup \sim B . \tag{14}
\]

Its validity follows from the "ping-pong" proof

\[
A \cap B \subseteq C
\]

\( \triangleright \) (monotonicity of \( \subseteq \) w.r.t. \( \cup \); complement rule)

\[
A \cup \sim B \subseteq C \cup \sim B
\]

\( \triangleright \) (U is a supremum for partial order \( \preceq \)
A \subseteq C \cup \neg B
\Rightarrow \text{monotonicity of } \subseteq \text{ w.r.t. } \cap; \text{ complement rule}
A \cap B \subseteq C \cap B
\Rightarrow \cap \text{ is an infimum for partial order } \leq
A \cap B \subseteq C.
\square

Next, we turn to the set addition and subtraction operators + and −, introduced in (3) and (4). The most important reason to introduce these operators in the first place lies in the property

\[ A = B + C \iff A - B = C. \tag{15} \]

This elegant property, reminding us of integer arithmetic, follows directly from its counterpart

\[ A = B \cup C \land B \cap C = \emptyset \iff A \setminus B = C \land B \subseteq A, \tag{16} \]

but, of course, the former is easier to remember. A proof for (16) reads

\[ A = B \cup C \land B \cap C = \emptyset \]
\Rightarrow \text{Leibniz; property of } \cup
A \setminus B = (B \cup C) \cap \neg B \land B \cap C = \emptyset \land A \supseteq B
\Rightarrow \text{complement rule; use second conjunct}
A \setminus B = (C \cap \neg B) \cup (C \cap B) \land A \supseteq B
= \text{distributivity; } \neg B \cup B = V
A \setminus B = C \land A \supseteq B
\Rightarrow \text{Leibniz; property of } \cap
B \cup (A \cap \neg B) = B \cup C \land A \supseteq B \land C \subseteq \neg B
= \text{complement rule; (14)}
B \cup A = B \cup C \land A \supseteq B \land B \cap C = \emptyset
\Rightarrow \text{use second conjunct}
A = B \cup C \land B \cap C = \emptyset.
\square

Unfortunately, not all properties are this simple. Although addition is commutative and associative, and for subtraction we have
\[(A - B) - C = (A - C) - B = A - (B + C), \quad (17)\]

The equation \((A+B) - C = (A-C) + B\) does not hold in general! The following rules are convenient to simplify formulas:

\[
\begin{align*}
(A + B) - B &= A, \quad \text{if } A \cap B = \emptyset, \\
(A - B) + B &= A, \quad \text{if } A \supseteq B. 
\end{align*}
\]

(18) \hspace{1cm} (19)

Some distributivity properties are

\[
\begin{align*}
(A + B) \cap C &= (A \cap C) + (B \cap C), \quad \text{if } A \cap B = \emptyset, \\
(A + B) \setminus C &= (A \setminus C) + (B \setminus C), \quad \text{if } A \cap B = \emptyset, \\
A - (B \cup C) &= (A - B) \cap (A - C), \quad \text{if } A \supseteq B \cup C, \\
A \setminus (B - C) &= (A \setminus B) + (A \cap C), \quad \text{if } C \subseteq B, \\
(A + C) \cap (B + C) &= (A \cap B) + C, \quad \text{if } A \cap C = \emptyset \text{ and } B \cap C = \emptyset.
\end{align*}
\]

(20) \hspace{1cm} (21) \hspace{1cm} (22) \hspace{1cm} (23) \hspace{1cm} (24)

the conditions being such that, if the left hand side is defined, so is the right hand side. Consequently, they are best used from left to right.

A convenient rule of thumb turns out to be not to introduce operators + and - for \(\cup\) and \(\setminus\), until the application of one of the rules (15) or (17)-(19) has become unavoidable, or the transition to primitive expressions is compulsory (see section 4.2). For the translation of \(\cup\) and \(\setminus\) into + and -, respectively, we have the following two rules:

\[
\begin{align*}
A \cup B &= A + (B \setminus A), \quad \text{and} \\
A \setminus B &= A - (B \cap A).
\end{align*}
\]

(25) \hspace{1cm} (26)

The cardinality function \# obeys the following rules

\[
\begin{align*}
\#\emptyset &= 0, \quad \text{and} \quad \#\{x\} = 1, \\
\#(A \cup B) + \#(A \cap B) &= \#A + \#B, \\
\#(A \setminus B) + \#B &= \#(B \setminus A) + \#A, \\
\#(A + B) &= \#A + \#B, \quad \text{if } A \cap B = \emptyset, \\
\#(A - B) &= \#A - \#B, \quad \text{if } A \supseteq B.
\end{align*}
\]

(27) \hspace{1cm} (28) \hspace{1cm} (29) \hspace{1cm} (30) \hspace{1cm} (31)

Once more we note that the + and - set operators behave more pleasantly, under certain conditions (going from left to right!).

Many set quantor calculus rules are related to predicate calculus rules,
with which the reader is assumed to be familiar. The connection is established by the equivalence
\[
x \in (\cup i: P \cdot i: A \cdot i) \equiv (\exists i: P \cdot i: x \in A \cdot i).
\]
(32)

In turn, the set quantifier is expressed in the union quantifier using
\[
\{i: P \cdot i: E \cdot i\} = (\cup i: P \cdot i: (E \cdot i)).
\]
(33)

Finally, we need the basic equivalences
\[
\begin{align*}
x \in \emptyset & \equiv \text{false}, \\
x \in \{u\} & \equiv x = u, \\
x \in \neg A & \equiv x \in V \land x \notin A.
\end{align*}
\]

As all other set operators are expressible in terms of \(\neg\), \(\cup\) and \(\in\), the given rules suffice to derive all set (quantor) rules. Only domain disjunctions and intersections of the collectifier are worth mentioning separately here:
\[
\begin{align*}
\{i: P \cdot i: Q \cdot i: E \cdot i\} & = (\cup i: Q \cdot i: (E \cdot i)) \cup \{i: P \cdot i: (E \cdot i)\}, \quad \text{and} \\
\{i: P \cdot i: Q \cdot i: E \cdot i\} & \subseteq (\cup i: Q \cdot i: (E \cdot i)) \cap \{i: Q \cdot i: (E \cdot i)\}.
\end{align*}
\]
(34) (35)

The latter inclusion may be replaced by an equality if \(E \cdot i\) corresponds to a unique \(i\), for all \(i\) (i.e. \(E\) is injective).

This concludes the set calculus. For a more comprehensive overview of applicable rules the reader is referred to [18].

3.2 Function calculus

For function types we have the so-called lift rules, providing this structured type with operators related to those of the range type. A commonly used version of lifting, by means of the lift-meet rule, is defined as follows. Let \(f \in F \rightarrow T_0\) and \(g \in G \rightarrow T_1\) be given functions. Then the lifted version of the infix operator \(\oplus \in (T_0 \times T_1) \rightarrow T\), denoted by \(\hat{\oplus}\), is defined by
\[
f \hat{\oplus} g = (\lambda i: i \in F \otimes G: f \cdot i \circ g \cdot i) \in F \otimes G \rightarrow T.
\]
(36)
So far, we only have rules at our disposal to restrain function domains, for example (10), (11), and (36) introduced above. In programming, if a function is to be computed, we need to be able to extend a function domain, or to combine functions with disjoint domains. To that end we introduce the so-called lift-split rule, which, assuming $F \cap G = \emptyset$, is given by

$$f \downarrow g = (\lambda i : i \in F + G: \begin{cases} \text{if } i \in F & f \cdot i \\ \text{if } i \in G & g \cdot i \end{cases}) \in F + G \rightarrow \top \cup \top \cup \top .$$

(37)

As it happens, the functions encountered in programming practice rarely have disjoint domains. In fact, a combination of (36) and (37) turns out to be very fruitful. An obvious definition for another lifted operator, denoted by $\top$, would be

$$f \top g = (f \circ g) \cup (\sim G \cap f) \cup (\sim F \cap g) \in F \cap G \cup \top \cup \top \cup \top .$$

(38)

From (36), (37), (38), and (10) a more convenient rule, coined lift-join rule in the following, is readily derived. It reads

$$f \top g = (\lambda i : i \in F \cup G: \begin{cases} \text{if } i \in F \cup G \cap f & f \cdot i \\ \text{if } i \in F \cup G \cap g & g \cdot i \end{cases}) .$$

(39)

The power of rule (39) is that it generalizes both (36) and (37): if $F = G$ it reduces to (36), and if $F \cap G = \emptyset$ it reduces to (37). In the latter case the $\circ$ operator on the left hand side may be chosen arbitrarily, or, alternatively, be omitted. Yet another frequently occurring special case is $F \cup G$, when (39) reduces to the lift-include rule

$$f \top g = (\lambda i : i \in F: \begin{cases} \text{if } i \in G \cap f & f \cdot i \\ \text{if } i \in G \cap g & g \cdot i \end{cases}) \in F \rightarrow \top \cup \top \cup \top .$$

(40)

The case $F \cup G$ is treated analogously, of course.

As for operator properties, it turns out that commutativity, associativity, idempotency and distributivity simply carry over to the lifted operators.
In addition to the definitions above we need a modest number of calculus rules, describing the properties of domain restricted functions, function modifiers and lifted functions, and relating these to each other. Apart from the functions \( f \) and \( g \), and the operator \( \circ \), introduced above, in the remainder of this section \( A \) and \( B \) denote sets, and \( h \) is a function of type \( \mathbb{H} \rightarrow \mathbb{I} \).

First the so-called empty function rules are mentioned, which are

\[
\begin{align*}
(\emptyset \vdash f) &= \emptyset, \quad (41) \\
(f \vdash \emptyset) &= f, \quad (42) \\
\check{\circ} \emptyset &= \emptyset \check{\circ} f = f. \quad (43)
\end{align*}
\]

The proofs of these rules are trivial. The following rule, called distribution of \( \check{\circ} \) over \( \vdash \), reads

\[
f \check{\circ} (A \vdash g) = (f \vdash A \vdash f \check{\circ} g), \quad \text{for } A \subseteq \mathbb{F}, \quad (44)
\]

and it is proved by equating the terms of the lambda quantifiers on the common domain \( \mathbb{F} \) of the left and right hand side of (44). Using the lift-include rule (40) and domain restriction (10) we derive

\[
\begin{align*}
\text{if } i \notin A \& \& \& g \rightarrow f \downarrow i \quad \text{0 } i \in A \& \& \& g \rightarrow f \downarrow i \check{\circ} g \cdot i
\end{align*}
\]

\[
\begin{align*}
= \langle \text{regrouping case analysis; use } i \in \mathbb{F} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{if } \emptyset A & \rightarrow f \downarrow i \\
\text{0 } i \in A & \rightarrow \langle \text{if } i \in \mathbb{F} \& \& \& g \rightarrow f \downarrow i \check{\circ} g \cdot i \rangle \\
\text{0 } i \in \mathbb{F} \& \& \& \emptyset \rightarrow f \downarrow i \\
\text{0 } i \in \mathbb{F} \& \& \& g \rightarrow g \downarrow i
\end{align*}
\]

\[
\text{fi}
\]

\[
\begin{align*}
= \langle \text{iff } i \in A = i \in A \& \& \& (f \check{\circ} g) \text{ using } i \in \mathbb{F}; \text{ lift-join rule (39) } \rangle
\end{align*}
\]

\[
\begin{align*}
\text{if } i \notin A \& \& \& (f \check{\circ} g) \rightarrow f \downarrow i \\
\text{0 } i \in A \& \& \& (f \check{\circ} g) \rightarrow (f \check{\circ} g) \cdot i
\end{align*}
\]

the last expression being the term of the right hand side of (44). A direct consequence of (44) is the more restricted rule

\[
f \check{\circ} g = (f \vdash G \vdash f \check{\circ} g). \quad (45)
\]

The next rule, called domain chaining, is given by
\[(A \cap B \vdash f) = (A \vdash (B \vdash f)) = (B \vdash (A \vdash f)) \quad (46)\]

The validity of the first equality of (46) follows from (10) and
\[f \cdot i = (B \vdash f) \cdot i = (A \vdash (B \vdash f)) \cdot i\]
on the common domain \(i \in ANB\).

For an arbitrary idempotent operator \(\ll \in (\mathbb{T} \times \mathbb{T}) \rightarrow \mathbb{T}\) we have
\[(A \cup B \vdash f) = (A \vdash f) \ll (B \vdash f), \quad \text{for idempotent } \ll. \quad (47)\]

This property, termed domain disjunction, follows from the term equality
\[f \cdot i = \begin{cases} \text{if } i \in (ANF) \cap (BNF) \rightarrow f \cdot i \ll f \cdot i \\ i \in (ANF) \backslash (BNF) \rightarrow f \cdot i \\ i \in (BNF) \backslash (ANF) \rightarrow f \cdot i \end{cases} \]
of the left and right hand sides on the common domain \((AUB) \cap F = ANF \cup BNF\),
using domain restriction (10), the lift-join rule (39) and the idempotency of \(\ll\). On condition that \(ANB = \emptyset\) holds, and hence \((ANF) \cap (BNF) = \emptyset\), we may replace in (47) \(AUB\) by \(A+B\), and \(\ll\) by \(\vee\), thus obtaining the domain split rule
\[(A + B \vdash f) = (A \vdash f) \vee (B \vdash f), \quad \text{for } ANB = \emptyset. \quad (48)\]

Apart from domain disjunction we also have term disjunction, so
\[(A \vdash f \circ g) = (A \vdash f) \circ (A \vdash g) \quad (49)\]
holds as well. Once more this is based upon (10) and (39). On the common
domain \(AN (FUG) = ANF \cup ANG\) we derive for the terms
\[(f \circ g) \cdot i = \begin{cases} \text{lift-join rule (39)} \end{cases} \]
\[\begin{cases} \text{if } i \in FNG \rightarrow f \cdot i \circ g \cdot i \\ i \in F \backslash G \rightarrow f \cdot i \\ i \in G \backslash F \rightarrow g \cdot i \end{cases}\]
\[
\text{if } i \in (\mathcal{A} \cap \mathcal{F}) \setminus (\mathcal{A} \cap \mathcal{G}) \implies f \cdot i \oplus g \cdot i
\]

.. which proves (49). If \( F \cap G = \emptyset \), and hence \((\mathcal{A} \cap \mathcal{F}) \setminus (\mathcal{A} \cap \mathcal{G}) = \emptyset \), then a direct consequence of (49) is the term split rule, based on the lift-split rule (37):

\[
(\mathcal{A} \cap f \cdot g) = (\mathcal{A} \cap f) \cdot (\mathcal{A} \cap g), \quad \text{for } F \cap G = \emptyset. \tag{50}
\]

Next we find that domain chaining (46) has an analogue in term chaining:

\[
(f \vert g \cdot h) = ((f \vert g) \cdot h) = ((f \vert h) \cdot g), \quad \text{for } G \cap H = \emptyset. \tag{51}
\]

Using the function modifier definition (11) and the lift-split rule (37) we derive, on the common domain \( F \), for the terms of the first equality of (51):

\[
\text{case analysis; (37)}
\]

\[
\text{if } i \in G \cup H \implies (g \cdot h) \cdot i \quad \text{or } i \notin G \cup H \implies f \cdot i \quad \text{fi}
\]

\[
\text{case analysis; (11)}
\]

\[
\text{if } i \in H \implies h \cdot i \quad \text{or } i \notin H \implies (f \cdot g) \cdot i \quad \text{fi}
\]

Finally, we mention the convenient lift-split associativity property of \( \circ \) with respect to \( \cdot \). For disjoint \( g \) and \( h \) domains we have

\[
f \circ (g \cdot h) = (f \circ g) \circ h = (f \circ h) \circ g, \quad \text{for } G \cap H = \emptyset. \tag{52}
\]

Due to the domain disjointness, all function expressions share the same domain \( F \cup G \cup H \). Using the extensive case analysis in the lift-join rule (39) as well as the lift-split rule (37), the validity of the first equality of (52) follows from the term equalities.
A general operator on functions, which is independent of operators on the type of the range of functions involved, is composition. This binary infix operator, denoted by \( \circ \), is defined by

\[
(f \circ g) \cdot t = f \cdot (g \cdot t), \quad \text{for all } t \in S, \text{ if } T_0 \subseteq \mathcal{F}.
\] (53)

A careful examination of the definition shows that \( f \circ g \) is in fact the function that results from applying the lifted unary "operator" \( f \) to \( g \), using the obvious definition for unary lifting, yielding \( (f \circ) = (\hat{f} \cdot) \). However, notation (53) is more current by far. On the other hand, the interpretation as unary lifted operator suggests, in analogy with properties of binary operators carrying over to the lifted versions, that composition \( (f \circ) \) distributes over \( \hat{e} \) if \( f \) distributes over \( \hat{e} \). In general, the implication

\[
(\forall s,t: s \in \mathcal{T}_0 \land t \in \mathcal{T}_1: f \cdot (s \circ t) = f \cdot s \circ f \cdot t) \Rightarrow f \circ (g \circ h) = (f \circ g) \hat{e} (f \circ h)
\] (54)

holds, provided that the ranges \( G \) and \( H \) of \( g \) and \( h \) are contained in \( T_0 \) and \( T_1 \), respectively. All other function domains and ranges of the consequent match automatically if those in the antecedent do, and vice versa. The proof of (54)
is straightforward and left to the reader.

A special class of functions with respect to composition, is the class of identity functions. For any type \( T \), identity function \( \text{id}_T : T \to T \) is defined by

\[
\text{id}_T : t = t, \quad \text{for all } t \in T. \quad (55)
\]

These functions act as left or right identities of the composition operator.

Finally, it is useful to introduce function exponentiation as a kind of repeated composition. For functions \( f : T \to T \) and \( k \in \mathbb{N} \) \( f^k \) is defined inductively by

\[
\begin{align*}
f^0 &= \text{id}_T, \\
f^{k+1} &= f \circ f^k.
\end{align*}
\]

(56)

(57)

Due to the associativity of composition, \( f \) and \( f^k \) in (57) may be interchanged.

### 3.3 Paths and cycles in directed graphs

In SAL a directed graph is represented by a tuple \( <V,S> \), or equivalently by \( <V,P> \), where \( V \) is the finite set of vertices of the graph, and \( S,P : V \to P \cdot V \) are the successor and predecessor functions, specifying for each vertex its set of successors and predecessors, respectively. Functions \( S \) and \( P \) are related to each other by

\[
\forall u,v : u \in S \cdot v \Rightarrow v \in P \cdot u.
\]

(58)

Often \( S \) and \( P \) are generalized to functions of type \( P \cdot V \to P \cdot V \), according to

\[
S \cdot A = (u : v \in A : S \cdot v),
\]

(59)

and analogously for \( P \). Denoting both the standard and generalized versions of functions \( S \) and \( P \) with the same symbol rarely causes confusion. From (59) it follows directly that the generalized \( S \) function satisfies

\[
\begin{align*}
S \cdot \emptyset &= \emptyset, \\
A \subseteq B &\Rightarrow S \cdot A \subseteq S \cdot B, \\
S \cdot (u : v : P \cdot v : A \cdot v) &= (u : v : P \cdot v : S \cdot (A \cdot v)).
\end{align*}
\]

(60)

(61)

(62)
and likewise for \( P \). These properties are called strictness, monotonicity and uni-disjunctivity, respectively. Another convenient rule, relating \( S \) and \( P \), is

\[
A \cap S \cdot B = \emptyset \iff P \cdot A \cap B = \emptyset,
\]

or, equivalently,

\[
S \cdot B \leq \sim A \iff P \cdot A \leq \sim B.
\]

Proof of (63):

\[
A \cap S \cdot B \neq \emptyset
\]

\[
\equiv \{ \text{properties of } \cap \text{ and } \emptyset\}
\]

\[
(\exists v:v \in A \land v \in S \cdot B)
\]

\[
\equiv \{ \text{trading; (59)}\}
\]

\[
(\exists v:v \in A: (\exists w:w \in B: v \in S \cdot w))
\]

\[
\equiv \{ \text{interchange quantifiers; (58)}\}
\]

\[
(\exists w:w \in B: (\exists v:v \in A: w \in P \cdot v))
\]

\[
\equiv \{ \text{retrace first two steps in reverse order}\}
\]

\[
B \cap P \cdot A \neq \emptyset.
\]

Specializing (63) to \( A = \{x\} \) and negating both sides we have

\[
x \in S \cdot B \equiv P \cdot x \cap B \neq \emptyset,
\]

and, because of symmetry, the same holds with \( S \) and \( P \) interchanged.

It is worth noting that all properties (58)-(65) remain valid when we replace \( S \) by \( S^k \) and \( P \) by \( P^k \) \((k \geq 0)\); see (56) and (57)) everywhere, to be proved by straightforward induction to \( k \). The same holds replacing \( S \) by \( S^* \) (or \( S^+ \)) and \( P \) by \( P^* \) (or \( P^+ \)), where the transitive-reflexive and transitive closures are defined by

\[
S^* \cdot A = (\cup k: 0 \leq k: S^k \cdot A), \quad \text{and}
\]

\[
S^+ \cdot A = (\cup k: 1 \leq k: S^k \cdot A),
\]

respectively, and likewise for \( P \). Unfolding definition (66) once, and using (67) and (62), we find the following relations between \( S^* \) and \( S^+ \):
\[ S^* \cdot A = A \cup S^* \cdot A, \quad \text{and} \]
\[ S^+ \cdot A = S \cdot (S^* \cdot A) = S^* \cdot (S \cdot A). \quad (68) \]
\[ (69) \]

If we wish to define cyclicity on an abstract level, we have to understand the concepts of path and cycle. For reasons of simplicity it is preferred here to express these only in terms of \( S \) and \( P \), rather than identifying a path with the sequence of vertices it consists of, and introducing a sequence calculus.

The existence of an empty path from vertex \( x \) to \( y \) is equivalent to \( x = y \), and for a path of length 1, \( y \) is a successor of \( x \), so using (55), (56) and (57) we have

\[
\text{"there is an empty path from } x \text{ to } y \" \equiv y \in S^0 \cdot x, \quad \text{and} \quad (70)
\]
\[
\text{"there is a path of length 1 from } x \text{ to } y \" \equiv y \in S \cdot x \quad (= S^1 \cdot x). \quad (71)
\]

From the obvious informal definition

\[
\text{"there is a path of length } k+1 \text{ from } x \text{ to } y \" \equiv \quad (72)
\]

for \( k > 0 \), straightforward induction to \( k \), with (57) and (62), yields

\[
\text{"there is a path of length } k \text{ from } x \text{ to } y \" \equiv y \in S^k \cdot x \quad (= x \in P^k \cdot y). \quad (72)
\]

Existential abstraction from \( x \) in (72), using (59), then leads to

\[
\text{"there is a path of length } k \text{ to } y \" \equiv y \in S^k \cdot x \quad (= P^k \cdot y \neq \emptyset). \quad (73)
\]

On the other hand, existential abstraction from \( k \) in (72), using (66) and (67), yields

\[
\text{"there is a path from } x \text{ to } y \" \equiv y \in S^* \cdot x, \quad \text{and} \quad (74)
\]
\[
\text{"there is a non-empty path from } x \text{ to } y \" \equiv y \in S^+ \cdot x. \quad (75)
\]

Finally we arrive at the important definition of cyclicity. Realizing that a cycle is generally understood to be a non-empty path of which start and endpoint coincide, we conclude from (75)
"there is a cycle through $x$" $\equiv$ $x \in S^\times \cdot x$.  \hfill (76)

The compact definitions of these concepts turn out to be very convenient for all graph problems in which the identity of the paths is of no interest.

4. Programming constructs

4.1 Levels of abstraction

In SAL four levels of abstraction are distinguished. The most abstract one is the specification level, which is used to initially express the programming problems. For these specifications we have in principle the entire language of mathematics at our disposal: all kinds of definitions, explicit or recursive, in terms of Hoare triples, assertions, predicates, quantors, expressions, or previously defined algorithms. Usually, a functional specification takes the form

$$\{ \text{in}: S \mid
\{ \text{out}: \top \mid
\{ \text{Pre}\cdot \text{in}\}
\begin{array}{c}
\text{PROG}\n\{ \text{Post}\cdot \text{in}\cdot \text{out}\}
\end{array}
\} \}
$$

where \text{in} and \text{out} represent the input and output variables, respectively, and statement list \text{PROG} should be constructed such that, given the initial condition \text{Pre}\cdot \text{in}, the condition \text{Post}\cdot \text{in}\cdot \text{out} is established, changing only the \text{out} parameters. The Hoare triple in (77) is called a specification statement.

The functional specification acts as a starting-point for a refinement process. The first simplification steps take place on the specification level. On this level, application of the specification statement

$$\{ \text{true} \} \begin{array}{c}
\text{out}+: \text{Post}\cdot \text{in}\cdot \text{out}
\{ \text{Post}\cdot \text{in}\cdot \text{out}\}
\end{array}
$$

turns out to be convenient sometimes. This miracle statement is equivalent to
a specification statement, whenever the precondition is irrelevant (true), and as such it means: out is assigned a value such that Post-in-out is validated. A detailed definition of these programming primitives is beyond the scope of this paper; the reader is referred to [9].

The driving force of the refinement process on the specification level is the urge to obtain an algorithm on the abstract level. Such an algorithm may contain variables of all SAL types, expressions using SAL operators (see section 2), and all SAL statements (to be treated hereafter in section 4.2 and 4.3), but no specification or miracle statements, no quantifiers and no unknown quantities. An algorithm on this level can in principle be executed on some computer, using a powerful language (e.g. SETL [19]), but the efficiency of the execution is hard to predict.

The next stage is the derivation of an algorithm on the primitive level, containing only a restricted set of operations and expressions of SAL, called primitive operations (see section 4.2), allowing a simple and efficient implementation. The complexity of algorithms on this level can be established in a straightforward way.

Finally, the representation level contains the programs in plain GCL. In order to transform an algorithm from the primitive level to this level, the variables and operations are grouped into structures that share the same representation. Since the number of primitive operations is limited, it is possible to make an exhaustive list of standard translations to GCL [20].

Note that not all algorithms need to traverse all levels: each level has a meaning in its own right. In fact, the last implementation step to the representation level, being of an almost mechanical nature, is never taken in this paper. For all standard representations and heuristics, necessary to perform this implementation step for the algorithms derived hereafter, the reader is referred to [20].

4.2 Programming primitives

With the set and function operators in section 2, very complex expressions and statements can be formed. If they were to be implemented, their time complexity could be constant, linear, quadratic or perhaps worse. Often this is not satisfactory. In order to gain insight in the time complexity and implementability of an algorithm, it is expressed in terms of a limited collection of so-called primitive statements and expressions. For each of
these a representation can be shown to exist that allows an implementation in constant time. Unfortunately, there is no representation that does the trick for all of them. There appear to be groups of compatible operations that correspond to a certain data structure with efficient implementation. So the programmer is left with the proof obligation that no conflicting operations are carried out on the same data structure. Experience teaches us that for most problems algorithms can be designed satisfying this proof obligation.

The primitive statements mentioned above take the form \( x \in X \) (a new statement for the selection of an arbitrary element of \( X \)), or the more usual form of the assignment statements \( X := \emptyset \), \( Y := X \), \( Y := \{x\} \) and \( Y := Y - \{x\} \). They are defined by their weakest precondition \( \wp \) (note that we restrict ourselves to terminating programs), for a given predicate \( P \), by

\[
\wp \cdot (x \in X) \cdot P = X \neq \emptyset \land (\forall v \in X: P(x := v)),
\]

\( \wp \cdot (X := \emptyset) \cdot P = P(X := \emptyset), \) \( \wp \cdot (Y := X) \cdot P = P(Y := X), \) \( \wp \cdot (Y := \{x\}) \cdot P = x \notin Y \land P(Y := \{x\}), \) and

\[
\wp \cdot (Y := Y - \{x\}) \cdot P = x \in Y \land P(Y := Y - \{x\}),
\]

where \( P(x := e) \) denotes predicate (or expression) \( P \) with all textual occurrences of \( x \) replaced by expression \( e \).

No expressions involving sets are allowed in a primitive algorithm, except for a few relational expressions to be used in guards, or in assignments to boolean variables. These are \( x \in X \), \( x \notin X \), \( X = \emptyset \) and \( X \neq \emptyset \). Of course combinations of these, using the boolean connectives \( \land \), \( \lor \), \( \neg \), \( = \), \( \neq \) and \( \in \) are allowed, too.

In order to simplify derivations, or to obtain a more elegant algorithm, set \( X \) in these relational expressions, as well as in (79) and (81), may be replaced by expressions like \( A + B \) or \( A - B \). This is only advisable if \( A + B \) or \( A - B \) can be implemented as one single representation data structure. Their counterparts \( A \cup B \) and \( A \cap B \) are not permitted, however, since implementation of these always causes trouble. This is a valid heuristic to introduce + and -, but the conditions in (3) and (4) must always be checked, of course.

A rule of thumb that guarantees easy representation of \( A + B \) or \( A - B \) is that no set assignment to \( A \) or \( B \) occurs in the algorithm, unless the condition \( A \cap B = \emptyset \) (or \( A \cup B \), respectively) evidently remains valid. For instance, the concurrent assignment \( A, B := \emptyset, V \) does not obstruct the implementation of \( A + B \),
whereas the assignment \( A, B := W, W \) does.

An assignment to a function is primitive only if a one-point modifier of the same function appears on the right hand side, i.e. a statement of the form

\[
f := (f \cdot s : \mathcal{E} \cdot s),
\]

(84)

for \( f \in \mathcal{F} \). The definition of this primitive statement follows directly from the axiom of assignment. Using (12) we find that (84) reduces to a skip if \( s \not\in \mathcal{F} \). On the other hand, if \( s \in \mathcal{F} \) holds we rewrite (84) as

\[
f \cdot s := \mathcal{E} \cdot s.
\]

(85)

Thus, the definition of the latter statement reads

\[
wp(f \cdot s := \mathcal{E} \cdot s) \cdot \mathcal{P} = s \in \mathcal{S} \land \mathcal{P}(f := (f \cdot s : \mathcal{E} \cdot s)).
\]

(86)

Otherwise, in expressions only function applications to variables of a simple type within the function domain are allowed. Sometimes, initializations with constant functions will not be refined any further (their coding into a trivial repetition is then postponed to the implementation phase).

Apart from the statements and expressions mentioned here no set or function operations are allowed to appear in primitive level algorithms.

### 4.3 Structured statements

For reasons of convenience two new structured statements are introduced into SAL, in addition to concatenation, if ... fi and do ... od. The as statement is an alternative statement comparable to the if statement, but if none of the guards is satisfied a skip is executed, instead of an abort. Apart from being an obvious counterpart of the if, it has the advantage of making programs shorter and more transparent, at least for the majority of graph algorithms considered. For the one guard case it is defined by

\[
as \ B0 \rightarrow S0 \ s a \ is \ equivalent \ to \ if \ B0 \rightarrow S0
\]

\[
0 \ -B0 \rightarrow \ skip
\]

\[
f1,
\]

(87)
where $B_0$ is a boolean expression and $S_0$ a statement list. Generalizations to more guards are obvious, and left to the reader. A proof rule in terms of Hoare triples without reference to the if statement is given by:

\[
\{P\} \ as \ B_0 \rightarrow S_0 \ sa \ \{Q\} \quad \text{is valid if and only if} \\
\{P \land B_0\} \ S_0 \ \{Q\} \quad \text{and} \quad P \land B_0 \Rightarrow Q \quad \text{hold.} 
\]

Another structured statement to be introduced in SAL is the for statement, which is a repetition, where a variable takes the value of all elements of a given set, in arbitrary order. If $W$ is some set expression, it is defined by

\[
\text{for } x \in W \rightarrow SL \ \text{ref} \quad \text{is equivalent to} \quad \begin{cases} 
A : P \cdot W; \ x : W \mid A := W; \\
\quad \text{do } A \neq \emptyset \rightarrow \\
\quad \quad x \in A; \ A := A \setminus \{x\}; \ SL \\
\quad \text{od} \\
\end{cases}
\]

Evidently, the for statement not only turns out to be shorter than its GCL version, but it also saves the introduction of the fresh set variable $A$. The occurrence of $x$ in the for clause for $x \in W$ is supposed to implicitly open a block enclosing the repetition in which $x$ is declared as a local variable. Of course $x$ may be used in statement list $SL$.

### 4.4 Expression accumulation theorem

Application of the for-statement is bound to be cumbersome, if it replaces the corresponding repetition only after the latter is derived in full. Preferably, we would apply a proof rule directly to the for-statement itself, like we did for the as-statement in the previous section. Several versions of proof rules are feasible [18], but since they use assertions and invariants, they still bear the seed of the underlying repetition in them, so application of these is still somewhat elaborate. On the other hand, the theorem below addresses the special case of refining a complex assignment statement to a for-statement. Yet it turns out to be adequate and practical for many applications. Not surprisingly, it reminds us of the so-called reduction from functional programming [II]. The theorem is formulated as follows.
Expression accumulation theorem

Let \( r \in T \) be a program variable, \( W \in P \cdot V \) a set expression, \( \mathcal{F} \in P \cdot V \times T \to T \) and \( \ell \in V \times S \) given mappings, and let \( @ \in T \times S \to T \) be a binary infix operator, for types \( S \) and \( T \). If for all \( A \in P \cdot V \), \( c \in T \), and \( x \in W \cdot A \)

\[
\mathcal{F} \cdot (A+\{x\}) \cdot c = \mathcal{F} \cdot A \cdot c \oplus \ell \cdot x
\]  

(90)

holds, then the statement \( r := \mathcal{F} \cdot W \cdot r \) is refined by

\[
\text{for } x \in W \Rightarrow r := r \oplus \ell \cdot x \quad \text{ref}, \quad (91)
\]

provided that the precondition of \( r := \mathcal{F} \cdot W \cdot r \) implies \( r = \mathcal{F} \cdot \emptyset \cdot r \).

Proof:

According to the axiom of assignment, the desired refinement, say \( \text{FOR} \), of the assignment statement \( r := \mathcal{F} \cdot W \cdot r \) should satisfy:

\[
\{P(r := \mathcal{F} \cdot W \cdot r) \land r = \mathcal{F} \cdot \emptyset \cdot r\} \quad \text{FOR} \quad \{P\},
\]  

(92)

for arbitrary predicates \( P \). In (92) the precondition has been strengthened with the conjunct \( r = \mathcal{F} \cdot \emptyset \cdot r \). Using the one-point rule in the precondition of (92), strengthening the postcondition somewhat and using a substitution rule, the validity of (92) follows from

\[
\{P(r := \mathcal{F} \cdot \Omega \cdot r) \land r = \mathcal{F} \cdot \emptyset \cdot r \land W = \Omega\} \quad \text{FOR} \quad \{P\}
\]  

(93)

\[
\{P(r := \mathcal{F} \cdot \Omega \cdot r) \land r = \mathcal{F} \cdot \Omega \cdot r\},
\]

and this in turn follows from

\[
\{r = \mathcal{F} \cdot \emptyset \cdot r \land W = \Omega\} \quad \text{FOR} \quad \{r = \mathcal{F} \cdot \Omega \cdot r\}
\]  

(94)

for all values of the logical variables \( r \) and \( \Omega \), taking advantage of the universal disjunctivity property for Hoare triples. This suggests a refinement of \( \text{FOR} \) by a repetition with invariant

\[
\mathcal{F}; r = \mathcal{F} \cdot (\Omega - A) \cdot r,
\]  

(95)
with a set variable \( A \), initially set to \( W (=\Omega) \), shrinking to \( \emptyset \). The implicitly required invariance of \( A \in \Omega \) is evident. The correctness of the repetition below then follows from the annotation in brackets:

\[
\begin{align*}
[A; & P \cdot W; x: W] & \quad \{ r = P \cdot r \wedge W = \Omega \} \\
A := W; & \quad \{ r = P \cdot (\Omega - A) \cdot r \} \\
\textbf{do} & \quad \{\emptyset \neq A \subseteq \Omega \wedge r = P \cdot (\Omega - A) \cdot r \} \\
\textbf{while} & \quad \{ x \in A \subseteq \Omega \wedge r = P \cdot (\Omega - A) \cdot r \} \\
A := A - \{x\}; & \quad \{ x \in \Omega - A \wedge r = P \cdot (\Omega - (A + \{x\})) \cdot r \} \\
r := r \ominus \ell \cdot x & \quad \{ r = P \cdot \Omega \cdot r \}, \text{ so } r = P \cdot \Omega \cdot r \}
\end{align*}
\]

The only non-trivial step is the assignment to \( r \) at the end of the repetition body; evaluating the right hand side expression, and using the precondition, we derive

\[
\begin{align*}
P \cdot (\Omega - A) \cdot r \\
= \{ x \in \Omega - A; (19) \} \\
P \cdot ((\Omega - A) - \{x\}) + \{x\} \cdot r \\
= \{ (90) \text{ with } (\Omega - A) - \{x\} \text{ for } A \} \\
P \cdot ((\Omega - A) - \{x\}) \cdot r \ominus \ell \cdot x \\
= \{ (17) \} \\
P \cdot (\Omega - (A + \{x\})) \cdot r \ominus \ell \cdot x \\
= \{ r = P \cdot (\Omega - (A + \{x\})) \cdot r \} \\
r \ominus \ell \cdot x.
\end{align*}
\]

Now, applying the definition of the for-statement (89), we find that (91) is indeed a correct refinement of FOR, and hence of \( r := P \cdot W \cdot r \).

Contrary to \( P \), it is essential that \( \ell \) does not depend on \( A \) or \( c \). If it does depend on \( c \), copying \( r \) to an auxiliary variable first may help. The possibility of \( P \cdot A \cdot c \) depending on \( c \) has been incorporated to allow for frequently occurring statements like \( r := r \cup W \) to be implemented using the expression accumulation theorem (yielding for \( x \in W \Rightarrow r := r \cup \ell \cdot x \)) using \( P \cdot A \cdot c = c \cup A \). If \( P \cdot A \cdot c \) does not depend on \( c \), the second parameter of \( P \) may be omitted.

Note that the result variable \( r \) and the accumulation set \( W \) may in fact be
one and the same variable \( Wr \) (for this reason we use \( Q \) instead of \( W \) in the proof of the theorem). Thus it is also possible to refine \( Wr := S \cdot Wr U X \) by replacing \( Wr \) by \( A \), so \( F \cdot A = S \cdot A U X \), provided that \( Wr = F \cdot \emptyset = X \) holds initially. It is even permitted to replace only one occurrence of \( Wr \) by \( A \) in the right hand side of \( Wr := X U Wr U S \cdot Wr \), yielding \( F \cdot A \cdot c = X U c U S \cdot A \) or \( F \cdot A \cdot c = X U A U S \cdot c \), requiring preconditions \( Wr = X U Wr (\equiv X \subseteq Wr) \) and \( Wr = X U S \cdot Wr \), respectively. In all these cases one must be very careful to distinguish between the parameters used: which are to be kept constant, which parameter is to be accumulated and which is the result parameter.

If precondition \( r = F \cdot \emptyset \cdot r \) does not hold, the following corollary may help.

**Corollary: expression accumulation theorem with initialization**

If mapping \( F \) satisfies condition (90) of the expression accumulation theorem, and set expression \( W \in \mathcal{P} \cdot V \) does not depend on variable \( r \in \mathcal{T} \), then the assignment \( r := F \cdot W \cdot r \) is refined by

\[
 r := F \cdot \emptyset \cdot r; \quad \text{for } x \in W \Rightarrow r := r \cdot \xi \cdot x \quad \text{ref.} \quad (97)
\]

**Proof:**

The Hoare triple

\[
\{ r = \rho \land W = \Omega \} \quad r := F \cdot \emptyset \cdot r \quad \{ r = F \cdot \emptyset \cdot \rho \land W = \Omega \} \quad (98)
\]

demonstrates that the assignment \( r := F \cdot \emptyset \cdot r \), prior to the \( \text{for} \) statement, suffices to establish the desired precondition of (94) once again, but this time on the condition that \( W \) does not depend on variable \( r \), otherwise the axiom of assignment affects both \( r \) and \( W \) in (98). So, concatenation of the statements in (98) and (94) yields the desired refinement.

The reader may verify that refining \( Wr := Wr U X \), using \( F \) with \( F \cdot A = A U X \), to

\[
Wr := X; \quad \text{for } x \in W \Rightarrow Wr := Wr U \langle x \rangle \quad \text{ref} \quad (99)
\]

is indeed incorrect, due to the fact that the accumulation set is identical to the result parameter. The correct choice would be \( F \cdot A = Wr U A \), of course. This example shows that the reader should be careful to check the conditions of the expression accumulation theorem and its corollary each time he uses them.
4.5 Function assignment theorems

Another useful consequence of the expression accumulation theorem follows if we apply it to the special case of an assignment of a function replacement modifier to the function itself. Since this is a very common statement, it is convenient to dedicate a new theorem to it.

Function Modification Theorem:

Let \( f \in \mathcal{F} \) be a function variable, \( g \in \mathcal{G} \) a functional expression not depending on \( f \), and \( W \) a set expression with \( W \subseteq \mathcal{F} \) and \( W \cap \mathcal{G} \). Then assignment

\[
f := (f \upharpoonright W \downharpoonleft g)
\]

is refined by

\[
(\text{ref})
\]

for \( x \in W \rightarrow f \cdot x := g \cdot x \).

Proof:

Obviously, we try to apply the expression accumulation theorem from section 4.4. Relying on the fact that \( g \) does not depend on \( f \), mapping \( \mathcal{F} \) defined by \( \mathcal{F} \cdot A \cdot c = (c \upharpoonright A \downharpoonleft g) \) is a suitable candidate for the accumulation function. Using equations (41) and (42), we find

\[
\mathcal{F} \cdot \emptyset \cdot f = (f \upharpoonright \emptyset \downharpoonleft g) = f,
\]

establishing the desired precondition. Then we derive, for \( x \in W - A \),

\[
\mathcal{F} \cdot (A + (x)) \cdot c
= \langle \text{definition of } \mathcal{F} \rangle
(c \upharpoonright A + (x) \downharpoonleft g)
= \langle \text{domain split (48)} \rangle
(c \upharpoonright (A \downharpoonleft g) - ((x) \downharpoonleft g))
= \langle \text{term chaining (51)} \rangle
((c \upharpoonright A \downharpoonleft g) \upharpoonright ((x) \downharpoonleft g))
= \langle \text{definition of } \mathcal{F} ; (9) \text{ and (10), using } x \in W \cap \mathcal{G} \rangle
(\mathcal{F} \cdot A \upharpoonright x : g \cdot x).
\]

So, \( \mathcal{F} \cdot A \cdot c \) satisfies the conditions of the expression accumulation theorem, if
we take \( \odot \) to be defined by \( h \odot x = (h \mid x : q \cdot x) \) and \( k = \text{id}_W \). Application of the
theorem yields

\[
\begin{align*}
\text{for } x \in W & \Rightarrow f := (f \mid x : q \cdot x) \text{ pref}, \\
\end{align*}
\]

(103)

as refinement of (100), which, using (86) with \( x \in \mathcal{W} \mathcal{F} \), is equivalent to (101).

The required independence of \( q \) on \( f \) sometimes appears too strict, when we try to apply the foregoing theorem to frequently occurring assignments like

\[ f := (f \mid W \vdash f \circ q), \text{ or equivalently } f := f \circ (W \vdash q), \text{ according to (44). } \]

In such a case, function \( f \) must be adjusted on a part of its domain, but the new value depends on the original value. The condition in the function modification theorem cannot be dropped, however, which is demonstrated by a counterexample, where \( q \) is defined by \( q \cdot x = (\Sigma i : f \cdot i) \), for all \( x \). We decide to introduce a separate theorem for the assignments mentioned above.

**Lift Accumulation Theorem:**

Let \( f \in \mathcal{F} \mathcal{T} \) be a function variable, \( q \in \mathcal{E} \mathcal{S} \) a functional expression not depending on \( f \), \( W \) a set expression with \( \mathcal{W} \mathcal{F} \) and \( \mathcal{W} \mathcal{E} \), and \( \odot \in \mathcal{T} \times \mathcal{E} \times \mathcal{T} \) a binary
infix operator. Then the abstract assignment

\[ f := f \circ (W \vdash q) \]

is refined by

\[
\begin{align*}
\text{for } x \in W & \Rightarrow f \cdot x := f \cdot x \circ q \cdot x \text{ pref}, \\
\end{align*}
\]

(105)

**Proof:**

This time we take the accumulation function \( \mathcal{F} \) of section 4.4 to be defined by \( \mathcal{F} \cdot A \cdot c = c \circ (A \vdash q) \). Then, using equations (41) and (43), we find

\[
\mathcal{F} \cdot \circ f = f \circ (\circ \vdash q) = f, 
\]

(106)

again establishing the precondition. Then, for \( x \in W \cdot A \), we derive

\[
\begin{align*}
\mathcal{F} \cdot (A+(x)) \cdot c \\
= \langle \text{definition of } \mathcal{F} \rangle \\
= c \circ (A+(x) \vdash q) \\
= \langle \text{domain split (48)} \rangle \\
\end{align*}
\]
\[
\begin{align*}
& c \circ ((A \& q) \cdot ((x) \& q)) \\
& = \{ \text{lift-split associativity (52)} \} \\
& (c \circ (A \& q)) \circ ((x) \& q) \\
& = \{ \text{definition of } \mathcal{F}; (9) \text{ and } (10), \text{ using } x \in \mathcal{W} \} \\
& \mathcal{F} \cdot A \cdot c \circ (x: q \cdot x).
\end{align*}
\]

So, the proposed function \( \mathcal{F} \) again satisfies the conditions of the expression accumulation theorem, if we replace \( \circ \) by \( \circ \), and take \( f \) to be defined by \( f \cdot x = (x: q \cdot x) \). Application of the theorem yields

\[
\text{for } x \in \mathcal{W} \rightarrow f := f \circ (x: q \cdot x) \quad \text{ref.}
\]

(107)

With the help of (45), using the assumption \( x \in \mathcal{W} \Rightarrow \) and (86), the assignment may be simplified, and (107) turns out to yield refinement (105).

\[
\square
\]

5. Definitions of cyclicity

5.1 Basic definition

Before we can even try to solve programming problems in acyclic graphs, we have to know first what cyclicity means, and, if possible, find an algorithm that establishes whether a given directed graph is cyclic, with specification

\[
\begin{align*}
\text{[ V: P \cdot U; S: V \Rightarrow P \cdot V ]} \\
& \{ G = \langle V, S \rangle \land V \text{ is finite}\} \\
& \text{[ cyc: } \\ & \quad \text{cyclicitytest} \\
& \quad \{ \text{cyc } = \text{ "graph } G \text{ is cyclic"}\} \\
& \}
\end{align*}
\]

(108)

\[
]\]

In [14] Martin Rem tries to solve this problem without defining cyclicity altogether: the postcondition of the specification is left as it is, relying on common understanding with respect to its meaning. Then, in the course of program development, cyclicity properties are used wherever appropriate. Of
course, these properties cannot be proved if a proper definition of cyclicity is lacking.

For instance, for the solution to the problem a repetition is suggested with tail invariant

\[ \text{"the subgraph spanned by } V_0 \text{ is cyclic" } \equiv \text{ "} G \text{ is cyclic}, \] (109)

introducing yet another undefined concept, that of "subgraph". In the corresponding repetition progress is made by removing vertices without predecessors, based on the property that vertices without predecessors cannot be part of a cycle. The repetition ends if there are no vertices without predecessors left, and the property a non-empty graph containing only vertices with predecessors is cyclic is called for. To me, this approach seems ad hoc and undesirable.

Another possibility is to define cyclicity in such a way as to allow a smooth algorithm derivation from the start. This may result in definitions that are not quite obvious [13]. As with any new problem, it is not at all clear in advance, which definition is most suited as a starting point for program derivation. Since the determination of a suitable definition is an essential part of the problem, and by no means the easiest one, it should not be left untreated. In my opinion, the original definition of some concept should be a careful transcription of the intuitive notion, and if that is not a suitable one, an explicit motivation, based on general principles, should be given to choose an alternative definition, together with a proof for the equivalence of both definitions. Furthermore, all properties used should be proved from the definition.

Regarding the problem under consideration, the most natural definition seems to be one based on the notions of paths and cycles in section 3.3. Using (76) we find that the postcondition of specification (108) can be rewritten as

\[ \text{cyc } \equiv (\exists v :: v \in S^+ \cdot v), \] (110)

or, equivalently,

\[ \text{cyc } \equiv C \neq \emptyset, \] (111)

where \( C \), the cycle set, is defined as
In my opinion set $C$ is not easily computable, because of the pointwise nature of definition (112), and more in particular the fact that the vertices in its domain are being characterized in an implicit and non-constructive way. Definition (112) appears to leave no alternative than to calculate $S^+ \cdot v$ for each $v$, and then to test on $v$-membership, leading to an inefficient algorithm. So we introduce a general rule of methodology for graph problems, based on experience: try to eliminate pointwise set quantifiers. We aim at explicit, constructive expressions.

5.2 The endpoint set as fixed point

Trying to abstract from the pointwise nature of definition (112) for $C$, we note that it is not necessary to identify vertices on a cycle. The only thing we are interested in is a special property of $C$: its (non-)emptiness. Hence we should exploit (112) to find a suitable property of $C$, enabling a simple empty test. In this context it is noted that one of the most obvious properties of vertices on a cycle is that any such vertex has a successor. So, abstracting from individual vertices, it may be interesting to examine $S \cdot C$. We derive

$$w \in S \cdot C$$

1. $(\exists v \in S^+ \cdot v: w \in S \cdot v)$
2. $(\exists S^+ \cdot v = S^* \cdot (S \cdot v))$ (69) for $S^*$, with (32)
3. $(\exists v: (\exists u: u \in S \cdot v: v \in S^* \cdot u): w \in S \cdot v)$
4. $\{\text{instantiation of } u \text{ to } w; \text{ trading}\}$
5. $(\exists v: v \in S^* \cdot w: w \in S \cdot v)$
6. $\{\text{(59) with (32)}\}$
7. $w \in S \cdot (S^* \cdot w)$
8. $(S^* \cdot w = S \cdot (S^* \cdot w))$ (69); (112)
9. $w \in C$,

and from this derivation we conclude

$$S \cdot C \supseteq C.$$
It turns out that cycle set $C$ is a prefixed point of $S$ in the complete lattice $(\mathcal{P} \cdot \mathcal{V}, \geq)$ (see [21]). This is the so-called powerset lattice, with the reverse order: containment instead of inclusion. In this lattice intersection $\cap$ acts as supremum (least upper bound), whereas $\mathcal{V}$ acts as bottom of the lattice.

In [22,23] it was demonstrated that least fixed points of operators on a directed graph have a constructive definition, and consequently are easy to be computed. So, what about the least fixed point of $S$ in this case? According to (61) $S$ is monotonic with respect to the order $\geq$. Hence, the lattice being finite, it is also continuous with respect to the supremum $\cap$, meaning that

$$S \cdot (\cap i : \mathcal{P} \cdot i : A \cdot i) = (\cap i : \mathcal{P} \cdot i : S \cdot (A \cdot i))$$

holds for ordered chains $A \cdot i$. Then a fixed point theorem [21] states that the least prefixed point of $S$ exists, and equals the least fixed point. Calling this fixed point $E$, we find that $E$ is uniquely determined by

$$S \cdot E = E, \quad (\forall X: S \cdot X \geq X: E \geq X). \quad (114) \quad (115)$$

An explicit expression for $E$ also follows from a fixed point theorem [21]:

$$E = (\cap k: 0 \leq k: S^k \cdot \mathcal{V}). \quad (116)$$

Comparing (116) to (73), we find that $y \in E$ means there exist paths of any length $k$ to $y$. That is why $E$ is called the endpoint set, i.e. the set of all vertices in which paths of unbounded length end.

With the introduction of $E$, satisfying (114) and (115), we have satisfied the following empirical rule: try to specify the problem in terms of simple, recursive equations. According to [22,23,24], recursive equations tend to lead to more elegant derivations and algorithms, because they are more closely related to the process of proving conditions invariant over a single execution of the body of a repetition. So computing $E$ is no problem. However, originally we were interested in cycle set $C$! From (113) and (115) we may conclude

$$C \subseteq E, \quad (117)$$

but not the equality of $C$ and $E$, and (117) is not sufficient to establish postcondition (111). In the next subsection we address this problem.
5.3 Central cyclicity theorem

Because of the considerations in the previous section we like to shift our attention from \( C \) to \( E \). According to (117) we have \( C \subseteq E \). However, in general \( C = E \) does not hold. Nor does \( C = B \cap E \), with \( B \) the beginpoint set (the dual of \( E \)), although the latter looks plausible. A counterexample to both conjectures is \( G = (V, S) \) with \( V = \{x, y, z\} \), \( S \cdot x = \{x, y\} \), and \( S \cdot y = S \cdot z = \{z\} \), in which case \( C = \{x, z\} \) and \( E = B = V \). The same counterexample invalidates \( C = (\forall k : 0 < k : S^k \cdot B) \) and its dual.

Fortunately, equality of \( C \) and \( E \) is not required. In order to rewrite condition (111) in terms of \( E \), according to

\[
\text{cyc} = \{E \neq \emptyset, \quad (118)
\]

it suffices to prove the equivalence \( C \neq \emptyset \iff E \neq \emptyset \). From (117) we have \( C \neq \emptyset \implies E \neq \emptyset \). The reverse follows from the theorem below.

Central Cyclicity Theorem

In a directed graph with cycle set \( C \) (112) and endpoint set \( E \) (116) we have

\[
C \neq \emptyset \iff E \neq \emptyset . \quad (119)
\]

Proof:

The remaining proof obligation is the implication to the left. Assuming \( E \) is non-empty, we will construct a member of \( C \). From (65) and (114) we conclude

\[
x \in E \iff E \cap P \cdot x \neq \emptyset, \quad (120)
\]

expressing the fact that each element of \( E \) has a predecessor in \( E \). On the validity of (120) the correctness of the following algorithm is based.

\[
\begin{align*}
\text{if } x : V & \land W : P \cdot V \land \{E \neq \emptyset\} \\
\text{\hspace{1cm} x \in E & \land W := \emptyset; } \\
\{\text{invariant: } x \in E \land W \subseteq E \land W \subseteq S^+ \cdot x; \text{ bound function: } \#(E-W)\} \\
\text{do } x \notin W & \Rightarrow W := W + \{x\}; \ x := E \cap P \cdot x \quad \text{od} \\
\{x \in W \subseteq S^+ \cdot x, \text{ so with (112): } C \neq \emptyset\}
\end{align*}
\]
The invariance of $x \in E$ and $W \in E$ is trivial. The invariance of $W \in S^+ \cdot x$ follows from

$$wp \cdot (W := W+(x); x : \in E \cap P \cdot x) \cdot (W \leq S^+ \cdot x)$$

$$= \{\text{property of } wp; (79) \text{ and } (82)\}$$

$$E \cap P \cdot x \neq \emptyset \land (\forall v: v \in E \cap P \cdot x: W+(x) \leq S^+ \cdot v) \land x \notin W$$

$$= \{(120); \text{domain extension; property of } S\}$$

$$x \in E \land (\forall v: v \in P \cdot x: W \leq S^+ \cdot v \land x \in S^+ \cdot v) \land x \notin W$$

$$= \{(58); \text{definition (67)} \text{ of } S^+ \text{ implies } x \in S^+ \cdot v \Rightarrow x \in S^+ \cdot v\}$$

$$x \in E \land (\forall v: x \in S^+ \cdot v: W \leq S^+ \cdot x) \land x \notin W$$

$$= \{\text{monotonicity (61)} \text{ of } S \text{ and } S^+: x \in S^+ \cdot v \Rightarrow S^+ \cdot x \leq S^+ \cdot (S \cdot v) \leq S^+ \cdot v\}$$

$$x \in E \land (\forall v: x \in S^+ \cdot v: W \leq S^+ \cdot x) \land x \notin W$$

$$= \{\text{quantor term is independent of dummy } v\}$$

$$x \in E \land W \leq S^+ \cdot x \land x \notin W.$$

Algorithm (121), formalizing an informal proof in [14], provides a nice example of an algorithm being used to prove an abstract, mathematical theorem.

Note that algorithm (121) above not only validates the choice of (118) as our new postcondition, but also indicates how a cycle may be constructed, if one exists. The problem of finding all cycles, or set $C$, or the maximal strong components of a graph [4], can probably be solved with an algorithm based on (121). These problems will be the subject of future investigations.

It is possible to draw yet another conclusion from (121). First note that $CUS \cdot E = E$ follows from (114) and (117), so $E$ is a fixed point of $(CUS \cdot)$. Hence, according to [22], $S^+ \cdot C \subseteq E$, i.e. all vertices reachable from $C$ are contained in $E$. Now, unfolding the repetition in (121) once, it turns out that $\xi \in W$ is an invariant of the repetition, with $\xi$ the original value of $x$, chosen arbitrarily from $E$. The postcondition of the repetition then implies $\xi \in W 
S^+ \cdot x$, with $x \in C$. So each vertex in $E$ is reachable from $C$, which finally implies

$$E = S^+ \cdot C.$$  \hspace{1cm} (122)

The central cyclicity theorem finally settles the question of finding a definition for cyclicity that is well-suited for derivations, being $E \neq \emptyset$, with $E$ defined by (114) and (115). Moreover, the theorem turns out to be important for acyclic graph problems, to be demonstrated in section 7.4 and 8.1.
6. Derivation of an algorithm for cyclicity test

6.1 Abstract algorithm

In this section an algorithm is derived establishing postcondition (118), using definitions (114) and (115). These suggest the first approximation of an algorithm to be

\[ V_0 := E; \]
\[ \{V_0 = E\} \]
\[ \text{cyc} := V_0 \neq \emptyset \]
\[ \{118\}, \text{and with (119): (111)}\].

In the first statement \( E \) is unknown, and, obviously, a repetition is needed to compute it. We propose the simplest invariant conceivable, using the order \( 2 \) to express that \( V_0 \) is an approximation of \( E \) from below, i.e. a value between bottom and \( E \). This is another heuristic rule based on experience: if a least fixed point of some function on a lattice is to be calculated, approximate it from below. This rule is closely related to the linear search theorem from [2]. Realizing that in this case we work with the reverse order, the rule yields invariant

\[ P_0: \quad V_0 \geq E. \]

Then \( V_0 \) is easily initialized with \( V \) (bottom approximates any element of the lattice), and \( \#(V_0 \setminus E) \) serves as bound function, which we choose to decrease one at a time. Using (79) and (83), the second approximation is found to be

\[ V_0 := V; \quad \{P_0\} \]
\[ \text{do } V_0 \setminus E \neq \emptyset \rightarrow \]
\[ x \in V_0 \setminus E; \quad V_0 := V_0 \setminus \{x\} \]
\[ \text{od}; \]
\[ \{P_0 \land V_0 \setminus E = \emptyset, \text{ so } V_0 = E\} \]
\[ \text{cyc} := V_0 \neq \emptyset \]
\[ \{111\}\].

This algorithm would be perfect, apart from the fact that the unknown set \( E \) still appears in the algorithm. This unknown quantity is eliminated from the
algorithm by using the only data available: its definition and the invariant. Again we have to approximate from below, with the order 2:

\[ E = \{1+4\} \]
\[ S \cdot E \]
\[ \leq \{P0 and monotonicity of S (61)\} \]
\[ S \cdot V0. \]

(123)

Expression \( V0 \setminus E \), appearing twice in the algorithm, could accordingly be replaced by \( V0 \setminus S \cdot V0 \). In order to simplify the guard of the repetition, it is convenient to introduce a new variable \( V1 \) and invariant

\[ P1: \quad V1 = S \cdot V0. \]

Derivation (123) tells us that the new guard \( V0 \setminus V1 \neq \emptyset \) is stronger than the original one, so we must verify whether the postcondition still holds:

\[ V0 \setminus V1 = \emptyset \]
\[ = \{P\}; \text{ set calculus} \]
\[ S \cdot V0 \geq V0 \]
\[ \rightarrow \{115\}: E \text{ is the least prefixed point of } S \]
\[ E \geq V0 \]
\[ = \{P0\} \]
\[ V0 = E. \]

(124)

Thus, with the help of (19), we find the third approximation of the algorithm:

\[ V0 := V; V1 := S \cdot V; \]
\[ \text{do } V0 \setminus V1 \neq \emptyset \rightarrow \]
\[ \quad x \in V0 \setminus V1; \quad \{x \in V0 \setminus E\} \]
\[ \quad V0 := V0 \setminus \{x\}; \quad \{P0 \land P1(V0 := V0 \setminus \{x\})\} \]
\[ \quad V1 := S \cdot V0; \quad \{P0 \land P1\} \]
\[ \text{od; } \]
\[ \text{cyc := } V0 \neq \emptyset . \]

This is an abstract algorithm in the sense of section 4.1: it no longer contains unknown quantities or quantifiers, and it has been proven correct.
Only questions about its time complexity require further refinement.

Note that the choice of the order of statements in the body of the repetition is deliberate. In this way we are suggested to refine the last statement by evaluating its precondition, leading to an expression of the form \( S \cdot (V_0 + x) \). On the other hand, if we reverse the last two statements, we would evaluate the postcondition \( P_I(V_0 := V_0 - x) \), leading to an expression of the form \( S \cdot (V_0 - x) \). From (62) we know that \( S \) distributes over set union \( U \), so the former expression is easily evaluated, but \( S \) does not distribute over \( \setminus \) or \( \cap \), so evaluation of the latter expression is expected to be cumbersome. This experience is expressed in the heuristic rule: choose the order of assignments in accordance with the available calculus rules for the quantities involved.

6.2 Efficiency driven optimizations

The third approximation of the algorithm is still not satisfactory. For instance, the \( S \cdot V \) and \( S \cdot V_0 \) expressions are expected to have quadratic time complexity, if they were to be coded, so the algorithm as a whole is probably cubic. To improve this, we try to expand the precondition of assignment \( V_1 := S \cdot V_0 \), in order to exploit our knowledge of the original value of \( V_1 \):

\[
P_I(V_0 := V_0 + x)
= \{\text{substitution}\} \quad V_1 = S \cdot (V_0 + x)
= \{\text{disjunctivity of } S \text{ (62); } x \notin V_0\} \quad V_1 = S \cdot V_0 \cup S \cdot x
= \{+ \text{ introduction (25): heading for application of (15)}\} \quad V_1 = S \cdot V_0 + S \cdot x \setminus S \cdot V_0
= \{(15)\} \quad V_1 = S \cdot x \setminus S \cdot V_0 = S \cdot V_0
= \{\text{substitution}\} \quad P_I(V_1 := V_1 - S \cdot x \setminus S \cdot V_0).
\]

This derivation is driven by the wish to end up with a predicate of the form \( P_I(V_1 := \mathcal{C} \cdot V_0 \cdot V_1) \), since the statement under consideration is supposed to change \( V_1 \), not \( V_0 \). So we are trying to restore \( S \cdot V_0 \) on the right-hand side of the equality. This obeys the heuristic rule: when working from the precondition, try to rewrite it as a substitution of the variable to be changed.
From the derivation we conclude, using the axiom of assignment, that

\[ V_1 := V_1 - S \cdot x \backslash S \cdot V_0 \]  

(126)

is a valid refinement of \( V_1 := S \cdot V_0 \). This statement teaches us two important facts. Firstly, \( V_1 \) shrinks, implying the invariance of

\[ P_2: \; V_0 \supseteq V_1, \]

which holds initially and is not disturbed by the statement \( V_0 := V_0 \backslash \{x\} \), because \( x \) is selected outside \( V_1 \). Invariant \( P_2 \) states that \( V_1 \) is a better approximation of \( E \) than \( V_0 \) (\( V_1 \supseteq E \) follows from (123) and \( P_1 \)). This pleasant property allows expression \( V_0 \backslash V_1 \) to be rewritten as \( V_0 \backslash V_1 \), thus rendering both guard and selection statement primitive (see section 4.2). This could also have been expressed more explicitly by choosing \( V_1 = V_0 - S \cdot V_0 \), instead of \( P_1 \), but this complicates the calculations slightly and leads to the same result.

Secondly, \( V_1 \) changes by at most \( \#S \cdot x \) vertices, indicating a considerable decrease in time complexity, provided that expression \( S \cdot V_0 \) can be evaluated efficiently. Let us try:

\[ y \in S \cdot V_0 \]

\[ \{65\} \]

\[ P \cdot y \cap V_0 \neq \emptyset \]

\[ \{27\}: \text{discarding superfluous information}\]

\[ \#(P \cdot y \cap V_0) \neq 0 \]

\[ \{ \text{introducing variable } T: T \in V \backslash N, \text{ and } P_3 \text{ below}\} \]

\[ T \cdot y \neq 0, \]

(127)

additionally requiring the invariance of

\[ P_3: \; T = \# \circ \gamma \cdot V_0, \; \text{ with} \]

\[ \gamma \cdot A = (\lambda v \in V: P \cdot (V \cap A)), \]  

(128)

introducing mapping \( \gamma \in P \cdot V \rightarrow (V \rightarrow P \cdot V) \) as a shorthand notation, and using (53).

The introduction of \( T \) allows statement (126) to be rewritten as

\[ V_1 := V_1 - S \cdot x \backslash T^{-1} \cdot 0, \]  

(129)
provided that \( P_3 \) holds in its precondition. In (129) the inverse function notation \( T^{-1} \cdot c \) is used to denote the subset of \( V \) with \( T \)-value \( c \).

Note that \( T \) could have been introduced directly in the third approximation, yielding the assignment

\[ V_1 := V - T^{-1} \cdot 0 \]  \( (130) \)

instead of (129). However, the expected time complexity of (130) is \( \#V \) instead of \( \#S \cdot x \), so derivation (125) really helps. This experience is expressed in the general heuristic rule: exploit, as much as possible, the precondition of a statement to be refined.

The fourth approximation to the algorithm now reads

\[
\begin{align*}
V_0, V_1 &:= V, V; \\
T &:= \# \circ \mathcal{B} \cdot V; \\
V_1 &:= V_1 - V \cap T^{-1} \cdot 0;
\end{align*}
\]

\[ \text{do } V_0 \neq \emptyset \rightarrow \\
\begin{align*}
x &\in V_0 - V_1; \ V_0 := V_0 - \{x\}; & \{P1 \land P3 | (V_0 := V_0 + \{x\})\} \\
T &:= \# \circ \mathcal{B} \cdot V_0; & \{P1 | (V_0 := V_0 + \{x\}) \land P3\} \\
V_1 &:= V_1 - S \cdot x \cap T^{-1} \cdot 0 & \{P1 \land P3\} \\
\text{od;}
\end{align*}
\]

\[ \text{cyc} := V_0 \neq \emptyset . \]

The initialization of \( V_1 \) has been reformulated slightly, using trivial set calculus, to stress the similarity to the adjustment of \( V_1 \) in the body of the repetition. Once again notice the order of the statements in the body of the repetition: \( T \) must be adjusted before \( V_1 \), because in (127) \( P_3 \) was used, but it is placed after the adjustment of \( V_0 \), since expression (128) allows an easy domain split if \( A \) is replaced by \( V_0 + \{x\} \) (remember the heuristic rule!).

It appears that with the introduction of \( T \) we have only shifted the problem. Fortunately, updating function \( T \) turns out to be no big deal. For the initialization we simply apply the corollary of the expression accumulation theorem from section 4.4 to expression \( \# \circ \mathcal{B} \cdot V \). In order to find the required split property we derive

\[ \mathcal{B} \cdot (A + \{x\}) \]

\[ = \ \{\text{definition of } \mathcal{B} (128); \text{ distributivity (20)}\} \]
\[
(\lambda v:: P \cdot V \cap A + P \cdot V \cap \{x\})
\]

\[= \{ \text{lift-join rule (39) used as (36)} \}
(\lambda v:: P \cdot V \cap A) \uparrow (\lambda v:: P \cdot V \cap \{x\})
\]

\[= \{ \text{(128); case analysis, with S/P relation (58)} \}
\]

\[\# \cdot A \uparrow (\lambda v:: \text{if } v \notin S \cdot x \rightarrow \emptyset \ 	ext{v} \in S \cdot x \rightarrow \{x\} \text{ f}])\]

\[= \{ \text{lift-split rule (37); choosing operator \uparrow because of associativity} \}
\]

\[\# \cdot A \uparrow (\lambda v:: v \notin S \cdot x : \emptyset) \uparrow (\lambda v:: v \in S \cdot x : \{x\})\]

\[= \{ \text{lift-include rule (40), } V \ni S \cdot x; \emptyset \text{ unit of } +; \text{ constant function (9)} \}
\]

\[\# \cdot A \uparrow \{S \cdot x: \{x\}\},\] (131)

and observe that, due to the fact that \# distributes over + (30), and lifting this property with (54), the actual function \# \cdot \# \cdot (A + (x)) splits according to

\[\# \cdot \{\# \cdot A \uparrow (S \cdot x: \{x\})\} = \# \cdot \# \cdot \{S \cdot x: \{x\}\} = \# \cdot \# \cdot A \uparrow (S \cdot x: 1),\] (132)

where the first + operator acts on sets and the last two on integers. For the initialization we note that \# \cdot \# \cdot \emptyset = \# \cdot \emptyset = \emptyset, and that V does not depend on T, and find that statement\n
\[T := \emptyset; \text{ for } X \in V \rightarrow T := T \uparrow (S \cdot x: 1) \text{ ref}\.\] (133)

Using an analogous statement list for \(T := \# \cdot \# \cdot V 0\) would be inefficient. Again we must try to exploit our knowledge of the original value of T as much as possible. Then, with the precondition rule in mind as well, we derive

\[\#3(V 0 := V 0 + (x))\] = \{substitution\}

\[T = \# \cdot \# \cdot (V 0 + (x))\] = \{(132)\}

\[T = \# \cdot \# \cdot V 0 \uparrow (S \cdot x: 1)\]

\[= \{+/= \text{ properties; heading for T-adjustment}\}
\]

\[T \uparrow (S \cdot x: 1) = \# \cdot \# \cdot V 0\]

\[= \{\text{substitution}\}\]

\[\#3(T := T \uparrow (S \cdot x: 1)),\] (134)

resulting in assignment \(T := T \uparrow (S \cdot x: 1)\) for the adjustment of T. Together with (133) this yields the fifth approximation:
CSN 92/04  \[ \text{VO,VI:}= V,V; \]
\[ T:= \emptyset; \]
\[ \text{for } x \in V \rightarrow T:= T \cup \{S \cdot x: 1\} \, \text{end}; \]
\[ \text{VI:= } V - V \cap T^{-1} \cdot 0; \]
\[ \text{do } \{ \text{VO} - \text{VI}\neq \emptyset \rightarrow \}
\[ x \in \text{VO} - \text{VI}; \, \text{VO:= } \text{VO} - \{x\}; \]
\[ T:= T \cup \{S \cdot x: 1\}; \]
\[ \text{VI:= } V - S \cdot x \cap T^{-1} \cdot 0 \]
\[ \text{od}; \]
\[ \text{cyc:= } \text{VO}\neq \emptyset. \]

This algorithm is the final version on the abstract level. It even allows an estimate of the time complexity. Observing that in the initialization of \( T \),
and in the main repetition, each \( x \in V \) is selected at most once (\( x \) comes from \( \text{VO} \), and is subsequently removed from it), and assuming that the statements containing \( S \cdot x \) can be implemented in \( O \cdot (\#S \cdot x) \), we find the time complexity to be \( O \cdot (N+M) \), with \( N=\#V \), and \( M \), the total number of edges, defined by
\[ M = (\Sigma v:: \#S \cdot v). \quad (135) \]

Still, it is desirable to proceed to the primitive level, in order to see exactly how the abstract statements can be implemented.

6.3 Primitive algorithm

In the fifth approximation of the algorithm two assignments to \( \text{VI} \) and three assignments to \( T \) (including \( T:= \emptyset \), the refinement of which is not strictly necessary) remain to be refined. Starting with the former two, we note that both right hand sides are instances of
\[ \mathcal{F} \cdot A = \text{VI} - A \cap T^{-1} \cdot 0, \quad (136) \]
with \( A \) equal to \( V \) and \( S \cdot x \), respectively. Deriving a split property for \( \mathcal{F} \):
\[ \mathcal{F} \cdot (A+(y)) \]
\[ = \{ \text{\footnotesize (136)} \} \]
\[ \text{VI} - (A+(y)) \cap T^{-1} \cdot 0 \]
\[ \text{\textbackslash{}text{distributivity (20); subtraction (17)}} \]
\[ V_l - A \cap T^{-1} \cdot 0 - (y) \cap T^{-1} \cdot 0 \]
\[ \{36\} \]
\[ \mathcal{F} \cdot A - (y) \cap T^{-1} \cdot 0. \quad (137) \]

Together with the validity of \[ \mathcal{F} \cdot \varnothing = V_l \] \[ (137) \] implies that \( \mathcal{F} \) meets the conditions of the expression accumulation theorem from 4.4, yielding a for-statement over \( V \) or \( S \cdot x \), respectively, with statement

\[ V_l := V_l - (y) \cap T^{-1} \cdot 0 \]

in the body. Some case analysis, the definition of \( T^{-1} \) in section 6.2, and the as proof rule \[ (88) \], show that this statement in turn is refined by

\[ \text{as } T \cdot y = 0 \rightarrow V_l := V_l - (y) \text{ sa,} \]

which is primitive (see section 4.2). Accordingly, the assignment to \( V_l \) in the initialization is replaced by

\[ \text{for } y \in V \rightarrow \text{as } T \cdot y = 0 \rightarrow V_l := V_l - (y) \text{ sa rof,} \quad (138) \]

whereas the adjustment of \( V_l \) in the repetition body is replaced by

\[ \text{for } y \in S \cdot x \rightarrow \text{as } T \cdot y = 0 \rightarrow V_l := V_l - (y) \text{ sa rof.} \quad (139) \]

The assignments to \( T \) in \[ (133) \] and \[ (134) \] are refined by a direct application of the lift accumulation theorem from 4.5, yielding

\[ \text{for } y \in S \cdot x \rightarrow T \cdot y := T \cdot y + 1 \text{ rof.} \quad (140) \]

For completeness' sake, we expand statement \( T := \hat{0} \) \[ (= (T \mid V_l \hat{0}) \) using the function modification theorem from section 4.5, resulting in the statement

\[ \text{for } x \in V \rightarrow T \cdot x := 0 \text{ rof.} \quad (141) \]

Using results \[ (138), (139), (140) \] and \[ (141) \], we finally arrive at the sixth approximation, which is also the primitive algorithm:
The cyclicity test problem is now supposed to be solved. Note that the refinements have retained their expected complexity, so the algorithm is still \(O(N+M)\), and the implementation in [25] shows that a consistent representation can be found leaving the time complexity unchanged. It is believed that this is feasible for most primitive algorithms in programming practice.

7. General function computation problem: standard solution

7.1 Specification of the problem

The functional specification of the central problem of this paper reads

\[
\begin{align*}
I & : V_0, V_1 : P \cdot V ; T : V \rightarrow \mathbb{N} \\
V_0, V_1 & := V, V ; \\
\text{for } x \in V & \rightarrow T \cdot x := 0 \ \text{rof} ; \\
\text{for } x \in V & \rightarrow \\
\quad \text{for } y \in S \cdot x & \rightarrow T \cdot y := T \cdot y + 1 \ \text{rof} ; \\
\text{for } y \in V & \rightarrow \text{as } T \cdot y = 0 \rightarrow V_1 := V_1 \setminus \{y\} \ \text{sa } \text{rof} ; \\
\text{do } & V_0 \setminus V_1 \neq \emptyset \rightarrow \\
\quad \text{[ } & x : V \ | \\
\quad \quad x & \in V_0 \setminus V_1 ; V_0 := V_0 \setminus \{x\} ; \\
\quad \quad \text{for } y \in S \cdot x & \rightarrow T \cdot y := T \cdot y - 1 \ \text{rof} ; \\
\quad \quad \text{for } y \in S \cdot x & \rightarrow \text{as } T \cdot y = 0 \rightarrow V_1 := V_1 \setminus \{y\} \ \text{sa } \text{rof} \\
\text{] } & \\
\text{od} ; \\
cyc & := V_0 \neq \emptyset
\end{align*}
\]
with \( f \in V \rightarrow \mathbb{T} \), the function to be computed, being specified by the general recursive equations

\[
(V \cdot v :: f \cdot v = H \cdot (P \cdot v + f) \cdot v),
\]

(142)

with \( H \) an arbitrary, polymorphic mapping from functions and elements of \( V \) to \( \mathbb{T} \). In other words: \( f \cdot v \) is expressed in terms of (the function values of) \( v \)'s predecessors and \( v \) itself.

Often (142) is derived from an explicit definition of \( f \), unfolding it once or more. In agreement with the corresponding rule in 5.2, it turns out once again that the recursive definition is more convenient than the explicit one. In the present form, (142) may be interpreted as a fixed point equation for \( f \).

For cyclic graphs, (142) may not have a unique solution. In such cases, the equations are supplemented by an extra requirement, saying, for instance, that it is the least solution of (142) we are interested in, using some appropriate ordering [23]. This extra condition cannot be obtained by simply unfolding the explicit definition, but instead a fixed point theorem must be applied, like in section 5.2. The theorem below shows that such a heavy piece of equipment is not necessary for acyclic graphs. It generalizes a theorem in [26,27].

**Uniqueness Theorem:**

Recursive equations (142) have a unique solution for acyclic graphs.

**Proof:**

The proof that a solution exists is postponed to the next subsections, where it follows from the correctness of an algorithm constructing a solution. In order to prove uniqueness, we postulate the existence of two solutions \( f_0 \) and \( f_1 \) of (142). Hence our proof obligation becomes

\[
(V \cdot v :: f_0 \cdot v = f_1 \cdot v).
\]

(143)

Then we derive

(143)

\[
\text{§ Central cyclicity theorem: } E=\emptyset, \text{ so } V\in\neg E; \text{ definition of } E \text{ (116)}\}
\]

\[
(V \cdot v :: v \in \neg\{n \cdot k : 0 \leq k : S \cdot v\}; f_0 \cdot v = f_1 \cdot v)
\]

\[
\text{§ De Morgan; (32)}\}
\]
The latter predicate is most conveniently proved using induction on \( k \), with induction hypothesis

\[
\forall v: v \in S^k \cdot V : f_0 \cdot v = f_1 \cdot v.
\]

**Base:** For \( n=0 \) we have \( S^0 \cdot V = \emptyset \), so the domain is empty.

**Step:** Let \( u \in S^{n+1} \cdot V \). Then we derive

\[
\begin{align*}
u & \in S^{n+1} \cdot V \\
& \equiv \{\text{exponentiation (57); composition (53)}\}
\end{align*}
\]

\[
\begin{align*}
& u \neq S \cdot (S^n \cdot V) \\
& \equiv \{S/P \text{ property (65)}\}
\end{align*}
\]

\[
\begin{align*}
P \cdot u \cap S^n \cdot V = \emptyset \\
& \equiv \{\text{shunting (14)}\}
\end{align*}
\]

\[
P \cdot u \in S^n \cdot V.
\]

Combining this with induction hypothesis (144) we find

\[
\forall v: v \in P \cdot u: f_0 \cdot v = f_1 \cdot v,
\]

which, using (10), is equivalent to

\[
(P \cdot u \vdash f_0) = (P \cdot u \vdash f_1).
\]

Then for all \( u \in S^{n+1} \cdot V \) it follows that

\[
f_0 \cdot u
\]

\[
\begin{align*}
& \equiv \{(142) \text{ for } f_0\}
\end{align*}
\]

\[
\begin{align*}
& \vdash (P \cdot u \vdash f_0) \cdot u \\
& \equiv \{(146)\}
\end{align*}
\]

\[
\begin{align*}
& \vdash (P \cdot u \vdash f_1) \cdot u \\
& \equiv \{(142) \text{ for } f_1\}
\end{align*}
\]

\[
f_1 \cdot u.
\]
so this proves

\[(\forall u: u \in S^{n+1} \cdot V: f^0 \cdot u = f^1 \cdot u).\]  \hspace{1cm} (147)

\[\square\]

### 7.2 Division into subproblems

All the usual recursive equations, with \(f \cdot v\) expressed in a limited number of other \(f \cdot u, u \in V\), have a straightforward, well-known solution. Simply assign the explicitly given values of \(f \cdot v\) for \(v\) in the base collection, and then proceed, assigning \(f\) values that can be expressed in already known values, using the recursive equations, thus constructing \(f\) "bottom-up" in a natural way. Could there be such a natural order for solving the general problem on an acyclic graph? Assuming that this is indeed the case, let function \(h \in [0,N) \times V\) be used to denote this so-called topological ordering. It assigns to each number in the ordered interval \([0,N)\), with \(N = \#V\), a vertex in \(V\). If \(h\) is to be a proper ordering, it is of course mandatory that it is injective, so

\[(\forall p: 0 \leq p < N: h \cdot p \neq h(0,p)),\]  \hspace{1cm} (148)

where notation \(h(0,p)\) for \((i: 0 \leq i < p: h \cdot i)\) has been introduced for reasons of conciseness. Using the presumed topological ordering \(h\), and invariant

\[R_0: 0 \leq k < N \land (\forall p: 0 \leq p < k: f^0 \cdot (h \cdot p) = f \cdot (h \cdot p)),\]

the following algorithm is an obviously correct first approximation:

\[
k := 0;
\[\text{do } k \neq N \rightarrow \]
\[
x := h \cdot k; k := k + 1;
\[f^0 \cdot x := f \cdot x
\[\text{od}
\]

\{k = N, so with \(R_0\) and (148) postcondition \(R\) follows\},

where variable \(x\) has been introduced solely to reduce the amount of writing.

Trying to eliminate the unknown \(f\), we derive
\[ f \cdot x = \langle 142 \rangle \]
\[ \mathcal{H} \cdot (P \cdot x \cdot f) \cdot x \]
\[ = \langle \text{introduce restriction on } h \text{ below: } P \cdot x \subseteq h(0,k); \mathcal{R} \rangle \]
\[ \mathcal{H} \cdot (P \cdot x \cdot f_0) \cdot x , \]
resulting in the abstract assignment
\[ f_0 \cdot x := \mathcal{H} \cdot (P \cdot x \cdot f_0) \cdot x , \]
\[ (149) \]
\[ (150) \]
to be substituted into the algorithm, provided that \( h \) satisfies the following condition, which also incorporates (148):
\[ (V \ p : 0 \leq p < N : h(p) \notin h[0,p) \wedge P \cdot h(p) \subseteq h[0,p)) . \]
\[ (151) \]
The second, new conjunct expresses the fact that the predecessors of the \( k \)-th vertex in the order are all contained in the vertices preceding it in the order. The problem of determining a function \( h \) that satisfies (151) is well-known [3,7,16], and is called topological sorting. This problem will be addressed in section 7.4; for the time being, let \textit{topsort} be an algorithm that establishes (151). Note that the form of (151) obeys the important heuristic rule to \textit{formulate specifications as concisely and abstractly as possible.}

If \( \mathcal{H} \) is a computable function, then (150) may in principle be refined, until an executable (primitive or GCL) algorithm results, but of course this process of refining depends on the actual form of \( \mathcal{H} \). Just to give an example, assume that \( \mathcal{H} \) satisfies the so-called split property
\[ \mathcal{H} \cdot (g \cdot (u : c)) \cdot v = \mathcal{H} \cdot g \cdot v \circ \mathcal{H} \cdot c \cdot u \cdot v , \]
\[ (152) \]
for all \( g \in G \cdot T \), \( c \in T \), and \( u,v \in V \) with \( G \subseteq V \) and \( u \notin G \), for some operator \( \circ \) and mapping \( \mathcal{H} \). In (152) operator-independent lifting lifting is used, according to the lift-split rule (37) for disjoint domains. Many problems in practice turn out to satisfy property (152); a few examples are treated in section 8.

If \( \mathcal{H} \) satisfies split property (152), the expression accumulation theorem with initialization may be applied, using \( \mathcal{S} \cdot A = \mathcal{H} \cdot (A \cdot f_0) \cdot x \), for constant \( x \). Note that \( \mathcal{S} \cdot (P \cdot x) \) does not depend on the result parameter \( f_0 \cdot x \), because \( x \notin P \cdot x \): the graph is acyclic. From (41) we find \( \mathcal{S} \cdot \emptyset = \mathcal{H} \cdot \emptyset \cdot x \) for the initialization,
and the domain split rule (48) and split property (152) yield

\[ f \cdot (A^+(y)) = \mathcal{H} \cdot ((A \cdot f^0) \circ (y \cdot f^0 \cdot y)) \cdot x = f \cdot A \circ K \cdot (f^0 \cdot y) \cdot y \cdot x. \]  

(153)

Then application of the expression accumulation theorem immediately results in the following refinement for assignment (150):

\[
\begin{align*}
f^0 \cdot x &:= \mathcal{H} \cdot \mathcal{O} \cdot x; \\
\text{for } y \in P \cdot x &\rightarrow \\
f^0 \cdot x &:= f^0 \cdot x \circ K \cdot (f^0 \cdot y) \cdot y \cdot x
\end{align*}
\]

(154)

From (154) another problem becomes apparent: in the guard of the for statement \( P \cdot x \) is used, but from the specification in section 7.1 we read that only the successor function \( S \) is given. This problem not only pertains to this special case satisfying the split property, but is expected to arise in all straightforward refinements of (150), simply because \( P \cdot x \) appears as parameter on the right hand side. Instead of computing \( P \cdot x \) on the spot, we choose the more efficient solution of computing the predecessor function \( P \) in advance. This problem is also well-known [2], and referred to as graph inversion. A solution is derived in the next subsection; until then, let \text{invert} denote an algorithm that establishes \( P^0 = P \).

Picking up the pieces, the second approximation to the central problem is

\[
\begin{align*}
\text{if } &k: \mathbb{N}; x: V; \\
h: [0,\#V) \rightarrow V; P^0: V \rightarrow P \cdot V \mid \\
\text{invert}; &\quad \langle P^0 = P \rangle \\
\text{topsort}; &\quad \langle \{ 151 \} \rangle \\
k:=0; \\
\text{do } &k \neq \mathbb{N} \rightarrow \\
\quad &x := h \cdot k; k := k + 1; \\
f^0 \cdot x &:= \mathcal{H} \cdot (P^0 \cdot x \circ f^0) \cdot x
\end{align*}
\]

The order of \text{invert} and \text{topsort} is arbitrary. Note that the above algorithm terminates if \text{topsort} does, in which case a solution to (142) exists: \( f^0 \). This matter will be settled in section 7.4.
7.3 Graph inversion

The graph inversion problem, emerging from the general solution to the central problem, also pops up in many other problems, where a derivation leads to an algorithm in terms of $P$, instead of $S$, which is not uncommon. It is also interesting by itself, because the derivation of its solution offers a nice example of what abstraction can accomplish in combination with a few simple theorems from the earlier sections.

In [2], Dijkstra and Feijen present a solution to this problem, without giving invariants, because, as they say, they do not help understanding. Indeed, they are right, as long as one sticks to the level of abstraction they use, with integers and arrays as data types. The story changes drastically if the problem is reformulated in terms of sets and functions.

The desired postcondition $PO = P$ is established by the first approximation to the algorithm:

$$PO := P.$$ 

Trivial as it may seem, it provides an easy starting point for application of the expression accumulation theorem, if we expand $P$, using (7):

$$P = (\lambda v : v \in V : P \cdot v).$$

(155)

The right-hand side expression can be interpreted as an instance of several possible accumulation functions, found by replacing a constant set by a set parameter $A$. Replacing $V$ in the domain of the $\lambda$-quantifier, the most obvious thing to do, turns out to lead to an inefficient solution. But then, noting that $X = X \cap V$ for all $X$, an alternative presents itself: the replacement of the (invisible) $V$ in the term of the $\lambda$-quantifier. In this way mapping $\mathcal{G}$ (128) may act as accumulation function. In section 6.2 we found $\mathcal{G} \cdot \mathcal{G} = \mathcal{G}$, as well as split property (131). Application of the expression accumulation theorem with initialization (section 4.4) consequently yields the second approximation:

$$PO := \mathcal{G};$$

for $x \in V \rightarrow PO := PO + S \cdot x : \{x\}$ ref.
After transformation of \((S \cdot x: (x))\) into \((S \cdot x+: (x)^+)\), using (9) and (10) and notation \((x)^+\) for \((V: \{x\})\), the lift accumulation theorem from section 4.5 directly provides us with the following refinement of the assignment in the body of the repetition:

\[
\text{for } y \in S \cdot x \rightarrow P_0 \cdot y := P_0 \cdot y + (x) \text{ rof}.
\]

Notice that here we profit from the continued calculation with the set + in (131): had it been replaced in any step by its more general counterpart \(U\), we would have been left with additional work here, making \(P_0 \cdot y U \{x\}\) primitive.

Finally, the initialization of \(P_0\) can be taken care of using the function modification theorem with \(\hat{\delta} = (P_0 | V \cdot \hat{\delta})\), and we get the primitive algorithm as third approximation:

\[
\begin{align*}
\text{for } x \in V &\rightarrow P_0 \cdot x := \emptyset \text{ rof}; \\
\text{for } x \in V &
\quad \text{for } y \in S \cdot x \rightarrow P_0 \cdot y := P_0 \cdot y + (x) \text{ rof} \\
&\text{rof}.
\end{align*}
\]

Assuming that all primitive operations are \(O \cdot 1\), which is realistic, it is easily seen that this solution is \(O \cdot (N+M)\), with \(N=\#V\) and \(M\) defined in (135).

The solution to the graph inversion problem and its derivation turned out to be trivial. However, it is only fair to say that it is not the whole story, if one uses the representation of Dijkstra and Feijen [2]. The implementation phase being beyond the scope of this paper, the reader is referred to [20] for a further treatment, where the present algorithm serves as an example. Suffice it to say here that the implementation in GCL, though a little elaborate, is as standard as the derivation of the primitive algorithm above.

### 7.4 Topological Sorting

In this section an algorithm will be derived, computing a function \(h\) that satisfies (151). We use the well-known technique of replacing a constant by a variable [2], in order to obtain the obvious invariants

\[
\begin{align*}
P_0: &\quad 0 \leq k \leq N, \\
P_1: &\quad (\forall p: 0 \leq p < k: h \cdot p \notin h[0,p) \land P \cdot (h \cdot p) \leq h[0,p])
\end{align*}
\]
for the simple repetition, and first approximation

\[ k := 0; \]
\[ \textbf{do} \quad k \neq N \rightarrow \]
\[ \begin{align*}
  & x := x \notin h[0, k) \land P \cdot x \leq h[0, k); \\
  & h \cdot k := x; k := k + 1
\end{align*} \]
\[ \textbf{od}. \]

Next, let us rewrite the selection predicate into a more manageable form:

\[ x \notin h(0, k) \land P \cdot x \leq h[0, k) \]
\[ \equiv \{ \text{complement; shunting (14)} \} \]
\[ x \in \neg h(0, k) \land P \cdot x \cap h(0, k) = \emptyset \]
\[ \equiv \{ \text{introducing new variable } V_0, \text{ and invariant } P_2 \text{ (see below)} \} \]
\[ x \in V_0 \land P \cdot x \cap V_0 = \emptyset \]
\[ \equiv \{ \text{S/P rule (65)} \} \]
\[ x \in V_0 \land x \notin S \cdot V_0 \]
\[ \equiv \{ \text{calculus} \} \]
\[ x \in V_0 \setminus S \cdot V_0, \]

so with the additional invariant

\[ P_2: \quad V_0 = \neg h[0, k) \]

the selection of \( x \) can be written very compactly as \( x \in V_0 \setminus S \cdot V_0 \). The question is whether or not this selection will abort. Inspection of the guard of the repetition teaches us that

\[ k \neq N \equiv V_0 \neq \emptyset \quad \text{(156)} \]

holds, due to \( P_2 \) and the injectivity of \( \cdot \) (in \( P_1 \)). Unfortunately, \( V_0 \neq \emptyset \) does not appear strong enough to guarantee non-abortion of the selection statement. Strengthening the guard of the repetition to \( V_0 \setminus S \cdot V_0 \neq \emptyset \) offers an easy way out, but then we must check whether the postcondition still holds. We derive

\[ V_0 \setminus S \cdot V_0 = \emptyset \]
\[ \equiv \{ \text{set calculus} \} \]
S \cdot V_0 \supseteq V_0
\Rightarrow \{ E \text{ is the least prefixed point of } S \text{ (115), for ordering } \geq \}
E \supseteq V_0
\equiv \{ \text{acyclic graph, } E=\emptyset \text{ with (119)} \}
V_0 = \emptyset
\Rightarrow \{ \text{set calculus} \}
V_0 \setminus S \cdot V_0 = \emptyset, \tag{157}

where once again we use the central cyclicity theorem from 5.3 to advantage. From "ping-pong" derivation (157) we conclude

\begin{equation}
V_0 \neq \emptyset \equiv V_0 \setminus S \cdot V_0 \neq \emptyset. \tag{158}
\end{equation}

Both guards are equivalent after all! Thus we obtain the second approximation:

\begin{verbatim}
k:=0; V_0:=V;
do V_0\neq\emptyset 
x:= V_0 \setminus S \cdot V_0; V_0:= V_0 \setminus \{x\};
\hfill h \cdot k:= x; k:= k+1
od.
\end{verbatim}

Finally, if we decide to simplify the selection statement by introducing a new set variable \( V_1 \), with invariant

\( P_2: V_1 = S \cdot V_0, \)

and make the necessary substitutions, we come to the striking conclusion that, apart from the assignments to \( h \) and \( k \), we have exactly the third approximation for the cyclicity test in section 6.1, with the same invariants! Hence, the assignments to \( V_1 \) can be made primitive in exactly the same way, and we only have to insert the assignments to \( h \) and \( k \) in the primitive algorithm derived in section 6.3. Consequently, the resulting algorithm is also \( O \cdot (N+M) \).

At the same time this means that a cyclicity test prior to \textit{topsort} is never needed: the value of \( V_0 \), being equal to \( E \) upon termination of the repetition, may serve to test for success in finding a topological order. If \( V_0 = \emptyset \), the graph is acyclic, and postcondition (151) holds, if \( V_0 \neq \emptyset \) the graph contains cycles, and a topological order does not exist.
7.5 Streamlining the solution: several alternatives

The general solution of the function computation problem is obtained by substituting the resulting algorithms for *invert* and *topsort* from sections 7.3 and 7.4 into the second approximation in 7.2. Note that the existence of a topological order for acyclic graphs guarantees the existence of a solution, \( f_0 \), thus completing the proof for the uniqueness theorem of section 7.1.

A slight optimization of the general solution is effectuated as follows. Why first calculate a topological order, storing it in \( h \), and next use \( h \), vertex by vertex? According to (156), the repetition following *topsort* has the same guard, so they can be merged, making \( h \) and \( k \) obsolete. The result is

\[
V_0,V_1 := V,S \cdot V;
\]

\[
do V_0 \neq \emptyset \rightarrow \\
\quad x \in V_0 \setminus V_1; \; V_0 := V_0 - \{x\}; \; V_1 := S \cdot V_0;
\]

\[
f_0 \cdot x := \#(P_0 \cdot x < f_0) \cdot x
\]

(159)

One may wonder if it is possible to avoid the graph inversion, too, if we assume that the graph is represented by the predecessor function \( P \) to begin with. This would not seem too drastic a change of perspective, considering that (142) specifies the function itself in terms of the graph's predecessor function. But this implies that also the references to \( S \) in *topsort*, or *cyclicitytest*, must be eliminated. This can be done by dualizing *topsort*. In order to accomplish this we try to reformulate its postcondition (151) in terms of \( S \). The first conjunct (injectivity) is rewritten in a straightforward way, and for the second one we derive

\[
(\forall p: 0 < p < N: P \cdot (h \cdot p) \leq h(0,p))
\]

\[= \{ \text{calculus} \}
\]

\[
(\forall p: 0 < p < N: (\forall v: v \in P \cdot (h \cdot p): v \in h(0,p)))
\]

\[= \{ \text{change dummy } v := h \cdot q, \text{ injectivity of } h \}
\]

\[
(\forall p: 0 < p < N: (\forall q: 0 < q < N \land h \cdot q \in P \cdot (h \cdot p): 0 < q < p))
\]

\[= \{ \text{interchange quantifiers; (58)} \}
\]

\[
(\forall q: 0 < q < N: (\forall p: 0 < p < N \land h \cdot p \in S \cdot (h \cdot q): q < p < N))
\]

\[= \{ \text{retrace first two steps in reverse order} \}
\]

\[
(\forall q: 0 < q < N: S \cdot (h \cdot q) \leq h(q+1,N)),
\]
hence (151) turns out to be equivalent to

\[(\forall p: 0 \leq p < N: h \cdot p \not\in h[p+1,N] \land S \cdot (h \cdot p) \subseteq h[p+1,N])\].

(160)

An algorithm establishing the latter postcondition is derived in an analogous way as the one in section 7.4. This time, however, the repetition proceeds in the direction of decreasing \(k\). We find the following algorithm:

\[
k := N; \ V0 := V; \ V1 := P \cdot V0;
\]

\[
do \ V0 \not= \emptyset \rightarrow \odo;
\]

\[
\begin{align*}
x & \in V0 \setminus V1; \ V0 := V0 \setminus \{x\}; \ V1 := P \cdot V0; \\
h \cdot (k-1) & := x; \ k := k - 1
\end{align*}
\]

Though apparently a different algorithm altogether, it still computes the same topological orders (both algorithms are nondeterministic, of course).

Now both topological sort and the calculation of \(f\) are expressed in terms of \(P\). Unfortunately, this time the optimization of merging both algorithms is inhibited: calculating and using \(h\) is carried out in opposite directions, so the less elegant two-step algorithm is inevitable, and \(h\) cannot be eliminated. From the point of view of memory requirements, one might prefer solution (161) preceded by \texttt{topsort} to (159) preceded by \texttt{invert}: the latter requires \(O \cdot M\) extra storing space, due to auxiliary variable \(P0\), with \(M = O \cdot N^2\) worst case.

8. Special solution for disjunctive fixed points

8.1 General derivation

In many cases the problems with preprocessing algorithms, requiring extra storing space, mentioned in section 7.5, can be avoided. For instance, if mapping \(\mathcal{F}\) satisfies split property (152), it is possible to keep track of the contribution of a vertex to the function values of all its successor vertices "in advance", instead of calculating the function values from all the known values of the predecessor vertices "in retrospect".

Trying to establish \(f0 = f\), with \(f\) satisfying (142), we approximate \(f\) by a
function $f_0 \in V \rightarrow \mathbb{F}$, equal to $f$ in those vertices outside a shrinking set $V_0$. Contrary to the previous section, we do not yet know in which order vertices should be removed from $V_0$, in the absence of an available topological order, but we expect this to follow from the derivation. So, with invariants

$\mathcal{P}_0$: $V_0 \subseteq V$,
$\mathcal{P}_1$: $(\forall i: i \in V_0: f_0 \cdot i = f \cdot i),$  

the first approximation reads

$V_0 := V$;
$\text{do } V_0 \neq \emptyset \rightarrow$
$\quad x \in V_0; V_0 := V_0\{x\};$
$\quad f_0 \cdot x := f \cdot x$
$\text{od.}$

Next the unknown quantity $f \cdot x$ must be eliminated from the algorithm:

$f \cdot x$
$= \{(142)\}$
$= \mathcal{H} \cdot (P \cdot x \uparrow f) \cdot x$
$= \{\text{in view of } \mathcal{P}_1: \text{choose } x \text{ such that } P \cdot x \cap V_0 = \emptyset, \text{ so } P \cdot x = P \cdot x \setminus V_0\}$
$= \{\text{introduce } \mathcal{P}_2 \text{ below, and variable } f_1 \in V \rightarrow \mathbb{F}\}$
$= f_1 \cdot x,$

(162)

provided that $P \cdot x \cap V_0 = \emptyset$, and, generalizing the last step to be valid for all $v: v \in V$, we are led to the additional invariant

$\mathcal{P}_2$: $(\forall v: f_1 \cdot v = \mathcal{H} \cdot (P \cdot v \setminus V_0 \uparrow f) \cdot v).$

According to (65), condition $P \cdot x \cap V_0 = \emptyset$ is equivalent to $x \notin S \cdot V_0$, hence it seems useful to introduce another invariant

$\mathcal{P}_3$: $V_1 = S \cdot V_0,$

replace the selection by $x \in V_0 \setminus V_1$, and strengthen the guard to $V_0 \setminus V_1 \neq \emptyset$. Then we use (41) to initialize $f_1$, and obtain the second approximation:
In the initialization of $f_1$ the second parameter is understood to be curried. Of course the introduction of $f_1$ forces us to adjust it, which is performed in $SL(f_1)$. Since in principle $f_1 \cdot x$, required in the assignment to $f_0 \cdot x$, may be changed by $SL(f_1)$, its value is temporarily stored in auxiliary variable $h_{fl}$. Note furthermore that the order of the statements is chosen in agreement with the heuristic rule at the end of section 6.1: the adjustment of $f_1$ before, and the assignment of $V_1$ after $V_0 := V_0 -(x)$; the expression for $f_1$ behaves nicely when $V_0$ is replaced by $V_0 -(x)$, to be shown below, whereas $S' (V_0 +(x))$ is easier to be manipulated than $S \cdot (V_0 -(x))$, as shown in section 6.1.

Because of the fact that we have strengthened the guard, we have to check whether the postcondition still holds. Fortunately, this follows from the corresponding derivation (157) for topsort in section 7.4, and is again based on the acyclicity of the graph, and the central cyclicity theorem in 5.3.

Once again we observe that the heart of the algorithm is identical to the cyclicity algorithm in section 6.1, hence we skip further refinement of the assignments to $V_1$. The assignments to $f_1$ remain. The initialization cannot be refined any further, without additional knowledge of $H$. Calculating the expression to be assigned to $f_1$ in the repetition we find, for each $v \in V$:

$$
H \cdot (P \cdot v \setminus (V_0-(x)) \vdash f') \cdot v
$$

= \langle set calculus rule (23) \rangle

$$
H \cdot (P \cdot v \setminus V_0 + P \cdot v \in(x) \vdash f') \cdot v
$$

= \langle case analysis; S/P-calculus (58) \rangle

if $v \in S \cdot x \rightarrow H \cdot (P \cdot v \setminus V_0 + (x) \vdash f') \cdot v$

0 $v \notin S \cdot x \rightarrow H \cdot (P \cdot v \setminus V_0 + f') \cdot v$ $f_1$

= \langle domain split (48); (162) implies $f \cdot x = f_1 \cdot x = h$ \rangle

if $v \in S \cdot x \rightarrow H \cdot ((P \cdot v \setminus V_0 + f) \cdot (x; h)) \cdot v$

0 $v \notin S \cdot x \rightarrow H \cdot (P \cdot v \setminus V_0 + f) \cdot v$

$f_1$
\[
= \{\text{split property (152); } P2\}
\]
\[
\begin{align*}
& \text{if } v \in S \cdot x \Rightarrow f_1 \cdot v \oplus K \cdot h \cdot x \cdot v \\
& 0 \quad v \notin S \cdot x \Rightarrow f_1 \cdot v \\
& f_1.
\end{align*}
\] (163)

According to the lift–include rule (40), applied to (163), \( SL(f_1) \) refines to
\[
f_1 := f_1 \circ (S \cdot x \oplus K \cdot h \cdot x),
\] (164)

again currying mapping \( K \). Applying the lift accumulation theorem of section 4.5 to (164), we find that this assignment in turn is refined by
\[
\text{for } y \in S \cdot x \Rightarrow f_1 \cdot y := f_1 \cdot y \oplus K \cdot h \cdot x \cdot y \quad \text{ref}
\] (165)

From this repetition we conclude that the value of \( f_1 \cdot x \) is not changed after all: in an acyclic graph \( x \notin S \cdot x \) holds for every \( x \). Auxiliary variable \( h \) can consequently be eliminated.

Another optimization follows from a close examination of \( P2 \): we find that upon termination \( V_0 = \emptyset \), so \( f_1 \cdot v = K \cdot (P \cdot v \oplus f) \cdot v = f \cdot v \), according to (142), so \( f_1 \) equals \( f \), too. Since nowhere in the algorithm \( f_0 \) is referenced, \( f_0 \) can be dismissed altogether. In fact, the algorithm could have been developed right from the start without using \( f_0 \) or invariant \( P1 \) at all. We chose not to, because the present derivation appears to be heuristically better.

Substituting (165) into the second approximation, and carrying out the abovementioned optimizations, the third approximation emerges:
\[
V_0, V_1 := V, S \cdot V; \quad f_1 := K \cdot \emptyset;
\]
do \( V_0 \setminus V_1 \neq \emptyset \) →
\[
\begin{align*}
& x \in V_0 \setminus V_1; \\
& \text{for } y \in S \cdot x \Rightarrow f_1 \cdot y := f_1 \cdot y \oplus K \cdot (f_1 \cdot x) \cdot x \cdot y \quad \text{ref}; \\
& V_0 := V_0 \setminus \{x\}; \quad V_1 := S \cdot V_0
\end{align*}
\] od
\[\{f_1 = f\}.
\]

Finally, we use the results of section 6 to refine the operations on \( V_0 \) and \( V_1 \). It then turns out that only the set difference \( V_0 \setminus V_1 \) influences the flow of the algorithm, so \( V_0 \) and \( V_1 \) separately have become superfluous, assuming that \( V_0 \) is no longer needed to check the cyclicity of the graph. In the fourth
approximation below, which is the primitive algorithm, we have therefore introduced a variable \( V_2 = V_0 - V_1 \), eliminated \( V_0 \) and \( V_1 \), and performed some loop fusion.

\[
\begin{align*}
\text{if} & \quad V_2: P \cdot V; T: V \rightarrow \mathbb{N} \mid V_2 := \emptyset; \\
\text{for} & \quad x \in V \rightarrow T \cdot x := 0 \quad \text{rof}; \\
\text{for} & \quad x \in V \rightarrow \text{for} \quad y \in S \cdot x \rightarrow T \cdot y := T \cdot y + 1 \quad \text{rof} \quad \text{rof}; \\
\text{for} & \quad x \in V \rightarrow \text{as} \quad T \cdot x = 0 \rightarrow V_2 := V_2 + \{x\} \quad \text{sa}; \quad f_1 \cdot x := K \cdot \phi \cdot x \quad \text{rof}; \\
\text{do} & \quad V_2 \neq \emptyset \rightarrow \\
\text{do} & \quad x \in V \mid x \ll V_2; V_2 := V_2 - \{x\}; \\
\text{for} & \quad y \in S \cdot x \rightarrow \\
\quad & \quad f_1 \cdot y := f_1 \cdot y \ll K \cdot (f_1 \cdot x) \cdot x \cdot y; \\
\quad & \quad T \cdot y := T \cdot y - 1; \quad \text{as} \quad T \cdot y = 0 \rightarrow V_2 := V_2 + \{y\} \quad \text{sa} \quad \text{rof}; \\
\text{od} \quad \text{rof} \quad \text{rof}; \\
\text{od} \quad \text{rof}.
\end{align*}
\]

Implementing this algorithm, using only low-level data structures, is straightforward; the reader is referred to [20]. Suffice it to remark here that all sets \( S \cdot x \) may be stored as array segments, consecutively if necessary, and \( V_2 \) is represented by a single array segment, behaving like a queue. If we choose this representation, the worst case time complexity is \( O(N+M) \), as expected. This efficiency cannot be improved upon if all values \( f \cdot v \), \( v \in V \), are to be computed. The computation of a single value (or a limited number of values) may be performed considerably faster in special cases. Take for instance the calculation of the \( N \)-th fibonacci number in \( O(\log N) \) time.

8.2 Predecessor sums and finite two-person games

Consider the following simple problem on an acyclic graph: function value \( f \cdot v \) in each vertex \( v \in V \) is equal to the sum of function values of \( v \)'s predecessors, except in those vertices without predecessors, where \( f \cdot v \) equals \( a \cdot v \), for given function \( a \in V \rightarrow \mathbb{N} \). This function \( f \) can be interpreted as a generalization of the fibonacci function, where each function value is the sum of exactly two (special) predecessor values. In the present example, the recursive equation for \( f \) reads
\[ f \cdot v = \begin{cases} P \cdot v=\emptyset & a \cdot v \\ \emptyset & P \cdot v\neq\emptyset \rightarrow (\Sigma w: w \in P \cdot v: f \cdot w) \\ \end{cases}, \] 

\[ H \cdot (P \cdot v=\emptyset \cdot v) = H \cdot (P \cdot v=\emptyset \cdot v), \quad (166) \]

for all \( v: v \in V \). Some possible choices for \( H \) are

\[ H \cdot g \cdot v = \begin{cases} P \cdot v=\emptyset & a \cdot v \\ \emptyset & P \cdot v\neq\emptyset \rightarrow (\Sigma w: w \in d \cdot g: g \cdot w) \\ \end{cases}, \]

\[ H \cdot g \cdot v = \begin{cases} P \cdot v=\emptyset & a \cdot v \\ \emptyset & P \cdot v\neq\emptyset \rightarrow (\Sigma w: w \in d \cdot g: g \cdot w) \\ \end{cases}, \]

\[ H \cdot g \cdot v = \begin{cases} P \cdot v=\emptyset & a \cdot v \\ \emptyset & P \cdot v\neq\emptyset \rightarrow 0 \end{cases} + (\Sigma w: w \in d \cdot g: g \cdot w), \quad (169) \]

for all functions \( g \in G \to \mathbb{T} \). These definitions are all distinct, but coincide when instantiated with \( g = (P \cdot v=\emptyset \cdot f) \). The choice between these alternatives is not arbitrary, but is governed by the form of the resulting split property. It is easy to check that \( H \) in (167) does not satisfy any split property. Definition (168) leads to a split property with a case analysis in split function \( H \), and, finally, (169) yields the most elegant form. This can be expected in advance, because the case analysis of (169) is \( g \)-independent, and the \( g \)-dependent quantifier is a separate term, allowing a simple domain split. Contrary to this, in (168) the \( g \)-dependence appears in one of the alternatives, and in (167), even worse, in the guards. Accordingly, the following rule of thumb for such a choice is formulated: localize and restrict the function dependence of the fixed point operator as much as possible. Note that the \( g \)-dependence in the domain of the quantifier is indispensable for guarding the appearance of \( g \) in the term against illegal parameter values.

The remainder will be based on definition (169). Needless to say, that (168) will eventually lead to the same algorithm. First we derive the form of split property (152):

\[ H \cdot (g \cdot (u: c)) \cdot v \]

\[ = \langle(169)\rangle \]

\[ = \begin{cases} a \cdot v & P \cdot v=\emptyset \\ \emptyset & P \cdot v\neq\emptyset \rightarrow 0 \end{cases} + (\Sigma w: w \in d \cdot (g \cdot (u: c)): (g \cdot (u: c)) \cdot w) \]

\[ = \begin{cases} a \cdot v & P \cdot v=\emptyset \\ \emptyset & P \cdot v\neq\emptyset \rightarrow 0 \end{cases} + (\Sigma w: w \in d \cdot (g \cdot (u: c)): (g \cdot (u: c)) \cdot w) + (g \cdot (u: c)) \cdot u \]

\[ = \begin{cases} a \cdot v & P \cdot v=\emptyset, (37), (169) \end{cases} \]
Evidently, Φ is instantiated by +, and $\kappa \cdot \mathbf{u} \cdot v = c$. Substituting this into the fourth approximation of the algorithm in section 8.1 the adjustment of $f_1 \cdot y$ turns out to be

\[
\begin{align*}
\text{f}_1 \cdot y &= f_1 \cdot y + f_1 \cdot x. 
\end{align*}
\]

Calculating the initial expression for $f_1$ we find

\[
\begin{align*}
\mathcal{H} \cdot \varnothing \cdot x &= \{169\}\} \\
&= \{ \begin{align*}
\text{if } P \cdot x = \varnothing &\rightarrow a \cdot x \\
\text{if } P \cdot x \neq \varnothing &\rightarrow 0 \text{ f}_1 + (\Sigma w: w \in d \cdot \varnothing; \varnothing \cdot w) \\
&= \{ \text{initial value } T \cdot x = \#(P \cdot x \cap V) = \#P \cdot x \ (P3, \text{section 6.2); empty domain}\}
\end{align*} \\
&= \{ \begin{align*}
\text{if } T \cdot x = 0 &\rightarrow a \cdot x \\
\text{if } T \cdot x \neq 0 &\rightarrow 0 \text{ f}_1, 
\end{align*} \}
\end{align*}
\]

implying that in the initialization the assignment to $f_1 \cdot x$ can be replaced by

\[
\begin{align*}
\text{if } T \cdot x = 0 &\rightarrow f_1 \cdot x := a \cdot x \\
\text{if } T \cdot x \neq 0 &\rightarrow f_1 \cdot x := 0 \\
\text{f}_1 ,
\end{align*}
\]

and the assignment in the preceding as-statement may be included in the first alternative, if one so desires. This completes the solution.

Another simple, recursively defined problem arises if we consider finite, two-person games. In those games two persons in turn make a move, starting in some given game position, and ending if no move is possible, in which case the last person to move is said to have won. Such games may be modelled by a directed acyclic graph, where the vertices represent the game positions, and the predecessor function $P$ yields the possible game positions reachable from a given position. The finiteness of the game implies acyclicity of the graph.

A position is obviously won if and only if one of the predecessor positions is lost. In particular, a position with no possible moves, i.e. a vertex without predecessors, is lost. If we represent a winning position by vertex label true, and a losing position by vertex label false, we are asked to compute a function $f \in V \cdot \mathcal{B}$ satisfying

\[
f \cdot x = (\exists w: w \in P \cdot x: \neg f \cdot w).
\]
This recursive definition almost inevitably yields
\[ \mathcal{H} \cdot g \cdot v = (\exists w : w \in d \cdot g ; \neg f \cdot w) \]
as definition for \( \mathcal{H} \), and a simple calculation shows that it satisfies split property and empty function property
\[ \mathcal{H} \cdot (g \cdot (u: c)) \cdot v = \mathcal{H} \cdot g \cdot v \cdot \neg c, \text{ and} \]
\[ \mathcal{H} \cdot \emptyset \cdot v = \text{false}, \]
respectively, allowing the reader to construct the corresponding algorithm in a straightforward way.

In [14] a generalization to infinite, possibly cyclic graphs is treated, including undecided positions, using an explicit definition of winning and losing positions. That rather informal definition, however, is questionable, because it is based on game strategies defined as functions \( S \cdot V \rightarrow V \), whereas a move in a strategy from some position may in principle be different each time that same position arises. The recursive treatment above looks more lucid.

8.3 Ancestor maxima

A less straightforward problem arises if the function to be computed is not specified by a fixed point equation, but instead by an explicit expression. For example, given a constant function \( a \in V \rightarrow \mathbb{Z} \), we are asked to compute a function \( f \in V \rightarrow \mathbb{Z} \) on an acyclic graph, such that each \( f \cdot x \) is the maximum over the \( a \)-values of all \( x \)'s ancestors; so, \( f \cdot x \) is, for all \( x \in V \), defined by
\[ f \cdot x = (\uparrow u : u \in P \cdot x ; a \cdot u). \quad (172) \]

According to the dual of (66), \( P \) in the right hand side of (172) is an infinite union, but this should not discourage us: we encountered a similar problem in definition (116) of \( E \). However, contrary to the case in section 5.2, application of a fixed point theorem is not necessary here. This is due to the uniqueness theorem for acyclic graphs, proven in section 7.1.

Explicit equations like (172) can often be transformed into recursive equations by unfolding the definition once (or more). Thus we derive
\[ f \cdot x = \{(172)\} \]
\[ = \{(68); (69); \text{domain split}\} \]
\[ = \{(59) \text{ applied to } P^*\} \]
\[ = \{(32); \text{generalized domain split}\} \]
\[ = \{(172)\} \]
\[ = \{(173)\} \]

and thus obtain the desired recursive equation, like (166) for predecessor sum. By the way, note that (166) does not have a simple explicit counterpart to (172). The apparent candidate of \( f \cdot x \) being the sum of all source ancestor values \( a \cdot u \) will not do: the domain split in the last-but-one step of (173) is not valid, because the sum quantifier requires the domains to be disjoint.

From (173) the definition of \( H \) is obvious:

\[ H \cdot g \cdot v = a \cdot v \uparrow (\uparrow w: w \in d \cdot g: g \cdot w) . \] (174)

The corresponding empty function and split property are easily found to be

\[ H \cdot \emptyset \cdot v = a \cdot v, \] (175)
\[ H \cdot (g \cdot (u; c)) \cdot v = H \cdot g \cdot v \uparrow c, \] (176)

and from these the assignment to \( fl \cdot x \) in the initialization and the adjustment of \( fl \cdot y \) in the repetition body can be deduced, resulting in

\[ fl \cdot x := a \cdot x, \] (177)
\[ fl \cdot y := fl \cdot y \uparrow fl \cdot x, \] (178)

respectively. Substituting (177) and (178) into the fourth approximation in section 8.1, one obtains the desired solution.

An interesting variation of this problem is the computation of the maximum over all labels \( t \) of ancestor edges of a given vertex. In that case each edge
\( <w,u> \) is labelled \( t \cdot w \cdot u \), where \( t \in V \times V \times Z \) is a given function. The explicit definition of \( f \) is then given by

\[
f \cdot x = (\uparrow u, w: u \in P^* \cdot x \wedge w \in P \cdot u: t \cdot w \cdot u),
\]

As (179) can be rewritten into the form

\[
f \cdot x = (\uparrow u: u \in P^* \cdot x: (\uparrow w: w \in P \cdot u: t \cdot w \cdot u)),
\]

it turns out to be just an instance of (172), with \( a \cdot u \) defined by

\[
a \cdot u = (\uparrow w: w \in P \cdot u: t \cdot w \cdot u).
\]

Substituting this into (173) we find

\[
f \cdot x = (\uparrow w: w \in P \cdot x: t \cdot w \cdot x) \uparrow (\uparrow w: w \in P \cdot x: f \cdot w) = (\uparrow w: w \in P \cdot x: t \cdot w \cdot x \uparrow f \cdot w),
\]

leading to

\[
H \cdot g \cdot v = (\uparrow w: w \in d \cdot g: t \cdot w \cdot v \uparrow g \cdot w)
\]

as definition for \( H \). From this we derive split property

\[
H \cdot (g \cdot (u: c)) \cdot v = H \cdot g \cdot v \uparrow t \cdot u \cdot v \uparrow c,
\]

so we take \( K \cdot c \cdot u \cdot v = t \cdot u \cdot v \uparrow c \), and furthermore we have

\[
H \cdot \phi \cdot v = \phi.
\]

Finding the corresponding statements is not difficult and left as an exercise to the reader.

Instead of the present choice for \( H \), we could have taken the first alternative of (182) as our starting point, replacing \( P \cdot x \) only in the second maximum by \( d \cdot g \). Our rule of thumb to localize the \( g \)-dependence is not specific about this: in both cases it is limited to two instances. On the other hand, the first expression is more complicated than the second, in favour of our choice. One could say that striving for simplicity is the heuristic rule "par
excellence". The reader may verify that the other choice in this case leads to a slightly simpler split property for $H$, but a significantly more complicated initialization.

8.4 Minimal and maximal distances in a graph

Now we turn to the collection of well-known problems [2,3,7,8,16] about minimal and maximal distances in a graph, from or to a given set, a source or a sink, with or without labelling the edges. We choose a few representative examples from the collection.

The first problem we call shortest path from a source, and the function to be computed is for each vertex $x$ equal to the minimal number of edges on any path from a source to $x$, in an acyclic graph. Defining source set $P\emptyset$ by

\[ P\emptyset = \{v: P\cdot v = \emptyset : v\} , \]

an explicit expression for $f \in V \rightarrow \mathbb{N}$, the function to be computed, can be found, using the path length interpretation in (72). It reads

\[ f \cdot x = \left\langle \left\langle (\downarrow k: 0 \leq k \land x \in S^k \cdot P\emptyset : k) \right\rangle \right\rangle \]

Immediately we derive the corresponding recursive equation:

\[ f \cdot x = \left\langle \left\langle (\downarrow k: 0 \leq k \land x \in S^k \cdot P\emptyset : k) \right\rangle \right\rangle \]

which, bearing in mind the heuristic rule from section 8.2, results in
as definition for \( \mathcal{H} \). From (189) we derive split and empty function property:

\[
\begin{align*}
\mathcal{H} \cdot (g \cdot (u \cdot c)) \cdot v &= \mathcal{H} \cdot g \cdot v \cdot (c + 1), \\
\mathcal{H} \cdot \emptyset \cdot v &= (\omega \mid P \emptyset; 0) \cdot v.
\end{align*}
\]

(190)

(191)

The corresponding adjustment of \( \mathcal{H} \cdot Y \) is evident, and for the initialization of \( \mathcal{H} \cdot x \) we have

\[
\begin{align*}
\text{if } T \cdot x = 0 & \rightarrow \mathcal{H} \cdot x := 0 \\
\text{if } T \cdot x \neq 0 & \rightarrow \mathcal{H} \cdot x := \omega
\end{align*}
\]

(192)

again making use of the initial value \#P of variable \( T \).

The similar problem of the longest path from a source is specified by

\[
f \cdot x = (\uparrow k: 0 \leq k \land x \in S^k \cdot P \emptyset; k),
\]

(193)

analogously to (187). It would not be worth mentioning separately, if it were not for some interesting new aspects, to be investigated below.

Firstly, it is possible to eliminate the source set \( P \emptyset \) and to simplify the explicit definition to

\[
f \cdot x = (\uparrow k: 0 \leq k \land P^k \cdot x \neq \emptyset; k),
\]

(194)

which follows from the derivation

\[
(\uparrow k: 0 \leq k \land P^k \cdot x \neq \emptyset; k)
= \{\text{generalized S/P property (65)}\}
(\uparrow k: 0 \leq k \land x \in S^k \cdot V; k)
= \{V = P \emptyset \cup \neg P \emptyset; \text{disjunctivity of } S^k (62)\}
(\uparrow k: 0 \leq k \land x \in S^k \cdot P \emptyset \cup S^k \cdot (\neg P \emptyset); k)
= \{\text{domain split; (65) with V for B implies P \emptyset = \neg S \cdot V}\}
(\uparrow k: 0 \leq k \land x \in S^k \cdot P \emptyset; k) \uparrow \ (\uparrow k: 0 \leq k \land x \in S^k \cdot (S \cdot V); k)
= \{\text{function exponentiation (57); dummy change}\}
(\uparrow k: 0 \leq k \land x \in S^k \cdot P \emptyset; k) \uparrow (\uparrow k: 1 \leq k \land x \in S^k \cdot V; k-1)
= \{the 2nd term is obviously smaller than, hence different from, the 2nd
equation in this derivation; but: a maximum must equal one of its terms\}

\[ \uparrow k: 0 \leq k \land x \in S^k \cdot P \emptyset: k \].

The recursive equation may be derived from (194), it reads

\[ f \cdot x = 0 \uparrow (\uparrow v: v \in P \cdot x: f \cdot v + 1), \quad (195) \]

but this is not different from the expected analogue of (188), if we replace \( \downarrow \) by \( \uparrow \), and \( \emptyset \), the unit of \( \downarrow \), by \( 0 \), the unit of \( \uparrow \) on \( \mathbb{N} \). Relative to the shortest path problem, the \( f \cdot y \) adjustment remains unchanged, and the initialization simply becomes \( f \cdot x = 0 \) everywhere.

Another interesting aspect of the longest path problem is a generalization to possibly cyclic graphs. For shortest paths this generalization is rather delicate, requiring an entirely different approach [2,23]. However, in this case a slight modification of the algorithm suffices.

Scrutinizing the derivation, we come up with two places where we use the acyclicity property: in establishing that upon termination \( V_0 = \emptyset \) holds, and hence that \( f_0 \cdot v \) equals \( f \cdot v \) everywhere on \( V \), and in the remark that \( f_1 \cdot x \) is not changed in the body of the repetition, because \( x \notin S \cdot x \). Noting that the unoptimized algorithm is in fact the cyclicity test algorithm, we may add invariant \( V_02E \), so \( V_1 = S \cdot V_0 \supseteq S \cdot E = E \), and we find that \( x \in V_0 - E \) holds after the selection statement. But then from \( E2C \) we conclude that \( x \notin C \), and hence \( x \notin S \cdot x \), so the latter usage of acyclicity is validated. Furthermore we conclude that upon termination \( V_0 = E \) holds, implying that at least on \( \sim E \) we have \( f_0 \cdot v = f \cdot v \). So the question is: how do we compute \( f \) on \( E \)?

For \( x \in E \) we have, according to (116):

\[ (\forall k: 0 \leq k: x \in S^k \cdot V), \]

and, with (65), this is equivalent to

\[ (\forall k: 0 \leq k: P^k \cdot x \neq \emptyset). \]

Using this information in the explicit definition of \( f \cdot x \) (194) we see that

\[ f \cdot x = (\uparrow k: 0 \leq k: k) = \emptyset. \]
Consequently, all we have to do is append an extra repetition after the main repetition, setting all $f_0 \cdot x$ values for $x \in E$, or equivalently $x \in V_0$, to $\alpha$, i.e.

$$\text{for } x \in V_0 \rightarrow f_0 \cdot x := \alpha \text{ ref}.$$ 

We leave it to the reader to check that all optimizations (including the elimination of $f_0$), except the elimination of $V_0$, are still feasible.

Next we turn to a generalization of the shortest path problem. We would like to compute for all $x$ the minimal distance from a given fixed set $X$, which need not be $P\emptyset$. Moreover, the distance along some path, instead of the number of edges, is taken to be the sum over all labels $t \cdot w \cdot u$ of the consecutive edges $\langle w, u \rangle$ of the path, where $t \in V \times V \rightarrow \mathbb{Z}$ is a given function. Note that these labels may even be negative.

Contrary to the computation of the maximum over all ancestor edge labels in (179), where the order of ancestor edges was irrelevant, this time the edges must be grouped into paths, if a meaningful distance is to be assigned to them. It appears that the introduction of a path formalism cannot be avoided any longer. For this single occasion we settle for an informal description.

We consider a path $p$ to be a sequence of vertices, numbered from 0 up to and including $#p$, the length of $p$ (the number of edges on the path). Thus $p=[x]$ is the empty path starting and ending in $x$, and $p=[x,y,z]$ is the path from $x$ via $y$ to $z$, having length 2. The $k$-th vertex on path $p$ is denoted by $p \cdot k$. Of course for all $k$: $0 \leq k \leq $#p$ we have $p \cdot (k+1) \in S \cdot (p \cdot k)$. Two paths $p$ and $q$ may be concatenated if $p \cdot #p = q \cdot 0$, giving path $p \cdot q$. Note that this is not the ordinary sequence concatenation! In this notation the explicit definition of the function to be computed reads

$$f \cdot x = \{ \downarrow p: p \cdot 0 \in X \land p \cdot #p = x: \Sigma \cdot p \}, \quad \text{with} \quad \Sigma \cdot p = \{ \Sigma k: 0 \leq k \leq p \cdot #p \in S \cdot (p \cdot k) \}.$$ 

(196)  \(197\)

From (196) and (197) we derive the recursive equation:

$$f \cdot x = \{ (196) \}$$

$$= \{ \downarrow p: p \cdot 0 \in X \land p \cdot #p = x: \Sigma \cdot p \}$$

= \{ \text{case analysis} \}$$

$$= \{ \downarrow p: p \cdot 0 \in X \land p \cdot #p = x \land #p = 0: \Sigma \cdot p \} \downarrow \{ \downarrow p: p \cdot 0 \in X \land p \cdot #p = x \land #p > 0: \Sigma \cdot p \}$$
\[ \text{one-point rule with } p = [x]; \text{ path split } p = q\# [u,x] \text{ with } x \in S \cdot u \text{ and } q \cdot \# q = u \]
\[ \downarrow \quad (q,u) \ni \quad (q\# [u,x]) \cdot \in X \land q \cdot \# q = u \land x \in S \cdot u: \Sigma \cdot (q\# [u,x]) \]
\[ \{ \text{empty domain and domain split; path property; (58)} \}
\[ \text{if } x \in X \Rightarrow 0 \Rightarrow x \notin X \Rightarrow \omega \Rightarrow \}
\[ \downarrow \quad (q,u) : q \cdot 0 \in X \land q \cdot \# q = u \land u \in P \cdot x: \Sigma \cdot q + t \cdot u \cdot x \]
\[ \{ \text{constant function modifier (12), distributivity of + over } \downarrow \{ \omega + k = \omega \} \}
\[ \{ \text{(197)} \}
\[ \{ \text{(196)} \}
\[ \{ \text{(196)} \}\]

leading to the definition of \( H \), being

\[ H \cdot g \cdot v = (\omega | X : 0) \cdot v \downarrow (\downarrow \quad u : u \in d \cdot g : g \cdot u + t \cdot u \cdot v) \quad (198) \]

Note that any reference to paths has disappeared, a fortunate consequence of our endeavour towards recursive equations. Fortunate, because the present example is only a very simple application of the path formalism, which tends to get very awkward in more complicated situations. So we turn it into a separate rule: get rid of path expressions as soon as possible.

From here the rest is straightforward; for completeness' sake we state the split and empty function property below:

\[ H \cdot (g \cdot (u : c)) \cdot v = (c + t \cdot u \cdot v), \quad (199) \]
\[ H \cdot \emptyset \cdot v = \text{if } v \in X \Rightarrow 0 \Rightarrow v \notin X \Rightarrow \omega \Rightarrow \} \quad (200) \]

The rest is left as an exercise for the reader.

8.5 Information extension

For several problems the program scheme of section 8.1 fails, because none of the alternatives for mapping \( H \) displays a proper split property. In this final subsection a possible way out for some cases is discussed, called information extension, to be illustrated using two examples.

The first example I will term predecessor balance. The function \( f \in V \cdot \mathbb{B} \) to be computed has no explicit definition; \( f \cdot x \) is defined recursively as the equality of the number of predecessors \( y \) of \( x \) with \( f \cdot y \) and the number with \( \neg f \cdot y \), or more formally:
\( f \cdot x = (\# v: v \in P \cdot x: f \cdot v) = (\# v: v \in P \cdot x: \neg f \cdot v). \) 

(201)

There appears to be no sensible alternative to the following definition for \( H \):

\( H \cdot g \cdot v = (\# v: v \in d \cdot g: g \cdot v) = (\# v: v \in d \cdot g: \neg g \cdot v). \)

(202)

Trying to derive a split property from (202), however, we find

\[
(\# v: v \in d \cdot g: g \cdot v) \quad \text{if} \quad c < \pm 1 \quad \text{and} \\
(\# v: v \in d \cdot g: \neg g \cdot v)
\]

which cannot be expressed in terms of \( H \cdot g \cdot v \). On the other hand, (203) shows that it can be expressed in terms of two other functions, in which not the equality of both numerical quantifiers matters, but instead their difference being \( \pm 1 \). However, introducing a tuple of all three functions will not help, requiring yet other functions with difference \( \pm 2 \), et cetera. It appears that we lack information: the knowledge if two numbers are equal is not sufficient, we need to know how much they differ. Thus we come to introduce function \( h \in V \rightarrow Z \), instead of \( f \), defined by

\[
h \cdot x = (\# v: v \in P \cdot x: f \cdot v) - (\# v: v \in P \cdot x: \neg f \cdot v).
\]

(204)

This step is called information extension: \( h \) contains more information than \( f \). Indeed, each \( f \cdot x \) may be expressed in terms of \( h \cdot x \) only, by

\[
f \cdot x = h \cdot x = 0,
\]

(205)

while the reverse is not possible. We can consequently switch to the problem of computing \( h \), because equivalence (205) enables us to find a recursive equation for \( h \) as well; substitution into (204) gives

\[
h \cdot x = (\# v: v \in P \cdot x: h \cdot v = 0) - (\# v: v \in P \cdot x: h \cdot v \neq 0).
\]

(206)

It is now a trivial matter to verify that the corresponding function \( H \), where

\[
H \cdot g \cdot v = (\# v: v \in d \cdot g: g \cdot v = 0) - (\# v: v \in d \cdot g: g \cdot v \neq 0).
\]

(207)
satisfies properties

\begin{align}
\mathcal{H}(g。(u: c)) \cdot v &= \mathcal{H} \cdot g \cdot v + \begin{cases} 1 & \text{if } c = 0 \\ 0 & \text{if } c \neq 0 \end{cases} \cdot \mathcal{H} \cdot \emptyset \cdot v, \\
\mathcal{H} \cdot \emptyset \cdot v &= 0.
\end{align}

Finally, the resulting algorithm computing \( h \) is supplemented by the repetition

\begin{equation}
\text{for } x \in V \rightarrow \begin{array}{l}
\text{f0} \cdot x := h0 \cdot x = 0 \quad \text{ref,}
\end{array}
\end{equation}

with obvious interpretations of \( f0 \) and \( h0 \), and, if desired, loop fusion may be applied, transferring the assignment to \( f0 \cdot x \) to the main repetition, directly after the assignment to \( h0 \cdot x \).

The second and last problem is to compute the Grundy function of an acyclic directed graph. A family of these functions was introduced by Grundy [26], in connection with game theory. More in particular the function considered here, be it generalized to possibly cyclic graphs, turns out to play an important role to find a winning strategy for the game of Nim, as explained quite lucidly in [27]. Again, only a recursive definition is available for this function \( f \in V \rightarrow \mathbb{N}:

\begin{align}
f \cdot x &= (\downarrow k: 0 \leq k \land k \notin GR \cdot x: k), \quad \text{with} \\
GR \cdot x &= \{ v: v \in P \cdot x: f \cdot v \},
\end{align}

i.e. \( f \cdot x \) is the smallest natural number not appearing among the \( f \)-values of \( x \)'s predecessors. Taking \( \mathcal{H} \) to be defined by

\begin{equation}
\mathcal{H} \cdot g \cdot v = (\downarrow k: 0 \leq k \land k \notin \{ v: v \in d \cdot g \cdot g \cdot v \}: k),
\end{equation}

it follows that

\begin{equation}
\mathcal{H}(g \cdot (u: c)) \cdot v = (\downarrow k: 0 \leq k \land k \notin \{ v: v \in d \cdot g \cdot g \cdot v \} \land k \neq c: k),
\end{equation}

which, in case \( c \neq \mathcal{H} \cdot g \cdot v \) equals \( \mathcal{H} \cdot g \cdot v \), but if \( c = \mathcal{H} \cdot g \cdot v \) is not expressible in terms of \( \mathcal{H} \cdot g \cdot v, u, c \) and \( v \).

How do we apply information extension here? To this end we once more need a function containing more information than \( f \) itself. But which function could that be? Well, the answer is right before our eyes: according to (211) \( f \cdot x \) can
be expressed directly in terms of $GR\cdot x$, whereas the reverse is obviously not possible. Hence function $GR$ is a candidate for a function with extended information. So we interpret definition pair (211) and (212) just the other way around: substitute the former into the latter. The alternative definition of $H$, this time for the determination of $GR$, then reads

$$H\cdot g\cdot v = \{v: v \in d\cdot g: \{k: 0 \leq k \land k \neq g\cdot v: k\}\},$$

(215)

and for the split and empty function property we find

$$H\cdot (g\cdot (u\cdot c))\cdot v = H\cdot g\cdot v \cup \{\{k: 0 \leq k \land k \neq c: k\}\},$$

$$H\cdot \emptyset\cdot v = \emptyset.$$  

(216)  
(217)

The latter presents no particular problems, and, introducing $gr1 \in V+P\cdot N$ as the variable approximating $GR$, the initialization becomes

$$gr1\cdot x := \emptyset.$$  

(218)

On the other hand, (216) would lead to the adjustment

$$gr1\cdot y := gr1\cdot y \cup \{\{k: 0 \leq k \land k \neq gr1\cdot x: k\}\},$$

(219)

which can hardly be called primitive. A little extra refinement work is called for. That turns out to be not difficult. The quantifier expression on the right hand side of (219) allows a straightforward application of the linear search theorem [2]. After the corresponding repetition

$$h:=0; \text{ do } h \in gr1\cdot x \implies h:= h+1 \text{ od}$$

(220)

variable $h$ has the value of the quantifier in (219). Since (220) is probably an $O\cdot N$ statement list, placing it before (219), inside the inner repetition, will be inefficient. Noting that the value of the quantifier, and consequently of $h$, does not depend on $y$, it is feasible to place (220) before the $\text{for } y \in S\cdot x$ statement. Furthermore, as $gr1\cdot x = GR\cdot x$, (211) implies that $h = f\cdot x$, and hence insertion of $fl\cdot x := h$ after (220) will establish the desired postcondition $fl=f$ after the main repetition.

Carrying out the abovementioned refinements, statement (219) reduces to
grl·y := grl·y U {h},

which, according to (82) and (88), is equivalent to the standard primitive statement list

\[ \text{as } h \notin \text{grl} \cdot y \Rightarrow \text{grl} \cdot y := \text{grl} \cdot y + \{h\} \text{ sa}. \]  

Substituting (218), (220), the assignment to fl·x, and (222) into the primitive algorithm at the end of section 8.1, we obtain the desired solution for the problem of the grundy function. Because this solution deviates from the general solution in details, it is, for once, written out in full below.

\[
\begin{align*}
\text{II} & : \quad \text{V2: } P \cdot V; \text{ T: } V \rightarrow N; \text{ grl: } V \rightarrow P \cdot N \ | \ V2:=\emptyset; \\
& \quad \text{for } x \in V \Rightarrow T \cdot x := 0 \ \text{ rof}; \\
& \quad \text{for } x \in V \Rightarrow \text{for } y \in S \cdot x \Rightarrow T \cdot y := T \cdot y + 1 \ \text{ rof} \ \text{ rof}; \\
& \quad \text{for } x \in V \Rightarrow \text{as } T \cdot x = 0 \Rightarrow V2 := V2 + (x) \text{ sa}; \text{ grl} \cdot x := \emptyset \ \text{ rof}; \\
& \quad \text{do } V2 \neq \emptyset \Rightarrow \\
& \quad \quad [ \quad [ \quad \left[ \quad \text{for } x \in V \Rightarrow h: N \ | \ x \in V2; V2 := V2 - (x); h := 0; \\
& \quad \quad \quad \text{do } h \in \text{grl} \cdot x \Rightarrow h := h + 1 \ \text{ od}; \text{ fl} \cdot x := h; \\
& \quad \quad \quad \text{for } y \in S \cdot x \Rightarrow \\
& \quad \quad \quad \quad \text{as } h \notin \text{grl} \cdot y \Rightarrow \text{grl} \cdot y := \text{grl} \cdot y + \{h\} \text{ sa}; \\
& \quad \quad \quad \quad T \cdot y := T \cdot y - 1; \text{ as } T \cdot y = 0 \Rightarrow V2 := V2 + (y) \text{ sa} \\
& \quad \quad \quad \text{rof} \\
& \quad \quad ] ] \\
& \quad \text{od} \\
& ]].
\end{align*}
\]

A slight complication arises if we try to establish the time complexity of this algorithm. In particular, the repetition in (220) looks awkward, being \(O \cdot \#IN\) in principle. Fortunately, observing the quantifier in (219), \(h\) never exceeds \#grl·x, and according to (212) \#grl·x = \#GR·x < \#P·x. This means that (220) is \(O \cdot (\#P·x)\), and the entire algorithm is of order

\[ N + (\Sigma v:: \#P·v) + (\Sigma v:: \#S·v), \]

which, according to (135) and (58), is \(O \cdot (N+M)\).

The bounding argument for \(h\) also has implications for the implementation of grl. According to [20], each set grl·x should be represented by a boolean
array on $\mathbb{N}$, in order to make the element tests $\neg \text{grl} \cdot x$. Of course, arrays with infinite domains are not feasible. Fortunately, values exceeding $\#P \cdot x$ are irrelevant, implying that each set $\text{grl} \cdot x$ can be implemented by a boolean array on domain $[0,\#P \cdot x]$, all consecutively stored if desired, leading to an overall auxiliary storage of $O \cdot (N+M)$. Going into the details of the implementation process will take us too far.

If the reader is not too puzzled by now, he may try to apply the theory of most of section 8 to the following exercise, which is due to Roland Backhouse. The problem is to compute, in an acyclic graph, the maximum distance from any source, if the distance, measured along a path, is defined as the product of the, possibly negative, labels of all edges on the path. Good luck!

9. Concluding remarks

What have we learned from this paper? Firstly, we have seen derivations of solutions to a nice collection of special problems, being cyclicity testing, graph inversion, topological sorting, predecessor sums, two-person game, ancestor vertex and edge maxima, shortest and longest distance from a source or a given set, predecessor balance and the grundy function. Despite the fact that not all of these problems are new, hopefully the reader recognizes some interesting new aspects in their approach, derivation or solution.

On the other hand, the problems mentioned above are just illustrations, and constitute only a minor part of the message. More important, in my opinion, is the derivation of two solution schemes to a generally formulated problem of the computation of a function on an acyclic graph. These have a wider impact, since they allow every new problem in the problem class to be solved almost effortlessly. The examples may help to solve the problems that possibly arise in applying the solution schemes.

Even more generally applicable, but with more modest advantages, is the SAL formalism, introduced in section 2, 3 and 4. It comprises the abstract types, set and especially function calculus rules, abstract statements and refinement theorems. This formalism has proved useful for other graph programming problems [22,23,24], and is expected to affect even more general abstract problems. In that case, it will be necessary to extend the formalism to include other types, such as lists, bags and trees.
Finally, the reader must have noticed the presence of a programming aid of a still higher level of generality: the whole of heuristic rules, spread out over the paper. Some of them are very general indeed, but here generality is inherently proportional to vagueness, and the potentiality of the rules is hard to be measured.

In the preceding treatment we did not address the problem of finding a suitable specification for a given problem. This is by no means a simple problem, since often it involves translating natural language into a formal specification. It is expected though, that if a verbal specification is sufficiently precise, it can always be translated into formulae containing quantifiers, using translations for standard phrases like "for all ....", "upper bound", "the least solution of ....", etc. Then formal manipulation can be applied to formulate the specification as concisely and abstractly as possible. This approach was used for the examples in the paper, and explicitly advocated.

Comparing the derivations in the present paper to those in [8,14] by Martin Rem, two distinctions attract attention, apart from the level of abstraction and formality. Firstly, none of the problems in this paper are solved with the aid of the so-called tripartitioning technique advocated in [8], and secondly, nowhere have we used a tail invariant.

In my opinion the tripartitioning technique, where $V$ is subdivided into three disjoint sets to begin with, is not particularly helpful. If three disjoint sets are necessary, this should follow from the derivation. If more than three sets are needed, this interferes with a tripartitioning. As it happens, two sets suffice for all algorithms in this paper, avoiding the introduction of a superfluous variable. These two sets, $V_0$ and $V_1$ from section 6.1, are not even disjoint. Admittedly, $V_0$, $V_0-V_1$ and $-V_0$ form a partition in three sets. However, as remarked in section 6.2, calculating with $V_0-V_1$ is more complicated than with $V_1$.

The absence of tail invariants is a consequence of personal experience with these, playing with all kinds of graph problems. Possibly due to the increased complexity of the problems, it turns out that tail invariants mean trouble. By nature they are more implicit than forward invariants, and as such force us to consider the entire invariant, instead of the defining right-hand side of a forward invariant, and sometimes require a little foresight concerning the direction of the derivation. They also carry less information, because they may allow more than one solution for the variables appearing in them. Since in
practice each variable is single-valued, why not use this information? On the other hand, many problems have no known elegant solution without using a tail invariant. Concluding, I would like to formulate the following heuristic rule: avoid, if possible, the use of tail invariants in complex problems.

Some elements in the formalism presented in this paper are still not satisfactory, and leave room for future improvements. One aspect concerns the hybrid nature of the calculations and algorithms, using sets and functions promiscuously. Calculating with functions, using the lift rules and function assignment theorems, is rather elegant, but why do we have to introduce sets as well? If a set is to be computed, like $E$ in section 6, or the set of reachables in [22], this is only natural, but if the unknown is a function, like in section 7.1, the sets popping up in 7.2 and 8.1 seem out of place.

One possibility for the elimination of the sets in section 8.1 would be to let the function variables absorb them in the form of domain restrictions. In that case we would replace $\mathcal{P}1$ by

$$\mathcal{P}1': \quad f_0 = \langle -V_0 \cdot f \rangle.$$

Naturally, this approach requires $f_0$ to be a partial function: a function with expanding domain, contrary to $\mathcal{P}1$, which presumes $f_0$ to be of type $V \rightarrow \mathbb{T}$, be it that the values on $V_0$ are "don't-cares". The present formalism does not allow this: our intermediate and help functions may have any domain we like, and we can combine functions with various domains in our calculations, but program variables have a fixed domain. However, it is worthwhile to consider a removal of this inconsistency [23]. In that case, the statement adjusting $f_0$ would be

$$f_0 := f_0 \vee (x: f \cdot x), \quad (223)$$

and derivation (162) should accordingly be replaced by

$$f \cdot x$$

$$= \langle (142) \rangle$$

$$\mathcal{H}(P \cdot x \downarrow f) \cdot x$$

$$= \langle \text{choose } x \text{ such that } P \cdot x \cap V_0 = \varnothing; \text{ then with } \mathcal{P}0 (P \cdot x \downarrow f) = (P \cdot x \downarrow f_0) \rangle$$

$$\mathcal{H}(P \cdot x \downarrow f_0) \cdot x$$

$$= \langle \text{introduce } \mathcal{P}2', \text{ and variable } f_1 \rangle$$

$$f_1 \cdot x, \quad (224)$$
where in this case the last step is generalized to invariant

\[ P2': \quad (\forall v \cdot f1 \cdot v = \mathcal{K} \cdot (P \cdot v \cdot f0) \cdot v). \]

This time \( f1 \) depends on \( f0 \), rather than \( v0 \), which is equally satisfactory. Note that, just like in (162), \( P \cdot v \) could have been replaced by \( P \cdot v \setminus v0 \), but this was deliberately avoided, in agreement with the heuristic rule: choose invariants always such that they depend on as few variables as possible. The motivation for this rule is that every change in one of the primary variables leads to a separate calculation of the necessary corresponding change in the secondary variable, causing a lot of extra work. In most cases, like here, invariants depending on two variables suffice.

Again it is possible to calculate the adjustment of \( f1 \), this time in terms of the change in \( f0 \), according to the derivation

\[
\begin{align*}
\mathcal{K} \cdot (P \cdot v \cdot f0 \cdot (x: h)) \cdot v &= \{\text{term split (50)}\} \\
&= \mathcal{K} \cdot ((P \cdot v \cdot f0) \cdot (P \cdot v \cdot (x: h)) \cdot v \\
&= \{\text{case analysis; (10)}\} \\
&\quad \text{if } x \in P \cdot v \rightarrow \mathcal{K} \cdot ((P \cdot v \cdot f0) \cdot (x: h)) \cdot v \\
&\quad \quad \text{if } x \notin P \cdot v \rightarrow \mathcal{K} \cdot ((P \cdot v \cdot f0) \cdot \emptyset) \cdot v \\
&\quad \quad \text{if } x \in P \cdot v \rightarrow \mathcal{K} \cdot (P \cdot v \cdot f0) \cdot v \\
\end{align*}
\]

resulting in the same statement for \( f1 \) as before. The only problem is, how to simulate the calculations containing \( V1 \) (\( x \) selection, postcondition checking), without letting the sets enter again through the back door. In [23] this is demonstrated to be feasible. However, more research on the subject is needed.

Note that in some special cases it is possible to use \( P2' \) with a fixed-type variable \( f0 \in V \cdot T \), e.g. for the ancestor maximum problem, where it becomes
\[ P2^*: (\forall v:: f_1 \cdot v = (\uparrow w:: w \in P \cdot v:: f_0 \cdot w)), \]

but this forces us to define \( f_0 \) explicitly for every vertex in \( V \), though it is not clear yet with which value.

Letting \( h = f_1 \cdot x \), derivation (225) is replaced by

\[
(\uparrow w:: w \in P \cdot v:: (f_0 | x: h) \cdot w)
\]

\[
= \text{domain split; (12)}
\]

\[
(\uparrow w:: w \in P \cdot v \setminus \{x\}:: f_0 \cdot w) \uparrow (\uparrow w:: w \in P \cdot v \cap \{x\}:: h)
\]

\[
= \text{case analysis}
\]

\[
\begin{cases}
\text{if } x \in P \cdot v & \Rightarrow (\uparrow w:: w \in P \cdot v \setminus \{x\}:: f_0 \cdot w) \uparrow (\uparrow w:: w=x:: h) \\
\text{if } x \notin P \cdot v & \Rightarrow (\uparrow w:: w \in P \cdot v:: f_0 \cdot w) \uparrow (\uparrow w:: w=x:: h)
\end{cases}
\]

\[
f_1
\]

\[
= \text{S/P calculus (58); one-point rule; empty domain; choose } f_0 \cdot x = -v \]

\[
\begin{cases}
\text{if } v \in S \cdot x & \Rightarrow (\uparrow w:: w \in P \cdot v:: f_0 \cdot w) \uparrow h \\
\text{if } v \notin S \cdot x & \Rightarrow (\uparrow w:: w \in P \cdot v:: f_0 \cdot w)
\end{cases}
\]

\[
f_1
\]

\[
= \text{P2*; lift-include rule (40); constant function notation}
\]

\[
( f_1 \uparrow (S \cdot x:: h)) \cdot v , \quad (226)
\]

where the special choice for the initial value of \( f_0 \cdot x \) allows us to restore the domain of the quantifier. Maybe the reader can benefit from this way of calculating on occasions.

Another possibility to get rid of the sets is to hide them in the hidden variables of \texttt{for}-statements, and use the expression accumulation theorem more often. Thus we could, along the same lines as the graph inversion problem in section 7.3, try to refine statement \( f_1 := f \), using accumulation function

\[
\mathcal{F} \cdot A = (\lambda v:: \mathcal{H} \cdot (P \cdot v \cap A = f) \cdot v) . \quad (227)
\]

With the aid of split property (152) the expression accumulation theorem leads us straight to the statement list (currying both \( \mathcal{H} \) and \( \mathcal{K} \)):

\[
 f_1 := \mathcal{H} \cdot \emptyset ; \text{ for } x \in V \Rightarrow f_1 := f_1 \uparrow (S \cdot x \uparrow \mathcal{K} \cdot (f \cdot x) \cdot x) \quad \text{ref} \quad (228)
\]

Subsequent application of the lift accumulation theorem then yields
We have come quite a long way in only two steps! However, in order to eliminate $f \cdot x$ from (229), we must come up with an equivalent of derivation (162) again, which seems to be impossible without having V0 at our disposal. There appears to be no alternative then to unfold the for-statement into its do equivalent, supplied with the corresponding invariant. Unfortunately, if we proceed in that way, all the advantages of this approach vanish. The present SAL formalism is not yet flexible enough to allow this to run smoothly.

Finally, for the sake of convenience, we will list below all heuristic rules mentioned in this paper, more or less in order of decreasing generality.

Strive for simplicity

Formulate specifications as concisely and abstractly as possible

Exploit, as much as possible, the precondition of a statement to be refined

Choose invariants such that they depend on as few variables as possible

Avoid, if possible, the use of tail invariants in complex problems

Choose the order of assignments in accordance with the available calculus rules for the quantities involved

When working from the precondition, try to rewrite it as a substitution of the variable to be changed

Try to eliminate pointwise set quantifiers

Try to specify the problem in terms of simple, recursive equations

Get rid of path expressions as soon as possible

If a least fixed point of some function on a lattice is to be calculated, approximate it from below.

Localize and restrict the function dependence of the fixed point operator as much as possible
10. Acknowledgements

I would like to thank Jaap van der Woude for supplying the idea of the uniqueness theorem, and for his stimulating criticism during my lectures on the subject: the abstraction he desired was always one level higher than I could offer. Rudolf Mak and Gerard Zwaan are gratefully acknowledged for their help in shaping my ideas about the formalism. To the latter I am also indebted for scrutinizing the draft version of this paper, and for suggesting many improvements. I am also grateful to Lex Bijlsma for his comments on the paper. Wim Feijen and Netty van Gasteren are thanked for the idea of inverting toposort, reformulating it in terms of P (although it did not lead to the effect they expected).

11. References


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
A systolic design for dynamic programming.
P.Struik

89/3 H.M.M. Ten Eikelder
Some category theoretical properties related to a model for a polymorphic lambda-calculus.
C. Hemerik

89/4 J.Zwiers
Compositionality and modularity in process specification and design: A trace-state based approach.
W.P. de Roever

89/5 Wei Chen
Networks of Communicating Processes and their (De-)Composition.
T.Verhoeff
J.T.Udding

89/6 T.Verhoeff
Characterizations of Delay-Insensitive Communication Protocols.

89/7 P.Struik
A systematic design of a parallel program for Dirichlet convolution.

89/8 E.H.L.Aarts
A general theory of genetic algorithms.
A.E.Eiben
K.M. van Hee

89/9 K.M. van Hee
Discrete event systems: Dynamic versus static topology.
P.M.P. Rambags

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
A concise formal framework for data modeling.
K.M. van Hee

89/13 A.T.M.Aerts
A program generator for simulated annealing problems.
K.M. van Hee
M.W.H. Hesen

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
Towards a framework for comparing data models.
K.M. van Hee

89/17 M.J. van Diepen
A formal semantics for Z and the link between Z and the relational algebra.
K.M. van Hee
<table>
<thead>
<tr>
<th>Page</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>90/2</td>
<td>K.M. van Hee P.M.P. Rambags</td>
<td>Dynamic process creation in high-level Petri nets, pp. 19.</td>
</tr>
<tr>
<td>90/4</td>
<td>A. Peeters</td>
<td>Decomposition of delay-insensitive circuits, p. 25.</td>
</tr>
<tr>
<td>90/5</td>
<td>J.A. Brzozowski J.C. Ebergen</td>
<td>On the delay-sensitivity of gate networks, p. 23.</td>
</tr>
<tr>
<td>90/8</td>
<td>M.B. Josephs</td>
<td>Receptive Process Theory, p. 16.</td>
</tr>
<tr>
<td>90/9</td>
<td>A.T.M. Aerts P.M.E. De Bra K.M. van Hee</td>
<td>Combining the functional and the relational model, p. 15.</td>
</tr>
<tr>
<td>90/10</td>
<td>M.J. van Diepen K.M. van Hee</td>
<td>A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).</td>
</tr>
<tr>
<td>90/11</td>
<td>P. America F.S. de Boer</td>
<td>A proof system for process creation, p. 84.</td>
</tr>
<tr>
<td>90/12</td>
<td>P. America F.S. de Boer</td>
<td>A proof theory for a sequential version of POOL, p. 110.</td>
</tr>
<tr>
<td>90/14</td>
<td>F.S. de Boer</td>
<td>A proof system for the language POOL, p. 70.</td>
</tr>
<tr>
<td>90/15</td>
<td>F.S. de Boer</td>
<td>Compositionality in the temporal logic of concurrent systems, p. 17.</td>
</tr>
<tr>
<td>90/16</td>
<td>F.S. de Boer C. Palamidessi</td>
<td>A fully abstract model for concurrent logic languages, p. 23.</td>
</tr>
<tr>
<td>90/17</td>
<td>F.S. de Boer C. Palamidessi</td>
<td>On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.</td>
</tr>
</tbody>
</table>
90/18 J. Coenen
E. v. d. Sluis
E. v. d. Velden

Design and implementation aspects of remote procedure calls, p. 15.

90/19 M. M. de Brouwer
P. A. C. Verkoulen

Two Case Studies in ExSpect, p. 24.

90/20 M. Rem

The Nature of Delay-Insensitive Computing, p. 18.

90/21 K. M. van Hee
P. A. C. Verkoulen

Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.

91/01 D. Alstein


91/02 R. P. Nederpelt
H. C. M. de Swart

Implication. A survey of the different logical analyses "if...then...", p. 26.

91/03 J. P. Katoen
L. A. M. Schoenmakers

Parallel Programs for the Recognition of P-invariant Segments, p. 16.

91/04 E. v. d. Sluis
A. F. v. d. Stappen

Performance Analysis of VLSI Programs, p. 31.

91/05 D. de Reus

An Implementation Model for GOOD, p. 18.

91/06 K. M. van Hee

SPECIFICATIEMETHODEN, een overzicht, p. 20.

91/07 E. Poll

CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.

91/08 H. Schepers

Terminology and Paradigms for Fault Tolerance, p. 25.

91/09 W. M. P. v. d. Aalst

Interval Timed Petri Nets and their analysis, p. 53.

91/10 R. C. Backhouse
P. J. de Bruin
P. Hoogendijk
G. Malcolm
E. Voermans
J. v. d. Woude

POLYNOMIAL RELATORS, p. 52.

91/11 R. C. Backhouse
P. J. de Bruin
G. Malcolm
E. Voermans
J. van der Woude

Relational Catamorphism, p. 31.

91/12 E. van der Sluis


91/13 F. Rietman

A note on Extensionality, p. 21.

91/14 P. Lemmens

The PDB Hypermedia Package. Why and how it was built, p. 63.
<table>
<thead>
<tr>
<th>Page</th>
<th>Authors</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<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</td>
<td>25.</td>
</tr>
<tr>
<td>91/18</td>
<td>Rik van Geldrop</td>
<td>Transformational Query Solving</td>
<td>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</td>
<td>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</td>
<td>16.</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</td>
<td>52.</td>
</tr>
<tr>
<td>91/26</td>
<td>P. de Bra, G.J. Houben, J. Paredaens</td>
<td>The GOOD based hypertext reference model</td>
<td>12.</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</td>
<td>17.</td>
</tr>
<tr>
<td>91/28</td>
<td>F. de Boer</td>
<td>A compositional proof system for dynamic process creation</td>
<td>24.</td>
</tr>
<tr>
<td>91/30</td>
<td>J.C.M. Baeten, F.W. Vaandrager</td>
<td>An Algebra for Process Creation</td>
<td>29.</td>
</tr>
<tr>
<td>ID</td>
<td>Author(s)</td>
<td>Title</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>91/31</td>
<td>H. ten Eikelder</td>
<td>Some algorithms to decide the equivalence of recursive types, p. 26.</td>
<td></td>
</tr>
<tr>
<td>91/33</td>
<td>W. v.d. Aalst</td>
<td>The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.</td>
<td></td>
</tr>
<tr>
<td>91/34</td>
<td>J. Coenen</td>
<td>Specifying fault tolerant programs in deontic logic, p. 15.</td>
<td></td>
</tr>
<tr>
<td>91/35</td>
<td>F.S. de Boer</td>
<td>Asynchronous communication in process algebra, p. 20.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>J.W. Klop</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C. Palamidessi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>92/01</td>
<td>J. Coenen, J. Zwiers, W.-P. de Roever</td>
<td>A note on compositional refinement, p. 27.</td>
<td></td>
</tr>
<tr>
<td>92/02</td>
<td>J. Coenen, J. Hooman</td>
<td>A compositional semantics for fault tolerant real-time systems, p. 18.</td>
<td></td>
</tr>
<tr>
<td>92/03</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Real space process algebra, p. 42.</td>
<td></td>
</tr>
<tr>
<td>92/05</td>
<td>J.P.H.W.v.d.Eijnde</td>
<td>Conservative fixpoint functions on a graph, p. 25.</td>
<td></td>
</tr>
<tr>
<td>92/06</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Discrete time process algebra, p.45.</td>
<td></td>
</tr>
<tr>
<td>92/07</td>
<td>R.P. Nederpelt</td>
<td>The fine-structure of lambda calculus, p. 110.</td>
<td></td>
</tr>
<tr>
<td>92/10</td>
<td>P.M.P. Rambags</td>
<td>Composition and decomposition in a CPN model, p. 55.</td>
<td></td>
</tr>
</tbody>
</table>