Conservative fixpoint functions on a graph

Citation for published version (APA):

van den Eijnde, J. P. H. W. (1992). *Conservative fixpoint functions on a graph*. (Computing science notes; Vol. 9205). Technische Universiteit Eindhoven.

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.

Eindhoven University of Technology
Department of Mathematics and Computing Science

Conservative Fixpoint Functions
on a Graph

by

J.P.H.W. van den Eijnde

92/05

Computing Science Note 92/05
Eindhoven, April 1992

COMPUTING SCIENCE NOTES

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

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

All rights reserved
editors: prof.dr.M.Rem
 prof.dr.K.M.van Hee.

CONSERVATIVE FIXPOINT FUNCTIONS ON A GRAPH

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

wsinjd@win.tue.nl

Abstract

In this paper we present a derivation of a general solution for a class of programming problems. In these problems a function over the vertices of a directed graph is to be computed, being defined as a least fixed point of some monotonic operator. If this operator satisfies a certain restriction with respect to its image for a differential change in its argument, it is called conservative, and an elegant general solution may be derived. It is stipulated that a strictly calculational derivation is only possible if the level of abstraction is sufficiently high. To that end a modest extension to the functional calculus is proposed, including partial functions, and a few simple high level programming constructs are introduced. The program scheme obtained is applied to a particular example, for which so far no derivation, other than informal ones, is known to exist. The solutions presented are not new, but the calculational, abstract and compact technique of deriving them is meant to improve and complement the current techniques [KNU77, REY81, REM84]. It is believed to simplify the derivations for a wider algorithm class [EIJ92] than the one treated here.

1. Introduction

From a fundamental point of view solving fixed-point equations for functions on a graph is interesting, because they represent generalizations of recursive equations "par excellence": each function value may depend on an arbitrary, possibly empty, set of other function values. These predecessor or successor sets simply define the graph. Obviously, the cyclicity of recursive equations is then reflected by the cyclicity of the corresponding graph. On the other hand, these fixed-point equations are of great practical use. They arise in many different fields: topography, digital networks, electronic circuits, compiler theory, computer graphics, and process scheduling. Recently the importance of solving fixed-point equations on graphs was recognized by Cai and Paige [CAI89], who devoted a comprehensive paper with many examples to the subject, although their objectives differ from ours, e.g. automatic code generation.

The technique used in this paper to solve graph programming problems is based on three corner-stones: heuristics, abstraction and calculus. Some twenty-five years ago, E.W. Dijkstra was one of the first to advocate the use of proper heuristics [DIJ76]. For a more up-to-date treatment the reader is referred to [GAS88]. These heuristic principles, and some others, are applied wherever appropriate in this paper, and an attempt is made to indicate at crucial points in the derivation why certain design decisions were taken. In [EIJ92] the reader may find explicitly formulated "rules of thumb", applying more in particular to set, graph or fixed-point problems.

The need for abstraction is widely accepted [REY81, CAI89], although in some modern textbooks [MEH84, COR90] graph algorithms are treated in terms of low level data structures. In my opinion, abstraction is the only way to reason about more complicated algorithms like graph algorithms, allowing the methods for solving small problems [DIJ88] to be applicable without requiring major changes. However, the way to abstract is subject to discussion. Since we are interested in computing functions the most logical conclusion seems to be to introduce abstract functions as data structures into the basic formalism, for which we choose the guarded command language (GCL) [DIJ88]. Because in repetitions we would like these functions to grow towards their final value we also introduce partial functions. Finally, we need abstract sets, in the spirit of [REM84]. The details of this **data abstraction** are summarized in section 2, together with a touch of **control abstraction** [LIS77] by means of some convenient programming structures, the most important one being a

for-statement that repeats a statement list for all elements of a fixed set. A straightforward proof rule is expressed by the **precondition for theorem**, and section 2 is concluded with the **expression accumulation theorem**, needed to make efficient use of the for-statement.

The third corner-stone of our technique, a calculational style, is widely believed to be a prerogative of transformational programming, be it imperative [MOR90], recursive [BUR77], functional [BIR88] or even relational [ROL90]. One of the goals of this paper is to show that programming with conventional Hoare triples can also be done purely calculationally. The required set and function calculus rules are introduced in the beginning of section 2. The resulting style is, admittedly, hybrid, with program notation on the one hand, and derivations using predicates on the other. 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.

The problem class under consideration is defined in section 3. Many members have been treated in literature, from minimal distance problems, dating as far back as 1959 [DIJ59, TAR81], reachability problems [REY81, CAI89], and least ancestor problem [REM86], to the capacity or tunnel problem [CAR79, REM85]. The derivation presented in section 4 solves the general problem. Compared to [CAI89, REM85], it has the advantage of being purely calculational, requiring inventiveness only at certain crucial points.

Many of the problems in the class can be treated very elegantly using fixed-point theory and a regular algebra of matrices [BAC75, CAR79, TAR81]. However, the present class of problems is more general, because these matrices carry less information than the fixed-point operators on functions in section 3. Therefore, in section 5, the general problem is instantiated to an example that, as far as I know, cannot be described by such a regular algebra: the problem of ascending reachability [REM85]. It does, however, turn out to be an instance of the problem class in [KNU77], but there a non-calculational proof of the solution algorithm is only given afterwards.

2. The SAL formalism

In the program notation of SAL (Set Algorithmic Language), which is an extension of GCL, all variables are assigned a - constant - type at the beginning of the block with respect to which they are local, by a declaration

of the form $v: \mathbb{T}$ (v a variable and $\mathbb{T}$ a type). Types are considered sets, supplemented with a collection of operators, satisfying certain laws.

In addition to the **simple types** $\mathbb{Z}$ (integers) and $\mathbb{B}$ (booleans), adapted from GCL, with the usual operators and laws, a separate, unstructured type $\mathbb{U}$ is introduced, the elements of which represent graph vertices. Apart from the **array type constructor** in GCL, we also have $\mathbb{P}$ (**powerset type constructor**), supporting the usual set operators and laws, $\rightarrow$ (**function type constructor**), and $\mapsto$ (**partial function type constructor**).

By convention sets are denoted with capitals, and u, v, w, x, y , and z are understood to be vertices. Unless stated otherwise, they are of standard type $V \in \mathbb{P} \cdot \mathbb{U}$. Lower case letters d, e, f, g , and h denote partial or total function variables. On the other hand, for predicates and other mappings *script* letters are used. Simple and generic types, and the powerset type constructor, are in **DOUBLE CAPITALS**. Finally, statement list names are in *italics*.

Although in principle the Eindhoven Quantifier Notation (EQN) [DIJ88] is adequate to formulate and prove the usual mathematical statements, I agree with [BAC89] that for complicated problems it tends to get very lengthy and cumbersome. So whenever possible, the dummies are abstracted from, and calculations are carried out on the function or set level. If not, we use the quantifiers $\forall$ (universal), $\exists$ (existential), $\downarrow$ (minimum) and $\uparrow$ (maximum), in an EQN-like notation, with an explicit dummy, a domain and a term, all separated by colons. The same convention is used for sets, so $\{v: \mathcal{D} \cdot v: \mathcal{E} \cdot v\}$ denotes the set of values $\mathcal{E} \cdot v$ for those v that satisfy predicate $\mathcal{D}$. In order to avoid confusion between set notation on the one hand, and the notation of hints in proofs and assertions in algorithms on the other, we adopt the convention of replacing the hint and assertion brackets $\{\}$ by the special brackets $\langle \rangle$.

Total function $f \in \mathbb{F} \rightarrow \mathbb{T}$ has domain $\mathbb{F}$ and range $\mathbb{T}$, and the so-called **domain operator** d satisfies $d \cdot f = \mathbb{F}$. On the other hand, for a partial function $g \in \mathbb{G} \mapsto \mathbb{T}$, $d \cdot g$ denotes the subset of domain $\mathbb{G}$ on which g is defined. So partial functions can be considered total functions with a variable domain; they are needed as program variables. All function notations and properties in the remainder of this section apply to total as well as partial functions, and $\rightarrow$ may denote either type constructor; $f \in \mathbb{F} \rightarrow \mathbb{T}$ and $g \in \mathbb{G} \mapsto \mathbb{T}$ are supposed to be arbitrary functions. By convention, the domain of a named function is denoted by the corresponding name in **CAPITALS**, hence

$$F = d \cdot f \quad \wedge \quad G = d \cdot g.$$

Instances of functions can be denoted as sets of pairs, and $\emptyset$ is the **empty function**. The **constant function** with domain S and value c for all $s \in S$ is denoted by $(S: c)$, and if $S = \{s\}$ is a singleton, this will be abbreviated to $(s: c)$, the **singleton function**. A constant function with standard domain V will be written $\hat{c}$, or $\hat{\mathcal{E}}$ if $\mathcal{E}$ is an expression. The **lambda quantifier** offers a general notation for functions:

$$(\lambda i: \mathcal{D}. i: \mathcal{E}. i)$$

is the function which, for all values of dummy i that satisfy $\mathcal{D}. i$, takes the value $\mathcal{E}. i$. In this notation a **domain restricted function** is defined by

$$(A \vdash f) = (\lambda i: i \in A \cap F: f \cdot i), \quad (0)$$

for arbitrary set A .

The operators on functions are defined in terms of the operators on their respective ranges, lifted to the function level. Let $\oplus \in \mathcal{T} \rightarrow \mathcal{T} \rightarrow \mathcal{T}$ be a binary infix operator. Then the lifted operator $\hat{\oplus}$ is defined by the **lift-meet rule**:

$$f \hat{\oplus} g = (\lambda i: i \in F \cap G: f \cdot i \oplus g \cdot i) \in F \cap G \rightarrow \mathcal{T}. \quad (1)$$

So far, rules (0) and (1) only allow us to restrain function domains. 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

$$f \vee g = (\lambda i: i \in F \cup G: \text{if } i \in F \rightarrow f \cdot i \quad \square \quad i \in G \rightarrow g \cdot i \quad \text{fi}) \in F \cup G \rightarrow \mathcal{T}. \quad (2)$$

As it happens, the functions encountered in programming practice rarely have disjoint domains. In fact, a combination of (1) and (2) turns out to be very fruitful. An obvious definition for yet another lifted operator $\check{\oplus}$ would be

$$\begin{aligned} f \check{\oplus} g &= (f \hat{\oplus} g) \vee (\sim G \vdash f) \vee (\sim F \vdash g) \\ &= (\lambda i: i \in F \cup G: \text{if } i \in F \cap G \rightarrow f \cdot i \oplus g \cdot i \quad \square \quad i \in F \setminus G \rightarrow f \cdot i \\ &\quad \square \quad i \in G \setminus F \rightarrow g \cdot i \quad \text{fi}) \in F \cup G \rightarrow \mathcal{T}. \end{aligned} \quad (3)$$

This rule is called the **lift-join rule**. Its power is that it generalizes both (1) and (2): using (0) and (1) we have

$$f \hat{\otimes} g = (F \cap G \vdash (f \check{\otimes} g)), \quad (4)$$

and if $F \cap G = \emptyset$ (3) reduces to (2); in that case the operator $\otimes$ on the left-hand side may be chosen arbitrarily, or omitted if it is irrelevant.

As for operator properties, it turns out that commutativity, associativity, idempotency and distributivity simply carry over to the lifted operators. For commutative and associative operators $\otimes$ we sometimes need a generalized version of (3), which reads

$$(\check{\otimes} k: \mathcal{D} \cdot k: h \cdot k) = (\lambda i: i \in (\cup k: \mathcal{D} \cdot k: H \cdot k): (\otimes k: \mathcal{D} \cdot k \wedge i \in H \cdot k: h \cdot k \cdot i)), \quad (5)$$

where $h \cdot k \in H \cdot k \rightarrow \mathbb{T}$ is a k -indexed family of functions.

Some function calculus rules, relating (0), (2) and (3), are summarized below. For arbitrary sets A and B we have the **chaining rule**

$$(A \cap B \vdash f) = (A \vdash (B \vdash f)). \quad (6)$$

If, in addition, $\otimes \in \mathbb{T} \rightarrow \mathbb{T} \rightarrow \mathbb{T}$ is an arbitrary binary infix operator on the common range $\mathbb{T}$ of f and g , the **term split rule** expresses

$$(A \vdash f \check{\otimes} g) = (A \vdash f) \check{\otimes} (A \vdash g). \quad (7)$$

For idempotent operators $\parallel \in \mathbb{T} \rightarrow \mathbb{T} \rightarrow \mathbb{T}$ the **domain disjunction rule**

$$(A \cup B \vdash f) = (A \vdash f) \parallel (B \vdash f) \quad (8)$$

holds. If $A \cap B = \emptyset$, rule (8) is called **domain split rule**, and operator $\parallel$ may be omitted. Rules (6), (7) and (8) are easily derived from (0), (2) and (3). Detailed proofs are given in [EIJ92]. An application of (5), the **domain shift to term rule**, reads

$$(\check{\otimes} k, i: \mathcal{D} \cdot k \wedge i \in W \cdot k: (\{i\} \vdash h \cdot k)) = (\check{\otimes} k: \mathcal{D} \cdot k: (W \cdot k \vdash h \cdot k)), \quad (9)$$

with $\otimes$ commutative and associative, and $W \cdot k$ k -indexed sets. It follows from

$$\begin{aligned}
& (\check{\otimes} k, i: \mathcal{D} \cdot k \wedge i \in W \cdot k: (\{i\} \vdash h \cdot k)) \\
= & \{ (0); (5) \} \\
& (\lambda i: i \in (\cup k, j: \mathcal{D} \cdot k \wedge j \in W \cdot k: \{j\} \cap H \cdot k): \\
& \quad (\otimes k, j: \mathcal{D} \cdot k \wedge j \in W \cdot k \wedge i \in \{j\} \cap H \cdot k: h \cdot k \cdot i)) \\
= & \{ \text{case analysis } j \in H \cdot k; \text{ property of } \cap \} \\
& (\lambda i: i \in (\cup k, j: \mathcal{D} \cdot k \wedge j \in W \cdot k \cap H \cdot k: \{j\}): \\
& \quad (\otimes k, j: \mathcal{D} \cdot k \wedge j \in W \cdot k \wedge i = j \wedge i \in H \cdot k: h \cdot k \cdot i)) \\
= & \{ (\cup j: j \in A: \{j\}) = A; \text{ one-point rule for } j \} \\
& (\lambda i: i \in (\cup k: \mathcal{D} \cdot k: W \cdot k \cap H \cdot k): (\otimes k: \mathcal{D} \cdot k \wedge i \in W \cdot k \cap H \cdot k: h \cdot k \cdot i)) \\
= & \{ (0); (5) \} \\
& (\check{\otimes} k: \mathcal{D} \cdot k: (W \cdot k \vdash h \cdot k)) .
\end{aligned}$$

Finally, we introduce a shorthand for a quantification over a function term, comparable to the reduce ($/$) from the Bird-Meertens formalism [BIR88], in the same way as function composition ($\circ$) is comparable to the map ($*$). For an associative and commutative operator $\otimes$ we define

$$\otimes \cdot f = (\otimes i: i \in F: f \cdot i) . \quad (10)$$

We conclude with two domain split rules for $\otimes \cdot f$, with a straightforward proof:

$$\otimes \cdot (f \vee g) = \otimes \cdot f \otimes \otimes \cdot g, \quad \text{and} \quad (11)$$

$$\otimes \cdot (f \check{\otimes} g) = \otimes \cdot f \otimes \otimes \cdot g . \quad (12)$$

Next we turn from **data abstraction** to **control abstraction** [LIS77]. Firstly, a convenient **miracle statement** is introduced. To that end we define statement

$\text{out} : \vdash \text{Post} \cdot \text{in} \cdot \text{out}$

with meaning: variable out is assigned a value such that $\text{Post} \cdot \text{in} \cdot \text{out}$ is validated. A detailed definition of this programming primitive is beyond the scope of this paper; the reader is referred to [MOR90].

A related programming primitive is element selection. Statement $x : \in X$ is defined using the weakest precondition $\wp$ for arbitrary predicate $\mathcal{P}$, by

$$\wp \cdot (x : \in X) \cdot \mathcal{P} \equiv X \neq \emptyset \wedge (\forall v: v \in X: \mathcal{P}_v^x) .$$

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 a single guard it is defined by

as $B \rightarrow SL$ sa is equivalent to if $B \rightarrow SL$ $\square$ $\neg B \rightarrow \text{skip}$ fi,

where B is a boolean expression and SL 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:

$$\{P\} \text{ as } B \rightarrow SL \text{ sa } \{Q\} \text{ is valid if and only if} \quad (13)$$

$$\{P \wedge B\} SL \{Q\} \text{ and } P \wedge \neg B \Rightarrow Q \text{ hold.}$$

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. It is defined by

$$\begin{aligned} \text{for } u \in W \rightarrow SL \text{ rof is equivalent to } & \llbracket A: P \cdot W; u: W \mid A := W; \quad (14) \\ & \text{do } A \neq \emptyset \rightarrow \\ & \quad u: \in A; A := A \setminus \{u\}; SL \\ & \text{od } \\ & \rrbracket. \end{aligned}$$

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

Application of the for-statement evidently offers no special advantages, if it replaces the corresponding repetition only after the latter is derived in full. Preferably, we would like to apply a proof rule directly to the for-statement itself. The following theorem offers one that is directly related to (14).

Precondition for-theorem

Let $W \in \mathcal{P} \cdot V$ be a program expression yielding a finite set, and let $u \in W$ and $A \in \mathcal{P} \cdot W$ be fresh variables. If there exists a predicate $\mathcal{P}$ depending on A , but not on u , and a statement list SL that does not refer to A , then

the validity for all A and u of $\langle u \in A \wedge \mathcal{P}_{AU\{u\}}^A \rangle SL \langle \mathcal{P} \rangle$,

(15)

implies the validity of $\langle \mathcal{P}_W^A \rangle \text{ for } u \in W \rightarrow SL \text{ rof } \langle \mathcal{P}_\emptyset^A \rangle$.

This theorem is proved in appendix A. It is generally applicable, but a drawback is the use of predicates, making application still rather elaborate. On the other hand, the theorem below addresses the special case of refining a complex assignment statement directly to a for-statement. It turns out to be adequate and practical for many applications. Not surprisingly, it reminds us of the so-called **reduction** from functional programming [BIR88]:

Expression accumulation theorem

Let $r \in T$ be a program variable, $W \in \mathcal{P} \cdot V$ a program expression yielding a finite set, $\mathcal{G} \in \mathcal{P} \cdot V \rightarrow T \rightarrow T$ and $f \in V \rightarrow T$ mappings, and let $\oplus \in T \rightarrow T \rightarrow T$ be a binary infix operator, for some type T . If for all $A \in \mathcal{P} \cdot W$, $t \in T$, and $u \in W \setminus A$

$$\mathcal{G} \cdot (A \cup \{u\}) \cdot t = \mathcal{G} \cdot A \cdot t \oplus f \cdot u \quad (16)$$

holds, then statement $r := \mathcal{G} \cdot W \cdot r$, with precondition $r = \mathcal{G} \cdot \emptyset \cdot r$, is refined by

$$\text{for } u \in W \rightarrow r := r \oplus f \cdot u \text{ rof} . \quad (17)$$

A proof outline is given in appendix A. For a detailed proof the reader is referred to [EIJ92]. Note that, contrary to the so-called **accumulation function** $\mathcal{G}$, it is essential that f does not depend on A or t .

3. Specification of the conservative fixpoint function problem

Consider a directed graph $G = \langle V, S \rangle$, with vertex set $V \in \mathcal{P} \cdot U$, and successor function $S \in V \rightarrow \mathcal{P} \cdot V$, mapping each vertex to the set of its direct successor

vertices. Alternatively, the predecessor function $P \in V \rightarrow P \cdot V$ may be used instead of S , the relation between the two being given by

$$(\forall u, v: v \in S \cdot u \equiv u \in P \cdot v).$$

Often S and P are generalized to functions of type $P \cdot V \rightarrow P \cdot V$, according to

$$S \cdot A = (\cup v: v \in A: S \cdot v), \quad (18)$$

and analogously for P . Denoting both the standard and generalized versions of S and P with the same symbol rarely causes confusion. Both S and P are strict with respect to $\emptyset$, monotonic, and uni-disjunctive (universally disjunctive).

Let $(T, \sqsubseteq)$ be a totally ordered, complete lattice, with bottom $\perp$, supremum $\sqcup$ and infimum $\sqcap$ [DAV90]. We consider partial functions $V \rightarrow T$ on the vertices of the graph. Then, using lift-join rule (3), we define lifted versions $\tilde{\sqcup}$ and $\tilde{\sqcap}$ of supremum and infimum, respectively. These, in turn, define in a natural way a partial order on $V \rightarrow T$ (in fact a complete lattice [DAV90]), consequently denoted by $\check{\sqsubseteq}$, though itself not being defined by (3), but by

$$d \check{\sqsubseteq} e \equiv D \sqsubseteq E \wedge (\forall v: v \in D: d \cdot v \sqsubseteq e \cdot v). \quad (19)$$

For this partial order $(A \vdash e)$ and $d \check{\sqsubseteq} e$ turn out to be monotonic.

We are asked to determine the least fixed point $f \in V \rightarrow T$ of some operator $\mathcal{F} \in (V \rightarrow T) \rightarrow (V \rightarrow T)$. So f satisfies

$$\mathcal{F} \cdot f = f, \quad (20)$$

$$(\forall e: \mathcal{F} \cdot e = e: f \check{\sqsubseteq} e). \quad (21)$$

Operator $\mathcal{F}$ is assumed to be monotonic, hence it satisfies

$$d \check{\sqsubseteq} e \Rightarrow \mathcal{F} \cdot d \check{\sqsubseteq} \mathcal{F} \cdot e. \quad (22)$$

Often recursive formulations can be derived from explicit expressions, using fixed-point theorems like that of Knaster-Tarski [DAV90]. In section 5 an instance of (20)-(21) is derived using the theorem below.

Generalized fixed-point theorem

Let $\mathcal{A} \in L \rightarrow L$ be a uni-disjunctive operator on the complete lattice $(L, \sqsubseteq, \sqcup, \sqcap)$, and let e be an element of L . If r is given by

$$r = (\bigcup k: 0 \leq k: A^k \cdot e), \quad (23)$$

then r is uniquely defined as the least (pre)fixed point of $(e \cup A)^*$, i.e.

$$e \cup A \cdot r = r, \quad \text{and} \quad (24)$$

$$(\forall s: s \in L: e \cup A \cdot s \subseteq s \Rightarrow r \subseteq s). \quad (25)$$

Here $(e \cup A)^*$ corresponds to $\mathcal{F}$ in (20)-(21). A proof is given in appendix B.

For the special class of problems considered here, $\mathcal{F}$ also satisfies a restriction on its image, with respect to a "differential" change in its argument, a kind of "limited growth"-property, expressed by

$$\mathcal{F} \cdot (e \vee (x: c)) \subseteq \mathcal{F} \cdot e \cup \hat{c}, \quad \text{for } x \notin E. \quad (26)$$

An operator satisfying (26) is called **conservative**, and so are its fixpoints.

It is not hard to prove that, if one restricts the context-free grammars from [KNU77] to generate only graph problems, then fixed-point functions turn out to be conservative precisely if their generating functions are **superior** (see [KNU77]). Though restricted to graphs, the present formalism is more general with respect to operators and function domains.

The class of programming problems defined by (18)-(26) can not necessarily be mapped to a regular algebra of matrices [BAC75, CAR79, TAR81]. On the other hand, if the regular algebra satisfies the property that the unit of its multiplication is top ($\top$) of the lattice $(\mathbb{T}, \subseteq)$, a property that corresponds to (26), it is an instance of the problem class defined above.

The required f computation is now accomplished by realizing postcondition

$$\mathcal{R}: \quad g = f.$$

where $g \in V \mapsto \mathbb{T}$ is a partial function variable.

4. Derivation of an algorithm

Because the quantity to be computed is a least fixed point we approximate it from below (cf. linear search) and propose the simplest invariant possible:

$\mathcal{P}0: \quad g \preceq f.$

From $\mathcal{P}0$ immediately follows the **first approximation** to the desired algorithm, using the miracle statement:

```

g := ∅;
do g ≱ f →
    d := g ≱ d ≱ f;  g := d
od ,

```

which is obviously (partially) correct. However, in this algorithm f is an unknown quantity, which should be eliminated. Again it has to be approximated from below, according to the derivation

$$f = \langle (20) \rangle \mathcal{F} \cdot f \preceq \langle \mathcal{P}0; (22) \rangle = \mathcal{F} \cdot g, \quad (27)$$

so it appears useful to introduce variable $h \in V \mapsto \mathbb{T}$, specified by

$\mathcal{P}1: \quad h = \mathcal{F} \cdot g.$

Naturally we want to replace f everywhere in the algorithm by h . Since, from (27), this means strengthening the guard and selection statement, this has no consequences for the invariance of $\mathcal{P}0$, so we obtain as **second approximation**:

```

g, h := ∅,  $\mathcal{F} \cdot \emptyset$ ;
do g ≱ h →
    d := g ≱ d ≱ h;  h :=  $\mathcal{F} \cdot d$ ;  g := d
od .

```

However, strengthening the guard implies weakening the postcondition, so we ought to make sure that $\mathcal{R}$ still holds upon termination. Negation of the guard could mean that g and h are not comparable, so in order to still conclude something useful we are forced to introduce invariant

$\mathcal{P}2: \quad g \preceq h.$

Then the postcondition remains valid, for we have

$$\begin{aligned}
& \neg g \checkmark h \\
\equiv & \{P2; P1\} \\
& g = \mathcal{F} \cdot g \\
\Rightarrow & \{(21)\} \\
& f \checkmark g \\
\equiv & \{P0\} \\
& \mathcal{R}.
\end{aligned} \tag{28}$$

Evidently, $P0$ and $P1$ remain valid in the latest algorithm. The validity of $P2_{d, \mathcal{F} \cdot d}^{g, h}$ just after the selection follows from $P1$ and $\mathcal{F}$ monotonicity (22):

$$d \checkmark h = \mathcal{F} \cdot g \checkmark \mathcal{F} \cdot d, \tag{29}$$

so the abstract, second approximation turns out to be (partially) correct.

So far everything has gone smoothly, requiring hardly any inventiveness. But at this point we take an important design decision. The only drawback of the second approximation seems to be the lack of clarity about the efficiency, or for that matter, termination at all. We decide to settle this question rigorously, and strive for a solution where domain G of g is explicitly **extended with one vertex** $x \in H$, hence $x \notin G$, in each turn of the repetition, and choose for $d \cdot x$ the **greatest value** permitted by upper bound h for d , i.e. $h \cdot x$. Such a refinement would consequently result in the **third approximation**:

```

g, h := ∅,  $\mathcal{F} \cdot \emptyset$ ;
do H \ G  $\neq \emptyset \rightarrow$ 
    x := H \ G;  c := h · x;
    d := g ∨ (x: c);
    h :=  $\mathcal{F} \cdot d$ ;  g := d
od.

```

Note that domains G and H of functions g and h , respectively, need not be adjusted separately. Variable $c \in T$ is only introduced as a shorthand. However, the above algorithm is only a true refinement of the previous one if

$$P3: \quad (G \vdash h) = g$$

holds, for this implies that $g \checkmark h$ is equivalent to $H \setminus G \neq \emptyset$, using (19).

Invariant $\mathcal{P}3$ means that h equals g on domain G of the latter, see (0).

Can we keep $\mathcal{P}3$ invariant? It holds initially, but before checking its invariance, we first mention a convenient property of d , directly following from conservativity property (26) and $\mathcal{P}1$:

$$\mathcal{F} \cdot d \check{\sqsubseteq} h \check{\sqcup} \hat{c}. \quad (30)$$

This also implies $c = \mathcal{F} \cdot g \cdot x \sqsubseteq \mathcal{F} \cdot d \cdot x \sqsubseteq h \cdot x \sqcup c = c$, proving

$$\mathcal{F} \cdot d \cdot x = c. \quad (31)$$

For the invariance of $\mathcal{P}3$ we then derive

$$\begin{aligned} & \mathcal{P}3_{d, \mathcal{F} \cdot d}^{g, h} \\ \equiv & \{ \mathcal{P}2; \text{substitution} \} \\ & (D \vdash \mathcal{F} \cdot d) \check{\sqsubseteq} d \\ \equiv & \{ d = g \vee (x: c); \text{domain split (8); (19)} \} \\ & (G \vdash \mathcal{F} \cdot d) \check{\sqsubseteq} g \wedge \mathcal{F} \cdot d \cdot x \sqsubseteq c \\ \Leftarrow & \{ (30); \text{monotonicity of } \vdash; (31) \} \\ & (G \vdash h \check{\sqcup} \hat{c}) \check{\sqsubseteq} g \\ \equiv & \{ \vdash \text{ term split (7)} \} \\ & (G \vdash h) \check{\sqcup} (G: c) \check{\sqsubseteq} g \\ \equiv & \{ \mathcal{P}3; \text{property of } \check{\sqcup} \} \\ & (G: c) \check{\sqsubseteq} g \\ \equiv & \{ \text{definition of } \check{\sqsubseteq} \text{ (19); reduce (10)} \} \\ & c \sqsubseteq \Pi \cdot g \\ \Leftarrow & \{ c = h \cdot x \text{ and } x \in H \setminus G \text{ arbitrary; definition of } \vdash \text{ (0); reduce (10)} \} \\ & \mathcal{P}4, \end{aligned} \quad (32)$$

where

$$\mathcal{P}4: \quad \sqcup \cdot (\sim G \vdash h) \sqsubseteq \Pi \cdot g.$$

In other words: all $h \cdot y$ for $y \in \sim G$ ($\sim G$ is G 's complement with respect to V) should be at most all $g \cdot z$ values. Establishing in turn the invariance of $\mathcal{P}4$:

$$\begin{aligned} & \sqcup \cdot (\sim D \vdash \mathcal{F} \cdot d) \sqsubseteq \Pi \cdot d \\ \Leftarrow & \{ d = g \vee (x: c); \text{reduce domain split (11); } \sim D \subseteq \sim G \} \end{aligned}$$

$$\begin{aligned}
& \mathbb{U} \cdot (\sim G \vdash \mathcal{F} \cdot d) \sqsubseteq \Pi \cdot g \sqcap c \\
\Leftarrow & \{(30); x \in H \setminus G, \text{ so from } \mathcal{P}4: c = h \cdot x \sqsubseteq \Pi \cdot g\} \\
& \mathbb{U} \cdot (\sim G \vdash h \checkmark \hat{c}) \sqsubseteq c \\
\equiv & \{\vdash \text{ term split (7); reduce domain split (12)}\} \\
& \mathbb{U} \cdot (\sim G \vdash h) \sqcup \mathbb{U} \cdot (\sim G: c) \sqsubseteq c \\
\equiv & \{\text{reduce constant term; property of } \mathbb{U}\} \\
& \mathbb{U} \cdot (\sim G \vdash h) \sqsubseteq c. \tag{33}
\end{aligned}$$

It follows that the invariance of $\mathcal{P}4$ in turn is guaranteed if x is chosen such that its h value c is maximal on domain $H \setminus G$! Now a maximum search is a rather trivial matter; however, since a witness x of the maximum is required it gives us the opportunity to demonstrate the use of the precondition for theorem.

If, in proof rule (15), we use for invariant $\mathcal{P}$

$$\mathcal{P}: \quad \mathbb{U} \cdot (\sim G \setminus A \vdash h) \sqsubseteq h \cdot x \quad \wedge \quad x \in H \setminus G,$$

then we conclude that for $A = \emptyset$ $\mathcal{P}$ implies the last member of (33), whereas for $A = H \setminus G$ the restricted domain function collapses, hence $x := H \setminus G$ suffices as initialization. Next, we evaluate precondition $y \notin A \wedge \mathcal{P}_{A \cup \{y\}}^A$ of the for-body statement list SL to be determined, first under the condition $h \cdot y \sqsubseteq h \cdot x$:

$$\begin{aligned}
& y \notin A \wedge \mathbb{U} \cdot (\sim G \setminus (A \cup \{y\}) \vdash h) \sqsubseteq h \cdot x \wedge h \cdot y \sqsubseteq h \cdot x \wedge x \in H \setminus G \\
\equiv & \{\text{set calculus; property of } \mathbb{U}\} \\
& y \notin A \wedge \mathbb{U} \cdot ((\sim G \setminus A) \setminus \{y\} \vdash h) \sqcup h \cdot y \sqsubseteq h \cdot x \wedge x \in H \setminus G \\
\Rightarrow & \{\text{domain split of } \vdash (8) \text{ and reduce (11)}\} \\
& \mathbb{U} \cdot (\sim G \setminus A \vdash h) \sqsubseteq h \cdot x \wedge x \in H \setminus G,
\end{aligned}$$

which equals $\mathcal{P}$. Alternatively, under the condition $h \cdot x \sqsubseteq h \cdot y$ we derive

$$\begin{aligned}
& y \notin A \wedge \mathbb{U} \cdot (\sim G \setminus (A \cup \{y\}) \vdash h) \sqsubseteq h \cdot x \wedge h \cdot x \sqsubseteq h \cdot y \wedge x \in H \setminus G \\
\Rightarrow & \{\text{transitivity of } \sqsubseteq; \text{property of } \mathbb{U}; \text{type information on } y\} \\
& y \notin A \wedge \mathbb{U} \cdot ((\sim G \setminus A) \setminus \{y\} \vdash h) \sqcup h \cdot y \sqsubseteq h \cdot y \wedge y \in H \setminus G \\
\Rightarrow & \{\text{domain split of } \vdash (8) \text{ and reduce (11)}\} \\
& \mathbb{U} \cdot (\sim G \setminus A \vdash h) \sqsubseteq h \cdot y \wedge y \in H \setminus G \\
\equiv & \{\text{substitution; definition of } \mathcal{P}\} \\
& \mathcal{P}_y^x,
\end{aligned}$$

so it appears that if we perform $x := y$ in the second alternative, both yield $\mathcal{P}$

in their postcondition. Hence, as proof rule (13) can be applied, and picking up the pieces we consequently find the **fourth approximation**

```

g,h:= ∅, ℱ·∅;
do H\G ≠ ∅ →
  x:= H\G;
  for y∈H\G → as h·x ⊆ h·y → x:=y sa rof;
  c:= h·x; h:= ℱ·(g ∨ (x: c)); g·x:= c
od,

```

where we eliminated d and simplified the g assignment. This algorithm is the end of the line if we have no further information about $\mathcal{F}$. Note that it is $O(\#V^2)$, if the assignment to h can be performed in $O(\#V)$ time.

In many problem instances $\mathcal{F}$ satisfies split property

$$\mathcal{F} \cdot (e \vee (x: c)) = \mathcal{F} \cdot e \oplus (S \cdot x \vdash K \cdot c \cdot x), \quad (34)$$

where K is an indexed family of mappings, with $K \cdot a \cdot u \in K \cdot a \cdot u \rightarrow T$ for all $a \in T$ and $u \in V$, and $\oplus$ is an associative binary infix operator. If $\mathcal{F}$ does not satisfy (34), one could say there is something wrong with the particular choice of the graph, because the successor function is supposed to precisely reflect the dependency relations between function values in all vertices.

If (34) does hold, the h assignment becomes

$$h := h \oplus (S \cdot x \vdash K \cdot c \cdot x), \quad (35)$$

and this allows us to refine the assignment further, using the expression accumulation theorem with accumulation function $\mathcal{G} \cdot A \cdot t = t \oplus (A \vdash K \cdot c \cdot x)$. Hence we check property (16) of the theorem:

$$\begin{aligned}
& \mathcal{G} \cdot (AU\{y\}) \cdot t \\
= & \text{\{definition of } \mathcal{G}\text{\}} \\
& t \oplus (AU\{y\} \vdash K \cdot c \cdot x) \\
= & \text{\{domain split of } \vdash \text{ (8), } y \notin A; \text{ choose } \oplus \text{ as arbitrary operator}\text{\}} \\
& t \oplus (A \vdash K \cdot c \cdot x) \oplus (\{y\} \vdash K \cdot c \cdot x) \\
= & \text{\{definition of } \mathcal{G}\text{\}} \\
& \mathcal{G} \cdot A \cdot t \oplus (\{y\} \vdash K \cdot c \cdot x),
\end{aligned}$$

so $f \cdot y$ is chosen to be $(\{y\} \vdash K \cdot c \cdot x)$, which is indeed independent of A and t . Finally, $\mathcal{F} \cdot \emptyset \cdot t = t$ is a tautology, so the restriction on the precondition of (35) holds. Hence (35) is refined by the for-statement

$$\underline{\text{for}} \ y \in S \cdot x \rightarrow \ h := h \ \dot{\oplus} \ (\{y\} \vdash K \cdot c \cdot x) \ \underline{\text{rof}}, \quad (36)$$

which may be expanded using (3) and (13). Also note that, since upon termination of the outer repetition $f=g=h$ holds, g has become superfluous, apart from its domain G . Substituting these findings into the fourth approximation we arrive at the final, **fifth approximation**

```

[[ G: P · V |
  h :=  $\mathcal{F} \cdot \emptyset$ ; G :=  $\emptyset$ ;
  do  $H \setminus G \neq \emptyset \rightarrow$ 
    [[ x: V; c: T | x ∈  $H \setminus G$ ;
      for  $y \in H \setminus G \rightarrow$  as  $h \cdot x \in h \cdot y \rightarrow x := y$  sa rof;
      c :=  $h \cdot x$ ;
      for  $y \in S \cdot x \rightarrow$ 
        as  $y \in K \cdot c \cdot x \cap H \rightarrow h \cdot y := h \cdot y \ \dot{\oplus} \ K \cdot c \cdot x \cdot y$ 
         $y \in K \cdot c \cdot x \setminus H \rightarrow h \cdot y := K \cdot c \cdot x \cdot y$ 
        sa
      rof; G :=  $G \cup \{x\}$ 
    ]]
  od  $\{ h = f \}$ 
]] .
```

Here, as in many applications, it is possible to implement a partial function using an array of size $\#V$, with "blanks" indicating vertices outside its domain, making membership tests for function domains ($y \in H$, $y \in K \cdot c \cdot x$) easy, $\mathcal{O}(1)$. Set $H \setminus G$ - remember that $G \subseteq H$! - may be implemented with a left adjusted array (a "stack"), making element selection also $\mathcal{O}(1)$, provided that in the maximum x search each new x value after $x := y$ is swapped to the back. Note that G itself is then obsolete. The details of the implementation phase are left to the reader, since the translation is a rather mechanical process. Also various optimizations, e.g. initializing h outside $d \cdot (\mathcal{F} \cdot \emptyset)$ with the left unit of $\dot{\oplus}$ and simplifying the central as-statement, take us beyond the scope of the paper.

Correcting for various differences in assumptions and notation the algorithm derived above corresponds to those in [BAC75, CAR79, TAR81, KNU77].

5. An example: ascending reachability

At this point we consider an application of the solution scheme derived in the previous section, called **ascending reachability**. This problem was proposed and solved in [REM85]. The problem is to determine the set of vertices reachable from a given set B via an ascending path. In this context "ascending path" means a path with the edge labels in ascending order, the edges being labelled by means of a weight function $t \in V \times V \rightarrow \mathbb{Z}$. On graph edges t is assumed to be finite.

It would not be fair to formally define the problem in a way most suited to formal manipulation, as one is tempted to do. Transformation of the "most natural" problem formulation to a suitable form is, in my opinion, an important part of the problem, and by no means the easiest. If possible, it should be based on heuristic principles that are more widely applicable.

To express the set of ascending reachables formally in a natural way, we need a path formalism. Paths are considered non-empty sequences of vertices, where all pairs of successive vertices x, y in such a sequence satisfy $y \in S \cdot x$. Thus $[x]$ is the zero-edge ("empty") path starting and ending in x , $[x, y]$ is the one-edge path from x to y , and so on. In the following p is understood to be a path in the graph. The vertices on path p are numbered, from 0 up to and including $\#p$, the path length, and $p \cdot i$, with $0 \leq i \leq \#p$, is the i -th vertex. If the last vertex of a path coincides with the first of another path, these paths may be concatenated with operator $\#$. Note that this is not the usual sequence concatenation! This brief summary will do for our purposes.

In terms of the path formalism the set of ascending reachables is given by

$$R = \{p: p \cdot 0 \in B \wedge asc \cdot p: p \cdot \#p\}, \quad (37)$$

with predicate asc being defined recursively by

$$asc \cdot [u], \quad (38)$$

$$asc \cdot (p \# [p \cdot \#p, u]) \equiv asc \cdot p \wedge t \cdot (p \cdot \#p) \cdot u \geq m \cdot p, \quad (39)$$

for $u \in S \cdot (p \cdot \#p)$, and function m in turn being defined by

$$m \cdot [u] = -\infty, \quad (40)$$

$$m \cdot (p \# [p \cdot \#p, u]) = t \cdot (p \cdot \#p) \cdot u, \quad (41)$$

again for $u \in S \cdot (p \cdot \#p)$, i.e. the weight of the last edge of path p , and, if restricted to ascending paths, indeed the maximum label weight on the path.

If one wishes to introduce a regular algebra [BAC75] to describe (38)–(41), a homomorphism is needed on the data structure of paths with concatenation, corresponding to the regular algebra product. However, (39) tends to violate associativity. In any case, $t \cdot u \cdot v$ interpreted as a matrix is not adequate to model the problem, and at least tupled matrix elements (e.g. ascendingness, first edge label, last edge label) are required. In this way associativity can be restored, but then the regular algebra product lacks a proper unit, and it is hard to define a regular algebra sum operator, especially one over which the product should distribute. If there is a way out, it may be rather messy.

On the other hand, the present problem is easily modeled using Knuth's formalism [KNU77]. The interested reader may verify that "superior" functions

$$g_{uv} \cdot c = \underline{\text{if}} \ t \cdot u \cdot v < c \rightarrow +\infty \ \square \ t \cdot u \cdot v \geq c \rightarrow t \cdot u \cdot v \ \underline{\text{fi}}$$

will do the trick. The superiority is expressed by the fact that $g_{uv} \cdot c \geq c$.

Returning to our problem we note that definition (37) has two drawbacks. Firstly, it requires the complicated path formalism, and secondly, it does not have the shape of a recursive equation, which proved to be so fruitful in last section's derivation, and for other graph problems [EIJ92]. It turns out that a simple recursive equation of that kind does not even exist for R : the ascending reachability of a given vertex x cannot be expressed in that of its predecessors. Intuitively, we need the "degree of ascending reachability" of x 's predecessors, in order to see whether paths to them can be extended with an edge to x . Therefore we apply a technique called **information extension** [EIJ92], and introduce a function $f \in V \rightarrow \mathbb{Z}$, defined by

$$f = (\lambda v: (\exists p:: ap \cdot p \cdot v): (\downarrow p: ap \cdot p \cdot v: m \cdot p)) , \quad (42)$$

using the shorthand

$$ap \cdot p \cdot v \equiv p \cdot 0 \in B \wedge p \cdot \#p = v \wedge asc \cdot p. \quad (43)$$

The minimum quantor in the term of (42) carries information about the most favourable incoming path in v , while the severe domain restriction serves two purposes: considering only interesting vertices with finite f values increases the computing efficiency, and in addition R is conveniently equal to $d \cdot f$ ($=F$).

Function f contains more information than R in the sense that R can be expressed in terms of f , whereas the reverse is not possible.

Next, we try to transform the, indeed cumbersome, expression for f into a recursive fixed point equation, by first expressing it as a countably infinite supremum, so that subsequently a fixed point theorem can be applied, like the one in section 3. We derive

$$\begin{aligned}
 & f \\
 = & \{ (42); \text{one-point rule} \} \\
 & (\lambda v: (\exists p, w: ap \cdot p \cdot w \wedge v \in \{w\}): (\downarrow p, w: ap \cdot p \cdot w \wedge v \in \{w\}: m \cdot p)) \\
 = & \{ \text{generalized lift-join rule (5), with } i:=v \text{ and } k:=(p, w) \} \\
 & (\downarrow p, w: ap \cdot p \cdot w: (w: m \cdot p)) \\
 = & \{ \text{domain split over all path lengths } k; \text{ introducing } Z \text{ below} \} \\
 & (\downarrow k: 0 \leq k: Z \cdot k), \tag{44}
 \end{aligned}$$

$$\text{with } Z \cdot k = (\downarrow p, w: ap \cdot p \cdot w \wedge \#p=k: (w: m \cdot p)). \tag{45}$$

If (44) is to be an explicit fixed-point expression like (23), $Z \cdot k$ should take the form $\mathcal{A}^k \cdot (Z \cdot 0)$, with $\mathcal{A}$ a uni-disjunctive operator. Substituting $k=0$ into (45), we find $p=[w]$, evaluate $ap \cdot [w]$ and $m \cdot [w]$ using (38) and (40), use (5) with $k:=w$, $\mathcal{D} \cdot k:= w \in B$, and $h \cdot k:= (w: -\omega)$, and arrive at

$$Z \cdot 0 = (\downarrow w: ap \cdot [w] \cdot w: (w: m \cdot [w])) = (\downarrow w: w \in B: (w: -\omega)) = (B: -\omega). \tag{46}$$

Next, trying to express $Z \cdot (k+1)$ in terms of $Z \cdot k$ we derive

$$\begin{aligned}
 & Z \cdot (k+1) \\
 = & \{ (45) \} \\
 & (\downarrow p, w: ap \cdot p \cdot w \wedge \#p=k+1: (w: m \cdot p)) \\
 = & \{ k \geq 0: \text{path split with dummy change } p:= p \# [v, w] \} \\
 & (\downarrow p, v, w: ap \cdot (p \# [v, w]) \cdot w \wedge \#p=k \wedge w \in S \cdot v: (w: m \cdot (p \# [v, w]))) \\
 = & \{ (39); (41); (43) \} \\
 & (\downarrow p, v, w: ap \cdot p \cdot v \wedge \#p=k \wedge t \cdot v \cdot w \geq m \cdot p \wedge w \in S \cdot v: (w: t \cdot v \cdot w)) \\
 = & \{ \text{term independent of } p; \text{generalized domain disjunction} \} \\
 & (\downarrow v, w: (\exists p: ap \cdot p \cdot v \wedge \#p=k \wedge t \cdot v \cdot w \geq m \cdot p) \wedge w \in S \cdot v: (w: t \cdot v \cdot w)) \\
 = & \{ \text{property of } \downarrow \} \\
 & (\downarrow v, w: (\exists p: ap \cdot p \cdot v \wedge \#p=k) \wedge \\
 & \quad (\downarrow p: ap \cdot p \cdot v \wedge \#p=k: m \cdot p) \leq t \cdot v \cdot w \wedge w \in S \cdot v: (w: t \cdot v \cdot w))
 \end{aligned}$$

$$\begin{aligned}
&= \{ \text{identifying domain and term of (45) using (5)} \} \\
&\quad (\downarrow v, w: v \in d \cdot (Z \cdot k) \wedge t \cdot v \cdot w \geq Z \cdot k \cdot v \wedge w \in S \cdot v: (w: t \cdot v \cdot w)) \\
&= \{ \text{introduce } \mathcal{L} \text{ below; calculus} \} \\
&\quad (\downarrow v, w: v \in d \cdot (Z \cdot k) \wedge w \in S \cdot v \cap \mathcal{L} \cdot (Z \cdot k) \cdot v: (w: t \cdot v \cdot w)) \\
&= \{ \text{shift } w \text{ to the term (9); definition of } \mathcal{A} \text{ below} \} \\
&\quad \mathcal{A} \cdot (Z \cdot k), \tag{47}
\end{aligned}$$

where

$$\mathcal{L} \cdot e \cdot v = \{ z: t \cdot v \cdot z \geq e \cdot v: z \}, \quad \text{and} \tag{48}$$

$$\mathcal{A} \cdot e = (\downarrow v: v \in E: (S \cdot v \cap \mathcal{L} \cdot e \cdot v \vdash t \cdot v)). \tag{49}$$

From (44), (46) and (47) we conclude

$$f = (\downarrow k: 0 \leq k: \mathcal{A}^k \cdot (B: -\omega)). \tag{50}$$

In order to prove that $\mathcal{A}$ is uni-disjunctive we use the last-but-two expression in derivation (47), abbreviating $w \in S \cdot v$ to $\mathcal{P}$ and $(w: t \cdot v \cdot w)$ to T . We derive

$$\begin{aligned}
&\mathcal{A} \cdot (\downarrow i: D \cdot i: \mathcal{E} \cdot i) \\
&= \{ \text{definition of } \mathcal{A}; \text{ domain of a generalized lift according to (5)} \} \\
&\quad (\downarrow v, w: v \in (U i: D \cdot i: E \cdot i) \wedge t \cdot v \cdot w \geq (\downarrow i: D \cdot i: \mathcal{E} \cdot i) \cdot v \wedge \mathcal{P}: T) \\
&= \{ \text{property of } U; \text{ term of generalized lift according to (5)} \} \\
&\quad (\downarrow v, w: (\exists i: D \cdot i: v \in E \cdot i) \wedge t \cdot v \cdot w \geq (\downarrow i: D \cdot i \wedge v \in E \cdot i: \mathcal{E} \cdot i \cdot v) \wedge \mathcal{P}: T) \\
&= \{ \text{property of } \downarrow \} \\
&\quad (\downarrow v, w: (\exists i: D \cdot i \wedge v \in E \cdot i: t \cdot v \cdot w \geq \mathcal{E} \cdot i \cdot v) \wedge \mathcal{P}: T) \\
&= \{ \text{generalized domain disjunction} \} \\
&\quad (\downarrow i: D \cdot i: (\downarrow v, w: v \in E \cdot i \wedge t \cdot v \cdot w \geq \mathcal{E} \cdot i \cdot v \wedge \mathcal{P}: T)) \\
&= \{ \text{definition of } \mathcal{A} \} \\
&\quad (\downarrow i: D \cdot i: \mathcal{A} \cdot (\mathcal{E} \cdot i)).
\end{aligned}$$

Hence all conditions for application of the generalized fixed point theorem are satisfied, and f in (50) is of the required shape (23). It follows that f is the least fixed point of an operator $\mathcal{F}$ defined by

$$\mathcal{F} \cdot e = (B: -\omega) \downarrow \mathcal{A} \cdot e, \tag{51}$$

and as such matches (20)-(21), the starting point of the algorithm derivation.

The uni-disjunctivity of $\mathcal{A}$ implies monotonicity of $\mathcal{F}$, if we take $\downarrow$ for $\cup$, and $\geq$ for $\sqcup$. It remains to check properties (26) and (35). We derive

$$\begin{aligned}
& \mathcal{F} \cdot (e \vee (x: c)) \\
= & \{ (51); (49) \} \\
& (B: -\infty) \quad \downarrow \quad (\downarrow v: v \in EU(x): (S \cdot v \cap \mathcal{L} \cdot (e \vee (x: c)) \cdot v \vdash t \cdot v)) \\
= & \{ \text{domain split}; (48); (49); (51) \} \\
& \mathcal{F} \cdot e \quad \downarrow \quad (\downarrow v: v = x: (S \cdot x \cap \mathcal{L} \cdot \hat{c} \cdot x \vdash t \cdot x)) \\
= & \{ \text{one-point rule; chaining } \vdash (6) \} \\
& \mathcal{F} \cdot e \quad \downarrow \quad (S \cdot x \vdash (\mathcal{L} \cdot \hat{c} \cdot x \vdash t \cdot x)),
\end{aligned}$$

so we recognize the required pattern of property (35) if we take $\downarrow$ for $\oplus$, and

$$\mathcal{K} \cdot c \cdot x = (\mathcal{L} \cdot \hat{c} \cdot x \vdash t \cdot x),$$

and at the same time, realizing that $\mathcal{K} \cdot c \cdot x \cdot y = t \cdot x \cdot y$ is defined only on domain $\mathcal{L} \cdot \hat{c} \cdot x$ (i.e. for $t \cdot x \cdot y \geq c$), it follows that (26) also holds. It seems we can instantiate the solution algorithm from the previous section with the above, refine the h initialization, from (46) and (51), by a for-statement, and after some minor simplifications we arrive at

```

[[ G: P · V | G := ∅;
  for x ∈ B → h · x := -∞ rof;
  do H \ G ≠ ∅ →
    [[ x: V; c: Z | x := H \ G;
      for y ∈ H \ G → as h · x ≥ h · y → x := y sa rof;
      c := h · x;
      for y ∈ S · x →
        as y ∈ H ∧ t · x · y ≥ c → h · y := h · y ↓ t · x · y
        y ∉ H ∧ t · x · y ≥ c → h · y := t · x · y
        sa
      rof; G := G ∪ {x};
    ]]
  od { h = f ∧ G = R }
]] .

```

After the repetition we have $G=H$, and in accordance with the remark at the beginning of this section the latter is equal to R , so we need not calculate R

separately after all. One more remark: if h is initialized to $+\infty$ outside B , the central as-statement reduces to

as $t \cdot x \cdot y \geq c \rightarrow h \cdot y := h \cdot y \downarrow t \cdot x \cdot y$ sa,

and upon termination h equals f , extended to the entire set V .

Comparing our solution to the one in [REM85], we observe that they are identical, except for the fact that Rem represents the domain part of function h with a left adjusted array, in which the order of the arguments matches the order of the vertices in HNG. That does not save memory, however, the worst case size of H still being $O(\#V)$.

The proof in [REM85] turns out to be entirely non-calculational, in fact verbal, though based on a complete set of more or less formal invariants, and it does not have the shape of a derivation. A flaw in the reasoning is the claim that the weight of any ascending path to the "minimal" vertex x in HNG must be at least $(\downarrow u: u \in \text{HNG}: h \cdot u)$, because any such path has an initial part in G ending in a vertex of HNG. This argument does not stick, because on HNG h is not necessarily equal to f , all these $h \cdot v$ values may still decrease.

It should be noted that the optimized solution using heaps in [REM85] is incomplete. Apart from restoring the heap-organized set HNG each time a vertex is added to H , also statement $h \cdot y := h \cdot y \downarrow t \cdot x \cdot y$ destroys the heap-order. Repairing this takes $O(\log \#V)$. It can be carried out many times for one and the same vertex, but is in total bounded by the number of edges $\#E$. So the entire solution then becomes $O((\#V + \#E) * \log \#V)$. Surprisingly, this is no worse than Rem claims, but then again his - faulty - solution is actually $O(\#V * \log \#V + \#E)$. As stated before, our solution is $O(\#V^2)$, which can be slightly worse than the heap solution, depending on the density of the graph.

6. Acknowledgements

I would like to thank Jaap van der Woude for his stimulating criticism during my lectures on graph algorithms: the abstraction he desired was always one level higher than I could offer. I also thank Lex Bijlsma and Roland Backhouse for their useful comments on the draft version of this paper, and one of the referees of the conference on mathematics of program construction, for pointing out an error in the derivation of the general solution.

7. References

- [BAC75] R.C. Backhouse and B.A. Carré, Regular Algebra Applied to Path-finding Problems, *J. Inst. Maths Applies* 15 (1975), 161-186
- [BAC89] R.C. Backhouse, An Exploration of the Bird-Meertens Formalism, *Lecture Notes of the International Summer School on Constructive Algorithmics, Part I* (Hollum, Ameland, 1989)
- [BIR88] R.S. Bird, *Constructive Functional Programming*, Lecture Notes for Marktoberdorf Summer School (1988)
- [BUR77] R.M. Burstall and John Darlington, A Transformation System for Developing Recursive Programs, *J. Ass. Comp. Mach.* 24 (1977), 44-67
- [CAI89] J. Cai and R. Paige, Program Derivation by Fixed Point Computation, *Sci. Comp. Progr.* 11 (1989), 197-261
- [CAR79] B. Carré, *Graphs and Networks* (Clarendon Press, Oxford, U.K., 1979)
- [COR90] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms* (McGraw-Hill, New York, 1990)
- [DAV90] B.A. Davey and H.A. Priestley, *Introduction to Lattices and Order* (Cambridge University Press, Cambridge, 1990)
- [DIJ59] E.W. Dijkstra, A Note on Two Problems in Connexion with Graphs, *Numer. Math.* 1 (1959), 269-271
- [DIJ76] E.W. Dijkstra, *A Discipline of Programming* (Prentice Hall, Englewood Cliffs, 1976)
- [DIJ88] E.W. Dijkstra and W.H.J. Feijen, *A Method of Programming* (Addison-Wesley, Reading, Mass., 1988)
- [EIJ92] J.P.H.W. van den Eijnde, *Program derivation in acyclic graphs and related problems* (Computing Science Notes 92/04, Technical Report, Eindhoven University of Technology, dept. Comp. Sci., 1992)
- [GAS88] A.J.M. van Gasteren, *On the Shape of Mathematical Arguments* (Thesis, Eindhoven University of Technology, 1988)
- [KNU77] D.E. Knuth, A generalization of Dijkstra's algorithm, *Inf. Proc. Lett.* 6 (1977), 1-5
- [LIS77] B. Liskov et al., Abstraction Mechanisms in CLU, *Comm. ACM* 20 (1977), 564-576
- [MEH84] K. Mehlhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP Completeness* (Springer-Verlag, Berlin, 1984)
- [MOR90] C. Morgan, *Programming from Specifications* (Prentice Hall, London, 1990)
- [REM84] M. Rem, Small Programming Exercises 5, *Sci. Comp. Progr.* 4 (1984),

323-333

- [REM85] M. Rem, Small Programming Exercises 7, *Sci. Comp. Progr.* 5 (1985), 219-229
- [REM86] M. Rem, Small Programming Exercises 13, *Sci. Comp. Progr.* 7 (1986), 243-248
- [REY81] J.C. Reynolds, *The Craft of Programming* (Prentice-Hall, Englewood Cliffs, N.J., 1981)
- [ROL90] Proceedings of the Ameland Workshop on Constructive Algorithmics: *The Role of Relations in Program Development* (Utrecht Univ., 1990)
- [TAR81] R.E. Tarjan, A Unified Approach to Path Problems, *J. ACM* 28 (1981), 577-593

Appendix A: Proof of the precondition for-theorem

Use $\mathcal{P}$ as invariant of the corresponding do statement in (14), and $\#A$ as variant function, then the correctness of (15) follows easily from the annotated version of the algorithm below.

[[	A: $\mathbb{P} \cdot W$; u: W;	$\{\mathcal{P}_W^A\}$
	A := W;	$\{\mathcal{P}\}$
	<u>do</u> $A \neq \emptyset \rightarrow$	$\{A \neq \emptyset \wedge \mathcal{P}\}$
	u: $\in A$;	$\{u \in A \wedge \mathcal{P}\}$
	A := $A \setminus \{u\}$;	$\{u \notin A \wedge \mathcal{P}_{A \cup \{u\}}^A\}$
	SL	$\{\mathcal{P}\}$
	<u>od</u>	$\{\mathcal{P} \wedge A = \emptyset, \text{ hence } \mathcal{P}_\emptyset^A\}$
]]		

All transitions are trivial, only when applying the definition of $u: \in A$ the independence of $\mathcal{P}$ on x is crucial, while the decrease of the variant function follows from $\#(A - \{u\}) = \#A - 1$ (for $u \in A$), provided that SL does not refer to A.

Proof outline for the expression accumulation theorem

It is easy to see that the desired refinement, say FOR, of the assignment statement $r := \mathcal{G} \cdot W \cdot r$ should satisfy

$$\{r = \mathcal{G} \cdot \emptyset \cdot \rho \wedge W = \Omega\} \text{ FOR } \{r = \mathcal{G} \cdot \Omega \cdot \rho\} \quad (\text{A0})$$

for all values of the logical variables ρ and Ω . The validity of the precondition requirement has been taken into account. Hoare triple (A0) suggests a refinement of *FOR* by a repetition with invariant

$$\mathcal{I}: r = \mathcal{G} \cdot (\Omega \setminus A) \cdot \rho, \quad (\text{A1})$$

with a set variable A , initially set to W , shrinking to $\emptyset$. The correctness of the repetition below follows from the annotation in brackets:

$$\begin{aligned} & \llbracket A: \mathbb{P} \cdot W; u: W \mid \{r = \mathcal{G} \cdot \emptyset \cdot \rho \wedge W = \Omega\} \\ & \quad A := W; \quad \{r = \mathcal{G} \cdot (\Omega \setminus A) \cdot \rho\} \\ & \quad \underline{\text{do}} \ A \neq \emptyset \rightarrow \quad \{A \neq \emptyset \wedge r = \mathcal{G} \cdot (\Omega \setminus A) \cdot \rho\} \\ & \quad \quad u \in A; \quad \{u \in A \wedge r = \mathcal{G} \cdot (\Omega \setminus A) \cdot \rho\} \\ & \quad \quad A := A \setminus \{u\}; \quad \{u \notin A \wedge r = \mathcal{G} \cdot (\Omega \setminus (A \cup \{u\})) \cdot \rho\} \\ & \quad \quad r := r \oplus f \cdot u \quad \{r = \mathcal{G} \cdot (\Omega \setminus A) \cdot \rho\} \\ & \quad \underline{\text{od}} \quad \{A = \emptyset \wedge r = \mathcal{G} \cdot (\Omega \setminus A) \cdot \rho, \text{ so } r = \mathcal{G} \cdot \Omega \cdot \rho\} \\ & \rrbracket. \end{aligned}$$

The reader is invited to carefully examine all steps. Now, applying the definition of the for-statement (14), we find that (17) is indeed a correct refinement of *FOR*, and hence of $r := \mathcal{G} \cdot W \cdot r$.

Appendix B: Proof of the generalized fixed point theorem

First we prove that r , as defined in (23), satisfies (24). We derive

$$\begin{aligned} & e \sqcup \mathcal{A} \cdot r \\ = & \{ (23); \mathcal{A} \text{ is uni-disjunctive} \} \\ & e \sqcup (\bigcup k: 0 \leq k: \mathcal{A} \cdot (\mathcal{A}^k \cdot e)) \\ = & \{ \text{definition of exponentiation; dummy } k := k-1 \} \\ & \mathcal{A}^0 \cdot e \sqcup (\bigcup k: 1 \leq k: \mathcal{A}^k \cdot e) \\ = & \{ \text{domain split; (23)} \} \\ & r. \end{aligned}$$

Secondly, it must be shown that r satisfies (25). Using a property of $\sqcup$ and definition (23) of r , we have, for arbitrary s :

$$r \sqsubseteq s \equiv (\forall k: 0 \leq k: A^k \cdot e \sqsubseteq s) . \quad (B0)$$

The right-hand side of (B0) can be proven with induction to k , assuming the validity of $e \sqcup A \cdot s \sqsubseteq s$. The base, $A^0 \cdot e = e \sqsubseteq s$, follows trivially. For the induction step it follows that

$$\begin{aligned} & A^{k+1} \cdot e \\ = & \{ \text{definition of exponentiation} \} \\ & A \cdot (A^k \cdot e) \\ \sqsubseteq & \{ A \text{ is monotonic; induction hypothesis } A^k \cdot e \sqsubseteq s \} \\ & A \cdot s \\ \sqsubseteq & \{ \text{assumption; property of } \sqcup \} \\ & s . \end{aligned}$$

Finally, the uniqueness of the solution of (24) and (25) is to be shown. Let r_0 and r_1 both satisfy (24) and (25). Application of (25), for r_0 , taking s to be r_1 , we find $r_0 \sqsubseteq r_1$, since r_0 satisfies (24), and hence $e \sqcup A \cdot r_0 \sqsubseteq r_0$. The reverse $r_1 \sqsubseteq r_0$ follows analogously. Antisymmetry then implies $r_0 = r_1$.

In this series appeared:

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

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

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

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

91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.