CALCULATING A GARBAGE COLLECTOR

Ulrich Berger

Mathematisches Institut, Ludwig-Maximilians-Universität
Theresienstr. 39, D-8000 München 2, Fed. Rep. Germany

Werner Meixner, Bernhard Möller

Institut für Informatik, Technische Universität München
Postfach 20 24 20, D-8000 München 2, Fed. Rep. Germany

1 Introduction

In this paper we give a calculational account of a garbage collection algorithm. There have been a number of papers with a similar programme (cf. [Broy, Pepper 82], [Dewar et al. 82], [van Diepen, de Roever 86], [Möller 87]). However, these treatments either were not completely formal or suffered from using an inconvenient representation of the problem. With this paper we want to introduce partial maps and their algebraic properties as a vehicle for treating pointer structures, as dealt with in garbage collection, both conveniently and formally. Moreover, we want to develop the algorithms to a level which can actually be transcribed directly into machine code allowing the use of overwriting and the like; this was not achieved in most of the papers cited. It also turns out that the theory of finite maps allows very concise high-level specifications of the subproblems involved in garbage collection.

The situation in which garbage collection becomes necessary is the following: The store, which accomodates a large number of records referencing one another through pointers, is exhausted, i.e., there is (almost) no more free storage left for the allocation of new records. Usually there is a distinguished set of **entry pointers** to the pointer structure which is given by the values of the currently active variables of the program that operates on the store. Only those records reachable through chains of references from the entry pointers actually need to be saved; all other records are inaccessible and thus the corresponding storage can be reclaimed.

What does garbage collection mean in a more abstract sense? To explain this, we liberate ourselves from the concrete contents of the records in the store and consider only their interrelationship through the pointers. This leads to a graph-like structure in which the nodes correspond to the records, and the arcs correspond to the pointers. More precisely, we consider a **graph** to be a partial map $G : \mathsf{node} \longrightarrow \mathsf{node}^*$ that assigns to each node from a subset of the set node of all possible nodes the *sequence* of its successor nodes. We use sequences rather than sets, since the fields of a record are ordered and may contain repetitions. If, by some process, we can distinguish a proper subgraph G' of G such that G' contains all the nodes accessible from the entry nodes, then G can be said to contain garbage about which we could as well forget. In this case, garbage collection means to compute G' from G and to operate on G' successively.

Getting more concrete again, we work with representations of such graphs. The task then consists in computing a representation of G' from one of G. In this paper we treat representations, called **states**, of graphs in a linear storage. For a state, the restriction to a substate usually leads to gaps in the storage; i.e., there are cells the contents of which have no meaning for the represented graph. Now, one possibility of garbage collection consists in detecting these gaps and compactifying the meaningful part by copying it to an initial interval of the storage; then a contiguous rest of the storage becomes free for further use.

We develop a corresponding algorithm from a specification at the level of graphs together with a notion of their representation in a linear storage. Although the final algorithm is fairly intricate in that it employs various ways of chaining and unchaining certain sets of storage cells, its structure becomes clear through a modular development by transformations.

2 Preliminaries

2.1 Transforming Nondeterministic Specifications

The development in this paper is based on the applicative part of the wide spectrum language CIP-L (cf. [Bauer et al. 85]), which includes a full typed lambda-calculus as well as pre-algorithmic specification constructs such as quantification, set comprehension, and non-deterministic choice. Typed λ-abstractions are written in the form

$$(\mathsf{m}\ x)\mathsf{n} : E$$

where m is the argument type and n is the result type. Recursion is introduced by the fixpoint construction

$$(\mathsf{fix}\ x : E)$$

where the type of x is that of E. Declarations can then be introduced by the definitional transformation rule

$$\frac{\lceil\ \mathsf{m}\ x\ \equiv\ E\ ;\ F\ \rfloor}{((\mathsf{m}\ x)\mathsf{n} : F)(\mathsf{fix}\ x : E)\ .}\ \Updownarrow\ [$$

CIP-L has a mathematical semantics that associates with each expression E the set $\mathcal{B}[\![E]\!]$ of possible values; $\mathcal{B}[\![E]\!]$ is called the **breadth** of E. The special value $\bot$ models the possibility of an erroneous or nonterminating computation. We set

$$\begin{aligned}\mathrm{DEF}[\![E]\!] &\overset{\text{def}}{\Leftrightarrow} \bot \notin \mathcal{B}[\![E]\!]\ ,\\ \mathrm{DET}[\![E]\!] &\overset{\text{def}}{\Leftrightarrow} |\mathcal{B}[\![E]\!]| = 1\ ,\end{aligned}$$

and call E **defined** resp. **determinate** if $\mathrm{DEF}[\![E]\!]$ resp. $\mathrm{DET}[\![E]\!]$ holds. For convenience, the semantical values are also considered as expressions. The details of the semantic description can be found in [Bauer et al. 85].

Based on the breadth, two fundamental relations between expressions are defined: E_1 and E_2 are called **equivalent** if $\mathcal{B}[\![E_1]\!] = \mathcal{B}[\![E_2]\!]$; we denote this by $E_1\ \equiv\ E_2$. This "strong" or meta-equality is not to be confused with "weak" (i.e. strict) equality tests $=$ in the language itself: The formula

$$\bot \equiv \bot$$

is valid. However,

$$\bot{=}\bot\ \not\equiv\ \mathsf{true}\ ;$$

rather we have

$$\bot{=}\bot\ \equiv\ \bot\ .$$

Equivalences are also denoted in the form of transformation rules, viz. as

$$\begin{array}{ccl} & E_1 & \\ \rule{3em}{0.4pt} & \updownarrow & \rule{3em}{0.4pt}[C \\ & E_2 & \end{array}$$

where C is a (possibly empty) list of applicability conditions, i.e., of conditions sufficient for the validity of the equivalence. Similarly, E_2 is called a **descendant** of E_1 if $\mathcal{B}[\![E_2]\!] \subseteq \mathcal{B}[\![E_1]\!]$; we denote this by $E_1 \supseteq E_2$ and, in the form of a transformation rule, as

$$\begin{array}{ccl} & E_1 & \\ \rule{3em}{0.4pt} & \downarrow & \rule{3em}{0.4pt}[C \\ & E_2 & \end{array}$$

where C again is a list of applicability conditions.

As an important aid in specifying and developing recursive routines we use assertions about the objects involved. They are formulated as Boolean expressions of the language. Given such an expression P, we use the notation

$$P \rhd E$$

as an abbreviation for the expression

$$\mathsf{if}\ P\ \mathsf{then}\ E\ \mathsf{else}\ \perp\ \mathsf{fi}\ .$$

A collection of useful algebraic properties of this construct can be found in [Möller 89]. Our principal use of it is within parameter restrictions for functions (cf. [Bauer, Wössner 82, Bauer et al. 85]): Let R be a Boolean expression possibly involving the identifier x. Then the declaration

$$\mathsf{funct}\ f \equiv (\mathsf{m}\ x : R)\mathsf{n} : E$$

of function f with parameter x restricted by R and with body E is by definition equivalent to

$$\mathsf{funct}\ f \equiv (\mathsf{m}\ x)\mathsf{n} : R \rhd E\ .$$

This means that f is undefined for all arguments x that violate the restriction R. If f is recursive, R has to hold also for the parameters of the recursive calls to ensure definedness; hence in this case R corresponds to invariants as known from imperative programming.

We now want to develop our central transformation rule for obtaining descendants of nondeterministic functions. Consider a function abstraction

$$(\mathsf{m}\ x)\mathsf{n} : E\!<\!x, f\!>$$

and the corresponding recursion

$$R \mathrel{\widehat{=}} (\mathsf{fix}\ f : (\mathsf{m}\ x)\mathsf{n} : E\!<\!x, f\!>)\ .$$

The notation $E < x, f >$ indicates that only the free identifiers x and f are of interest in E; for expressions A and G then $E < A, G >$ is a shorthand for the substitution $E[\![A, G\ \mathsf{for}\ x, f]\!]$. We want to find a criterion under which the recursion R is a descendant (and hence a correct implementation) of some function F which may be viewed as the specification. The idea is to employ noetherian induction on the arguments of F and R. Let therefore $\prec$ be a determinate expression of kind $\mathsf{funct}(\mathsf{m}, \mathsf{m})\mathsf{bool}$ that denotes a noetherian strict-order on the set of values of kind m different from $\perp$. We define, for arbitrary function g of kind $\mathsf{funct}(\mathsf{m})\mathsf{n}$,

$$\mathrm{BELOW}[\![x, g]\!]\!<\!y\!> \overset{\text{def}}{\equiv} y \prec x \rhd g(y)\ .$$

This denotes an expression that agrees with $g(y)$ for $y \prec x$ and is undefined for all other y; hence it corresponds to a restriction of g to values strictly less than x. Now we call $E{<}x, f{>}$ **recurrent wrt.** $\prec$ iff

$$E{<}x, g{>} \equiv E{\ll}x, \text{BELOW}[\![x, g]\!]{\gg},$$

where g is a fresh identifier and $E \ll x, \text{BELOW}[\![x, g]\!] \gg$ results from $E < x, g >$ by replacing all calls $g(A)$ by $\text{BELOW}[\![x, g]\!]{<}A{>}$. This means that for all g the value of $E < x, g >$ depends at most on values $g(y)$ with $y \prec x$. We call $E < x, f >$ **recurrent** if there is some $\prec$ such that $E{<}x, f{>}$ is recurrent wrt. $\prec$. Then we have

Theorem 2.1.1
The following rule (DESCENDANT-FIXPOINT) is correct:

$$\frac{\qquad\qquad}{} \begin{array}{c} F \\ \big\downarrow \\ (\mathsf{fix}\ f : (\mathsf{m}\ x)\mathsf{n} : E{<}x, f{>}) \end{array} \frac{\qquad\qquad}{} \left[\begin{array}{l} E{<}x, f{>} \ \text{recurrent} \\ F(x) \ \supseteq \ E{<}x, F{>} \end{array} \right.$$

Proof: We abbreviate the output scheme again to R. Let $E < x, f >$ be recurrent w.r.t $\prec$ and assume $F(x) \supseteq E{<}x, F{>}$. We show $F(u) \supseteq R(u)$ by noetherian induction on the values $u \not\equiv \perp$ of kind m. By induction hypothesis we may assume $F(v) \supseteq R(v)$ for all $v \prec u$, and thus

$$\text{BELOW}[\![u, F]\!] \ \supseteq \ \text{BELOW}[\![u, R]\!] \ . \qquad\qquad (+)$$

Then

$$\begin{array}{lll} & F(u) & \\ \supseteq & E{<}u, F{>} & \\ \equiv & E{\ll}u, \text{BELOW}[\![u, F]\!]{\gg} & \text{(since } E{<}x, f{>} \text{ is recurrent)} \\ \supseteq & E{\ll}u, \text{BELOW}[\![u, R]\!]{\gg} & \text{(by (+) and } \supseteq\text{-monotonicity of } E{<}x, f{>}) \\ \equiv & E{<}u, R{>} & \text{(since } E{<}x, f{>} \text{ is recurrent)} \\ \equiv & R(u) & \text{(by definition of } R). \end{array}$$

Finally, since all expressions of kind $\mathsf{funct(m)n}$ denote strict functions in CIP-L, we have $F(\perp) \equiv \perp \equiv R(\perp)$ and thus also $F(\perp) \supseteq R(\perp)$. ■

Note that the descendant relation implies that for totally defined F the recursively defined function f always terminates.

2.2 Notations for Specific Data Structures

2.2.1 Sequences

Given a set M, we denote by M^* the set of all finite sequences of M-elements. The empty sequence is denoted by $<>$. Then $M^+ \overset{\text{def}}{\equiv} M^* \backslash \{<>\}$ is the set of all nonempty finite sequences of M-elements. The singleton sequence consisting just of $x \in M$ is denoted by $<x>$. Concatenation of sequences is denoted by $+$. Accordingly, for sequences α_i we write

$$\sum_{i \in [1:n]} \alpha_i$$

for

$$\alpha_1 + \ldots + \alpha_n \ .$$

We write $x{\in}\alpha$ to express that x occurs in the sequence α. Moreover, $set(\alpha) \overset{\text{def}}{\equiv} \{x \mid x{\in}\alpha\}$. The length of a sequence α is denoted by $|\alpha|$. For $0 < i \leq |\alpha|$ we write $\alpha[i]$ for the i-th element of α.

Given a function $f : M \longrightarrow N$ we denote by $f*$ its unique homomorphic extension mapping M^* to N^*, i.e.,

$$f * (<x_1, \ldots, x_n>) \equiv <f(x_1), \ldots, f(x_n)> .$$

2.2.2 Sets and Orders

The cardinality of a set M is denoted by $|M|$. Moreover, the complement of a subset N of M is denoted by $\overline{N}$.

For a finite subset $s \subseteq M$ of a linearly ordered set M we denote by $sort(s)$ the unique repetition-free sequence that contains exactly the elements of s in ascending order from left to right. M is **well-ordered** if every nonempty subset $s \subseteq M$ contains a least element $min(s)$. If then M also contains a greatest element ∞, we set

$$inf(s) \overset{\text{def}}{\equiv} min(s \cup \{\infty\}) .$$

The greatest element of s (if any) is denoted by $max(s)$. If M contains a least element $-\infty$ we set

$$sup(s) \overset{\text{def}}{\equiv} max(s \cup \{-\infty\}) .$$

Define for a subset $s \subseteq M$ of an ordered set M

$$\begin{array}{rcl} s^{\vee} & \overset{\text{def}}{\equiv} & \{y \mid \exists\, x \in s : x \leq y\} \\ s^{\wedge} & \overset{\text{def}}{\equiv} & \{y \mid \exists\, x \in s : y \leq x\} \\ isinterval(s) & \overset{\text{def}}{\Leftrightarrow} & s \;\equiv\; s^{\vee} \cap s^{\wedge} . \end{array}$$

s is an **initial interval** of M if $s^{\wedge} \;\equiv\; s$. For abbreviation we write $\overset{\vee}{x}$, $\hat{x}$ instead of $\{x\}^{\vee}$, $\{x\}^{\wedge}$. Moreover, we set

$$\begin{array}{l} [x : y] \overset{\text{def}}{\equiv} \overset{\vee}{x} \cap \hat{y} , \\ [x : y[\overset{\text{def}}{\equiv} [x : y]\backslash\{y\} , \\ ]x : y] \overset{\text{def}}{\equiv} [x : y]\backslash\{x\} , \\ ]x : y[\overset{\text{def}}{\equiv} [x : y]\backslash\{x, y\} . \end{array}$$

Suppose again that M is well-ordered and contains a greatest element ∞. Then for $s \subseteq M$ and $x \in M$ we define

$$succ_s(x) \overset{\text{def}}{\equiv} inf(s\backslash\{x\}) .$$

2.3 Relations and Maps

2.3.1 Basic Notions

A **binary relation** r between sets M and N is a subset $r \subseteq M \times N$. We write $\downarrow r$ and $\uparrow r$ for domain and range of r, resp. Moreover, we define

$$set(r) \overset{\text{def}}{\equiv} \downarrow r \;\cup \uparrow r .$$

For $s \subseteq M$ we denote by $s{\uparrow}r$ the image of s under r. Likewise, $t{\downarrow}r$ is the inverse image of $t \subseteq N$ under r.

A **(partial) map** m from a set M to a set N is a relation $m \subseteq M \times N$ such that $(x,y) \in m \ \wedge\ (x,z) \in m \ \Rightarrow\ y \ \equiv\ z$. Some of our notation derives from this set view of maps. E.g., by $\emptyset$ we denote the empty partial map from M to N.

Next, we define

$$[s \mapsto y] \ \overset{\text{def}}{\equiv}\ \{(x,y) \mid x \in s\}\ ;$$

this is the constant map assigning y to every element of s. In this construction, y frequently is obtained by applying another map m. To cope in an algebraically convenient way with partialities, we set

$$[s \mapsto m(x)] \ \overset{\text{def}}{\equiv}\ \emptyset \quad \text{for } x \notin {\downarrow}m\ .$$

Symmetrically, we also define

$$[\{n(x)\} \mapsto y)] \ \overset{\text{def}}{\equiv}\ \emptyset \quad \text{for } x \notin {\downarrow}n\ .$$

In using these notations we omit singleton set braces, i.e., we write $x{\uparrow}m$, $y{\downarrow}m$, and $[x \mapsto y]$ instead of $\{x\}{\uparrow}m$, $\{y\}{\downarrow}m$, $[\{x\} \mapsto y]$. Note that $[x \mapsto y] \ \equiv\ \{(x,y)\}$.

Lemma 2.3.1 (Functionality)

Let m be a map.
(1) $|m| \ \equiv\ |{\downarrow}m|$.
(2) $t_1 \cap t_2 \equiv\ \emptyset \ \Rightarrow\ t_1{\downarrow}m \cap t_2{\downarrow}m \ \equiv\ \emptyset$.

A map is **injective** if the predicate *isinjective*(m) holds, where

$$\mathit{isinjective}(m) \ \overset{\text{def}}{\Leftrightarrow}\ \forall\ x,y \in {\downarrow}m:\ m(x) \ \equiv\ m(y) \ \Rightarrow\ x \ \equiv\ y\ .$$

In this case we denote its inverse map by m^{-1}.

Lemma 2.3.2

$\mathit{isinjective}(m) \ \Leftrightarrow\ \forall\ s,t : (s \cap t){\uparrow}m \ \equiv\ s{\uparrow}m \cap t{\uparrow}m$.

By $\circ$ we denote the composition of partial maps:

$$s{\uparrow}(n \circ m) \ \equiv\ (s{\uparrow}m){\uparrow}n\ .$$

2.3.2 Map Union

Two maps $m, n : M \longrightarrow N$ are **compatible** if $m(x) \ \equiv\ n(x)$ for all $x \in {\downarrow}m \ \cap {\downarrow}n$. In particular this holds if ${\downarrow}m \ \cap {\downarrow}n \ \equiv\ \emptyset$. For compatible m, n their union $m \cup n$ is again a map. This generalizes to families $(m_i)_{i \in I}$ of maps (I may even be infinite) if the maps m_i are pairwise compatible; we then write $\bigcup\limits_{i \in I} m_i$ for the union map. If $I \ \equiv\ \emptyset$, we set $\bigcup\limits_{i \in I} m_i \ \equiv\ \emptyset$ as well.

Lemma 2.3.3

$$\begin{array}{ll}
(1) & m \equiv \bigcup_{x \in \downarrow m} [x \mapsto m(x)] \qquad \text{(domain-oriented representation)} \\
(2) & m \equiv \bigcup_{z \in \uparrow m} [(z\downarrow m) \mapsto z] \qquad \text{(range-oriented representation)} \\
(3) & k \circ \bigcup_{i \in I} m_i \equiv \bigcup_{i \in I} (k \circ m_i) \\
(4) & (\bigcup_{i \in I} m_i) \circ k \equiv \bigcup_{i \in I} (m_i \circ k) \\
(5) & \mathit{isinjective}(m) \Rightarrow m^{-1} \equiv \bigcup_{x \in \downarrow m} [m(x) \mapsto x] \ .
\end{array}$$

We want to develop a recursive routine for calculating particular unions over finite, well-ordered index sets. The specification reads

```
funct $mapunion$ ≡ (finset $s$, map $f, g$ : $isinjective(f)$) map :
    $\bigcup_{x \in s} [f(x) \mapsto g(x)]$ .
```

The parameter restriction serves to make the union well-defined. If $s \equiv \emptyset$, by definition $mapunion(s, f, g) \equiv \emptyset$. Otherwise, for arbitrary $z \in s$ we have $s \equiv \{z\} \cup s\backslash\{z\}$ and hence

$$\begin{array}{ll}
 & mapunion(s, f, g) \\
\equiv & \bigcup_{x \in \{z\} \cup s\backslash\{z\}} [f(x) \mapsto g(x)] \\
\equiv & \bigcup_{x \in \{z\}} [f(x) \mapsto g(x)] \cup \bigcup_{x \in s\backslash\{z\}} [f(x) \mapsto g(x)] \\
\equiv & [f(z) \mapsto g(z)] \cup mapunion(s\backslash\{z\}, f, g) \ .
\end{array}$$

In particular, this holds for $z \equiv min(s)$. Thus we obtain the recursion (for well-founded M termination is obvious)

```
funct $mapunion$ ≡ (finset $s$, map $f, g$ : $isinjective(f)$) map :
    if $s = \emptyset$ then $\emptyset$
                else m $z \equiv min(s)$ ;
                     $[f(z) \mapsto g(z)] \cup mapunion(s\backslash\{z\}, f, g)$  fi .
```

To make this into a tail-recursion and to avoid the repeated use of *min* , we assume that m has a greatest element ∞ and define the embedding

```
funct $mu$ ≡ ( finset $s$, map $f, g$, map $r$, m $x$ :
                 $isinjective(f) \wedge x = min(s)$) map :
    $r \cup mapunion(s, f, g)$ .
```

We have

$$mapunion(s, f, g) \equiv mu(s, f, g, \emptyset, min(s))$$

Now we calculate

```
    $mu(s, f, g, r, x)$
≡   $isinjective(f) \wedge x = min(s) \rhd$
    $r \cup$ if $s = \emptyset$ then $\emptyset$
                  else m $z \equiv min(s)$ ;
                       $[f(z) \mapsto g(z)] \cup mapunion(s\backslash\{z\}, f, g)$  fi
≡   $isinjective(f) \wedge x = min(s) \rhd$
    if $s = \emptyset$ then $r \cup \emptyset$
              else m $z \equiv min(s)$ ;
```

```
                    $r \cup [f(z) \mapsto g(z)] \cup mapunion(s\backslash\{z\}, f, g)$  fi
    $\equiv$  $isinjective(f) \;\wedge\; x = min(s) \rhd$
        if $s = \emptyset$ then $r$
                    else  $r \cup [f(x) \mapsto g(x)] \cup mapunion(s\backslash\{x\}, f, g)$  fi
    $\equiv$  $isinjective(f) \;\wedge\; x = min(s) \rhd$
        if $s = \emptyset$ then $r$
                    else  $mu(s\backslash\{x\}, f, g, r \cup [f(x) \mapsto g(x)], min(s\backslash\{x\}))$  fi
    $\equiv$  $isinjective(f) \;\wedge\; x = min(s) \rhd$
        if $s = \emptyset$ then $r$
                    else  $mu(s\backslash\{x\}, f, g, r \cup [f(x) \mapsto g(x)], succ_s(x))$  fi .
```

Since the termination behaviour has not changed, we are left with

```
funct $mu$ $\equiv$ ( finset $s$, map $f, g$, map $r$, m $x$ :
                  $isinjective(f) \;\wedge\; x = min(s)$) map :
    if $s = \emptyset$ then $r$
                else  $mu(s\backslash\{x\}, f, g, r \cup [f(x) \mapsto g(x)], succ_s(x))$  fi .
```

2.3.3 Restriction and Overwriting

The **restriction** of a map $m : M \longrightarrow N$ to a set $s \subseteq M$ is

$$m|s \overset{\text{def}}{\equiv} m \cap (s \times N) \ .$$

Moreover,

$$m \ominus s \overset{\text{def}}{\equiv} m|\overline{s}.$$

Here again we omit singleton set braces, i.e., we write $m \ominus x$ instead of $m \ominus \{x\}$.

Another useful operation is **map overwriting**: Given maps $m, n : M \longrightarrow N$ we define

$$m \mathbin{\leftarrow\!\!+} n \overset{\text{def}}{\equiv} (m \ominus {\downarrow}n) \cup n \ .$$

Hence,

$$(m \mathbin{\leftarrow\!\!+} n)(x) \equiv \text{if } x \in {\downarrow}n \text{ then } n(x) \text{ else } m(x) \text{ fi}.$$

Lemma 2.3.4

1. $(m \ominus s) \cup n \equiv m \mathbin{\leftarrow\!\!+} n$ provided ${\downarrow}m \cap s \equiv {\downarrow}m \cap {\downarrow}n$.

2. $l \mathbin{\leftarrow\!\!+} (m \cup n) \equiv l \mathbin{\leftarrow\!\!+} n$ provided $m \subseteq l$ and m and n are compatible (Annihilation).

A number of further properties of these operations can be found in the appendix; we shall use them freely without explicit reference.

3 Storage Graphs and Their Representation

3.1 Storage Graphs

Storage graphs are intended to model the accessibility relations between records as given by the pointers in the records. Since the fields of a record are ordered and may contain repetitions of pointers, the usual notion of a directed graph where each node is connected to a *set* of successor nodes is not adequate for our purposes. Rather we consider *sequences* of successor nodes. Also, since we shall have to deal with arbitrary parts of such storage graphs, we generalize in another direction by allowing the successor map to "leave" the part under consideration.

Let node be a set of "nodes". A **pseudo-graph** is a partial map $G : \mathsf{node} \longrightarrow \mathsf{node}^*$. The nodes in $G(x)$ are called the **immediate successors** of the node $x \in \downarrow G$. We set

$$nodes(G) \overset{\text{def}}{\equiv} \downarrow G \cup \bigcup_{x \in \downarrow G} set(G(x)) \ .$$

A **storage graph** then is a pseudo-graph G such that $nodes(G) \subseteq \downarrow G$. The more general case of pseudo-graphs models "dangling references" which will occur e.g. during the copying phase of our garbage collection algorithm when only part of the accessible cells have been copied to their new locations.

3.2 Memories and Allocations

We also want to talk about the representation of such pseudo-graphs in a (linear) memory. A **memory** is a pair $\mathcal{M} \equiv (\mathsf{cell}, \Box)$, where cell is a denumerable set of **cells** and $\Box \in \mathsf{cell}$ is a distinguished cell called the **anchor** of the memory.

A (partial) map $m : \mathsf{cell} \longrightarrow \mathsf{cell}$ is called a **state** of M if the predicate $isstate(m)$ holds where

$$isstate(m) \overset{\text{def}}{\equiv} \Box \notin \downarrow m \ .$$

A **linear memory** is a memory $\mathcal{M} \equiv (\mathsf{cell}, \Box)$ such that cell is linearly ordered by some ordering $\leq$ in which $\Box$ is the least element. In the sequel we assume cell to be the set $\mathbb{N}$ of natural numbers and $\Box \equiv 0$.

Let now G be a pseudo-graph. We want to represent G by a state of a linear memory. The idea is to store each node of G together with its successors in a block of contiguous cells; the node itself is marked by cell contents $\Box$. In practice, this leading cell frequently is used for storing information about a record, such as its length, type information, and the like. However, as stated in the introduction, we abstract from such details; $\Box$ seems an adequate substitute here.

Let $x \in \downarrow G$. Then the size of the corresponding block is given by

$$size(x) \overset{\text{def}}{\equiv} 1 + |G(x)| \ .$$

An **allocation** of G is a partial map $g : \mathsf{node} \longrightarrow \mathsf{cell}$ such that $nodes(G) \subseteq \downarrow g$ and $\Box \notin \uparrow g$. g is supposed to assign to each node in $\downarrow G$ the starting cell of its block; the additional condition ensures that $\Box$ can in fact be used to characterize block beginnings. Given an allocation g of G we define for $x \in \downarrow G$

$$block(x, g) \overset{\text{def}}{\equiv} [g(x) \mapsto \Box] \cup \bigcup_{i \in [1:|G(x)|]} [(g(x) + i) \mapsto g(G(x)[i])] \ .$$

We set

$$fields(x, g) \overset{\text{def}}{\equiv} (\downarrow block(x, g)) \backslash \{g(x)\} \equiv \{g(x) + i \mid i \in [1 : |G(x)|]\} \ .$$

We have

Lemma 3.2.1

$$isinterval(\downarrow block(x, g)) \ .$$

An allocation g of G is **overlap-free** if

$$x \not\equiv y \;\Rightarrow\; \downarrow\!block(x,g) \cap \downarrow\!block(y,g) \;\equiv\; \emptyset\ .$$

For an overlap-free allocation we can extend the function *block* to sets $s \subseteq \downarrow\!G$ by setting

$$block(s,g) \;\overset{\text{def}}{\equiv}\; \bigcup_{x\in s} block(x,g)\ .$$

Then the following state is a **representation** of G:

$$blockrep(G,g) \;\overset{\text{def}}{\equiv}\; block(\downarrow\!G,g)\ .$$

Lemma 3.2.2
Let $G_1, G_2 \subseteq G$ and let g be an allocation of G. Then

$$blockrep(G_1 \cup G_2, g) \;\equiv\; blockrep(G_1,g) \cup blockrep(G_2,g)\ .$$

We call a state m a **pseudo-graph state** if the predicate $ispgstate(m)$ holds where

$$ispgstate(m) \;\overset{\text{def}}{\Leftrightarrow}\; \exists\, G, g : m \equiv blockrep(G,g)\ .$$

Then

$$keys(m) \;\overset{\text{def}}{\equiv}\; \Box\!\downarrow\!m \;\equiv\; (\downarrow\!G)\!\uparrow\!g$$

denotes the set of **keys** of m, i.e., the set of addresses that are the beginnings of blocks. We set

$$arcs(m) \;\overset{\text{def}}{\equiv}\; m \ominus keys(m)\ .$$

Moreover we define

$$followers(m,y) \;\equiv\; \textsf{if } y+1 \in \downarrow\!arcs(m) \textsf{ then } <y+1> + followers(m,y+1) \textsf{ else } <> \textsf{ fi.}$$

Lemma 3.2.3
Let g be an overlap-free allocation of G and let $m \;\equiv\; blockrep(G,g)$.

1. $\Box \notin \downarrow\!m \;\;\wedge\;\; \uparrow\!m \cap \downarrow\!m \subseteq keys(m)\ .$
2. g is injective.
3. $arcs(m) \;\equiv\; \bigcup_{x\in keys(m)} followers(m,x)\ .$
4. For all $x \in \downarrow\!G$ we have $fields(x,g) \;\equiv\; set(followers(m,g(x)))$.
5. $\downarrow\!G \;\equiv\; keys(m)\!\downarrow\!g$ and
   $G \;\equiv\; \bigcup_{x\in keys(m)} [x\!\downarrow\!g \;\mapsto\; g^{-1} * (m * followers(m,x))]$.

1. means that $\Box$ in m designates an "improper" cell that does not contain a value. 4. and 5. show how a graph can be reconstructed from its block representation.

Assume now that the set $\downarrow\!G$ of nodes is linearly ordered by some order $\leq$. Then an allocation g is called **order-preserving** if

$$\forall\, x,y \in \downarrow\!G : x \leq y \;\Rightarrow\; g(x) \leq g(y)\ .$$

Lemma 3.2.4
Let g be an injective and order-preserving allocation. Then for $x, y \in {\downarrow}G$ we have

1. $x < y \;\Rightarrow\; g(x) < g(y)$.
2. $x \leq y$ iff $g(x) \leq g(y)$, i.e., ${\downarrow}G$ and ${\uparrow}g$ are order-isomorphic.

Proof:
1. is immediate from the injectivity of g.
2. We only need to show ($\Leftarrow$). Assume $g(x) \leq g(y)$ but $x \not\leq y$. By linearity of $\leq$ then $y < x$ and hence also $g(y) < g(x)$ by 1. Contradiction!

■

This lemma holds for arbitrary order-preserving injections between linear orders. For the special case of storage linearization we get

Lemma 3.2.5
Let g be an overlap-free and order-preserving allocation. Then

$$x < y \;\Rightarrow\; {\downarrow}\mathit{block}(x, g) < {\downarrow}\mathit{block}(y, g) \ ,$$

where for subsets s, t of an ordered set

$$s < t \overset{\text{def}}{\Leftrightarrow} \forall\, x \in s : \forall\, y \in t : x < y \ .$$

Proof: Since g is order-preserving, $g(x) < g(y)$. Let now $u \in {\downarrow}\mathit{block}(x, g)$ and $v \in {\downarrow}\mathit{block}(y, g)$ and assume $v \leq u$. Since $g(y) \leq v$, we have then $g(x) \leq g(y) \leq u$ and hence $g(y) \in {\downarrow}\mathit{block}(x, g)$ by $\mathit{isinterval}({\downarrow}\mathit{block}(x, g))$. But then ${\downarrow}\mathit{block}(x, g) \cap {\downarrow}\mathit{block}(y, g) \not\equiv \emptyset$, a contradiction. ■

We now want to characterize contiguous block representations. Call an overlap-free allocation g of G **gap-free** if $\mathit{isinterval}({\downarrow}\mathit{blockrep}(G, g))$ holds. g is called **perfect** if it is overlap-free, order-preserving, and gap-free.

Theorem 3.2.6
Let $g : \mathsf{node} \longrightarrow \mathsf{cell}$ be a perfect allocation of G and $x \in {\downarrow}G$ such that x is not the maximum of ${\downarrow}G$. Then

$$g(\mathit{succ}_{{\downarrow}G}(x)) \;\equiv\; g(x) + \mathit{size}(x) \ .$$

Proof: Since g is order-preserving and injective, we have $g(x) < g(\mathit{succ}_{{\downarrow}G}(x))$. By the above lemma then ${\downarrow}\mathit{block}(x, g) < {\downarrow}\mathit{block}(\mathit{succ}_{{\downarrow}G}(x), g)$ and in particular ${\downarrow}\mathit{block}(x, g) < \{\mathit{succ}_{{\downarrow}G}(x)\}$. Let

$$z \overset{\text{def}}{\equiv} \mathit{max}({\downarrow}\mathit{block}(x, g)) \;\equiv\; g(x) + |G(x)| \ .$$

Assume $z < u < g(\mathit{succ}_{{\downarrow}G}(x))$ for some u. By $\mathit{isinterval}({\downarrow}\mathit{blockrep}(G, g))$ we get $u \in {\downarrow}\mathit{blockrep}(G, g)$, say $u \in {\downarrow}\mathit{block}(w, g)$ for some $w \in {\downarrow}G$. Then $g(w) \leq u < g(\mathit{succ}_{{\downarrow}G}(x))$ and hence $w < \mathit{succ}_{{\downarrow}G}(x)$ since g is order-preserving and injective. This is equivalent to $w \leq x$. But then ${\downarrow}\mathit{block}(w, g) \leq {\downarrow}\mathit{block}(x, g)$ contradicting $g(x) \leq z < u \in {\downarrow}\mathit{block}(w, g)$. Therefore

$$g(\mathit{succ}_{{\downarrow}G}(x)) \;\equiv\; z + 1 \;\equiv\; g(x) + |G(x)| + 1 \;\equiv\; g(x) + \mathit{size}(x) \ .$$

■

Corollary 3.2.7

Let $g : \mathsf{node} \longrightarrow \mathsf{cell}$ be a perfect allocation of G and $x \in \downarrow G$ such that $| \overset{\vee}{x} \cap \downarrow G| > i$. Then

$$g(\mathit{succ}^i_{\downarrow G}(x)) \;\equiv\; g(x) + \sum_{j<i} \mathit{size}(\mathit{succ}^j_{\downarrow G}(x)) \;.$$

Proof: Induction on i. ■

A pseudo-graph state m is called **compressed** if the predicate $\mathit{iscgstate}(m)$ holds, where

$$\mathit{iscgstate}(m) \;\overset{\text{def}}{\Leftrightarrow}\; \mathit{ispgstate}(m) \;\wedge\; \downarrow m \;\equiv\; (\downarrow m)^{\wedge} \backslash \{\square\} \;,$$

i.e., if its domain is an initial interval of $\mathsf{cell}\backslash\{\square\}$. By the above corollary, given a pseudo-graph G there is exactly one perfect allocation g of G such that $\mathit{blockrep}(G, g)$ is compressed; g is called the **compressing allocation** of G.

4 The Garbage Collection Problem

4.1 The Reachable Subgraph

When garbage collection becomes necessary, there is a set of immediate entries into the store. All blocks reachable from these entries need to be saved whereas everything else is garbage to be removed. We first treat the reachability problem at the level of storage graphs.

Let G be a pseudo-graph and let $x, y \in \mathsf{node}$. A sequence $p \in \mathsf{node}^*$ is called a **path in** G **from** x **to** y iff the predicate $\mathit{ispath}_G(p, x, y)$ holds, where

$$\begin{aligned}\mathit{ispath}_G(p, x, y) \;\overset{\text{def}}{\Leftrightarrow}\;\; & p \in \mathsf{node}^+ \;\wedge\; \mathit{set}(p)\backslash\{\mathit{last}(p)\} \subseteq \downarrow G \;\wedge \\ & x = \mathit{first}(p) \;\wedge\; y = \mathit{last}(p) \;\wedge \\ & \forall\, i \in [1 : |p| - 1] : p[i+1] \in G(p[i]).\end{aligned}$$

We define

$$\mathsf{nodeset} \;\overset{\text{def}}{\equiv}\; \{s | s \subseteq \mathsf{node} \;\wedge\; |s| < \infty\} \;.$$

Given a pseudo-graph G, a node $x \in \mathit{nodes}(G)$ is **reachable** from some set $s \in \mathsf{nodeset}$ iff the predicate $\mathit{isreachable}_G(x, s)$ holds where

$$\mathit{isreachable}_G(x, s) \;\overset{\text{def}}{\Leftrightarrow}\; \exists\, z \in s, p \in \mathsf{node}^+ : \mathit{ispath}_G(p, z, x)$$

The function $\mathit{rnset}_G : \mathsf{nodeset} \longrightarrow \mathsf{nodeset}$, defined by

$$\mathit{rnset}_G(s) \;\overset{\text{def}}{\equiv}\; \{\; x \in \mathsf{node} \mid \mathit{isreachable}_G(x, s) \;\}$$

computes the set of nodes reachable from a given set.

Now, for a storage graph G and a set $s \subseteq \downarrow G$, the **subgraph of** G **reachable from** s is $G_s \overset{\text{def}}{\equiv} G|\mathit{rnset}_G(s)$. It is easily verified that the pseudo-graph G_s indeed is a storage graph.

4.2 Statement of the Garbage Collection Problem

Consider a storage graph G with a perfect allocation g and a set $s \subseteq \downarrow G$. Moreover, set $n \overset{\text{def}}{\equiv} \mathit{blockrep}(G, g)$. The problem consists in computing the reachable subgraph G_s together with the compressing allocation g_s of G_s as well as the corresponding state $n_s \overset{\text{def}}{\equiv} \mathit{blockrep}(G_s, g_s)$. In fact, ultimately we are interested in an algorithm that computes n_s directly from n.

4.3 A First Analysis of the Problem

Assume G, g, n and G_s, g_s, n_s as in Section 4.2. Define

$$n_1 \overset{\text{def}}{\equiv} \mathit{blockrep}(G_s, g) \ .$$

Thus, n_1 is the accessible but not yet compressed part of the storage. To compute n_s from n_1, define a collapsing map $k : \downarrow n_1 \longrightarrow \mathsf{cell}$ by

$$k(g(x) + i) \overset{\text{def}}{\equiv} g_s(x) + i$$

for $x \in \downarrow G_s$ and $0 \leq i \leq |G_s(x)|$.

Lemma 4.3.1

(1) k is well-defined. Moreover, k is an order-embedding and $\uparrow k \ \equiv \ \downarrow n_s$ (which is an initial interval of cell).

(2) $n_s \circ k \ \equiv \ k \circ n_1$.

Proof: (1) is obvious.

(2)
$$\begin{aligned} & n_s(k(g(x) + i)) \\ \equiv\ & n_s(g_s(x) + i) \\ \equiv\ & g_s(G(x)[i]) \\ \equiv\ & k(g(G(x)[i])) \\ \equiv\ & k(n_1(g(x) + i)) \end{aligned}$$

■

The preceding considerations suggest a decomposition of the problem into the following parts:

1. Compute G_s from G and s (reachability).
2. Compute $n_c \ \equiv \ \mathit{blockrep}(G_s, g)$ from G_s and g.
3. Compute k from n_c.
4. Compute n_s from n_c and k (copying).

Diagrammatically the situation can be described as follows:

$$\begin{tikzcd} G \arrow[r, Rightarrow, "reach"] \arrow[d, Rightarrow, "g"] & G_s \arrow[d, Rightarrow, "g"] & g_s \\ \mathit{blockrep}(G, g) \arrow[r, phantom, "\supseteq"] & \mathit{blockrep}(G_s, g) \arrow[r, Rightarrow, "k"] & \mathit{blockrep}(G_s, g_s) \end{tikzcd}$$

Here the double arrows indicate that the respective functions are of second order, since their arguments, viz. pseudo-graphs and states, are mappings themselves.

5 Determining the Reachable Subgraph

5.1 A First Recursive Solution

Let pgraph be the set of pseudo-graphs and $\mathsf{nodeset}$ be the set of finite subsets of node. The specification of the reachability problem now reads

$$\begin{array}{l}\mathsf{funct}\ \ reach\ \equiv\ (\mathsf{pgraph}\ G, \mathsf{nodeset}\ s : s \subseteq \downarrow G)\mathsf{pgraph} :\\ \quad G|rnset_G(s)\ .\end{array}$$

From the specification of *rnset* we derive immediately the property

Lemma 5.1.1

$rnset_G(s \cup t)\ \equiv\ rnset_G(s) \cup rnset_G(t)$ for arbitrary $s, t \in \mathsf{nodeset}$.

To derive a recursive algorithm for *reach*, let a pseudo-graph G and a node set $s \subseteq \downarrow G$ be given. If $s \equiv\ \emptyset$, then

$$G|rnset_G(s)\ \equiv\ G|\emptyset\ \equiv\ \emptyset\ .$$

If $s \not\equiv \emptyset$, we may choose an arbitrary node $z \in s$. Then we have

Lemma 5.1.2

(1) $ispath_G(p, x, y)\ \wedge\ z \notin lead(p)\ \equiv\ ispath_{G\ominus z}(p, x, y)$,
where $lead(p)\ \overset{\text{def}}{\equiv}\ <p[1], \ldots, p[|p| - 1]>$.

(2) $\exists\ x \in s : \exists\ p : ispath_G(p, x, y)\ \wedge\ z \in lead(p)\ \equiv$
$\exists\ p : ispath_G(p, z, y)$.

(3) $\exists\ p : ispath_G(p, z, y)\ \wedge\ z \neq y\ \equiv$
$\exists\ u \in set(G(z))\backslash\{z\} : \exists\ p_1 : ispath_G(p_1, u, y)\ \wedge\ z \notin set(p_1)$.

Proof: (1) Holds, because $\downarrow(G \ominus z)\ \equiv\ \downarrow G\backslash\{z\}$.

(2) Assume $ispath_G(p, x, y) \wedge z \in lead(p)$. Let q be the shortest postfix of p that contains z. Then $first(q) = z$ and $last(q) = y$, so that $ispath_G(q, z, y)$. The reverse implication is trivial.

(3) Assume $ispath_G(p, z, y)\ \wedge\ z \neq y$. Let q be the shortest postfix of p that contains z. Then $first(q) = z$ and $last(q) = y$, so that $ispath_G(q, z, y)$. Since $z \neq y$ we have $|q| \geq 2$ and there must be a $u \in set(G(z))$ with $u = q[2]$. But then $ispath_G(rest(q), u, y)$ and, by construction, $z \notin set(rest(q))$ as well as $u \neq z$.
Assume conversely that $ispath_G(p_1, u, y)\ \wedge\ z \notin set(p_1)$ for some p_1 and $u \in set(G(z))\backslash\{z\}$. Then $ispath_G(<z> +p_1, z, y)$ and $z \neq y$.

■

Now we obtain

$$\begin{array}{ll} & rnset_G(s)\\ \equiv & \{y \mid \exists\ x \in s : \exists\ p : ispath_G(p, x, y)\}\\ \equiv & \{y \mid \exists\ x \in s : \exists\ p : ispath_G(p, x, y)\ \wedge\ z \notin lead(p)\} \cup\\ & \{y \mid \exists\ x \in s : \exists\ p : ispath_G(p, x, y)\ \wedge\ z \in lead(p)\}\\ \equiv & \{y \mid \exists\ x \in s\backslash\{z\} : \exists\ p : ispath_G(p, x, y)\ \wedge\ z \notin lead(p)\} \cup\\ & \{y \mid \exists\ x \in s : \exists\ p : ispath_G(p, x, y)\ \wedge\ z \in lead(p)\}\\ \equiv & \text{(by (1))}\end{array}$$

$$
\begin{array}{ll}
& \{y \mid \exists\, x \in s \backslash \{z\} : \exists\, p : \mathit{ispath}_{G \ominus z}(p,x,y)\} \cup \\
& \{y \mid \exists\, x \in s : \exists\, p : \mathit{ispath}_G(p,x,y) \ \wedge \ z \in \mathit{lead}(p)\} \\
\equiv & \mathit{rnset}_{G \ominus z}(s \backslash \{z\}) \cup \\
& \{y \mid \exists\, x \in s : \exists\, p : \mathit{ispath}_G(p,x,y) \ \wedge \ z \in \mathit{lead}(p)\} \\
\equiv & \text{(by (2))} \\
& \mathit{rnset}_{G \ominus z}(s \backslash \{z\}) \cup \\
& \{y \mid \exists\, p : \mathit{ispath}_G(p,z,y)\} \\
\equiv & \mathit{rnset}_{G \ominus z}(s \backslash \{z\}) \cup \mathit{rnset}_G(\{z\}) \ .
\end{array}
$$

Moreover,

$$
\begin{array}{ll}
& \mathit{rnset}_G(\{z\}) \\
\equiv & \{y \mid \exists\, p : \mathit{ispath}_G(p,z,y)\} \\
\equiv & \{y \mid \exists\, p : \mathit{ispath}_G(p,z,y) \ \wedge \ y = z\} \cup \\
& \{y \mid \exists\, p : \mathit{ispath}_G(p,z,y) \ \wedge \ y \neq z\} \\
\equiv & \text{(since } \mathit{ispath}(<z>,z,z) \text{ holds)} \\
& \{z\} \cup \{y \mid \exists\, p : \mathit{ispath}_G(p,z,y) \ \wedge \ y \neq z\} \\
\equiv & \text{(by (3))} \\
& \{z\} \cup \{y \mid \exists\, u \in \mathit{set}(G(z)) \backslash \{z\} : \exists\, p_1 : \mathit{ispath}_G(p_1,u,y) \ \wedge \ z \notin \mathit{set}(p_1)\} \\
\equiv & \{z\} \cup \{y \mid \exists\, u \in \mathit{set}(G(z)) \backslash \{z\} : \exists\, p_1 : \mathit{ispath}_G(p_1,u,y) \ \wedge \ z \notin \mathit{lead}(p_1) \ \wedge \ z \neq y\} \\
\equiv & \text{(by (1))} \\
& \{z\} \cup \{y \mid \exists\, u \in \mathit{set}(G(z)) \backslash \{z\} : \exists\, p_1 : \mathit{ispath}_{G \ominus z}(p_1,u,y) \ \wedge \ z \neq y\} \\
\equiv & \{z\} \cup \{y \mid \exists\, u \in \mathit{set}(G(z)) \backslash \{z\} : \exists\, p_1 : \mathit{ispath}_{G \ominus z}(p_1,u,y)\} \backslash \{z\} \\
\equiv & \{z\} \cup \mathit{rnset}_{G \ominus z}(\mathit{set}(G(z)) \backslash \{z\}) \backslash \{z\} \\
\equiv & \{z\} \cup \mathit{rnset}_{G \ominus z}(\mathit{set}(G(z)) \backslash \{z\}) \ .
\end{array}
$$

Hence,

$$
\begin{array}{ll}
& \mathit{rnset}_G(s) \\
\equiv & \mathit{rnset}_{G \ominus z}(s \backslash \{z\}) \cup \{z\} \cup \mathit{rnset}_{G \ominus z}(\mathit{set}(G(z)) \backslash \{z\}) \\
\equiv & \text{(by Lemma 5.1.1)} \\
& \mathit{rnset}_{G \ominus z}((s \backslash \{z\}) \cup (\mathit{set}(G(z)) \backslash \{z\})) \cup \{z\} \\
\equiv & \mathit{rnset}_{G \ominus z}((s \cup \mathit{set}(G(z))) \backslash \{z\}) \cup \{z\} \\
\equiv & \{z\} \cup \mathit{rnset}_{G_1}(\hat{s})
\end{array}
$$

where

$$
\begin{array}{rcl}
G_1 & \overset{\text{def}}{\equiv} & G \ominus z \ , \\
\hat{s} & \overset{\text{def}}{\equiv} & (s \cup \mathit{set}(G(z))) \backslash \{z\} \ .
\end{array}
$$

Now we obtain

$$
\begin{array}{ll}
& \mathit{reach}(G,s) \\
\equiv & G | \mathit{rnset}_G(s) \\
\equiv & G | (\{z\} \cup \mathit{rnset}_{G_1}(\hat{s})) \\
\equiv & ([z \mapsto G(z)] \cup G_1) | (\{z\} \cup \mathit{rnset}_{G_1}(\hat{s})) \\
\equiv & [z \mapsto G(z)] | \{z\} \cup [z \mapsto G(z)] | \mathit{rnset}_{G_1}(\hat{s}) \cup G_1 | \{z\} \cup G_1 | \mathit{rnset}_{G_1}(\hat{s}) \\
\equiv & [z \mapsto G(z)] \cup [z \mapsto G(z)] | \mathit{rnset}_{G_1}(\hat{s}) \cup \emptyset \cup G_1 | \mathit{rnset}_{G_1}(\hat{s}) \\
\equiv & \text{(since } [z \mapsto G(z)] | \mathit{rnset}_{G_1}(\hat{s}) \subseteq [z \mapsto G(z)]\text{)} \\
& [z \mapsto G(z)] \cup G_1 | \mathit{rnset}_{G_1}(\hat{s}) \ .
\end{array}
$$

Next, we observe that we can shrink $\hat{s}$ even further using

Lemma 5.1.3
For a pseudo-graph H and a set $s \in \mathsf{nodeset}$ we have

$$H|\mathit{rnset}_H(s) \;\equiv\; H|\mathit{rnset}_H(s \cap {\downarrow}H) \;.$$

Proof: Note first that, by definition of *ispath*, we have for $z \notin {\downarrow}H$ that

$$\mathit{ispath}(p, z, x) \;\equiv\; x = z \;\wedge\; p = <z> \;\;.$$

Therefore, if $t \cap {\downarrow}H \;\equiv\; \emptyset$ then $\mathit{rnset}_H(t) \;\equiv\; t$. Now,

$$\begin{aligned}
& \mathit{rnset}_H(s) \\
\equiv\;& \mathit{rnset}_H((s \cap {\downarrow}H) \cup (s \backslash {\downarrow}H)) \\
\equiv\;& \text{(by Lemma 5.1.1)} \\
& \mathit{rnset}_H(s \cap {\downarrow}H) \cup \mathit{rnset}_H(s \backslash {\downarrow}H) \\
\equiv\;& \mathit{rnset}_H(s \cap {\downarrow}H) \cup s \backslash {\downarrow}H
\end{aligned}$$

and hence

$$\begin{aligned}
& H|\mathit{rnset}_H(s) \\
\equiv\;& H|(\mathit{rnset}_H(s \cap {\downarrow}H) \cup s \backslash {\downarrow}H) \\
\equiv\;& H|\mathit{rnset}_H(s \cap {\downarrow}H) \cup H|(s \backslash {\downarrow}H) \\
\equiv\;& H|\mathit{rnset}_H(s \cap {\downarrow}H) \cup \emptyset \\
\equiv\;& H|\mathit{rnset}_H(s \cap {\downarrow}H) \;.
\end{aligned}$$

■

This gives us finally

$$\begin{aligned}
& \mathit{reach}(G, s) \\
\equiv\;& [z \mapsto G(z)] \cup G_1|\mathit{rnset}_{G_1}(\hat{s}) \\
\equiv\;& \text{(by the previous Lemma)} \\
& [z \mapsto G(z)] \cup G_1|\mathit{rnset}_{G_1}(\hat{s} \cap {\downarrow}G_1) \\
\equiv\;& [z \mapsto G(z)] \cup G_1|\mathit{rnset}_{G_1}(((s \cup \mathit{set}(G(z)))\backslash\{z\}) \cap {\downarrow}G_1) \\
\equiv\;& [z \mapsto G(z)] \cup G_1|\mathit{rnset}_{G_1}((s \cup \mathit{set}(G(z))) \cap {\downarrow}G_1) \\
\equiv\;& \text{(since } (s \cup \mathit{set}(G(z))) \cap {\downarrow}G_1 \subseteq {\downarrow}G_1) \\
& [z \mapsto G(z)] \cup \mathit{reach}(G_1, (s \cup \mathit{set}(G(z)) \cap {\downarrow}G_1) \\
\equiv\;& [z \mapsto G(z)] \cup \mathit{reach}(G_1, s_1)
\end{aligned}$$

where

$$s_1 \overset{\text{def}}{\equiv} (s \cup \mathit{set}(G(z))) \cap {\downarrow}G_1 \;.$$

Defining

$$\mathit{elem}(s) \overset{\text{def}}{\equiv} s \neq \emptyset \;\triangleright\; \mathsf{some\ node}\ x : x \in s \;,$$

we obtain from this the following recursive version for *reach* :

```
funct $reach \equiv$ (pgraph $G$, nodeset $s : s \subseteq {\downarrow}G$)pgraph :
     if $s = \emptyset$ then $\emptyset$
             else node $z \equiv elem(s)$ ;
                  pgraph $G_1 \equiv G \ominus z$ ;
                  nodeset $s_1 \equiv (s \cup set(G(z))) \cap {\downarrow}G_1$ ;
                  $[z \mapsto G(z)] \cup reach(G_1, s_1)$              fi .
```

Termination is guaranteed, since $|{\downarrow}G_1| < |{\downarrow}G|$. The restriction of *reach* to proper graphs yields a solution to the original subgraph problem, viz. $G_s \equiv reach(G, s)$ for every $s \in$ nodeset and every graph G.

5.2 Improvement of the Solution

Since map union is associative, we may by a standard method embed *reach* into a tail-recursive function *reach1* :

$$reach(G, s) \equiv reach1(\emptyset, G, s)$$

where

```
funct $reach1 \equiv$ (pgraph $H$, pgraph $F$, nodeset $s : s \subseteq {\downarrow}F$) pgraph :
    if $s = \emptyset$ then $H$
        else node $z \equiv elem(s)$ ;
            pgraph $H_1 \equiv H \cup [z \mapsto F(z)]$ ;
            pgraph $F_1 \equiv F \ominus z$ ;
            nodeset $s_1 \equiv (s \cup set(F(z))) \cap {\downarrow}F_1$ ;
            $reach1(H_1, F_1, s_1)$                    fi .
```

In a second step we eliminate the parameter F from the recursion: Obviously the property

$$P_G(H, F) \overset{\text{def}}{\Leftrightarrow} F = G \backslash H$$

is an invariant of *reach1* (i.e., $P_G(H, F) \Rightarrow P_G(H_1, F_1)$), and $P_G(\emptyset, G)$ holds. Therefore we may replace F by $G \backslash H$ in *reach1* . Moreover, $s \subseteq {\downarrow}F \subseteq {\downarrow}G \ \wedge \ F|s = G|s$ is an additional invariant for the call $reach1(\emptyset, G, s)$; hence it follows that

$$\begin{aligned} & s_1 \\ \equiv \ & (s \cup set(F(z))) \cap {\downarrow}F_1 \\ \equiv \ & (s \cup set(G(z))) \cap ({\downarrow}G \backslash {\downarrow}H_1) \\ \equiv \ & (s \cup (set(G(z))) \cap {\downarrow}G)) \backslash {\downarrow}H_1 \ . \end{aligned}$$

If furthermore G is even a storage graph, we have $set(G(z)) \subseteq {\downarrow}G$ and therefore

$$s_1 \equiv (s \cup set(G(z))) \backslash {\downarrow}H_1 \ .$$

Thus our final algorithm reads

```
funct $reach \equiv$ (pgraph $G$, nodeset $s : s \subseteq {\downarrow}G$) pgraph :
    $reach2(\emptyset, s)$
where
funct $reach2 \equiv$ (pgraph $H$, nodeset $s : s \subseteq {\downarrow}G$) pgraph :
    if $s = \emptyset$ then $H$
        else node $z \equiv elem(s)$ ;
            pgraph $H_1 \equiv H \cup [z \mapsto G(z)]$ ;
            nodeset $s_1 \equiv (s \cup set(G(z))) \backslash {\downarrow}H_1$ ;
            $reach2(H_1, s_1)$                    fi .
```

This version corresponds to algorithm (7) derived in [Berghammer et al. 87] which computes the set ${\downarrow}G_s$ of nodes reachable from s (replace pgraph by nodeset, $[z \mapsto G(z)]$ by z, and ${\downarrow}H_1$ by H_1).

6 Copying Pointer Structures

6.1 Statement of the Problem

We now consider the task of copying a state to another part of a memory. Let $\mathcal{M} = (\mathsf{cell}, \Box)$ be a memory and let m, n be states of $\mathcal{M}$ such that $\Box \in set(m) \cap set(n)$. We call n a **copy** of m if there is a total bijection $k : set(m) \longrightarrow set(n)$ such that $k(\Box) \equiv \Box$ and the following diagram commutes:

$$\begin{tikzcd}
{\downarrow}m \arrow[r, "m"] \arrow[d, "k"] & {\uparrow}m \arrow[d, "k"] \\
{\downarrow}n \arrow[r, "n"] & {\uparrow}n
\end{tikzcd}$$

This means that $k \circ m \equiv n \circ k$ and, since k is bijective, that $n \equiv k \circ m \circ k^{-1}$. Hence, given k, we can compute n from m in two passes: First we form $k \circ m$ which means that the cell contents as given by m are updated to contain the corresponding addresses of the copy ("pointer relocation pass"); then we compose with k^{-1} which means the actual transport of the new contents to the new locations ("copying pass").

6.2 Copying Pass

Given $p \equiv k \circ m$, the copying pass is easily performed. First, by totality of k, we have ${\downarrow}p \equiv {\downarrow}m$ $(\subseteq {\downarrow}k)$. Now

$$\begin{aligned}
& p \circ k^{-1} \\
\equiv\ & \text{(by Lemma 2.3.3)} \\
& p \circ \bigcup_{x \in {\downarrow}k} [k(x) \mapsto x] \\
\equiv\ & \text{(by Lemma 2.3.3)} \\
& \bigcup_{x \in {\downarrow}k} p \circ [k(x) \mapsto x] \\
\equiv\ & \bigcup_{x \in {\downarrow}k} [k(x) \mapsto p(x)] \\
\equiv\ & \bigcup_{x \in {\downarrow}p} [k(x) \mapsto p(x)] \ ,
\end{aligned}$$

since $[k(x) \mapsto p(x)] \equiv \emptyset$ for $x \in {\downarrow}k \backslash {\downarrow}p \equiv {\downarrow}k \backslash {\downarrow}m$. This union can be computed using the algorithm developed in Section 2.3.2. We set

$$copass(p, k) \overset{\text{def}}{\equiv} {\downarrow}p \subseteq {\downarrow}k \ \rhd \ mapunion({\downarrow}p, k, p) \ .$$

6.3 Pointer Relocation

The more difficult subtask consists in computing the composition $k \circ m$ efficiently. According to Lemma 2.3.3 there are essentially two ways of forming $k \circ m$:

1. domain-oriented:
$$k \circ m \equiv \bigcup_{x \in {\downarrow}m} [x \mapsto k(m(x))]$$

If we look at the union as a loop, this way of forming $k \circ m$ needs an explicit representation of k, since the same value of k may be needed repeatedly at irregular intervals.

2. range-oriented:
$$k \circ m \;\equiv\; \bigcup_{z \in \uparrow m} [(z{\downarrow}m) \mapsto k(z)]$$
For evaluating this by a loop we only need one value of k at a time to process a whole subset of ${\downarrow}m$. Hence we can avoid explicit representation of the complete k, which is particularly important in garbage collection, where storage is almost exhausted. Moreover, the repeated lookups are avoided and thus also time-efficiency is improved. Of course, this is only interesting if m is highly non-injective so that the inverse images $z{\downarrow}m$ are large. However, for block representations especially of dense graphs just this is the case.

We follow now the range-oriented variant. We need a way of representing the component maps $[z{\downarrow}m \mapsto y]$ suitably. For this we use an idea that is presented e.g. in [Dewar, McCann 77]: All elements of s are chained into a linked list; then $[s \mapsto y]$ can be formed as
$$\bigcup_{x \in s} [x \mapsto y]$$
following the chain.

6.3.1 Chains

In this section we list a number of basic properties of singly linked lists; proofs not stated here can be found in [Möller 90].

Since a partial map $m : \mathsf{cell} \longrightarrow \mathsf{cell}$ can be thought of as representing a graph of maximal outdegree 1, we may represent linked lists again by maps. The idea of following a chain starting from a point x in a graph of maximal outdegree 1, represented by $m : \mathsf{cell} \longrightarrow \mathsf{cell}$, is captured by considering m as a relation and passing to its transitive closure m^+: For $x, y \in set(m)$,
$$x\ m^+\ y \;\overset{\text{def}}{\Leftrightarrow}\; y \in \bigcup_{i \in \mathbb{N} \setminus \{0\}} x{\uparrow}m^i$$
where
$$\begin{array}{lcl} m^0 & \overset{\text{def}}{\equiv} & id_{set(m)} \\ m^{i+1} & \overset{\text{def}}{\equiv} & m \circ m^i \ . \end{array}$$
Hence $x\ m^+\ y$ holds iff y can be reached from x following the links of m (at least once). We also need the reflexive transitive closure m^* of m given by
$$x\ m^*\ y \;\overset{\text{def}}{\Leftrightarrow}\; y \in \bigcup_{i \in \mathbb{N}} x{\uparrow}m^i \ .$$

We call a state m a **chain**, if m^+ is a linear strict-order on $set(m)$, i.e., iff the predicate $ischain(m)$ holds where
$$\begin{array}{lcll} ischain(m) & \overset{\text{def}}{\Leftrightarrow} & \forall\, x, y, z \in set(m) : & \\ & & \neg\ x\ m^+\ x & \text{(irreflexivity)} \\ & \wedge & x \not\equiv y \;\Rightarrow\; (x\ m^+\ y \;\vee\; y\ m^+\ x) & \text{(linearity)}\ . \end{array}$$

Irreflexivity excludes the existence of cycles within the list, whereas linearity implies that the list is connected, i.e., that, given two distinct cells in the list, one of them can be reached from the other following the links of the list. The definition implies that a chain is injective and thus that each cell in a chain is referred to by at most one cell. Note also that $\emptyset$ is a chain. m is a **finite** chain if m is both a chain and a finite set of pairs.

Lemma 6.3.1

1. For a map $[x \mapsto y]$ we have $[x \mapsto y]^+ \equiv [x \mapsto y]$, i.e., all such maps are transitive.
2. $[x \mapsto y]$ is a chain iff $x \not\equiv y$.

Proof:
1. $[x \mapsto y] \circ [x \mapsto y] \equiv \mathsf{if}\ x = y\ \mathsf{then}\ [x \mapsto x]\ \mathsf{else}\ \emptyset\ \mathsf{fi}$. Now a straightforward induction shows that for all $i \geq 2$ we have $[x \mapsto y]^i \equiv \mathsf{if}\ x = y\ \mathsf{then}\ [x \mapsto x]\ \mathsf{else}\ \emptyset\ \mathsf{fi}$.
2. is immediate from 1.

■

The following lemma states a property that is very useful for treating combinations of chains:

Lemma 6.3.2

1. Let m and n be maps such that $\uparrow n \cap \downarrow m \equiv \emptyset$. Then
$$(m \cup n)^+ \equiv m^+ \cup m^+ n^+ \cup n^+ ,$$
where $m^+ n^+$ is the relational product of m^+ and n^+ (in diagrammatic order).
2. Let $(m_i)_{i \in I}$ be a family of maps such that $\downarrow m_i \cap \uparrow m_j \equiv \emptyset$ for $i \not\equiv j$. Then
$$(\bigcup_{i \in I} m_i)^* \equiv \bigcup_{i \in I} m_i^*$$
$$(\bigcup_{i \in I} m_i)^+ \equiv \bigcup_{i \in I} m_i^+$$

Proof:
1. From $\uparrow n \cap \downarrow m \equiv \emptyset$ it follows that $nm \equiv \emptyset$. Now a straightforward induction shows
$$(m \cup n)^i \equiv \bigcup_{k=0}^{i} m^{i-k} n^k$$
for $i > 1$. From this the claim is immediate.
2. Because of $\downarrow m_i \cap \uparrow m_j \equiv \emptyset$ we have $m_i m_j \equiv m_j m_i \equiv \emptyset$ for $i \not\equiv j$. Hence $(\bigcup_{i \in I} m_i)^k \equiv \bigcup_{i \in I} m_i^k$ for all k and the claim is immediate.

■

We now restrict our attention to finite chains.

Lemma 6.3.3

Let $m \not\equiv \emptyset$ be a finite chain. Then

1. $\mathit{set}(m)$ contains a least element $\mathit{first}(m)$ and a greatest element $\mathit{last}(m)$ w.r.t. m^+.
2. $\uparrow m \subseteq \downarrow m \cup \{\mathit{last}(m)\}$ and $\downarrow m \subseteq \uparrow m \cup \{\mathit{first}(m)\}$.
3. $\downarrow m \equiv \mathit{set}(m) \backslash \{\mathit{last}(m)\}$ and $\uparrow m \equiv \mathit{set}(m) \backslash \{\mathit{first}(m)\}$.
4. $\downarrow m \backslash \uparrow m \equiv \{\mathit{first}(m)\}$ and $\uparrow m \backslash \downarrow m \equiv \{\mathit{last}(m)\}$.
5. $\downarrow m \backslash \{\mathit{first}(m)\} \equiv \uparrow m \backslash \{\mathit{last}(m)\}$.

We set, for well-founded $m \not\equiv \emptyset$

$$\begin{aligned} \mathit{rest}(m) &\overset{\text{def}}{\equiv} \textsf{if } m = \emptyset \textsf{ then } \emptyset \textsf{ else } m \ominus \mathit{first}(m) \textsf{ fi} , \\ \mathit{inner}(m) &\overset{\text{def}}{\equiv} \downarrow m \backslash \{\mathit{first}(m)\} \equiv \uparrow m \backslash \{\mathit{last}(m)\} . \end{aligned}$$

Note that *rest* is total whereas *inner* is partial.

Lemma 6.3.4
Let $m \not\equiv \emptyset$ be a finite chain. Then

1. $m \equiv [\mathit{first}(m) \mapsto m(\mathit{first}(m))] \cup \mathit{rest}(m)$.
2. $\mathit{rest}(m) \not\equiv \emptyset \Rightarrow \mathit{first}(\mathit{rest}(m)) \equiv m(\mathit{first}(m))$.
3. If m is also anchored, $\mathit{rest}(m) \equiv \emptyset \Leftrightarrow m(\mathit{first}(m)) \equiv \square$.

6.3.2 Chained Representation of Sets

It would be attractive to call a chain m a representation of $\mathit{set}(m)$. However, we have

Lemma 6.3.5
Let $m \not\equiv \emptyset$ be chain. Then $|\mathit{set}(m)| \geq 2$.

Proof: By definition, $|\mathit{set}(m)| \geq |\downarrow m|$. Hence, if $|\downarrow m| \geq 2$ we are finished. Otherwise, $|\downarrow m| = 1$, i.e., $m \equiv [x \mapsto y]$ for some $x, y \in M$ and $\mathit{set}(m) \equiv \{x, y\}$. However, since m is irreflexive, we have $x \not\equiv y$ and thus $|\mathit{set}(m)| \equiv 2$. ■

According to this lemma, then sets of cardinality 1 would not be representable. To remedy this, we use the distinguished cell $\square$ as a chain terminator and consider all other elements in chains as the elements of the represented sets. Therefore we define the predicate

$$\begin{aligned} \mathit{isanchored}(m) \overset{\text{def}}{\Leftrightarrow}\ & \mathit{ischain}(m) \wedge \\ & (m \equiv \emptyset \vee \mathit{last}(m) \equiv \square) . \end{aligned}$$

Thus, a nonempty chain is **anchored** iff it is terminated by $\square$. An anchored chain m **represents** the set $\downarrow m$. A corresponding representation function is

```
funct $\mathit{chainrep} \equiv$ (cellset $s$) state :
      some state $m$ : $\mathit{isanchored}(m) \wedge \downarrow m = s$ .
```

We want to develop an incrementation function for extending a set representation to a representation of a larger set. To deal in a satisfactory way with the non-determinacy involved, we need an auxiliary notion concerning non-determinacy. Let E be a possibly non-determinate expression. An **atom** of E is a determinate descendant of E. From this definition it is immediate that $E_1 \subseteq E_2$ iff every atom of E_1 is also an atom of E_2; likewise, $E_1 \equiv E_2$ iff E_1 and E_2 have the same sets of atoms.

Let now s be a set of cells, m be an atom of *chainrep*(s), and $x \notin s \cup \{\Box\}$ be a cell. We want to extend m to an atom n of *chainrep*$(s \cup \{x\})$. Since this requirement implies $\downarrow n = \downarrow m \cup \{x\}$, we try to set $n \equiv m \cup [x \mapsto y]$ for some $y \in M$. The union is well-defined since $x \notin \downarrow m$ by the assumption. We now have to choose y in such a way that n becomes an anchored chain. First, we compute the transitive closure of n:

$$\begin{array}{ll} & n^+ \\ \equiv & (m \cup [x \mapsto y])^+ \\ \equiv & \text{(by Lemma 6.3.2)} \\ & [x \mapsto y]^+ \cup [x \mapsto y]^+ m^+ \cup m^+ \\ \equiv & \text{(by Lemma 6.3.1)} \\ & [x \mapsto y] \cup [x \mapsto y] m^+ \cup m^+ \\ \equiv & (*)\ . \end{array}$$

Hence *isanchored*(n) implies $x \not\equiv y$ by irreflexivity. Next we compute *first*(n). By Lemma 6.3.3,

$$\begin{array}{ll} & \{\mathit{first}(n)\} \\ \equiv & \downarrow n \backslash \uparrow n \\ \equiv & (\downarrow m \cup \{x\}) \backslash (\uparrow m \cup \{y\}) \\ \equiv & (\downarrow m \backslash \uparrow m \backslash \{y\}) \cup (\{x\} \backslash \uparrow m \backslash \{y\}) \\ \equiv & (\downarrow m \backslash \uparrow m \backslash \{y\}) \cup (\{x\} \backslash \{y\} \backslash \uparrow m) \\ \equiv & (\downarrow m \backslash \uparrow m \backslash \{y\}) \cup (\{x\} \backslash \uparrow m) \\ \equiv & (**)\ . \end{array}$$

Since $x \notin \downarrow m \cup \{\Box\} \supseteq \uparrow m$, we have $\{x\} \backslash \uparrow m \equiv \{x\}$. Hence

$$(**) \equiv (\downarrow m \backslash \uparrow m \backslash \{y\}) \cup \{x\}$$

and thus $x \in \{\mathit{first}(n)\}$, i.e. $x \equiv \mathit{first}(n)$. Now it follows that $\downarrow m \backslash \uparrow m \backslash \{y\} \subseteq \{x\}$ and hence $\downarrow m \backslash \uparrow m \backslash \{y\} \subseteq \downarrow m \cap \{x\} \equiv \emptyset$, i.e., $\downarrow m \backslash \uparrow m \backslash \{y\} \equiv \emptyset$. If $m \equiv \emptyset$ this holds trivially; however, if $m \not\equiv \emptyset$ this is equivalent to $\{\mathit{first}(m)\} \backslash \{y\} \equiv \emptyset$ and thus also to $y \equiv \mathit{first}(m)$. So we only need to find y in the case $m \equiv \emptyset$. But by *isanchored*(n) we know $\mathit{last}(n) \equiv \Box$. We calculate

$$\begin{array}{ll} & \{\mathit{last}(n)\} \\ \equiv & \uparrow n \backslash \downarrow n \\ \equiv & (\uparrow m \cup \{y\}) \backslash (\downarrow m \cup \{x\}) \\ \equiv & \{y\} \backslash \{x\} \\ \equiv & \{y\}, \end{array}$$

and thus $y \equiv \Box$. Therefore we define

$$\mathit{init}(m) \overset{\text{def}}{\equiv} \textsf{if } m = \emptyset \textsf{ then } \Box \textsf{ else } \mathit{first}(m) \textsf{ fi}\ .$$

If we can now show *isanchored*(n) for

$$n \overset{\text{def}}{\Leftrightarrow} m \cup [x \mapsto \mathit{init}(m)]\ ,$$

we have found an atom of *chainrep*$(s \cup \{x\})$.

From $(*)$ we obtain

$$\begin{array}{rcl} u\, n^+\, v & \Leftrightarrow & (u \;\equiv\; x \;\wedge\; v \;\equiv\; y) \vee \\ & & (u \;\equiv\; x \;\wedge\; y\, m^+ v) \vee \\ & & (u\, m^+\, v) \\ & \Leftrightarrow & (u \;\equiv\; x \;\wedge\; v \in \mathit{set}(m)) \vee \\ & & (u\, m^+\, v)\ . \end{array}$$

Now we show that n^+ is a linear strict-order. Let $u, v, w \in \mathit{set}(n)$.

(Irreflexivity) Assume $x\, n^+\, x$. Then $x \in \mathit{set}(m) \equiv \downarrow\!m \cup \{\square\}$, a contradiction to $\mathit{precond}(x, m)$. For $u \not\equiv x$, $u\, n^+\, u$ would mean $u \in \downarrow\!m \;\wedge\; u\, m^+\, u$, contradicting $\mathit{ischain}(m)$ and thus again $\mathit{precond}(x, m)$.

(Linearity) Let $u \;\not\equiv\; v$. If $u \;\equiv\; x$ then $v \in \mathit{set}(m)$ and thus $u\, n^+\, v$. Symmetrically, $v\, n^+\, u$ if $v \;\equiv\; x$. Otherwise, $u, v \in \mathit{set}(m)$ and therefore $u\, m^+\, v$ or $v\, m^+\, u$ by linearity of m. But then also $u\, n^+\, v$ or $v\, n^+\, u$, resp.

Finally we show that $\mathit{last}(n) \;\equiv\; \square$. If $m \;\equiv\; \emptyset$ this is immediate. Otherwise, since $x \;\equiv\; \mathit{first}(n)$, we have $x\, n^+\, \mathit{last}(m)$ and therefore $\mathit{last}(n) \;\equiv\; \mathit{last}(m) \;\equiv\; \square$ by $\mathit{isanchored}(m)$.

If we now define

$$\begin{array}{l} \mathsf{funct}\ \ \mathit{prefix} \;\equiv\; (\mathsf{cell}\ x, \mathsf{state}\ m : \mathit{isanchored}(m) \;\wedge\; x \notin \mathit{set}(m))\ \mathsf{cell} : \\ \qquad [x \mapsto \mathit{init}(m)] \cup m\ , \end{array}$$

the results of our development can be restated as

Lemma 6.3.6

1. If m is an atom of $\mathit{chainrep}(s)$ and $x \notin s \cup \{\square\}$, then $\mathit{prefix}(x, m)$ is an atom of $\mathit{chainrep}(s \cup \{x\})$.
2. $\mathit{first}(\mathit{prefix}(x, m)) \;\equiv\; x \;\wedge\; \mathit{rest}(\mathit{prefix}(x, m)) \;\equiv\; m$.

At this point, it is also interesting to note that

Lemma 6.3.7

If $m \;\not\equiv\; \emptyset$ is an anchored chain, we have $\mathit{prefix}(\mathit{first}(m), \mathit{rest}(m)) \;\equiv\; m$.

Proof:

$$\begin{array}{rl} & \mathit{prefix}(\mathit{first}(m), \mathit{rest}(m)) \\ \equiv & [\mathit{first}(m) \mapsto \mathit{init}(\mathit{rest}(m))] \cup \mathit{rest}(m) \\ \equiv & \mathsf{if}\ \mathit{rest}(m) = \emptyset\ \mathsf{then}\ [\mathit{first}(m) \mapsto \square] \\ & \qquad\qquad\quad \mathsf{else}\ \ [\mathit{first}(m) \mapsto \mathit{first}(\mathit{rest}(m))] \cup \mathit{rest}(m)\ \ \mathsf{fi} \\ \equiv & (\text{by } \mathit{isanchored}(m)) \\ & \mathsf{if}\ \mathit{rest}(m) = \emptyset\ \mathsf{then}\ m \\ & \qquad\qquad\quad \mathsf{else}\ \ [\mathit{first}(m) \mapsto \mathit{first}(\mathit{rest}(m))] \cup \mathit{rest}(m)\ \ \mathsf{fi} \\ \equiv & (\text{by Lemma 6.3.4}) \\ & \mathsf{if}\ \mathit{rest}(m) = \emptyset\ \mathsf{then}\ m\ \mathsf{else}\ m\ \mathsf{fi} \\ \equiv & m\ . \end{array}$$

■

6.3.3 Chained Representation of Maps

Let now n be a map. From the range-oriented decomposition $n \ \equiv \ \bigcup_{z \in \uparrow n} [z{\downarrow}n \mapsto z]$ we obtain the partition ${\downarrow}n \ \equiv \ \bigcup_{z \in \uparrow n} z{\downarrow}n$. We represent m by a union of chains each of which represents one of the sets $z{\downarrow}m$; the cell z is prefixed as a header cell to the respective chain. To avoid confusion between these chains we require that

$$\mathit{ischainable}(m) \overset{\text{def}}{\Leftrightarrow} {\downarrow}m \cap {\uparrow}m \ \equiv \ \emptyset$$

holds; otherwise there would be a link from one chain to the beginning of another and the partition would be lost. We now define

```
funct $\mathit{chain} \ \equiv$ (state $m$ : $\mathit{ischainable}(m)$) state :
    $\bigcup_{z \in \uparrow m} \mathit{prefix}(z, \mathit{chainrep}(z{\downarrow}m))$
```

Since the elements of $z{\downarrow}m$ are the starting points of mutually unconnected chains, they are also sources (in the graph-theoretic sense) of the chained map. We set, for arbitrary state n,

$$\mathit{src}(n) \ \overset{\text{def}}{\equiv} \ {\downarrow}m \backslash {\uparrow}m \ ;$$

this is the set of cells to which no pointer exists in n. To distinguish the sublist of a state n that emanates from a given cell x, we define

$$\mathit{from}(x, n) \ \overset{\text{def}}{\equiv} \ m|(\bigcup_{i \in \mathbb{N}} x{\uparrow}n^i) \ \equiv \ n|\{y \mid x \ n^* \ y\} \ .$$

Note that this sublist need not be a chain, since there may be a cycle. Note also, that $\mathit{from}(x, n) \ \equiv \ \emptyset$ if $x \notin {\downarrow}n$. With the help of these notions we can characterize chainings of maps by the following predicate *ischaining* :

$$\begin{array}{lll} \mathit{ischaining}(l) & \overset{\text{def}}{\equiv} & l = \bigcup_{z \in \mathit{src}(l)} \mathit{from}(z, l) \\ & \wedge & \forall \ z \in \mathit{src}(l) : \mathit{isanchored}(\mathit{from}(z, l)) \\ & \wedge & \forall \ z_1, z_2 \in \mathit{src}(l) : z_1 \neq z_2 \ \Rightarrow \ {\downarrow}\mathit{from}(z_1, l) \cap {\downarrow}\mathit{from}(z_2, l) = \emptyset \ . \end{array}$$

Moreover, we can define the inverse operation to chaining:

$$\mathit{unchain}(l) \ \overset{\text{def}}{\equiv} \ \mathit{ischaining}(l) \ \rhd \ \bigcup_{z \in \mathit{src}(l)} [\mathit{inner}(\mathit{from}(z, l)) \mapsto z] \ .$$

Lemma 6.3.8
Let l be an atom of $\mathit{chain}(m)$. Then $\mathit{unchain}(l) \ \equiv \ m$.

Proof: By definition, $l \ \equiv \ \bigcup_{z \in \uparrow m} l_z$ where each l_z is an atom of $\mathit{prefix}(z, \mathit{chainrep}(z{\downarrow}m))$. Since $z \in {\uparrow}m$ we have ${\downarrow}l_z \ \equiv \ \{z\} \cup z{\downarrow}m$ and ${\uparrow}l_z \ \equiv \ z{\downarrow}m \cup \{\Box\}$. Therefore, ${\downarrow}l_z \cap {\uparrow}l_y \ \equiv \ \emptyset$ for $z \not\equiv y$. Hence, for $z \in {\downarrow}m$, we have $z \ l^* \ x \ \Leftrightarrow \ z \ l_z^* \ x$. Moreover, by Lemma 6.3.2, $l^* \ \equiv \ \bigcup_{z \in \uparrow m} l_z^*$. This gives

$$\begin{array}{ll} & \mathit{from}(z,l) \\ \equiv & l|\{x : z\ l^*\ x\} \\ \equiv & l|\{x : z\ l_z^*\ x\} \\ \equiv & l_z|\{x : z\ l_z^*\ x\} \\ \equiv & \mathit{from}(z,l_z) \\ \equiv & l_z\ . \end{array}$$

Moreover,

$$\begin{array}{ll} & \mathit{inner}(l_z) \\ \equiv & \downarrow l_z\backslash\{\mathit{first}(l_z)\} \\ \equiv & (\{z\} \cup \downarrow \mathit{chainrep}(z\downarrow m))\backslash\{z\} \\ \equiv & \downarrow \mathit{chainrep}(z\downarrow m) \\ \equiv & z\downarrow m\ . \end{array}$$

Now the claim is immediate from Lemma 2.3.3. ■

As in the case of set representations, we want to develop an incrementation function for extending a chained representation of a map into one of a larger map. Let therefore m, n be maps and $x \not\equiv \square$ be a cell such that $x \not\in \downarrow m$ as well as $n \equiv m \cup [x \mapsto y]$ for some y and $\mathit{ischainable}(n)$ hold. Given a chaining of m, we then want to extend it to a chaining of n. First we calculate

$$\begin{array}{ll} & \mathit{ischainable}(n) \\ \Leftrightarrow & \downarrow(m \cup [x \mapsto y]) \cap \uparrow(m \cup [x \mapsto y]) \ \equiv\ \emptyset \\ \Leftrightarrow & (\downarrow m \cup \{x\}) \cap (\uparrow m \cup \{y\}) \ \equiv\ \emptyset \\ \Leftrightarrow & (\downarrow m \cap \uparrow m) \cup (\downarrow m \cap \{y\}) \cup (\{x\} \cap \uparrow m) \cup (\{x\} \cap \{y\}) \ \equiv\ \emptyset \\ \Leftrightarrow & \downarrow m \cap \uparrow m \ \equiv\ \emptyset \ \wedge\ \downarrow m \cap \{y\} \ \equiv\ \emptyset \ \wedge\ \{x\} \cap \uparrow m \ \equiv\ \emptyset \ \wedge\ \{x\} \cap \{y\} \ \equiv\ \emptyset \\ \Leftrightarrow & \mathit{ischainable}(m) \ \wedge\ y \not\in \downarrow m \ \wedge\ x \not\in \uparrow m \ \wedge\ x \not\equiv y\ . \end{array}$$

Let now l be an atom of $\mathit{chain}(m)$. This means that $l \ \equiv \ \bigcup\limits_{z\in\uparrow m} l_z$ where each l_z is an atom of the respective $\mathit{prefix}(z, \mathit{chainrep}(z\downarrow m))$. Hence $\mathit{rest}(l_z)$ is an atom of $\mathit{chainrep}(z\downarrow m)$.
We have

$$\downarrow l \ \equiv\ \bigcup_{z\in\uparrow m} (\{z\} \cup z\downarrow m)\ ,$$

and hence $x \not\in \downarrow l$.
Consider now an $u \in \uparrow n$. To achieve a more uniform calculation we set $l_u \overset{\text{def}}{\equiv} \emptyset$ if $u \not\in \uparrow m$. Then $\mathit{rest}(l_u) \equiv \emptyset \equiv \mathit{chainrep}(u\downarrow m)$, and hence $\mathit{rest}(l_u) \subseteq \mathit{chainrep}(u\downarrow m)$ also in this case.
Case 1: $u \not\equiv y$. Then $u\downarrow n \ \equiv\ u\downarrow m$, and hence l_u is an atom of $\mathit{prefix}(u, \mathit{chainrep}(u\downarrow m))$.
Case 2: $u \equiv y$. Then

$$\begin{array}{ll} & \mathit{prefix}(u, \mathit{chainrep}(u\downarrow n)) \\ \equiv & \mathit{prefix}(u, \mathit{chainrep}(u\downarrow m \cup \{x\})) \\ \supseteq & \mathit{prefix}(u, \mathit{prefix}(x, \mathit{chainrep}(u\downarrow m))) \\ \supseteq & \mathit{prefix}(u, \mathit{prefix}(x, \mathit{rest}(l_u)))\ . \end{array}$$

Hence

$$\begin{array}{ll} & \mathit{chain}(n) \\ \equiv & \bigcup\limits_{z\in\uparrow n} \mathit{prefix}(z, \mathit{chainrep}(z\downarrow n)) \\ \equiv & \bigcup\limits_{z\in\uparrow m\cup\{y\}} \mathit{prefix}(z, \mathit{chainrep}(z\downarrow n)) \end{array}$$

$$
\begin{array}{ll}
\equiv & \displaystyle\bigcup_{z\in\uparrow m\backslash\{y\}\cup\{y\}} \mathit{prefix}(z, \mathit{chainrep}(z{\downarrow}n)) \\
\equiv & \displaystyle\bigcup_{z\in\uparrow m\backslash\{y\}} \mathit{prefix}(z, \mathit{chainrep}(z{\downarrow}n)) \cup \mathit{prefix}(y, \mathit{chainrep}(y{\downarrow}n)) \\
\supseteq & \displaystyle\bigcup_{z\in\uparrow m\backslash\{y\}} l_z \cup \mathit{prefix}(y, \mathit{prefix}(x, \mathit{rest}(l_y))) \\
\equiv & l\backslash l_y \cup \mathit{prefix}(y, \mathit{prefix}(x, \mathit{rest}(l_y))) \\
\equiv & l \ominus {\downarrow} l_y \cup \mathit{prefix}(y, \mathit{prefix}(x, \mathit{rest}(l_y))) \\
\equiv & \text{(by Lemma 2.3.4, since } x \notin {\downarrow} l) \\
 & l \Leftarrow \mathit{prefix}(y, \mathit{prefix}(x, \mathit{rest}(l_y))) \\
\equiv & l \Leftarrow ([y \mapsto \mathit{first}(\mathit{prefix}(x, \mathit{rest}(l_y)))] \cup \mathit{prefix}(x, \mathit{rest}(l_y))) \\
\equiv & l \Leftarrow ([y \mapsto x] \cup [x \mapsto \mathit{init}(\mathit{rest}(l_y))] \cup \mathit{rest}(l_y)) \\
\equiv & \text{(by Lemma 2.3.4 (Annihilation), since } \mathit{rest}(l_y) \subseteq l) \\
 & l \Leftarrow ([y \mapsto x] \cup [x \mapsto \mathit{init}(\mathit{rest}(l_y))]) \\
\equiv & \mathsf{if}\ y \in \uparrow m\ \mathsf{then}\ l \Leftarrow ([y \mapsto x] \cup [x \mapsto \mathit{first}(\mathit{rest}(l_y))]) \\
 & \qquad\qquad \mathsf{else}\ \ l \Leftarrow ([y \mapsto x] \cup [x \mapsto \Box])\ \ \mathsf{fi} \\
\equiv & \mathsf{if}\ y \in \uparrow m\ \mathsf{then}\ l \Leftarrow ([y \mapsto x] \cup [x \mapsto l_y(y)]) \\
 & \qquad\qquad \mathsf{else}\ \ l \Leftarrow ([y \mapsto x] \cup [x \mapsto \Box])\ \ \mathsf{fi} \\
\equiv & \mathsf{if}\ y \in \uparrow m\ \mathsf{then}\ l \Leftarrow ([y \mapsto x] \cup [x \mapsto l(y)]) \\
 & \qquad\qquad \mathsf{else}\ \ l \Leftarrow ([y \mapsto x] \cup [x \mapsto \Box])\ \ \mathsf{fi}\ .
\end{array}
$$

Hence we define

$$
\begin{array}{ll}
 & \mathit{insert}(l, y, x) \\
\overset{\text{def}}{\equiv} & \mathit{ischaining}(l)\ \wedge\ x \notin \mathit{set}(l)\ \wedge\ y \notin \mathit{set}(l)\backslash \mathit{src}(l)\ \wedge\ x \neq y \rhd \\
 & \mathsf{if}\ y \in \mathit{src}(l)\ \mathsf{then}\ l \Leftarrow ([y \mapsto x] \cup [x \mapsto l(y)]) \\
 & \qquad\qquad \mathsf{else}\ \ l \Leftarrow ([y \mapsto x] \cup [x \mapsto \Box])\ \ \mathsf{fi}\ .
\end{array}
$$

The results of the above development then are summarized by

Lemma 6.3.9
Assume $\mathit{ischaining}(l)\ \wedge\ x \notin \mathit{set}(l)\ \wedge\ y \notin \mathit{set}(l)\backslash\mathit{src}(l)\ \wedge\ x \neq y$. Then $\mathit{insert}(l, y, x)$ is an atom of $\mathit{chain}(\mathit{unchain}(l) \cup [x \mapsto y])$.

6.3.4 Pointer Relocation Completed

We are now in a position to describe our efficient algorithm for computing $k \circ m$: We first construct a chained representation $l \equiv \bigcup_{z\in\uparrow m} l_z$ of m. Now we define

$$\mathit{relocate}(l, k) \overset{\text{def}}{\equiv} \mathit{ischaining}(l) \ \rhd\ k \circ \mathit{unchain}(l)\ .$$

We have

$$
\begin{array}{ll}
 & \mathit{relocate}(l, k) \\
\equiv & k \circ \displaystyle\bigcup_{z\in \mathit{src}(l)} [\mathit{inner}(l_z) \mapsto z] \\
\equiv & \displaystyle\bigcup_{z\in \mathit{src}(l)} [\mathit{inner}(l_z) \mapsto k(z)]\ .
\end{array}
$$

This is the main loop of our algorithm. We now want to develop a more direct version of the inner loops that form the maps $[\mathit{inner}(l_z) \mapsto k(z)]$ for $z \in \mathit{src}(l)$. To this end we define

$$\mathit{fibre}(l, x, y) \overset{\text{def}}{\equiv} \mathit{isanchored}(l)\ \wedge\ x \in {\downarrow} l\ \rhd\ [\mathit{ginner}(l, x) \mapsto y]$$

where

$$\mathit{ginner}(l,x) \overset{\text{def}}{\equiv} \{z \mid x\ l^+\ z\}\backslash\{\Box\}\ .$$

We have

$$\mathit{ginner}(l,x) \equiv \textsf{if } l(x)=\Box \textsf{ then } \emptyset \textsf{ else } \{l(x)\}\cup \mathit{ginner}(l,l(x)) \textsf{ fi}$$

and hence

$$\begin{aligned}
& \mathit{fibre}(l,x,y)\\
\equiv\ & \textsf{if } l(x)=\Box \textsf{ then } [\mathit{ginner}(l,x)\mapsto y]\\
& \qquad\qquad\quad \textsf{else } [\mathit{ginner}(l,x)\mapsto y] \textsf{ fi}\\
\equiv\ & \textsf{if } l(x)=\Box \textsf{ then } [\emptyset\mapsto y]\\
& \qquad\qquad\quad \textsf{else } [\{l(x)\}\cup \mathit{ginner}(l,l(x))\mapsto y] \textsf{ fi}\\
\equiv\ & \textsf{if } l(x)=\Box \textsf{ then } \emptyset\\
& \qquad\qquad\quad \textsf{else } [\{l(x)\}\mapsto y]\cup[\mathit{ginner}(l,l(x))\mapsto y] \textsf{ fi}\\
\equiv\ & \textsf{if } l(x)=\Box \textsf{ then } \emptyset\\
& \qquad\qquad\quad \textsf{else } [l(x)\mapsto y]\cup \mathit{fibre}(l,l(x),y) \textsf{ fi}\ .
\end{aligned}$$

Hence we have the recursion (termination is obvious)

```
funct $\mathit{fibre} \equiv$ (state $l$, cell $x, y$ : $\mathit{isanchored}(l) \wedge x \in \downarrow l$) state :
    $\lceil$ cell $z \equiv l(x)$ ; if $z = \Box$ then $\emptyset$
                                   else $[z \mapsto y] \cup \mathit{fibre}(l, z, y)$ fi $\rfloor$ .
```

Finally, we obtain

$$\begin{aligned}
& \mathit{relocate}(l,k)\\
\equiv\ & \bigcup_{x\in \mathit{src}(l)} [\mathit{inner}(l_x)\mapsto k(x)]\\
\equiv\ & \bigcup_{x\in \mathit{src}(l)} \mathit{fibre}(l,x,k(x))\ .
\end{aligned}$$

6.4 Combining Relocation and Copying

Suppose that $\mathit{unchain}(l) \equiv m$. Then

$$\begin{aligned}
& k\circ m\circ k^{-1}\\
\equiv\ & k\circ \mathit{unchain}(l)\circ k^{-1}\\
\equiv\ & \mathit{relocate}(l,k)\circ k^{-1}\\
\equiv\ & \mathit{copass}(\mathit{relocate}(l,k),k)\ .
\end{aligned}$$

Therefore we define

$$\begin{aligned}
\mathit{copy}(l,k) \overset{\text{def}}{\equiv}\ & \mathit{ischaining}(l) \wedge \mathit{isinjective}(k) \wedge {\downarrow} k = \mathit{set}(m) \rhd\\
& \mathit{copass}(\mathit{relocate}(l,k),k)\ .
\end{aligned}$$

Then the following diagram commutes:

$$\begin{tikzcd}
\mathit{set}(m) \arrow[r, "\mathit{unchain}(l)"] \arrow[d, "k"] & \mathit{set}(m) \arrow[d, "k"] \\
\mathit{set}(m)\uparrow k \arrow[r, "\mathit{copy}(l,k)"] & \mathit{set}(m)\uparrow k
\end{tikzcd}$$

This concludes our treatment of the pointer relocation pass.

7 Merging Reachability and Chaining

7.1 Statement of the Problem

In the previous section we have derived an efficient copying algorithm based on a chaining of the state to be copied. In this section we want to integrate the construction of such a representation with the computation of the reachable part.

A central assumption for chainable maps was that their domains should be disjoint from their ranges. Since, however, pseudo-graph states do not have this property, we cannot chain the whole reachable substate, but only its arcs (see 3.2).

More precisely, assume a pseudo-graph G, a set $s \subseteq \downarrow G$, and a perfect allocation g of G. Define

$$\begin{array}{lcl} n & \overset{\text{def}}{\equiv} & \mathit{blockrep}(G,g) \ , \\ G_1 & \overset{\text{def}}{\equiv} & \mathit{reach}(G,s) \ , \\ n_1 & \overset{\text{def}}{\equiv} & \mathit{blockrep}(G_1,g), \\ t & \overset{\text{def}}{\equiv} & s{\uparrow}g \ . \end{array}$$

We want to compute a chaining of $\mathit{arcs}(n_1)$. To this end, we shall first compute $\mathit{arcs}(n_1)$ from n and t by a function *sreach* and then chain this using a function *csreach* . Hence *sreach* and *csreach* should satisfy the equations

$$\begin{array}{lr} \text{(SR)} & \mathit{sreach}(n,t) \ \equiv \ \mathit{arcs}(n_1) \ , \\ \text{(CSR)} & \mathit{unchain}(\mathit{csreach}(n,t)) \ \equiv \ \mathit{arcs}(n_1) \ . \end{array}$$

In the sequel we shall assume that G is a fragment of a fixed storage-graph G_0 (see 3.1), i.e., that $G \subseteq G_0$. Consequently, $n \subseteq n_0$ where $n_0 \overset{\text{def}}{\equiv} \mathit{blockrep}(G_0,g)$. This will considerably simplify the invariants of the algorithms we shall derive in the sequel.

7.2 Reachability on Graph Representations

In Section 5.1 we have derived the following reachability algorithm for pseudo-graphs:

```
funct $reach$ $\equiv$ (pgraph $G$, nodeset $s : s \subseteq \downarrow G$) pgraph :
    if $s = \emptyset$ then $\emptyset$
            else node $z$ $\equiv$ $elem(s)$ ;
                 pgraph $G_1$ $\equiv$ $G \ominus z$ ;
                 nodeset $s_1$ $\equiv$ $(s \cup set(G(z))) \cap \downarrow G_1$ ;
                 $[z \mapsto G(z)] \cup reach(G_1, s_1)$          fi .
```

Our first task consists in transforming this into a corresponding algorithm for state representations of pseudo-graphs. Let g be an allocation of a pseudograph G. The set s of starting *nodes* in the pseudo-graph case is now replaced by a set t of starting *keys*. Thus we define

$$\begin{array}{lcl} \mathit{inreach1}(G,g,n,t) & \overset{\text{def}}{\equiv} & n = \mathit{blockrep}(G,g) \ \wedge \ t \subseteq \mathit{keys}(n) \ \triangleright \\ & & \mathit{blockrep}(\mathit{reach}(G, t{\downarrow}g), g) \ . \end{array}$$

Now assume $n \equiv \mathit{blockrep}(G,g)$ and $\emptyset \not\equiv t \subseteq \mathit{keys}(n)$. Define

$$\begin{array}{lcl} x & \overset{\text{def}}{\equiv} & \mathit{elem}(t{\downarrow}g) \ , \\ G_1 & \overset{\text{def}}{\equiv} & G \ominus x \ , \\ s_1 & \overset{\text{def}}{\equiv} & (t{\downarrow}g \cup \mathit{set}(G(x))) \cap \downarrow G_1 \ , \\ n_1 & \overset{\text{def}}{\equiv} & \mathit{blockrep}(G_1,g) \ . \end{array}$$

Then

$$\begin{array}{ll} & \mathit{inreach1}(G,g,n,t) \\ \equiv & \mathit{blockrep}([x \mapsto G(x)] \cup \mathit{reach}(G_1,s_1),g) \\ \equiv & \text{(by Lemma 3.2.2)} \\ & \mathit{blockrep}([x \mapsto G(x)],g) \cup \mathit{blockrep}(\mathit{reach}(G_1,s_1),g) \\ \equiv & \mathit{block}(x,g) \cup \mathit{inreach1}(G_1,g,n_1,s_1{\uparrow}g)\ . \end{array}$$

Now we calculate

$$\begin{array}{ll} & s_1{\uparrow}g \\ \equiv & ((t{\downarrow}g \cup \mathit{set}(G(x)))\ \ \cap {\downarrow}G_1){\uparrow}g \\ \equiv & \text{(by Lemma 2.3.2, since } g \text{ is injective)} \\ & ((t{\downarrow}g){\uparrow}g \cup \mathit{set}(G(x)){\uparrow}g) \cap ({\downarrow}G_1{\uparrow}g) \\ \equiv & (t\ \cup {\uparrow}(\mathit{block}(x,g) \ominus g(x))) \cap \mathit{keys}(n_1) \\ \equiv & (t\ \cup {\uparrow}\mathit{block}(x,g) \backslash \{\square\}) \cap \mathit{keys}(n_1) \\ \equiv & (t\ \cup {\uparrow}\mathit{block}(x,g)) \cap \mathit{keys}(n_1)\ . \end{array}$$

since $\square \notin \mathit{keys}(n_1)$. Thus we obtain the recursion

```
funct $\mathit{inreach1}$ $\equiv$ ( pgraph $G$, map $g, n$, cellset $t$ :
                        $n = \mathit{blockrep}(G,g) \ \wedge\ t \subseteq \mathit{keys}(n)$) state :
    if $t = \emptyset$
      then $\emptyset$
      else node $x \equiv \mathit{elem}(t{\downarrow}g)$ ;
           pgraph $G_1 \equiv G \ominus x$ ;
           state $n_1 \equiv \mathit{blockrep}(G_1,g)$ ;
           cellset $t_1 \equiv (t\ \cup {\uparrow}\mathit{block}(x,g)) \cap \mathit{keys}(n_1)$ ;
           $\mathit{block}(x,g) \cup \mathit{inreach1}(G_1,g,n_1,t_1)$            fi .
```

7.3 Computing the Arcs of the Reachable Substate

Our next task consists in deriving from *inreach1* an algorithm *sreach* satisfying (SR). An additional requirement for this algorithm is that it should not use a separate parameter for the set t which would occupy a lot of storage space. Everything should be done on the map-parameter n and on some auxiliary cell-parameters.

A first idea how to realize this would be to represent the set t as a chain and to overwrite n with this chain. However, one sees immediately that this is in conflict with the chaining for representing the map which requires a different overwriting of n. We shall overcome these difficulties by not storing t itself but (a code of) a set of cells pointing to the elements of t and, when adding $\mathit{block}(x,g)$, by chaining the cells not in their original order but in a different one arising during the traversal of the reachable part.

Let again

$$\begin{array}{lcl} n & \overset{\text{def}}{\equiv} & \mathit{blockrep}(G,g)\ , \\ G_1 & \overset{\text{def}}{\equiv} & \mathit{reach}(G,s)\ , \\ n_1 & \overset{\text{def}}{\equiv} & \mathit{blockrep}(G_1,g), \\ t & \overset{\text{def}}{\equiv} & s{\uparrow}g\ . \end{array}$$

Using the fact that

$$G_1 \ \equiv\ G|s \cup \mathit{reach}(\tilde{G},\tilde{s})$$

where $\tilde{G} \overset{\text{def}}{\equiv} G \ominus s$ and $\tilde{s} \overset{\text{def}}{\equiv} (\bigcup_{x \in s} set(G(x))) \cap {\downarrow}\tilde{G}$, we calculate

$$\begin{aligned}
& arcs(n_1) \\
\equiv\ & arcs(blockrep(G_1, g)) \\
\equiv\ & \text{(by Lemma 3.2.2)} \\
& arcs(blockrep(G|s, g)) \cup arcs(blockrep(reach(\tilde{G}, \tilde{s}))) \\
\equiv\ & \tilde{n}|u \cup arcs(inreach1(\tilde{G}, g, m, \tilde{s}{\uparrow}g))
\end{aligned}$$

where

$$\begin{aligned}
\tilde{n} &\overset{\text{def}}{\equiv} n \ominus t\ , \\
u &\overset{\text{def}}{\equiv} setfollowers(n, t) \overset{\text{def}}{\equiv} \bigcup_{z \in t} set(followers(n, z)) \quad \text{(see 3.2)}, \\
m &\overset{\text{def}}{\equiv} blockrep(\tilde{G}, g)\ .
\end{aligned}$$

Furthermore,

$$\tilde{s}{\uparrow}g \equiv u{\uparrow}\tilde{n} \cap keys(\tilde{n})\ .$$

This suggests the definition

$$\begin{aligned}
inreach2(G, g, n, m, u) \overset{\text{def}}{\equiv}\ & P_2(G, g, n, m, u) \rhd \\
& n|u \cup arcs(inreach1(G, g, m, u{\uparrow}n \cap keys(n)))\ ,
\end{aligned}$$

where

$$\begin{aligned}
P_2(G, g, n, m, u) \overset{\text{def}}{\equiv}\ & m = blockrep(G, g) \\
\wedge\ & n \supseteq m \\
\wedge\ & keys(n) = keys(m) \\
\wedge\ & u \subseteq {\downarrow}arcs(n) \\
\wedge\ & u \cap {\downarrow}m = \emptyset \\
\wedge\ & setfollowers(n, keys(n)) \cap u = \emptyset\ .
\end{aligned}$$

Using the notation from above, we obtain

$$arcs(n_1) \equiv inreach2(\tilde{G}, g, \tilde{n}, m, u)\ .$$

Now we have to develop a recursion for *inreach2*. This turns out to be the most difficult step in the whole development.

Suppose that $P_2(G, g, n, m, u)$ and $u \not\equiv \emptyset$ hold and define

$$\begin{aligned}
t &\overset{\text{def}}{\equiv} u{\uparrow}n \cap keys(n)\ , \\
y &\overset{\text{def}}{\equiv} elem(u)\ , \\
z &\overset{\text{def}}{\equiv} n(y)\ , \\
n_1 &\overset{\text{def}}{\equiv} n \ominus y\ , \\
u_1 &\overset{\text{def}}{\equiv} u \backslash \{y\}\ , \\
\tilde{t} &\overset{\text{def}}{\equiv} u_1{\uparrow}n_1 \cap keys(n_1)\ .
\end{aligned}$$

First,

$$\begin{aligned}
& n|u \\
\equiv\ & n|(\{y\} \cup u_1) \\
\equiv\ & n|\{y\} \cup n|u_1 \\
\equiv\ & [y \mapsto z] \cup n_1|u_1\ .
\end{aligned}$$

Moreover, since $y \in {\downarrow}arcs(n)$ we have $keys(n_1) \;\equiv\; keys(n)$. Hence

$$\begin{array}{ll} & t \\ \equiv & (u_1 \cup \{y\}){\uparrow}n \cap keys(n) \\ \equiv & (u_1{\uparrow}n \cap keys(n)) \cup (\{y\}{\uparrow}n \cap keys(n)) \\ \equiv & (u_1{\uparrow}n_1 \cap keys(n_1)) \cup (\{z\} \cap keys(n)) \\ \equiv & \tilde{t} \cup (\{z\} \cap keys(n)) \ . \end{array}$$

Case 1: $z \notin keys(n)$, i.e., $(\{z\} \cap keys(n)) \;\equiv\; \emptyset$.
Then $t \;\equiv\; \tilde{t}$ and thus

$$\begin{array}{ll} & inreach2(G, g, n, m, u) \\ \equiv & n|u \cup arcs(inreach1(G, g, m, t)) \\ \equiv & [y \mapsto z] \cup n_1|u \cup arcs(inreach1(G, g, m, \tilde{t})) \\ \equiv & [y \mapsto z] \cup inreach2(G, g, n_1, m, u_1) \ . \end{array}$$

■

Case 2: $z \in keys(n)$, i.e., $(\{z\} \cap keys(n)) \;\equiv\; \{z\}$.
Then $t \;\equiv\; \tilde{t} \cup \{z\}$. Now we specialize the choice of $elem(t{\downarrow}g)$ in $inreach1$ to $g^{-1}(z)$, i.e., we set

$$\begin{array}{lll} x & \overset{\text{def}}{\equiv} & g^{-1}(z) \ , \\ G_1 & \overset{\text{def}}{\equiv} & G \ominus x \ , \\ m_1 & \overset{\text{def}}{\equiv} & blockrep(G_1, g) \ , \\ t_1 & \overset{\text{def}}{\equiv} & (t \ \cup {\uparrow}block(x, g)) \cap keys(m_1) \ . \end{array}$$

We have $keys(m_1) \;\equiv\; keys(m) \backslash \{z\}$. Hence, setting

$$n_2 \overset{\text{def}}{\equiv} n_1 \ominus z$$

we have

$$keys(m_1) \;\equiv\; keys(n_2) \quad \text{and} \quad n_1|u_1 \;\equiv\; n_2|u_1 \ ,$$

since $z \notin {\downarrow}n_1 \;\equiv\; u_1 \subseteq {\downarrow}arcs(n)$. Moreover,

$$\begin{array}{ll} & inreach2(G, g, n, m, u) \\ \equiv & n|u \cup arcs(inreach1(G, g, m, t)) \\ \equiv & [y \mapsto z] \cup n_1|u_1 \cup arcs(block(x, g) \cup inreach1(G_1, g, m_1, t_1)) \\ \equiv & [y \mapsto z] \cup n_1|u_1 \cup arcs(block(x, g)) \cup arcs(inreach1(G_1, g, m_1, t_1)) \\ \equiv & [y \mapsto z] \cup n_2|u_1 \cup arcs(block(x, g)) \cup arcs(inreach1(G_1, g, m_1, t_1)) \ . \end{array}$$

Now we observe that, since $n \;\supseteq\; m \;\equiv\; blockrep(G, g)$,

$$\begin{array}{ll} & block(x, g) \\ \equiv & n|(\{g(x)\} \cup set(followers(n, g(x)))) \\ \equiv & n|(\{z\} \cup set(followers(n, z))) \ . \end{array}$$

Let therefore

$$u_2 \overset{\text{def}}{\equiv} set(followers(n, z)) \ .$$

Then

$$\begin{array}{ll} & arcs(block(x, g)) \\ \equiv & arcs(n|(\{z\} \cup u_2)) \\ \equiv & arcs(n|\{z\} \cup n|u_2) \\ \equiv & arcs(n|(\{z\}) \cup arcs(n|u_2) \end{array}$$

$$
\begin{array}{ll}
\equiv & (\text{since } z \in \mathit{keys}(n|\{z\}) \text{ and thus } \mathit{arcs}(n|\{z\}) \;\equiv\; \emptyset) \\
& \mathit{arcs}(n|u_2) \\
\equiv & (\text{since } u_2 \cap \mathit{keys}(n) \;\equiv\; \emptyset \text{ by } n \;\supseteq\; \mathit{block}(x,g)) \\
& n|u_2 \;.
\end{array}
$$

Hence

$$
\begin{array}{ll}
& [y \mapsto z] \cup n_2|u_1 \cup \mathit{arcs}(\mathit{block}(x,g)) \\
\equiv & [y \mapsto z] \cup n_2|u_1 \cup n|u_2 \\
\equiv & ([y \mapsto z] \cup n|u_2) \cup n_2|u_1 \\
\equiv & ([y \mapsto z] \cup n_2|u_2) \cup n_2|u_1 \\
\equiv & [y \mapsto z] \cup n_2|(u_2 \cup u_1) \;.
\end{array}
$$

If we now can show $(u_1 \cup u_2){\uparrow}n_2 \cap \mathit{keys}(n_2) \;\equiv\; t_1$ and $(u_1 \cup u_2) \cap {\downarrow}m_1 \;\equiv\; \emptyset$, we have derived a recursion relation for *inreach2* . We have

$$
t_1 \;\equiv\; (t \cap \mathit{keys}(m_1)) \cup ({\uparrow}\mathit{block}(x,g) \cap \mathit{keys}(m_1)) \;.
$$

Now,

$$
\begin{array}{ll}
& t \cap \mathit{keys}(m_1) \\
\equiv & u{\uparrow}n \cap \mathit{keys}(n) \cap \mathit{keys}(m_1) \\
\equiv & (\text{since } m_1 \subseteq m \subseteq n) \\
& u{\uparrow}n \cap \mathit{keys}(m_1) \\
\equiv & (\{z\} \cup u_1{\uparrow}n_1) \cap \mathit{keys}(m_1) \\
\equiv & u_1{\uparrow}n_1 \cap \mathit{keys}(m_1) \\
\equiv & (\text{since } n_2 \;\equiv\; n_1 \ominus z \text{ and } z \in \mathit{keys}(n) \text{ so that } z \notin u_1) \\
& u_1{\uparrow}n_2 \cap \mathit{keys}(m_1) \\
\equiv & u_1{\uparrow}n_2 \cap \mathit{keys}(n_2)
\end{array}
$$

and

$$
\begin{array}{ll}
& {\uparrow}\mathit{block}(x,g) \cap \mathit{keys}(m_1) \\
\equiv & {\uparrow}n|u_2 \cap \mathit{keys}(m_1) \\
\equiv & u_2{\uparrow}n \cap \mathit{keys}(m_1) \\
\equiv & (\text{since } n(y) \;\equiv\; z \notin \mathit{keys}(m_1)) \\
& u_2{\uparrow}n_1 \cap \mathit{keys}(m_1) \\
\equiv & (\text{since } z \notin u_2) \\
& u_2{\uparrow}n_2 \cap \mathit{keys}(m_1) \\
\equiv & u_2{\uparrow}n_2 \cap \mathit{keys}(n_2)
\end{array}
$$

and thus indeed $(u_1 \cup u_2){\uparrow}n_2 \cap \mathit{keys}(n_2) \;\equiv\; t_1$. Furthermore, $m_1 \;\equiv\; \mathit{blockrep}(G_1,g) \;\equiv\; m \backslash \mathit{block}(x,g)$ and $u_2 \subseteq {\downarrow}\mathit{block}(x,g)$ so that $(u_1 \cup u_2) \cap {\downarrow}m_1 \;\equiv\; \emptyset$. Therefore in this case

$$
\mathit{inreach2}(G,g,n,m,u) \;\equiv\; [y \mapsto z] \cup \mathit{inreach2}(G_1,g,n_2,m_1,u_1 \cup u_2) \;.
$$

■

Altogether, we have obtained the following recurrent recursion for *inreach2* :

$$
\begin{array}{ll}
& \mathit{inreach2}(G,g,n,m,u) \\
\supseteq & P_2(G,g,n,m,u) \rhd \\
& \textsf{if } u = \emptyset
\end{array}
$$

```
      then $\emptyset$
      else  cell $y \;\equiv\; elem(u)$ ;
            cell $z \;\equiv\; n(y)$ ;
            state $n_1 \;\equiv\; n \ominus y$ ;
            cellset $u_1 \;\equiv\; u \backslash \{y\}$ ;
            if $z \notin keys(n)$
               then $[y \mapsto z] \cup inreach2(G, g, n_1, m, u_1)$
               else  state $n_2 \;\equiv\; n_1 \ominus z$ ;
                     pgraph $G_1 \;\equiv\; G \ominus g^{-1}(z)$ ;
                     state $m_1 \;\equiv\; blockrep(G_1, g)$ ;
                     $[y \mapsto z] \cup inreach2(G_1, g, n_2, m_1, u_1 \cup set(followers(n, z)))$ fi fi .
```

According to the rule (DESCENDANT-FIXPOINT) now the least fixpoint of the corresponding functional is a descendant of *inreach2* . We will denote this fixpoint again by *inreach2* .

Obviously, the parameters G, g, and m do not contribute to the computation proper and thus may be eliminated. We define recursively

```
      $sreach2(n, u)$
  $\equiv$  $u \subseteq {\downarrow}arcs(n) \;\rhd$
    if $u = \emptyset$
      then $\emptyset$
      else  cell $y \;\equiv\; elem(u)$ ;
            cell $z \;\equiv\; n(y)$ ;
            state $n_1 \;\equiv\; n \ominus y$ ;
            cellset $u_1 \;\equiv\; u \backslash \{y\}$ ;
            if $z \notin keys(n)$
               then $[y \mapsto z] \cup sreach2(n_1, u_1)$
               else  state $n_2 \;\equiv\; n_1 \ominus z$ ;
                     $[y \mapsto z] \cup sreach2(n_2, u_1 \cup set(followers(n, z)))$ fi fi .
```

A trivial computational induction proves that

$$sreach2(n, u) \;\equiv\; inreach2(G, g, n, m, u)$$

provided $P_2(G, g, n, m, u)$ holds. Now we have

$$\begin{aligned} & arcs(n_1) \\ \equiv\; & inreach2(\tilde{G}, g, \tilde{n}, m, u) \\ \equiv\; & sreach2(\tilde{n}, u) \\ \equiv\; & sreach2(n, u) \;. \end{aligned}$$

Therefore we define the function

$$\begin{aligned} sreach(n, t) \;\overset{\text{def}}{\equiv}\; & t \subseteq keys(n) \;\rhd \\ & sreach2(n, setfollowers(n, t)) \;, \end{aligned}$$

which obviously satisfies (SR).

7.4 Specialization to a Depth-First Traversal

We now represent sets $u \subseteq {\downarrow}arcs(n)$ of cells by sequences α such that $set(\alpha) \equiv u$. Furthermore we concretize *elem* to *first* , replace set union by concatenation and obtain in this way a first deterministic algorithm for our problem. We specify *sreach3* by

$$sreach3(n, \alpha) \;\overset{\text{def}}{\equiv}\; P_3(n, \alpha) \;\rhd\; sreach2(n, set(\alpha))$$

with the invariant

$$\begin{array}{rcl} P_3(n,\alpha) & \overset{\text{def}}{\equiv} & \mathit{set}(\alpha) \subseteq {\downarrow}\mathit{arcs}(n) \\ & \wedge & \mathit{repetitionfree}(\alpha) \\ & \wedge & \mathit{setfollowers}(n, \mathit{keys}(n)) \cap \mathit{set}(\alpha) = \emptyset \end{array}$$

where

$$\mathit{repetitionfree}(\alpha) \overset{\text{def}}{\Leftrightarrow} \mathit{isinjective}(\bigcup_{i=1}^{|\alpha|} [i \mapsto \alpha[i]]) \ .$$

The last conjunct of P_3 corresponds to the last conjunct of P_2 and is needed in proving invariance of *repetitionfree*(α). We have the embedding

$$\mathit{sreach2}(n,u) \ \equiv \ \mathit{sreach3}(n, \mathit{sort}(u)) \ .$$

Using again the rule (DESCENDANT-FIXPOINT) we immediately obtain the recursion

```
    $sreach3(n,\alpha)$
 $\supseteq$  $P_3(n,\alpha) \rhd$
    if $\alpha = \langle\rangle$
      then $\emptyset$
      else cell $y \equiv first(\alpha)$ ;
           cell $z \equiv n(y)$ ;
           state $n_1 \equiv n \ominus y$ ;
           cellsequ $\alpha_1 \equiv rest(\alpha)$ ;
           if $z \notin keys(n)$
             then $[y \mapsto z] \cup sreach3(n_1, \alpha_1)$
             else state $n_2 \equiv n_1 \ominus z$ ;
                  cellsequ $\alpha_2 \equiv followers(n,z) + \alpha_1$ ;
                  $[y \mapsto z] \cup sreach3(n_2, \alpha_2)$            fi fi .
```

The choice $y \equiv \mathit{first}(\alpha)$ (rather than $y \equiv \mathit{last}(\alpha)$) leads to a depth-first traversal.

7.5 Encoding the Arcs

We have started this chapter considering a (fixed) pseudo-graph state n_0. From now on we will assume that the keys of n_0 are marked (for instance, using a bit-vector). We define, for $n \subseteq n_0$,

$$\begin{array}{lcl} \mathit{bound}(n) & \overset{\text{def}}{\equiv} & \mathit{sup}({\downarrow}n) + 1 \ , \\ \mathit{keys}_+(n) & \overset{\text{def}}{\equiv} & \mathit{keys}(n) \cup \{\mathit{bound}(n)\} \ , \\ \mathit{next}_n(z) & \overset{\text{def}}{\equiv} & \mathit{succ}_{\mathit{keys}_+(n)}(z) \ , \\ \mathit{keys}_+ & \overset{\text{def}}{\equiv} & \mathit{keys}_+(n_0) \ , \\ \mathit{bound} & \overset{\text{def}}{\equiv} & \mathit{bound}(n_0) \ . \end{array}$$

Obviously all submaps $n \subseteq n_0$ appearing in our reachability algorithms satisfy

$$\mathit{keys}(n) \subseteq \mathit{keys}_+ \ \wedge \ {\downarrow}\mathit{arcs}(n) \cap \mathit{keys}_+ \ \equiv \ \emptyset \ .$$

Moreover, we observe that the property

$$\exists \ \beta : \alpha \ \equiv \ \mathit{followers}(n, \beta) \ ,$$

where

$$followers(m, \beta) \equiv \sum_{i \in [1:|\beta|]} followers(m, \beta[i]) \ ,$$

(see 3.2 for *followers* of single cells) is an invariant of *sreach3*. Therefore we can reconstruct the input α from the sequence β. Packing some other obvious invariants into the assertion

$$\begin{array}{rcl} P_4(n,\beta) & \overset{\text{def}}{\equiv} & keys(n) \subseteq keys_+ \\ & \wedge & \downarrow arcs(n) \cap keys_+ = \emptyset \\ & \wedge & repetitionfree(\beta) \\ & \wedge & \downarrow n \cap set(\beta) = \emptyset \\ & \wedge & \forall\ z \in keys(n) \cup set(\beta) :]z : next_n(z)[\subseteq \downarrow arcs(n) \backslash set(\beta) \ , \end{array}$$

we define

$$sreach4(n,\beta) \overset{\text{def}}{\equiv} P_4(n,\beta) \ \triangleright\ sreach3(n, followers(n,\beta)) \ .$$

We obtain for $n \subseteq n_0$ and $t \subseteq keys(n)$ the embedding

$$\begin{array}{rl} & sreach(n,t) \\ \equiv & sreach2(n, setfollowers(n,t)) \\ \equiv & sreach2(n, set(followers(n, sort(t)))) \\ \equiv & sreach3(n, followers(n, sort(t))) \\ \equiv & sreach4(n, sort(t)) \ . \end{array}$$

Now let n and β be such that $P_4(n,\beta)$ holds. Assume $\beta \not\equiv <>$ and let

$$\begin{array}{rcl} y & \overset{\text{def}}{\equiv} & first(\beta) + 1 \ , \\ \gamma & \overset{\text{def}}{\equiv} & rest(\beta) \ . \end{array}$$

Case 1: $y \notin \downarrow arcs(n)$.
Then $followers(n, first(\beta)) \equiv <>$ and therefore

$$followers(n,\beta) \equiv followers(n,\gamma) \ .$$

Hence

$$sreach4(n,\beta) \equiv sreach4(n,\gamma) \ .$$

■

Case 2: $y \in \downarrow arcs(n)$.
Then, setting

$$\alpha \overset{\text{def}}{\equiv} followers(n,\beta) \ ,$$

we have

$$y \equiv first(\alpha)$$

and

$$followers(<y> + \gamma) \equiv rest(\alpha) \ .$$

Therefore, if $z \in keys(n)$ — where $z \overset{\text{def}}{\equiv} n(y)$ — then

$$followers(n, <z> + <y> + \gamma) \equiv followers(n,z) + rest(\alpha) \ .$$

■

The test $y \in \downarrow arcs(n)$ may be simplified as follows: P_4 implies that

$$\forall\, z \in set(\beta) :]z : next_n(z)[\subseteq \downarrow arcs(n)\ ,$$

which, in turn, entails that

$$y \notin keys_+ \ \Rightarrow\ y \in \downarrow arcs(n)\ .$$

Hence

$$y \notin \downarrow arcs(n) \ \Leftrightarrow\ y \in keys_+\ .$$

Therefore

```
    $sreach4(n, \beta)$
$\equiv$ $P_4(n, \beta) \rhd$
    if $\beta = <>$
      then $\emptyset$
      else cell $y \equiv first(\beta) + 1$ ;
           cellsequ $\gamma \equiv rest(\beta)$ ;
           if $y \in keys_+$
             then $sreach4(n, \gamma)$
             else cell $z \equiv n(y)$ ;
                  state $n_1 \equiv n \ominus y$ ;
                  cellsequ $\beta_1 \equiv <y> + \gamma$ ;
                  if $z \notin keys(n)$
                    then $[y \mapsto z] \cup sreach4(n_1, \beta_1)$
                    else state $n_2 \equiv n_1 \ominus z$ ;
                         cellsequ $\beta_2 \equiv <z> + \beta_1$ ;
                         $[y \mapsto z] \cup sreach4(n_2, \beta_2)$ fi fi fi .
```

7.6 Integration of Chaining

Next we transform *sreach4* into a tail recursion and add chaining. In a first step we add a parameter m that accumulates — not, as usual, the intermediate results themselves, but — chainings of the intermediate results. This will make the chaining step very easy. We specify

$$sreach5(n, \beta, m) \overset{\text{def}}{\equiv} P_5(n, \beta, m) \ \rhd\ unchain(m) \cup sreach4(n, \beta)\ .$$

The assertion P_5 collects all the invariants we need for the further development; in it we use the function

$$cinner(m) \overset{\text{def}}{\equiv} \bigcup_{x \in src(m)} inner(from(x, m))\ .$$

Then

$$\begin{array}{lcl}
P_5(n, \beta, m) & \overset{\text{def}}{\equiv} & P_4(n, \beta) \\
& \wedge & ischaining(m) \\
& \wedge & cinner(m) \cap keys(n) = \emptyset \\
& \wedge & \downarrow n \cap \downarrow m = \emptyset \\
& \wedge & \uparrow arcs(n) \subseteq keys(n) \cup src(m) \\
& \wedge & keys_+ = keys_+(n) \cup src(m) \\
& \wedge & \beta \neq \emptyset \ \Rightarrow\ set(rest(\beta)) \subseteq \Box\downarrow m \\
& \wedge & src(m) = \{z \in keys_+ \mid set(\beta) \cap [z : next_n(z)[\neq \emptyset\} \\
& \wedge & cinner(m) = \\
& & \bigcup \{]z : y] \mid z \in src(m) \ \wedge\ y \in]z : next_n(z)[\cap set(\beta)\}
\end{array}$$

We have the embedding

$$sreach(n_0, t) \;\equiv\; sreach5(n_0, sort(t), [t \mapsto \Box]) \;.$$

Note that $unchain([t \mapsto \Box]) \equiv \emptyset$. Recall now from Section 6.3.3 that for a chaining m and for $y, z \notin cinner(m)$

$$unchain(m) \cup [y \mapsto z] \;\equiv\; unchain(insert(m, z, y))$$

where

$$\begin{aligned} insert(m, z, y) \;\equiv\; & \textsf{if } z \in src(m) \textsf{ then } m \Leftarrow\!\!+([z \mapsto y] \cup [y \mapsto m(z)]) \\ & \qquad\qquad\qquad\quad \textsf{else } \; m \Leftarrow\!\!+([z \mapsto y] \cup [y \mapsto \Box]) \quad \textsf{fi} \;. \end{aligned}$$

Therefore we obtain the recursion

```
     $sreach5(n, \beta, m)$
  $\equiv$ $P_5(n, \beta, m) \rhd$
     if $\beta = <>$
       then $unchain(m)$
       else cell $y \equiv first(\beta) + 1$ ;
            cellsequ $\gamma \equiv rest(\beta)$ ;
            if $y \in keys_+$
              then $sreach5(n, \gamma, m)$
              else cell $z \equiv n(y)$ ;
                   state $n_1 \equiv n \ominus y$ ;
                   cellsequ $\beta_1 \equiv <y> + \gamma$ ;
                   state $m_1 \equiv insert(m, z, y)$ ;
                   if $z \notin keys(n)$
                     then $sreach5(n_1, \beta_1, m_1)$
                     else state $n_2 \equiv n_1 \ominus z$ ;
                          cellsequ $\beta_2 \equiv <z> + \beta_1$ ;
                          $sreach5(n_2, \beta_2, m_1)$ fi fi fi .
```

Now chaining can be added simply by replacing $unchain(m)$ by m in the body of $sreach5$. We take $P_6 \equiv P_5$ and define

```
     $sreach6(n, \beta, m)$
  $\equiv$ $P_6(n, \beta, m) \rhd$
     if $\beta = <>$
       then $m$
       else cell $y \equiv first(\beta) + 1$ ;
            cellsequ $\gamma \equiv rest(\beta)$ ;
            if $y \in keys_+$
              then $sreach6(n, \gamma, m)$
              else cell $z \equiv n(y)$ ;
                   state $n_1 \equiv n \ominus y$ ;
                   cellsequ $\beta_1 \equiv <y> + \gamma$ ;
                   state $m_1 \equiv insert(m, z, y)$ ;
                   if $z \notin keys(n)$
                     then $sreach6(n_1, \beta_1, m_1)$
                     else state $n_2 \equiv n_1 \ominus z$ ;
                          cellsequ $\beta_2 \equiv <z> + \beta_1$ ;
                          $sreach6(n_2, \beta_2, m_1)$ fi fi fi .
```

Now we have

$$\mathit{sreach5}(n, \beta, m) \;\equiv\; \mathit{unchain}(\mathit{sreach6}(n, \beta, m)) \ ,$$

(a proof of this fact is provided by the implementation rule RANGE in [Berghammer, Ehler 90]) and consequently

$$\mathit{sreach}(n_0, t) \;\equiv\; \mathit{unchain}(\mathit{sreach6}(n_0, \mathit{sort}(t), [t \mapsto \Box])) \ .$$

Therefore

$$\mathit{csreach}(n, t) \;\overset{\text{def}}{\equiv}\; \mathit{sreach6}(n, \mathit{sort}(t), [t \mapsto \Box])$$

satisfies the required equation (CSR) from Section 7.1.

7.7 Improving Space Efficiency

As standing, the algorithm for csreach still uses the three "large" parameters n, β and m. In the case of garbage collection, however, there is no space available for these parameters. So we need to represent them by *one* map parameter, viz. by the store itself. The assertion $P_5(\equiv \ P_6)$ clearly indicates how to proceed: The conjunct $\downarrow m \cap \downarrow n = \emptyset$ allows us to simply unify n and m; moreover, since $\mathit{set}(\mathit{rest}(\beta)) \subseteq \Box\downarrow m$ we may overwrite m with (a map representation of) $\mathit{rest}(\beta)$ without losing essential information about m. To this end, we define (using operations from Section 6.3) for sequences $\gamma \in (\mathsf{cell}\backslash\{\Box\})^*$

$$\begin{aligned}\mathit{map}(\gamma) \;\overset{\text{def}}{\equiv}\;& \mathit{repetitionfree}(\gamma) \rhd \\ & \textsf{if } \gamma = \langle\rangle \textsf{ then } \emptyset \textsf{ else } \mathit{prefix}(\mathit{first}(\gamma), \mathit{map}(\mathit{rest}(\gamma))) \textsf{ fi} \ ,\end{aligned}$$

and for maps c and cells v such that $\mathit{isanchored}(c) \ \wedge \ v \in \mathit{set}(c)$,

$$\mathit{seq}(v, c) \;\overset{\text{def}}{\equiv}\; \textsf{if } v = \Box \textsf{ then } \langle\rangle \textsf{ else } \langle v\rangle + \mathit{seq}(c(v), c \ominus v) \textsf{ fi} \ .$$

If $v \equiv \mathit{first}(c)$ this may be rewritten as

$$\mathit{seq}(v, c) \;\equiv\; \textsf{if } v = \Box \textsf{ then } \langle\rangle \textsf{ else } \langle v\rangle + \mathit{seq}(c(v), \mathit{rest}(c)) \textsf{ fi} \ .$$

Then obviously

$$\mathit{seq}(\mathit{tfirst}(\gamma), \mathit{map}(\gamma)) \;\equiv\; \gamma \ ,$$

where

$$\mathit{tfirst}(\gamma) \;\overset{\text{def}}{\equiv}\; \textsf{if } \gamma = \langle\rangle \textsf{ then } \Box \textsf{ else } \mathit{first}(\gamma) \textsf{ fi}$$

and

$$\mathit{set}(\gamma) \;\overset{\text{def}}{\equiv}\; \downarrow \mathit{map}(\gamma) \ .$$

Hence we have $\mathit{isanchored}(\mathit{map}(\gamma))$ and $\mathit{map}(\gamma)$ represents $\mathit{set}(\gamma)$.

We want to represent the sequence β in *sreach6* by the triple (y, v, c) where

$$\begin{aligned} y &\overset{\text{def}}{\equiv} \mathit{tfirst}(\beta) + 1 \ , \\ v &\overset{\text{def}}{\equiv} \mathit{tfirst}(\mathit{trest}(\beta)) \ , \\ c &\overset{\text{def}}{\equiv} \mathit{map}(\mathit{trest}(\beta)) \end{aligned}$$

and

$$\mathit{trest}(\gamma) \;\overset{\text{def}}{\equiv}\; \textsf{if } \gamma = \langle\rangle \textsf{ then } \langle\rangle \textsf{ else } \mathit{rest}(\gamma) \textsf{ fi} \ .$$

We can retrieve β from (y, v, c) by virtue of

$$\beta \;\equiv\; <y-1> +\mathit{seq}(v, c)\ .$$

Hence we define

$$\mathit{sreach7}(n, y, v, c, m) \;\stackrel{\text{def}}{\equiv}\; P_7(n, y, v, c, m) \;\triangleright\; \mathit{sreach6}(n, <y-1> +\mathit{seq}(v, c), m)$$

where

$$\begin{array}{rcl} P_7(n, y, v, c, m) & \stackrel{\text{def}}{\equiv} & \mathit{isanchored}(c) \;\wedge\; v \in \mathit{set}(c) \\ & \dot{\wedge} & P_5(n, <y-1> +\mathit{seq}(v, c), m) \\ & \wedge & y \notin \downarrow\!c \\ & \wedge & \mathit{set}(\mathit{seq}(v, c)) = \downarrow\!c\ . \end{array}$$

We have

$$\mathit{sreach}(n_0, t) \;\equiv\; \mathit{unchain}(\mathit{sreach7}(n_0, y, v, c, m))$$

where

$$\begin{array}{rcl} y & \stackrel{\text{def}}{\equiv} & \mathit{tmin}(t) + 1\ , \\ v & \stackrel{\text{def}}{\equiv} & \mathit{tmin}(t \backslash \{\mathit{tmin}(t)\})\ , \\ c & \stackrel{\text{def}}{\equiv} & \mathit{map}(t \backslash \{\mathit{tmin}(t)\})\ , \\ m & \stackrel{\text{def}}{\equiv} & [t \mapsto \Box] \end{array}$$

and

$$\mathit{tmin}(t) \;\stackrel{\text{def}}{\equiv}\; \mathit{min}(t \cup \{\Box\})\ .$$

Note that $v \equiv \mathit{first}(c)$. To find a recursion for *sreach7* we consider values n, y, v, c, m satisfying the invariant $P_7(n, y, v, c, m)$. Define

$$\beta \;\equiv\; <y-1> +\mathit{seq}(v, c)\ .$$

Then

$$\begin{array}{rcl} y & \equiv & \mathit{first}(\beta) + 1\ , \\ \mathit{seq}(v, c) & \equiv & \mathit{rest}(\beta)\ . \end{array}$$

Case 1: $y \in \mathit{keys}_+$.
Then

$$\mathit{sreach7}(n, y, v, c, m) \;\equiv\; \mathit{sreach6}(n, \mathit{seq}(v, c), m)\ .$$

If $v \;\equiv\; \Box$ then $\mathit{seq}(v, c) \;\equiv <>$ and hence

$$\mathit{sreach6}(n, \mathit{seq}(v, c), m) \;\equiv\; m\ .$$

Otherwise,

$$\mathit{seq}(v, c) \;\equiv <v> +\mathit{seq}(c(v), \mathit{rest}(c))\ ,$$

which implies that

$$\mathit{sreach6}(n, \mathit{seq}(v, c), m) \;\equiv\; \mathit{sreach7}(n, v+1, c(v), \mathit{rest}(c), m)\ .$$

■

Case 2: $y \notin \mathit{keys}_+$.
Define

$$z \;\stackrel{\text{def}}{\equiv}\; n(y) \;\text{ and }\; m_1 \;\stackrel{\text{def}}{\equiv}\; \mathit{insert}(m, z, y)\ .$$

If $z \notin \mathit{keys}(n)$

$$\begin{aligned}
&\mathit{sreach7}(n,y,v,c,m)\\
\equiv\ &\mathit{sreach6}(n\ominus y, <y> +\mathit{rest}(\beta), m_1)\\
\equiv\ &\mathit{sreach6}(n\ominus y, <y> +\mathit{seq}(v,c), m_1)\\
\equiv\ &\mathit{sreach7}(n\ominus y, y+1, v, c, m_1)\ .
\end{aligned}$$

Otherwise,

$$\begin{aligned}
&\mathit{sreach7}(n,y,v,c,m)\\
\equiv\ &\mathit{sreach6}(n\ominus y\ominus z, <z> + <y> +\mathit{seq}(v,c), m_1)\\
\equiv\ &\mathit{sreach7}(n\ominus y\ominus z, z+1, y, \mathit{prefix}(y,c), m_1)
\end{aligned}$$

because $y \notin {\downarrow}c$ implies

$$\mathit{seq}(y,\mathit{prefix}(y,c))\ \equiv\ <y> +\mathit{seq}(v,\mathit{rest}(\mathit{prefix}(y,c)))\ \equiv\ <y> +\mathit{seq}(v,c)\ .$$

Hence we obtain the recursion

```
      $sreach7(n, y, v, c, m)$
    $\equiv$ $P_7(n, y, v, c, m) \rhd$
     if $y \in keys_+$
       then if $v = \Box$
                then $m$
                else $sreach7(n, v+1, c(v), rest(c), m)$ fi
       else cell $z \equiv n(y)$ ;
            state $m_1 \equiv insert(m, z, y)$ ;
            if $z \notin keys(n)$
              then $sreach7(n \ominus y, y+1, v, c, m_1)$
              else $sreach7(n \ominus y \ominus z, z+1, y, prefix(y, c), m_1)$ fi fi .
```

Now we unite the parameters n, c, m into one map parameter l. We define

$$\mathit{sreach8}(n,y,v,c,m,l)\ \overset{\text{def}}{\equiv}\ P_8(n,y,v,c,m,l)\ \rhd\ n \mathbin{\triangleleft\!\!+} \mathit{sreach7}(n,y,v,c,m)$$

where

$$P_8(n,y,v,c,m,l)\ \overset{\text{def}}{\equiv}\ P_7(n,y,v,c,m)\ \wedge\ l = n\cup(m \mathbin{\triangleleft\!\!+} c)\ .$$

Let now n, y, v, c, m, l be such that $P_8(n,y,v,c,m,l)$ holds.

Case 1: $y \in \mathit{keys}_+$.
If $v\ \equiv\ \Box$ then $c\ \equiv\ \emptyset$ and hence

$$\mathit{sreach8}(n,y,v,c,m,l)\ \equiv\ n \mathbin{\triangleleft\!\!+} m\ \equiv\ n\cup m\ \equiv\ l\ .$$

Otherwise,

$$\mathit{sreach8}(n,y,v,c,m,l)\ \equiv\ \mathit{sreach8}(n, v+1, l(v), \mathit{rest}(c), m, l \mathbin{\triangleleft\!\!+} [v\mapsto\Box])$$

because $c(v)\ \equiv\ l(v)$ and $v\in{\downarrow}c \subseteq \Box{\downarrow}m$ together with $n\cup(m \mathbin{\triangleleft\!\!+} c)\ \equiv\ l$ implies $n\cup(m \mathbin{\triangleleft\!\!+} \mathit{rest}(c))$ $\equiv\ l \mathbin{\triangleleft\!\!+} [v\mapsto\Box]$. $\blacksquare$

Case 2: $y \notin \mathit{keys}_+$.
Define $z\ \overset{\text{def}}{\equiv}\ n(y)\ \equiv\ l(y)$ and $m_1\ \overset{\text{def}}{\equiv}\ \mathit{insert}(m,z,y)$. If $z\notin \mathit{keys}(n)$ then

$$\mathit{sreach8}(n,y,v,c,m,l)\ \equiv\ \mathit{sreach8}(n\ominus y, y+1, v, c, m_1, n\ominus y\cup(m_1 \mathbin{\triangleleft\!\!+} c))\ .$$

Since $z\in{\uparrow}\mathit{arcs}(n)\subseteq \mathit{keys}(n)\cup \mathit{src}(m)$ and $\mathit{cinner}(m)\cap \mathit{keys}_+\ \equiv\ \emptyset$, we have

$$z\notin \mathit{keys}(n)\ \Leftrightarrow\ z\in \mathit{src}(m)\ \Leftrightarrow\ m(z)\ \not\equiv\ \Box\ \Leftrightarrow\ l(z)\ \not\equiv\ \Box\ .$$

Moreover, $z \in src(m)$ implies

$$m_1 \;\equiv\; m \mathbin{\lessdot\!\!+}([z \mapsto y] \cup [y \mapsto m(z)])$$

by definition of *insert*. Furthermore, since

$$\downarrow\!c \;\equiv\; set(seq(v,c)) \subseteq \Box\!\downarrow\!m$$

and $y, z \notin \Box\!\downarrow\!m$, we get $y, z \notin \downarrow\!c$ and therefore

$$m_1 \mathbin{\lessdot\!\!+} c \;\equiv\; (m \mathbin{\lessdot\!\!+} c) \mathbin{\lessdot\!\!+}([z \mapsto y] \cup [y \mapsto m(z)]) \ .$$

Hence

$$n \ominus y \cup (m_1 \mathbin{\lessdot\!\!+} c) \;\equiv\; l \mathbin{\lessdot\!\!+}([z \mapsto y] \cup [y \mapsto m(z)]) \ .$$

If $z \in keys(n)$ (i.e., if $l(z) \;\equiv\; \Box$) then

$$sreach8(n,y,v,c,m,l) \;\equiv\; sreach8(n \ominus y \ominus z, z+1, y, prefix(y,c), m_1, l_1)$$

where

$$l_1 \;\overset{\text{def}}{\equiv}\; n \ominus y \ominus z \cup (m_1 \mathbin{\lessdot\!\!+} prefix(y,c)) \ .$$

Since $z \notin src(m)$,

$$m_1 \;\equiv\; m \mathbin{\lessdot\!\!+}([z \mapsto y] \cup [y \mapsto \Box]) \ .$$

Furthermore, since $z \in keys_+$ and $inner(m) \cap keys_+ \;\equiv\; \emptyset$, we get $z \notin \downarrow\!m$ and thus again $z \notin \downarrow\!c$. Therefore

$$l_1 \;\equiv\; l \mathbin{\lessdot\!\!+}([z \mapsto y] \cup [y \mapsto v]) \ .$$

■

Summarizing this we obtain

```
    $sreach8(n,y,v,c,m,l)$
$\equiv$ $P_8(n,y,v,c,m,l) \rhd$
    if $y \in keys_+$
      then if $v = \Box$
             then $l$
             else $sreach8(n, v+1, l(v), rest(c), m, l \mathbin{\lessdot\!\!+}[v \mapsto \Box])$ fi
      else cell $z \equiv l(y)$ ;
           cell $x \equiv l(z)$ ;
           state $m_1 \equiv insert(m,z,y)$ ;
           if $x \notin keys(n)$
             then state $c_1 \equiv prefix(y,c)$ ;
                  $sreach8(n \ominus y \ominus z, z+1, y, c_1, m_1, l \mathbin{\lessdot\!\!+}([z \mapsto y] \cup [y \mapsto v]))$
             else $sreach8(n \ominus y, y+1, v, c, m_1, l \mathbin{\lessdot\!\!+}([z \mapsto y] \cup [y \mapsto x]))$ fi fi .
```

Eliminating the redundant parameters n, c, m we obtain our final algorithm

```
    $sreach9(y,v,l)$
$\equiv$ if $y \in keys_+$
      then if $v = \Box$
             then $l$
             else $sreach9(v+1, l(v), l \mathbin{\lessdot\!\!+}[v \mapsto \Box])$ fi
      else cell $z \equiv l(y)$ ;
           cell $x \equiv l(z)$ ;
           if $x = \Box$
             then $sreach9(z+1, y, l \mathbin{\lessdot\!\!+}([z \mapsto y] \cup [y \mapsto v]))$
             else $sreach9(y+1, v, l \mathbin{\lessdot\!\!+}([z \mapsto y] \cup [y \mapsto x]))$ fi fi .
```

7.8 Assembling the Parts

We have to follow the chain of embeddings to obtain the final program for *sreach*. For n, y, v, c, m such that $P_7(n, y, v, c, m)$ holds we have

$$\begin{aligned}
& n \Lleftarrow \mathit{sreach7}(n, y, v, c, m) \\
\equiv\ & \mathit{sreach8}(n, y, v, c, m, n \cup (m \Lleftarrow c)) \\
\equiv\ & \mathit{sreach9}(y, v, n \cup (m \Lleftarrow c)) \ .
\end{aligned}$$

In the sequel we will — for convenience — consider only singleton sets $s \ \equiv\ \{z_0\}$ (denoted just by z_0) of starting keys $z_0 \in \mathit{keys}(n_0)$. To see what *sreach9* really does, we define

$$\begin{aligned}
m_0 &\overset{\text{def}}{\equiv} \mathit{sreach7}(n_0, z_0 + 1, \Box, \emptyset, [z_0 \mapsto \Box]) \ , \\
l_0 &\overset{\text{def}}{\equiv} \mathit{sreach9}(z_0 + 1, \Box, n_0) \ .
\end{aligned}$$

Since $n_0 \cup ([z_0 \mapsto \Box] \cup \emptyset) \ \equiv\ n_0 \cup [z_0 \mapsto \Box] \ \equiv\ n_0$ by $z_0 \in \mathit{keys}(n_0)$, we have $l_0 \ \equiv\ n_0 \Lleftarrow m_0$ and $\mathit{unchain}(m_0) \ \equiv\ \mathit{sreach}(n_0, z_0)$. Actually we want to compute m_0; but *sreach9* computes l_0 which is m_0 plus the garbage $l_0 \ominus \downarrow m_0$. Thus we must try to retrieve m_0 from l_0. Let now

$$\mathit{keys} \overset{\text{def}}{\equiv} \mathit{keys}(n_0) \ .$$

The invariant P_8 implies

$$\begin{aligned}
& \mathit{keys} \subseteq \downarrow l_0 \ , \\
& \mathit{src}(m_0) \ \equiv\ \{z \in \mathit{keys} \mid l_0(z) \ \not\equiv\ \Box\} \ .
\end{aligned}$$

Furthermore,

$$\downarrow m_0 \ \equiv\ \bigcup_{z \in \mathit{src}(m_0)} \mathit{ginner}(l_0, z)$$

because m_0 is a chaining (cf. Section 6.3.4). Hence

$$\begin{aligned}
& m_0 \\
\equiv\ & l_0 | \downarrow m_0 \\
\equiv\ & l_0 | (\bigcup \{\mathit{ginner}(l_0, z) \mid z \in \mathit{keys} \ \wedge\ l_0(z) \ \not\equiv\ \Box\}) \ .
\end{aligned}$$

We set

$$\mathit{retrieve}(l_0) \overset{\text{def}}{\equiv} l_0 | (\bigcup \{\mathit{ginner}(l_0, z) \mid z \in \mathit{keys} \ \wedge\ l_0(z) \ \not\equiv\ \Box\}) \ .$$

However, the operation *retrieve* will not be executed; we will just use it for further development.

We define a function *ocsreach* (**o**verwritten, **c**hained **s**tate reachability) by

$$\mathit{ocsreach}(n, z) \overset{\text{def}}{\equiv} \mathit{iscgstate}(n) \ \wedge\ z \in \mathit{keys}(n) \ \triangleright\ \mathit{sreach9}(z + 1, \Box, n)$$

(see Section 3.2 for *iscgstate*). We will not *compute* the test

$$\mathit{iskey}_+(z) \overset{\text{def}}{\equiv} z \in \mathit{keys}_+$$

occurring within *sreach9*, but regard it as an additional global parameter (to be realized e.g. by a bit vector). Thus the complete program reads

```
funct  $ocsreach$ $\equiv$ (state $n$, cell $z$) state :
        $sreach9(z + 1, \Box, n)$
where
funct  $sreach9$ $\equiv$ (cell $y, v$, state $l$) state :
```

```
if $iskey_+(y)$
  then if $v = \square$
          then $l$
          else $sreach9(v+1, l(v), l \Leftarrow [v \mapsto \square])$ fi
  else cell $z \equiv l(y)$ ;
       cell $x \equiv l(z)$ ;
       if $x = \square$
          then $sreach9(z+1, y, l \Leftarrow ([z \mapsto y] \cup [y \mapsto v]))$
          else $sreach9(y+1, v, l \Leftarrow ([z \mapsto y] \cup [y \mapsto x]))$ fi fi .
```

ocsreach meets the specification

$$sreach(n, z) \;\equiv\; unchain(retrieve(ocsreach(n, z)))$$

if n is a compressed state and $z \in keys(n)$.

8 Copying Chained Graph States

In the preceding chapter we have shown how to compute from a compressed graph state

$$n_0 \;\equiv\; blockrep(G_0, g) \;,$$

a membership test $iskey_+$ for $keys_+(n_0)$, and

$$s \;\equiv\; \{z_0\} \subseteq keys(n)$$

a map

$$l_0 \;\equiv\; ocsreach(n_0, iskey_+, z_0)$$

such that

$$m_0 \;\overset{\text{def}}{\equiv}\; retrieve(l_0)$$

is a chained representation of $arcs(n_1)$ (so that $unchain(m_0) \;\equiv\; arcs(n_1) \;\equiv\; sreach(n_0, s)$), where

$$n_1 \;\overset{\text{def}}{\equiv}\; blockrep(G_s, g)$$

and

$$G_s \;\overset{\text{def}}{\equiv}\; reach(G_0, s) \;.$$

According to our problem analysis in Section 4.3 we are left with the task of computing the compressed state $n_s \;\equiv\; blockrep(G_s, g_s)$ (where g_s is a perfect allocation of G_s) from n_1 and the collapsing map k defined by

$$k(g(x)+i) \;\overset{\text{def}}{\equiv}\; g_s(x)+i \quad (x \in {\downarrow}G_s,\ i \in [0 : |G(x)|]).$$

We generalize this to the following problem: Given a graph G and an overlap-free and order-preserving (but not necessarily gap-free) allocation g, compute from

$$n \;\overset{\text{def}}{\equiv}\; blockrep(G, g)$$

the compressed state

$$n_c \;\overset{\text{def}}{\equiv}\; blockrep(G, g_c)$$

for the unique compressing allocation g_c of G.

In Section 4.3 we have already seen that n_c is the copy of n via the collapsing map k; i.e., $n_c \equiv k \circ n \circ k^{-1}$, where k now is defined by

$$\text{(K)} \qquad k(g(x)+i) \overset{\text{def}}{\equiv} g_c(x)+i$$

for $x \in \downarrow G$ and $i \in [0 : |G(x)|])$. Assuming, in accordance with the situation above, that we have a map l such that $m \overset{\text{def}}{\equiv} \mathit{retrieve}(l)$ is a chained representation of $\mathit{arcs}(n)$, we have the decomposition

$$\begin{aligned} & n \\ \equiv\ & n_\square \cup \mathit{arcs}(n) \\ \equiv\ & n_\square \cup \mathit{unchain}(m) \end{aligned}$$

where

$$n_\square \overset{\text{def}}{\equiv} n|\mathit{keys}(n) \ (\equiv n\backslash \mathit{arcs}(n)) \ .$$

Now we can employ the functions derived in Chapter 6 to calculate

$$\begin{aligned} & n_c \\ \equiv\ & k \circ n \circ k^{-1} \\ \equiv\ & k \circ (n_\square \cup \mathit{unchain}(m)) \circ k^{-1} \\ \equiv\ & (k \circ n_\square \circ k^{-1}) \cup (k \circ \mathit{unchain}(m) \circ k^{-1}) \\ \equiv\ & (\text{since } k(\square) \equiv \square) \\ & (n_\square \circ k^{-1}) \cup \mathit{copy}(m,k) \\ \equiv\ & \mathit{copass}(n_\square, k) \cup \mathit{copass}(\mathit{relocate}(m,k),k) \ . \end{aligned}$$

By the definition of *copass* we have

$$\begin{aligned} & \mathit{copass}(n_\square, k) \\ \equiv\ & \bigcup_{z \in \downarrow n_\square} [k(z) \mapsto n_\square(z)] \\ \equiv\ & \bigcup_{z \in \downarrow n_\square} [k(z) \mapsto \square] \ . \end{aligned}$$

Furthermore we define

$$\begin{aligned} p &\overset{\text{def}}{\equiv} \mathit{relocate}(m,k) \equiv k \circ \mathit{arcs}(n) \ , \\ \mathit{size}(z) &\overset{\text{def}}{\equiv} \mathit{succ}_{\mathit{keys}_+}(z) - z \ . \end{aligned}$$

Using that

$$\begin{aligned} & \downarrow p \\ \equiv\ & \downarrow \mathit{arcs}(n) \\ \equiv\ & \bigcup_{z \in \mathit{keys}(n)} [z+1 : z+\mathit{size}(z)] \end{aligned}$$

we obtain

$$\begin{aligned} & \mathit{copass}(p,k) \\ \equiv\ & \bigcup_{x \in \downarrow p} [k(x) \mapsto p(x)] \\ \equiv\ & \bigcup_{z \in \mathit{keys}(n)} \ \bigcup_{i \in [1:\mathit{size}(z)]} [k(z+i) \mapsto p(z+i)] \ . \end{aligned}$$

Therefore

$$\begin{array}{ll} & n_c \\ \equiv & \mathit{copass}(n_\Box, k) \cup \mathit{copass}(p, k) \\ \equiv & \bigcup\limits_{z \in \mathit{keys}(n)} ([k(z) \mapsto \Box] \cup (\bigcup\limits_{i \in [1:\mathit{size}(z)]} [k(z) + i \ \mapsto \ p(z+i)])) \end{array}$$

since $k(z+i) \ \equiv \ k(z) + i$ for $i \in [1 : \mathit{size}(z)]$.

Thus our problem divides into two parts:

1. Compute $p \ \equiv \ \mathit{relocate}(m, k)$ from l.
2. Compute the union above.

8.1 Relocation Pass

By the definition of relocate (cf. Section 6.3.4) it is immediate that

$$\mathit{relocate}(m, k) \ \equiv \ \bigcup_{z \in \mathit{keys}(n)} \mathit{fibre}(m, z, k(z))$$

where

$$\mathit{fibre}(m, z, y) \ \equiv \ [\mathit{ginner}(m, z) \mapsto y] \ ,$$

because $\mathit{src}(m) \ \equiv \ \mathit{keys}(n)$. We aim at an algorithm that traverses the set $\mathit{keys}(n)$ in ascending order. Therefore we try to compute $k(\mathit{next}_n(z))$ (cf. Section 7.5) incrementally from $k(z)$ (for $z \in \mathit{keys}(n) \cup \{\Box\}$ such that $\overset{\vee}{z} \cap \mathit{keys}_+(n) \backslash \{z\} \ \not\equiv \ \emptyset$; cf. Section 2.2.2 for $\overset{\vee}{z}$). Let

$$\begin{array}{rcl} z_1 & \overset{\text{def}}{\equiv} & \mathit{next}_n(z) \ , \\ x & \overset{\text{def}}{\equiv} & g^{-1}(z) \ . \end{array}$$

Then, since g is order-preserving,

$$z_1 \ \equiv \ \mathit{next}_n(g(x)) \ \equiv \ g(\mathit{succ}_{\downarrow G}(x)) \ .$$

Therefore

$$\begin{array}{ll} & k(z_1) \\ \equiv & k(g(\mathit{succ}_{\downarrow G}(x))) \\ \equiv & \text{(by (K))} \\ & g_c(\mathit{succ}_{\downarrow G}(x)) \\ \equiv & \text{(by Theorem 3.2.6)} \\ & g_c(x) + \mathit{size}(x) \\ \equiv & (\text{since } n \ \equiv \ \mathit{blockrep}(G, g)) \\ & k(g(x)) + \mathit{size}(g(x)) \\ \equiv & k(z) + \mathit{size}(z) \ . \end{array}$$

Suppressing the parameter k we define

$$\mathit{relocate1}(m, z, y) \ \overset{\text{def}}{\equiv} \ y = k(z) \ \rhd \ \bigcup_{x \in \mathit{keys}(n) \backslash \hat{z}} \mathit{fibre}(m, x, k(x))$$

with the embedding

$$\mathit{relocate}(m, k) \ \equiv \ \mathit{relocate1}(m, \Box, \Box) \ .$$

Immediately we obtain the recursion

$$\begin{array}{rl} & \mathit{relocate1}(m,z,y) \\ \equiv & y=k(z) \ \rhd \ \textsf{if } \mathit{next}_n(z)\notin \mathit{keys} \\ & \qquad\qquad\qquad \textsf{then } \emptyset \\ & \qquad\qquad\qquad \textsf{else } \mathit{fibre}(m,\mathit{next}_n(z),y+\mathit{size}(z)) \cup \\ & \qquad\qquad\qquad\qquad \mathit{relocate1}(m,\mathit{next}_n(z),y+\mathit{size}(z)) \textsf{ fi} \ . \end{array}$$

Because we are given keys_+ rather than $\mathit{keys}_+(n)$, we want to derive an algorithm based on

$$\mathit{next}(x) \overset{\text{def}}{\equiv} \mathit{succ}_{\mathit{keys}_+}(x)$$

rather than on next_n. To this end we introduce

$$\mathit{pred}_n(x) \overset{\text{def}}{\equiv} \mathit{sup}(\mathit{keys}(n) \cap \hat{x} \setminus \{x\})$$

and define

$$\mathit{relocate2}(m,z,y,x) \overset{\text{def}}{\equiv} x \in \mathit{keys}_+ \ \wedge \ \mathit{pred}_n(x)=z \ \rhd \ \mathit{relocate1}(m,z,y)$$

with the embedding

$$\mathit{relocate}(m,k) \ \equiv \ \mathit{relocate1}(m,\square,\square) \ \equiv \ \mathit{relocate2}(m,\square,\square,\square+1) \ .$$

Now assume $\mathit{pred}_n(x) \ \equiv \ z$.

Case 1: $x \ \equiv \ \mathit{bound}$.
Then $z \ \equiv \ \mathit{bound}(n)$ and therefore $\mathit{relocate1}(m,z,y) \ \equiv \ \emptyset$. ■

Case 2: $x \in \mathit{keys}\setminus \mathit{keys}(n)$.
Then $\mathit{pred}_n(\mathit{next}(x)) \ \equiv \ \mathit{pred}_n(x) \ \equiv \ z$ and thus

$$\mathit{relocate2}(m,z,y,x) \ \equiv \ \mathit{relocate2}(m,z,y,\mathit{next}(x)) \ .$$

■

Case 3: $x \in \mathit{keys}(n)$.
Then $\mathit{pred}_n(\mathit{next}(x)) \ \equiv \ x \ \equiv \ \mathit{next}_n(z)$. Therefore, by the recursion for $\mathit{relocate1}$,

$$\begin{array}{rl} & \mathit{relocate2}(m,z,y,x) \\ \equiv & \mathit{relocate1}(m,z,y) \\ \equiv & \mathit{fibre}(m,x,y+\mathit{size}(z)) \cup \mathit{relocate1}(m,x,y+\mathit{size}(z)) \\ \equiv & \mathit{fibre}(m,x,y+\mathit{size}(z)) \cup \mathit{relocate2}(m,x,y+\mathit{size}(z),\mathit{next}(x)) \ . \end{array}$$

■

Thus, defining

$$R(m,z,y,x) \overset{\text{def}}{\equiv} x \in \mathit{keys}_+ \ \wedge \ y=k(x) \ \wedge \ \mathit{pred}_n(x)=z$$

we get the recursion

```
    $\mathit{relocate2}(m,z,y,x)$
  $\equiv$ $R(m,z,y,x) \rhd$
    if $x = \mathit{bound}$
      then $\emptyset$
      else if $x \notin \mathit{keys}(n)$
             then $\mathit{relocate2}(m,z,y,\mathit{next}(x))$
             else cell $y_1 \equiv y+\mathit{size}(z)$ ;
                  cell $x_1 \equiv \mathit{next}(x)$ ;
                  $\mathit{fibre}(m,x,y_1) \cup \mathit{relocate2}(m,x,y_1,x_1)$ fi fi .
```

The body of the function *copass* is, like the body of *relocate* , a union over the index set $keys(n)$. Thus, forming *copass* requires a second traversal of $keys(n)$ and therefore of *keys* since we only have *keys* (and not $keys(n)$) as a primitive. But if we *compute* $next_n$ (as a map) simultaneously with *relocate* , we can afterwards use it to traverse $keys(n)$, thus improving speed efficiency considerably. To this end we define

$$\begin{aligned} mapnext(n) &\overset{\text{def}}{\equiv} \Big(\bigcup_{y \in keys(n) \cup \{\Box\}} [y \mapsto next_n(y)]\Big) \cup [bound(n) \mapsto \Box] \\ &\equiv \Big(\bigcup_{y \in keys(n)} [pred_n(y) \mapsto y]\Big) \cup \\ &\quad\; [pred_n(bound(n)) \mapsto bound(n)] \cup [bound(n) \mapsto \Box] \ . \end{aligned}$$

Then

$$mapnext(n)(y) \ \equiv \ next_n(y) \ \text{ for } \ y \in keys(n) \cup \{\Box\} \ \ (\equiv \ \downarrow next_n) \ .$$

Applying similar techniques as in the case of *relocate* , we generalize *mapnext* to

$$\begin{aligned} mapnext1(n, z, x) &\overset{\text{def}}{\equiv} x \in keys_+ \ \wedge \ z = pred_n(x) \ \rhd \\ &\quad \Big(\bigcup_{y \in keys(n) \setminus \hat{z}} [pred_n(y) \mapsto y]\Big) \cup \\ &\quad [pred_n(bound(n)) \mapsto bound(n)] \cup [bound(n) \mapsto \Box] \end{aligned}$$

with the embedding

$$mapnext(n) \ \equiv \ mapnext1(n, \Box, n) \ .$$

We obtain the recursion

```
    $mapnext1(n, z, x)$
$\equiv$ $x \in keys_+ \ \wedge \ z = pred_n(x) \ \rhd$
    if $x = bound$
      then $[z \mapsto next(z)] \cup [next(z) \mapsto \Box]$
      else if $x \notin keys(n)$
             then $mapnext1(n, z, next(x))$
             else $mapnext1(n, x, next(x)) \cup [z \mapsto x]$ fi fi .
```

The recursions for *relocate2* and *mapnext1* fit together nicely. Furthermore, $\downarrow relocate2(n, z, y, x) \subseteq \downarrow arcs(n)$ and $\downarrow mapnext1(n, z, x) \subseteq keys_+(n) \cup \{\Box\}$, which implies that these domains are disjoint. Hence we may define

$$relocate3(m, z, y, x) \ \overset{\text{def}}{\equiv} \ relocate2(n, z, y, x) \cup mapnext1(m, z, x)$$

and obtain by function combination (see [Berghammer, Ehler 90]) the recursion

```
    $relocate3(m, z, y, x)$
$\equiv$ $R(m, z, y, x) \ \rhd$
    if $x = bound$
      then $[z \mapsto next(z)] \cup [next(z) \mapsto \Box]$
      else if $x \notin keys(n)$
             then $relocate3(m, z, y, next(x))$
             else cell $y_1 \ \equiv \ y + size(z)$ ;
                  cell $x_1 \ \equiv \ next(x)$ ;
                  $fibre(m, x, y_1) \cup [z \mapsto x] \cup relocate3(m, x, y_1, x_1)$ fi fi .
```

We have

$$relocate(m,k) \cup mapnext(m) \;\equiv\; relocate3(m,\Box,\Box,\Box+1)\ .$$

Now we transform *relocate3* in such a way that it works on l rather than on $m \;\equiv\; retrieve(l)$. Recall that

$$retrieve(l) \;\equiv\; l|\!\downarrow\! m \;\equiv\; l|(\bigcup \{ginner(l,z) \mid z \in keys \;\wedge\; l(z) \;\not\equiv\; \Box\})$$

and $keys(n) \;\equiv\; src(m)$ since $n \;\equiv\; unchain(m)$. Thus, for $x \in keys_+$,

$$x \notin keys(n) \;\Leftrightarrow\; l(z) \;\equiv\; \Box$$

and

$$fibre(m,x,y) \;\equiv\; fibre(l,x,y)\ .$$

Hence we obtain for

$$relocate4(l,z,y,x) \;\overset{\text{def}}{\equiv}\; relocate3(retrieve(l),z,y,x)$$

the recursion

```
    $relocate4(l,z,y,x)$
$\equiv$ $R(retrieve(l),z,y,x) \rhd$
    if $x = bound$
      then $[z \mapsto next(z)] \cup [next(z) \mapsto \Box]$
      else if $l(x) = \Box$
              then $relocate4(l,z,y,next(x))$
              else cell $y_1 \;\equiv\; y + size(z)$ ;
                   cell $x_1 \;\equiv\; next(x)$ ;
                   $fibre(l,x,y_1) \cup [z \mapsto x] \cup relocate4(l,x,y_1,x_1)$ fi fi .
```

Since for the initial call $relocate4(l,\Box,\Box,\Box{+}1)$ the assertion is satisfied, we may omit it (cf. [Möller 89] for a formal treatment of this step). Furthermore we may add an accumulating parameter q to obtain a tail-recursive form. Finally — to prepare overwriting — we replace in the second recursive call l by $l \;\ominus\; l^*(x)$ where

$$l^*(x) \;\overset{\text{def}}{\equiv}\; \bigcup_{i\in\mathbb{N}} x{\uparrow}l^i(x)\ ;$$

this does not affect the algorithm. The result is

```
    $relocate5(l,z,y,x,q)$
$\equiv$ $R(retrieve(l),z,y,x) \rhd$
    if $x = bound$
      then $q \cup [z \mapsto next(z)] \cup [next(z) \mapsto \Box]$
      else if $l(x) = \Box$
              then $relocate5(l,z,y,next(x),q)$
              else cell $y_1 \;\equiv\; y + size(z)$ ;
                   cell $x_1 \;\equiv\; next(x)$ ;
                   state $l_1 \;\equiv\; l \;\ominus\; l^*(x)$ ;
                   state $q_1 \;\equiv\; q \cup fibre(l,x,y_1) \cup [z \mapsto x]$ ;
                   $relocate5(l_1,x,y_1,x_1,q_1)$ fi fi .
```

Defining

$$nrelocate(l) \;\overset{\text{def}}{\equiv}\; relocate(retrieve(l),k) \cup mapnext(n)\ ,$$

where n is the state to be compressed, we obtain

$$nrelocate(l) \;\equiv\; relocate5\,(l, \Box, \Box, \Box + 1, \emptyset)\ .$$

Obviously the property $\downarrow\!l \cap \downarrow\!q = \emptyset$ is an invariant for *relocate5* . Thus we may carry $l \cup q$ on a new parameter *lq* . This gives our final version

```
    $relocate6\,(lq, z, y, x)$
$\equiv$ if $x = bound$
      then $lq \cup [z \mapsto next(z)] \cup [next(z) \mapsto \Box]$
      else if $lq(x) = \Box$
             then $relocate6\,(lq, z, y, next(x))$
             else cell $y_1 \;\equiv\; y + size(z)$ ;
                  cell $x_1 \;\equiv\; next(x)$ ;
                  state $lq_1 \;\equiv\; lq \mathbin{\triangleleft\!\!+} (fibre(lq, x, y_1) \cup [z \mapsto x])$ ;
                  $relocate6\,(lq_1, x, y_1, x_1)$                    fi fi .
```

relocate6 satisfies

$$l \mathbin{\triangleleft\!\!+} nrelocate(l) \;\equiv\; relocate6\,(l, \Box, \Box, \Box + 1) \;\overset{\text{def}}{\equiv}\; onrelocate(l)$$

for l such that $retrieve(l)$ is a chained map representation.

8.2 Copying Pass

We have seen that the relocation pass returns a union of the state $p \;\overset{\text{def}}{\equiv}\; relocate(m, k)$ and the map $l \;\overset{\text{def}}{\equiv}\; mapnext(n)$ where

$$p \;\equiv\; relocate(m, k) \;\equiv\; \bigcup_{z \in keys(n)} fibre(m, z, k(z))$$

and

$$l(y) \;\equiv\; mapnext(n)(y) \;\equiv\; next_n(y)$$

for $y \in keys(n) \cup \{\Box\}$. We recall that $l(\Box)$ is the minimum of $keys_+(n)$ and $l(bound(n)) \;\equiv\; \Box$. Moreover, $\downarrow\!p \cap \downarrow\!l \;\equiv\; \emptyset$ making the union $p \cup l$ a well-defined map. We set $lp \;\overset{\text{def}}{\equiv}\; p \cup l$. Now we have to construct n_c from l_p based on the values $size(z)$ for $z \in keys(n)$; we call the corresponding function *copypass* , so that

$$n_c \;\equiv\; copypass(lp)\ .$$

As stated before, n_c is represented by

$$n_c \;\equiv\; \bigcup_{z \in keys(n)} \Big([k(z) \mapsto \Box] \cup \bigcup_{i \in [1:size(z)]} [k(z) + i \mapsto p(z + i)]\Big)\ ,$$

where now *lp* can be substituted for p. We define, for $z \in keys(n)$,

$$fibrecopass(z, lp, k(z)) \;\overset{\text{def}}{\equiv}\; [k(z) \mapsto \Box] \cup \bigcup_{i \in [1:size(z)]} [k(z) + i \mapsto lp(z + i)])$$

and

$$fibrecopass(\Box, lp, \Box) \;\overset{\text{def}}{\equiv}\; \emptyset\ .$$

Next, for $y \in \mathit{keys}_+(n) \cup \{\Box\}$ we define

$$\mathit{copass1}(y, \mathit{lp}, k) \overset{\text{def}}{\equiv} \bigcup_{z \in \mathit{keys}(n) \setminus \overset{\vee}{y}} \mathit{fibrecopass}(z, \mathit{lp}, k(z)) \ .$$

Then

$$n_c \ \equiv \ \mathit{copass1}(\mathit{bound}(n), \mathit{lp}, k) \ .$$

We have the following recursion equations:

$$\begin{aligned} \mathit{copass1}(\mathit{next}_n(\Box), \mathit{lp}, k) \ &\equiv \ \emptyset \ , \\ \mathit{copass1}(\mathit{next}_n(y), \mathit{lp}, k) \ &\equiv \ \mathit{copass1}(y, \mathit{lp}, k) \cup \mathit{fibrecopass}(y, \mathit{lp}, k(y)) \end{aligned}$$

for $y \not\equiv \Box$. Embedding with an accumulator nc leads to the "ascending" recursion

```
    $\mathit{copass2}(nc, y, \mathit{lp}, k)$
 $\equiv$  $nc = \mathit{copass1}(y, \mathit{lp}, k) \rhd$
    if $y = \mathit{bound}(n)$
      then $nc$
      else  $\mathit{copass2}(nc \cup \mathit{fibrecopass}(y, \mathit{lp}, k(y)), \mathit{next}_n(y), \mathit{lp}, k)$ fi
```

with

$$\mathit{copass2}(\emptyset, \mathit{next}_n(\Box), \mathit{lp}, k) \ \equiv \ \mathit{copass1}(\mathit{bound}(n), \mathit{lp}, k) \ .$$

The last equation is a consequence of the recursion equations and of the finiteness of the function next_n. Hence we can compute n_c as

$$n_c \ \equiv \ \mathit{copass2}(\emptyset, \mathit{next}_n(\Box), \mathit{lp}, k) \ .$$

Obviously, the collapsing map k is only used in computing $\mathit{fibrecopass}(y, \mathit{lp}, k(z))$; moreover, in this k is only used to yield $k(y)$. As already shown in Section 3.2, k satisfies the recursion equation

$$k(\mathit{next}_n(y)) \ \equiv \ k(y) + \mathit{size}(y)$$

for $y \in \mathit{keys}(n)$ such that $\mathit{next}_n(y) \not\equiv \mathit{bound}(n)$. Hence we can eliminate k and use its values on the single cells instead. Second, $\mathit{copass2}(nc, y, \mathit{lp}, k)$ in fact only depends on the restriction of lp to $\overset{\vee}{y}$. Therefore we can replace $\mathit{copass2}$ by

```
    $\mathit{copass3}(nc, y, \mathit{lp}, c_1)$
 $\equiv$  if $y = \mathit{bound}(n)$
      then $nc$
      else  cell $y_1 \ \equiv \ \mathit{next}_n(y)$ ;
            cell $c_2 \ \equiv \ c_1 + \mathit{size}(y)$ ;
            $\mathit{copass3}(nc \cup \mathit{fibrecopass}(y, \mathit{lp}, c_1), y_1, \mathit{lp}| \overset{\vee}{y_1}, c_2)$ fi .
```

Furthermore, the condition $y = \mathit{bound}(n)$ can be equivalently replaced by $\mathit{lp}(y) \ \equiv \ \Box$; also, $\mathit{lp}(y)$ can be substituted for $\mathit{next}_n(y)$, since $\mathit{next}_n(y) \ \equiv \ \mathit{lp}(y)$ for all $y \in \mathit{keys}(n) \cup \{\Box\}$.

For our last modification we observe that under the assertion $nc \cap \mathit{lp}| \overset{\vee}{y_1} = \emptyset$ also

$$(nc \cup \mathit{fibrecopass}(y, \mathit{lp}, c_1)) \cap \mathit{lp}| \overset{\vee}{y_1} \ \equiv \ \emptyset \ .$$

Thus we can overwrite lp using

```
    $copass4(nlp, y, c_1)$
  $\equiv$ if $nlp(y) = \Box$
      then $nlp$
      else cell $y_1 \equiv nlp(y)$ ;
           cell $c_2 \equiv c_1 + size(y)$ ;
           $copass4(nlp \cup fibrecopass(y, nlp, c_1), y_1, c_2)$ fi .
```

Now we are in the position to compute n_c from lp by a copy pass, viz.

$$n_c \equiv copypass(lp) \equiv n'_c|\{|{\downarrow}n|\}^\wedge$$

where

$$n'_c \overset{\text{def}}{\equiv} copass4(lp, next_n(\Box), \Box + 1) .$$

Note that, in general, $n_c \not\equiv n'_c$ since n'_c may contain parts of lp that are not overwritten by *copass4*. It should also be clear that the computation of $fibrecopass(y, nlp, c_1)$ is elementary.

8.3 Integrating the Relocation and Copying Passes

Summarizing the results of the preceding sections we obtain the following: Assume that $m \equiv retrieve(l)$ and $n \equiv unchain(m)$. Then

$$\begin{aligned} & copypass(onrelocate(l)) \\ \equiv\ & copypass(l \twoheadleftarrow nrelocate(l)) \\ \supseteq\ & copass(relocate(m, k), k) \\ \equiv\ & copy(n, k) \\ \equiv\ & n_c \ . \end{aligned}$$

Thus,

$$ocopy(l) \overset{\text{def}}{\equiv} copypass(onrelocate(l))$$

is n_c, generally together with some garbage attached to the end of n_c which, however, is easily deleted. Hence *ocopy* solves the copying part of our garbage collection problem. The complete program for it reads

```
funct $ocopy \equiv$ (state $l$) state :
      ⌈ funct $fibre \equiv$ (state $m$, cell $z, y, x$) state :≪ body of $fibre$ ≫ ,
        funct $relocate6 \equiv$ (state $lq$, cell $z, y, x$) state :≪ body of $relocate6$ ≫ ,
        funct $onrelocate \equiv$ (state $l$) state :
              $relocate6(l, \Box, \Box, \Box + 1)$ ,
        funct $copass4 \equiv$ (state $nlp$, cell $y, c_1$) state :≪ body of $copass4$ ≫ ,
        funct $copypass \equiv$ (state $l$) state :
              $copass4(lp, next_n(\Box), \Box + 1)|\{|{\downarrow}n|\}^\wedge$ ;

        $copypass(onrelocate(l))$ ⌋ .
```

9 Summary and Conclusion

9.1 The Complete Algorithm

If we assemble the algorithms from Chapters 7 and 8, we obtain the following garbage collection program:

```
funct $garcoll$ $\equiv$ (state $n$, cell $z$ : $iscgstate(n) \ \wedge\ z \in keys(n)$) state :
    ⌈ cell $bound$ $\equiv$ $sup(\downarrow n) + 1$;
      funct $iskey_+$ $\equiv$ (cell $x$) bool : $x \in keys(n) \cup \{bound\}$,

      funct $ocsreach$ $\equiv$ (state $n$, cell $z$) state :
            $sreach9(z + 1, \Box, n)$ ,
      funct $sreach9$ $\equiv$ (cell $y, v$, state $l$) state :
            if $iskey_+(y)$
            then if $v = \Box$
                   then $l$
                   else $sreach9(v + 1, l(v), l \Leftarrow [v \mapsto \Box])$ fi
            else cell $z$ $\equiv$ $l(y)$ ;
                 cell $x$ $\equiv$ $l(z)$ ;
                 if $x = \Box$
                   then $sreach9(z + 1, y, l \Leftarrow ([z \mapsto y] \cup [y \mapsto v]))$
                   else $sreach9(y + 1, v, l \Leftarrow ([z \mapsto y] \cup [y \mapsto x]))$ fi fi ,

      funct $ocopy$ $\equiv$ (state $l$) state :
            $copypass(onrelocate(l))$ ,
      funct $onrelocate$ $\equiv$ (state $l$) state :
            $relocate6(l, \Box, \Box, \Box + 1)$ ,
      funct $relocate6$ $\equiv$ (state $lq$, cell $z, y, x$) state :
            if $x = bound$
            then $lq \cup [z \mapsto next(z)] \cup [next(z) \mapsto \Box]$
            else if $lq(x) = \Box$
                   then $relocate6(lq, z, y, next(x))$
                   else cell $y_1$ $\equiv$ $y + size(z)$ ;
                        cell $x_1$ $\equiv$ $next(x)$ ;
                        state $lq_1$ $\equiv$ $lq \Leftarrow (fibre(lq, x, y_1) \cup [z \mapsto x])$ ;
                        $relocate6(lq_1, x, y_1, x_1)$                      fi fi ,
      funct $fibre$ $\equiv$ (state $l$, cell $x, y$ : $isanchored(l) \ \wedge\ x \in \downarrow l$) state :
            ⌈ cell $z$ $\equiv$ $l(x)$ ;
              if $z = \Box$ then $\emptyset$ else $[z \mapsto y] \cup fibre(l, z, y)$ fi ⌋ ,
      funct $copypass$ $\equiv$ (state $l$) state :
            $copass4(lp, next_n(\Box), \Box + 1)|\{|\downarrow n|\}^\wedge$ ;
      funct $copass4$ $\equiv$ (state $nlp$, cell $y, c_1$) state :
            if $nlp(y) = \Box$
            then $nlp$
            else cell $y_1$ $\equiv$ $nlp(y)$ ;
                 cell $c_2$ $\equiv$ $c_1 + size(y)$ ;
                 $copass4(nlp \cup fibrecopass(y, nlp, c_1), y_1, c_2)$ fi ;

      $ocopy(ocsreach(n, z))$                                                ⌋ .
```

In this version the compressed state

$$n_c \overset{\text{def}}{\equiv} garcoll(n, z)$$

may still contain garbage in the cells greater than $bound(n_c)$. But this does not matter because we can easily calculate $bound(n_c)$ and thus we know which cells are free for subsequent re-use. The computation of $bound(n_c)$ can even be combined with the routine *relocate6* , since in the termination case $x = bound$ we have $bound(n_c) \equiv y + size(z)$. As a final step, one can now pass immediately to a procedural version of this algorithm and thus introduce selective updating. Note that no further development at that level is necessary.

9.2 Discussion

The algebra of maps has proved to be an invaluable tool in our derivation of the garbage collection algorithm. It has allowed us to carry out the development in full detail without leading to unmanageably complex expressions. It also has allowed us to stay entirely at the applicative language level where calculation is easy due to the strong algebraic properties of applicative languages. We are convinced that this way of approaching machine-oriented programs is an important step towards achieving systems software with guaranteed correctness.

It is to be hoped that the theory about states and chained representations developed in this paper can be re-used for further transformational studies of pointer algorithms. Apart from this, the development shows the importance and feasibility of early modularization of a specification; the transformational approach is strong enough to allow efficiency-increasing integration of program parts that have been developed independently.

Acknowledgement
We are grateful to R. Tobiasch for drawing our attention to the garbage collection problem. Valuable comments and suggestions concerning this paper have been provided by F.L. Bauer, R. Berghammer, R. Dewar, W. Dosch, M. Lichtmannegger, H. Partsch, P. Pepper, M. Sintzoff, and, in particular, by H. Ehler.

10 References

[Bauer, Wössner 82]
F.L. Bauer, H. Wössner: Algorithmic language and program development. New York: Springer 1982

[Bauer et al. 85]
F.L. Bauer et al.: The Munich project CIP. Volume I: The wide spectrum language CIP-L. Lecture Notes in Computer Science **183**. New York: Springer 1985

[Berghammer, Ehler 90]
R. Berghammer, H. Ehler: On the use of elements of functional programming in program development by transformations. This volume

[Berghammer et al. 87]
R. Berghammer, H. Ehler, H. Zierer: Development of graph algorithms by program transformation. In: H. Göttler, H.-J. Schneider (eds.): Graph-theoretic concepts in computer science. Lecture Notes in Computer Science **314**. Berlin: Springer 1988, 206–218

[Broy, Pepper 82]
M. Broy, P. Pepper: Combining algebraic and algorithmic reasoning: An approach to the Schorr-Waite-Algorithm. ACM TOPLAS **4**, 362–381 (1982)

[Dewar, McCann 77]
R. Dewar, A. McCann: MACRO SPITBOL — a SNOBOL4 compiler. Software — Practice and Experience **7**, 95–113 (1977)

[Dewar et al. 82]
R. Dewar, M. Sharir, E. Weixelbaum: Transformational derivation of a garbage collection algorithm. ACM TOPLAS **4**, 650–667 (1982)

[van Diepen, de Roever 86]
N. van Diepen, W. de Roever: Program derivation through transformations: The evolution of list-copying algorithms. Science of Computer Programming **6**, 213–272 (1986)

[Möller 87]
B. Möller: Entwicklung eines Speicherbereinigungsalgorithmus. Institut für Informatik der TU München, Manuscript, October 1987

[Möller 89]
B. Möller: Applicative assertions. In : J.L.A. van de Snepscheut (ed.): Mathematics of Program Construction, Groningen, 26–30 June 1989. Lecture Notes in Computer Science **375**. Berlin: Springer 1989, 348–362

[Möller 90]
B. Möller: Formal derivation of pointer algorithms. In: M. Broy (ed.): Informatik im Kreuzungspunkt von Numerischer Mathematik, Rechnerentwurf, Programmierung, Algebra und Logik. Festkolloquium, München, 12.–14. Juni 1989. Lecture Notes in Computer Science (to appear)

11 Appendix: Some Properties of Map Operations

All properties in this appendix are stated without proof. We group them according to the map operations they involve.

11.1 Domain and Range

(1)	$t{\downarrow}m \subseteq {\downarrow}m$	(2)	$s{\uparrow}m \subseteq {\uparrow}m$
(3)	$s \subseteq t \;\Rightarrow\; s{\downarrow}m \subseteq t{\downarrow}m$	(4)	$s \subseteq t \;\Rightarrow\; s{\uparrow}m \subseteq t{\uparrow}m$
(5)	$\overline{s{\downarrow}m} \;\equiv\; {\downarrow}m \backslash s{\downarrow}m \cup \overline{{\downarrow}m}$	(6)	$({\uparrow}m)\backslash t \;\equiv\; \overline{t{\downarrow}m}{\uparrow}m$
(7)	$s \subseteq {\downarrow}m \;\Rightarrow\; (s{\uparrow}m){\downarrow}m \supseteq s$	(8)	$t \subseteq {\uparrow}m \;\Rightarrow\; (t{\downarrow}m){\uparrow}m \equiv\; t$
(9)	$s \subseteq t{\downarrow}m \;\Rightarrow\; s{\uparrow}m \subseteq t$	(10)	$t \supseteq s{\uparrow}m \;\Rightarrow\; t{\downarrow}m \supseteq s$
(11)	$({\uparrow}m){\downarrow}m \;\equiv\; {\downarrow}m$	(12)	$({\downarrow}m){\uparrow}m \;\equiv\; {\uparrow}m$
(13)	$s{\downarrow}(m \cup n) \;\equiv\; s{\downarrow}m \cup s{\downarrow}n$	(14)	${\downarrow}(m \cup n) \;\equiv\; {\downarrow}m \;\cup {\downarrow}n$
(15)	$t{\uparrow}(m \cup n) \;\equiv\; t{\uparrow}m \cup t{\uparrow}n$	(16)	${\uparrow}(m \cup n) \;\equiv\; {\uparrow}m \cup {\uparrow}n$
(17)	$m \subseteq n \;\Rightarrow\; s{\downarrow}m \subseteq s{\downarrow}n$	(18)	$m \subseteq n \;\Rightarrow\; s{\uparrow}m \subseteq s{\uparrow}n$

11.2 Restriction

(1)	$m\|s \subseteq m$	(2)	$m \ominus s \subseteq m$
(3)	$s \subseteq \overline{{\downarrow}m} \;\Rightarrow m\|s \equiv \emptyset$	(4)	$s \subseteq \overline{{\downarrow}m} \;\Rightarrow m \ominus s \equiv m$
(5)	$m\|\emptyset \equiv \emptyset$	(6)	$m \ominus \emptyset \equiv m$
(7)	${\downarrow}m \subseteq s \;\Rightarrow m\|s \equiv m$	(8)	${\downarrow}m \subseteq s \;\Rightarrow m \ominus s \equiv \emptyset$
(9)	$m\|{\downarrow}m \equiv m$	(10)	$m \ominus {\downarrow}m \;\equiv \emptyset$
(11)	$(m\|s)\|t \equiv m\|(s \cap t)$	(12)	$(m \ominus s) \ominus t \equiv m \ominus (s \cup t)$
(13)	$m\|(s \cup t) \equiv m\|s \cup m\|t$	(14)	$m \ominus (s \cup t) \equiv m \ominus s \cap m \ominus t$
(15)	$m\|(s \cap t) \equiv m\|s \cap m\|t$	(16)	$m \ominus (s \cap t) \equiv m \ominus s \cup m \ominus t$
(17)	$m\|(s\backslash t) \equiv m\|s \cap m \ominus t$	(18)	$m \ominus (s\backslash t) \equiv m \ominus s \cup m\|t$
(19)	$t{\downarrow}(m\|s) \equiv (t{\downarrow}m) \cap s$	(20)	$t{\downarrow}(m \ominus s) \equiv (t{\downarrow}m)\backslash s$
(21)	${\downarrow}(m\|s) \equiv {\downarrow}m \cap s$	(22)	${\downarrow}(m \ominus s) \equiv {\downarrow}m\backslash s$
(23)	$t{\uparrow}(m\|s) \equiv (t \cap s){\uparrow}m$	(24)	$t{\uparrow}(m \ominus s) \equiv (t\backslash s){\uparrow}m$
(25)	${\uparrow}(m\|s) \equiv s{\uparrow}m$	(26)	${\uparrow}(m \ominus s) \equiv \overline{s}{\uparrow}m$
(27)	$(m \cup n)\|s \equiv m\|s \cup n\|s$	(28)	$(m \cup n) \ominus s \equiv m \ominus s \cup m \ominus t$

11.3 Overwriting

(1) $\emptyset \mathbin{\lhd\!\!-} m \;\equiv\; m \mathbin{\lhd\!\!-} \emptyset \;\equiv\; m$

(2) $(l \mathbin{\lhd\!\!-} m) \mathbin{\lhd\!\!-} n \;\equiv\; l \mathbin{\lhd\!\!-} (m \mathbin{\lhd\!\!-} n)$

(3) $m \mathbin{\lhd\!\!-} n \;\equiv\; n \mathbin{\lhd\!\!-} m$ iff m and n are compatible.
In this case, $m \mathbin{\lhd\!\!-} n \;\equiv\; m \cup n$.

(4) $s{\downarrow}(m \mathbin{\lhd\!\!-} n) \;\equiv\; (s{\downarrow}m\backslash{\downarrow}n) \cup s{\downarrow}n$

(5) ${\downarrow}(m \mathbin{\lhd\!\!-} n) \;\equiv\; {\downarrow}m \cup {\downarrow}n$

(6) $m \mathbin{\lhd\!\!-} n \;\equiv\; n \Leftrightarrow\; {\downarrow}m \subseteq {\downarrow}n$

(7) $m \mathbin{\lhd\!\!-} n \;\equiv\; m \Leftrightarrow\; n \subseteq m$

(8) $(m \mathbin{\lhd\!\!-} n) \ominus s \;\equiv\; (m \ominus ({\downarrow}n \cup s)) \cup n \ominus s \;\equiv\; (m \ominus ({\downarrow}n) \ominus s) \cup n \ominus s$

(9) ${\uparrow}(m \mathbin{\lhd\!\!-} n) \;\equiv\; \overline{{\downarrow}n}{\uparrow}m \;\cup {\uparrow}n$