RELATIONS, GRAPHS AND PROGRAMS

by

Jesús N. Ravelo

Technical Monograph PRG-125
ISBN 0-902928-99-6

April 1999

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building, Parks Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building, Parks Road
Oxford OX1 3QD
England

Relations, Graphs and Programs

Jesús N. Ravelo
Linacre College

Michaelmas Term 1998

A thesis submitted for the degree of
Doctor of Philosophy at the University of Oxford

Abstract

Much emphasis has been placed in recent years on deriving or calculating programs rather than proving them correct. Adequate calculational frameworks are needed to support such an approach. This thesis explores the use of a calculus of binary relations to express and reason about graph-theoretical concepts in the context of program construction. Since graphs play a prominent role in algorithmics and have applications in many other fields, such a calculational treatment of graphs via relations positively benefits the formal program construction field.

Phrasing the basics of graph theory with relations allows a formal compact presentation of well-known facts, as well as the development of novel proofs for such facts in a calculational fashion. Such a machinery is combined with predicate, refinement and fixed-point calculi to derive imperative programs that solve several graph computational problems. Relations are used to model graphs and sets as the data manipulated by programs and specifications. The case-studies put forward in this thesis include some generic problems with instances that correspond to graph algorithms as well as some individual graph problems. These examples demonstrate the applicability of the framework of relations to calculational graph algorithmics, yet some drawbacks are examined. Potential sources of improvement of this presentation and hints on future research are discussed.

A la conspiración del Universo...

Contents

Acknowledgements

First and foremost, I thank my supervisor Richard Bird for advice and support throughout the whole development of this thesis. I am particularly grateful to him for the humaneness, patience and encouragement he offered during the times when emotional support was what I needed most. Thanks also to Oege de Moor, Sharon Curtis, Jeremy Gibbons, Pedro Borges and Alvaro Arenas at the Computing Laboratory for help and encouragement. My examiners, Carroll Morgan and Roland Backhouse, provided valuable suggestions and criticisms about the contents and presentation of the thesis.

Acknowledgement is due to the following institutions for financial sponsorship and/or practical assistance: CONICIT (Consejo Nacional de Investigación Científica y Tecnológica) of Venezuela, Interamerican Development Bank, Universidad Simón Bolívar, the Overseas Research Students Awards Scheme of the United Kingdom, and both the Computing Laboratory and Linacre College at Oxford. The staff at Linacre College added outstanding friendliness and warmth to the practicalities of their support.

Special mention is due to my advisors at Universidad Simón Bolívar, Cristina Zoltan and Roger Soler. Their advice and encouragement were decisive for initiating, and carrying through, this whole adventure. I am also especially indebted to Elsa, Anne and "the Group" at the Oxford University Counselling Services for helping me sail through the hardest times.

I thank my family for their everlasting emotional and practical support. Their love makes life a worthwhile trip. Thanks to Stephen for the love, good times and dreams we shared during the final year of this stage of my life. Last but not least, many friends formed the network of support and fun that always kept me going on. Names are not necessary: *you* know who *you* are, thanks! One name must be made explicit since, without her, I would not be writing this today: Thanks, Martha!

Chapter 1
Introduction

Much attention has been given, over the last two decades or so, to the derivation or calculation of correct programs from their specifications, as opposed to the a posteriori verification of their correctness. Calculational techniques for program construction have thus become an established important area of research in computer science. The prime goal of this research field is the reduction of as many parts as possible of the process of program construction to syntactic manipulation. This requires the use of calculi that suitably combine precision and conciseness: the former provides the much needed assurance that the formulae in manipulation model the features of whatever "reality" we have in mind in a *precise* manner, while the latter helps to express such features in a *concise* manner, thereby avoiding the burden of having to manipulate gigantic formulae.

This thesis explores the use of a calculus of binary relations to express and reason about graph properties in the context of imperative program construction. Relations are used to model graphs and sets, thereby modelling the data manipulated by the programs and specifications. Such a calculational treatment of graphs via relations thus links together the areas of graph algorithmics and formal program construction, which are the main subjects this work touches upon: on the one hand, a large variety of algorithms linked to the well-developed subject of graph theory and, on the other hand, calculational tools for the development of imperative programs. Most of the rest of this introductory chapter is dedicated to surveying briefly the history of these subjects.

Section 1.1 surveys the area of graph algorithmics, while Section 1.2 deals with the calculus of binary relations and its application to reasoning about graph concepts and properties. The use of calculational techniques for the construction of imperative programs, which has evolved into the so-called refinement calculus, a uniform setting for the manipulation of specifications

and programs, is surveyed in Section 1.3. Finally, Section 1.4 presents an outline of the contents of the rest of this thesis.

1.1 Graph Algorithmics

Graph theory is nowadays a prominent tool in the solution of a wide variety of practical problems in many different fields. Much of the theory has indeed been developed motivated by its practical applications. This can be seen in what is recognised as the origin of graph theory, Euler's solution in 1736 of the Königsberg bridges puzzle [51, 52], and further developments such as Kirchhoff's theory of trees in 1847 for the study of electrical networks [86], and Ford and Fulkerson's theory of network flows in 1956 as an application in operations research [56, 57]. Other fields that have benefited from graph-theoretical results include chemistry, biology, economics, geography, architecture, and the social sciences –see e.g. [58]–. In most of the practical situations that arise in these fields, graphs that model a real-life problem must be analysed in some way or other, and such graphs tend to be large and complex. Computer assistance to perform such analyses is then most valuable and, consequently, so is the design of efficient algorithms. A large number of graph algorithms have indeed been developed over the last few decades, and a class of graph problems for which efficient algorithms are not likely to be found has been identified, giving birth to the theory of NP-completeness. Good introductions to graph theory can be found in e.g. [31, 71], and to graph theory and algorithmics in e.g. [61, 67, 121, 145].

Most algorithmics textbooks, e.g. [3, 36, 85], dedicate significant space to the study of graph algorithms. However, their presentations follow, more often than not, a traditional approach to algorithm design. Algorithms are first given and afterwards, if at all, shown correct with respect to their specification. Several well-known graph algorithms, though sometimes compactly presented, have complicated correctness proofs. Authors must take pains to present clear proofs and yet, due to the lack of adequate calculational frameworks to do so, their efforts often result in arguments that are still difficult to read. Calculational frameworks for program construction have proved to be profitable in revealing the core of the design decisions that lead to the development of some algorithm or other. Therefore, tackling the large variety of well-known graph algorithms under a calculational approach seems to be a goal worth pursuing.

Some graph algorithms are instances of general design paradigms. Kruskal's algorithm and Prim's algorithm for the computation of minimum spanning trees [90, 126], and Dijkstra's minimum paths algorithm [43] are examples of the greedy strategy. Others are designed in very specific and peculiar ways,

yet are based on similar structural graph properties. This is the case of a class of algorithms based on depth-first traversals of the input graph, including a version of topological sorting, Kosaraju and Sharir's algorithm and Tarjan's algorithm for the computation of strongly connected components [138, 141], Tarjan's algorithm for the computation of biconnected components [141], and many others. A few of these graph algorithmic problems are dealt with in this thesis using the calculational method. Some of them are treated as instances of more abstract problems while some others are tackled individually.

1.2 The Calculus of Relations

The foundations of the calculus of binary relations has its origins in work by Augustus de Morgan, Charles S. Peirce and Enrst Schröder during the second half of the nineteenth century. During the 1850s, de Morgan started work on a theory of dyadic relations, stating some of the laws that govern the behaviour of such relations [112]. Peirce elaborated upon de Morgan's work, drawing inspiration also from Boole's logical algebra, in a series of papers that started in the 1870s [123, 124]. Schröder further developed Peirce's "Logic of Relatives" in a systematic fashion in the third volume of his "Algebra der Logik", published in 1895 [137].

No particular interest was given to an axiomatic approach to the calculus of relations until Tarski undertook this task. In 1941, Tarski proposed an axiomatisation for a large part of the calculus [142]. This led to the development of relation algebras, devised as models of Tarski's axioms and later used by Lyndon to show the incompleteness of Tarski's axioms with respect to the set-theoretical approach to relations [93]. Among many others, Maddux has continued research on relation algebras; he has also written a detailed overview of the origins of the calculus of relations and relation algebras [94].

Relations have also been studied in the context of category theory. Just as categories were defined as a simple model of the algebra of functions, allegories have emerged as enriched categories to model the algehra of relations. It has been proved that a certain kind of allegories, viz. unitary tabular allegories, is axiomatically very close to set-theoretical relations. In other words, the axioms of a unitary tabular allegory are in some sense, which we will not detail here, complete. A comprehensive study of allegory theory has been written by Freyd and Ščedrov [59]; also, the book [28] by Bird and de Moor offers an introduction to the theory of categories and allegories along with applications to algorithm design and program construction.

Many other researchers have also used the calculus of binary relations for programming theory, e.g. [49, 75, 76, 95, 103, 104]. This is due to the fact

that modelling specifications and programs as relations provides an adequate framework in which to treat non-determinism. We use relations, though in the context of program construction, for a different purpose: to express and reason about graph concepts and properties. A great deal of research into using the calculus of binary relations in the realm of graph theory has already been undertaken, mainly in Germany. The standard reference is the book [136] by Schmidt and Ströhlein, in which a large number of graph concepts are phrased in terms of relations, and proofs of graph properties are carried out in an algebraic fashion. This approach to graphs has also been used within the area of formal program construction to derive programs that manipulate graphs from relational specifications [18, 19, 20, 83]. Some of such specifications can be directly executed using RELVIEW, a programming system for the manipulation of relations [17].

Arguing that a calculus of binary relations is too restricted, Möller has developed a calculus of formal languages and n-ary relations to serve the aims of program construction, in particular, the construction of programs that manipulate graphs [104, 105]. Möller and Russling have shown that within such a calculus many graph problems can be clearly specified and successfully manipulated to obtain algorithmic solutions [107, 131, 133, 134] –see also [33]–. This calculus is indeed well-suited to reason about graphs, whether or not such reasoning is aimed at developing programs.

1.3 The Refinement Calculus

The use of calculational techniques for the construction of imperative programs is nowadays firmly established. It originated in the late 1960s and early 1970s with precedent-setting work by Floyd, Hoare and Wirth [55, 73, 147], among others, and, more prominently, by Dijkstra's introduction of predicate transformers and their use in the derivation of programs [44, 45]. This approach to programming has, since then, been further developed and disseminated in several textbooks, e.g. [7, 47, 68, 82]. The 1980s then saw the emergence of the refinement calculus, a framework in which specifications are put together with executable programming constructs in a uniform formal setting. Its origins go back to Back's [4], being later further developed by Back himself, Morgan and Morris [5, 6, 114, 115, 117, 119]. The refinement calculus comprises, first, an extension to Dijkstra's language of guarded commands [44, 45] with specification statements and, second, a formal refinement relation based on weakest-precondition predicate transformers semantics [48].

In this thesis, we will use the notation of Morgan's refinement calculus [115] for specifying the computational problems we will be dealing with as well as for presenting the derivations of the corresponding programs. Morgan's

specification statement $w : [\, pre\,,\, post\,]$, where w is a list of program variables, and pre and $post$ are propositions, informally means: "If the initial state satisfies precondition pre, then change only the variables listed in w so that the resulting final state satisfies postcondition $post$". The refinement relation is denoted by $\sqsubseteq$. Roughly, $S \sqsubseteq P$ can be interpreted as program P being correct with respect to specification S, and it is read "P refines S" or "S is refined by P". Since specifications and executable programs reside in the same space and can be mixed unrestrictedly, both S and P above might be any combination of programming constructs and specification statements. The general informal interpretation for $S \sqsubseteq P$ is that, whenever S is required, P is good enough to fulfil such a need.

It is worth noting that, in presenting our derivations of programs, we will not make use of detailed laws of the refinement calculus, as in [115, 117]. Rather, we will develop our programs in a style similar to that of the aforementioned textbooks [7, 47, 68, 82], but presenting "formal refinement summaries" using the notation of Morgan's calculus. Such summaries could, however, be meticulously proved valid by means of the detailed refinement laws.

1.4 Outline

Chapter 2, "Relations and Graphs", introduces the calculus of relations, a few additional notions not inherent in relations but often used in conjunction with them –such as lattice-theoretical fixed points in general and some closure operators on relations in particular–, and a formalisation within such a calculational framework of most of the basic graph-theoretical concepts used in subsequent chapters. Much of the contents of this chapter is well-known material produced by previous research, but the formalisation of a few graph-theoretical concepts plus a couple of calculational proofs in it appear to be, to the best of our knowledge, novel. –The final Chapter 8, in page 161, makes explicit mention of what the novel points of Chapter 2 are.–

Chapters 3 to 7 form the core of what the rest of the title of this thesis, "and Programs", refers to. Each chapter takes up either single graph computational problems or a family of them in the form of one generic problem. In each case, the first step that is taken is the formalisation of the corresponding specification. Such specifications serve as a starter for showing the use this thesis puts forward for the calculus of binary relations in the formal construction of graph algorithms: the preconditions and postconditions are formulae of the extended predicate calculus that results from adding relational constructs to it. When refining the specifications to programs, these pre- and postconditions are manipulated via the calculational facilities provided by the combined predicate and relational calculi. Such is the point where the

adequacy of the framework of relations to derivational graph algorithmics is put to the test.

It is worth pointing out that our final programs will handle variables of type, e.g., "set" or "relation" and, in that respect, might still be considered abstract non-executable programs. However, these programs will be such that their further refinement to more concrete ones manipulating sequences, arrays, matrices, or any other particular construct offered by some or other imperative programming language should not be a complex, even if laborious, task.

Each of the five "Programs"-chapters is closed with a section that surveys previous and current work related to its contents. The final Chapter 8 provides some concluding remarks by summarising and assessing the results shown in this thesis. Such a discussion is supported by the aforementioned reviews of related work, in the light of which our results are judged.

Chapter 2
Relations and Graphs

This chapter serves two main purposes: to introduce the calculus of binary relations, and to illustrate the use of such a calculus in modelling graphs as well as in formalising concepts and proving properties related to graphs.

Section 2.1 introduces the basics of the calculus of binary relations and Section 2.2 then presents ways in which sets can be modelled with relations. A brief pass through lattice theory is then made in the following two sections. Section 2.3 presents fixed points and some of their properties, emphasising the calculational nature of the presentation. Section 2.4 presents orderings, equivalences, and a short review of the reflexive-transitive closure and transitive closure operators on relations. We then embark on the task of presenting, in Section 2.5 and Section 2.6, several basic graph-theoretical concepts within the framework of binary relations. Section 2.7 is dedicated to spanning trees and a novel calculational proof of a well-known property of them. Section 2.8 formalises the notion of paths in a graph, which will require a brief review of the treatment of datatypes within the calculus of relations.

Emphasis will be made all along in the use of calculational methods –see e.g. [14]–, not only when using relations but also in the context of lattice theory –as in e.g. [2, Part I] and [99]– and first-order logic –via predicate calculus, see e.g. [48, 69]–.

2.1 Basics of the Calculus of Relations

This section is devoted to presenting the basics of the calculus of binary relations. We will not be concerned with a strictly axiomatic approach but, rather, with using relations in a calculational style. More complete introductions to the calculus of binary relations can be found elsewhere [2, 28, 32].

Category Structure –Composition and Identities– A *relation* R to set X from set Y is a subset of the cartesian product $X \times Y$. In such a case we say that R is of *type* $X \leftarrow Y$ and denote it by $R : X \leftarrow Y$. For $(x, y) \in R$ we write $x \langle R \rangle y$.

Given relations $R : X \leftarrow Y$ and $S : Y \leftarrow Z$, their *composition* $R \cdot S$ is of type $X \leftarrow Z$ and is such that:

$$x \langle R \cdot S \rangle z \quad \equiv \quad \langle \exists y :: x \langle R \rangle y \wedge y \langle S \rangle z \rangle \ .$$

Also, for each set X there is an *identity* relation id_X of type $X \leftarrow X$ such that:

$$x_1 \langle id_X \rangle x_2 \quad \equiv \quad x_1 = x_2 \ .$$

It can be shown that composition is associative:

$$(R \cdot S) \cdot T \quad = \quad R \cdot (S \cdot T) \ , \tag{2.1}$$

and that identity relations act as units of composition:

$$id_X \cdot R \quad = \quad R \quad = \quad R \cdot id_Y \ , \tag{2.2}$$

where R is of type $X \leftarrow Y$. Type information given by the subscripts of the identity relations will usually be clear from context. We will thus omit such subscripts, writing just id instead.

Functions will be introduced later in this section as a particular kind of relation and, accordingly, we will also use the right-to-left arrow to denote their type. We prefer to use such an arrow for two reasons. First, it is consistent with the conventional notation for functional application in which arguments are given to functions on the right: for function $f : X \leftarrow Y$ and y of type Y, fy is of type X. Secoud, the types involved in relational composition and, therefore, also in functional composition take a more natural form: as in the general case of relations above, for functions $f : X \leftarrow Y$ and $g : Y \leftarrow Z$ their composition $f \cdot g$ is of type $X \leftarrow Z$. We will thus think of relations and functions in a rather operational way: as taking arguments, or inputs, on the right and delivering results, or outputs, on the left.

Statements like $x \langle R \rangle y$, with explicit reference to the elements x, y of the sets involved in the types of relations, are said to be written in a *pointwise* or set-theoretical style. On the other hand, statements like (2.1) and (2.2), expressed in terms of composition and with no reference to elements of the sets involved, are said to be written in a *point-free* style. We will prefer to manipulate relations in the point-free style since point-free calculations have proved to be, more often than not, more compact than their pointwise counterparts. However, we will occasionally use the pointwise style when it

aids a more intuitive or perhaps more effective presentation of a concept.

Lattice Structure The elements of lattice theory relevant to the calculus of binary relations can be found both in [2, Part I] and [136, Appendix A]. The former promotes the calculational style we use in this thesis. A complete introduction to lattice theory can be found elsewhere, e.g. [42].

For sets X and Y, we will denote the collection of all relations of type $X \leftarrow Y$ by $\mathsf{Rel}(X,Y)$. This collection forms a complete Boolean lattice, i.e. a complete, completely or infinitively distributive, complemented lattice, $(\mathsf{Rel}(X,Y), \cup, \cap, \bar{}, \emptyset, \Pi)$, where $\cup$ and $\cap$ denote the *union (join)* and *intersection (meet)* operators, $\bar{}$ the *complementation (negation)* operator, and $\emptyset$ and Π are the *empty (bottom)* and *universal (top)* relations.

Inclusion $\subseteq$ of relations is the partial order induced by the lattice structure: $R \subseteq S$ is equivalent to $S = R \cup S$ and also equivalent to $R = R \cap S$. In a pointwise fashion, the inclusion order satisfies:

$$R \subseteq S \;\equiv\; \langle \forall x, y :: x \,\langle R \rangle\, y \Rightarrow x \,\langle S \rangle\, y \rangle \;.$$

Strictly speaking, we should use subscripts related to the type of the operations and relations above: $\cup_{X,Y}$, $\emptyset_{X,Y}$, $\subseteq_{X,Y}$, etc. As it was the case for the identity relations, this information will usually be either clear from context or irrelevant. It will thus be omitted.

The union and intersection operators are characterised by the equivalences that now follow. Given a bag $\mathcal{R}$ of relations and a relation W, we have:

$$\langle \cup R : R \in \mathcal{R} : R \rangle \subseteq W \;\equiv\; \langle \forall R : R \in \mathcal{R} : R \subseteq W \rangle \;, \tag{2.3}$$

$$W \subseteq \langle \cap R : R \in \mathcal{R} : R \rangle \;\equiv\; \langle \forall R : R \in \mathcal{R} : W \subseteq R \rangle \;. \tag{2.4}$$

In particular:

$$R \cup S \subseteq W \;\equiv\; R \subseteq W \;\wedge\; S \subseteq W \;, \tag{2.5}$$

$$W \subseteq R \cap S \;\equiv\; W \subseteq R \;\wedge\; W \subseteq S \;. \tag{2.6}$$

In the point-free style, operators are usually characterised by this kind of equivalences called *universal properties*.

Using the theory of Galois connections –see e.g. [1] or [2, Chapter 5]–, mutual distribution of union and intersection guarantees the existence of a *subtraction* operator $-$ and an *implication* operator $\Rightarrow$ characterised by the following universal properties:

$$R - S \subseteq W \;\equiv\; R \subseteq W \cup S \;, \tag{2.7}$$

$$W \subseteq R \Rightarrow S \;\equiv\; W \cap R \subseteq S \;. \tag{2.8}$$

These operators would exist and be well-defined by the above equations even in a non-complemented lattice. However, in the presence of complementation, which satisfies

$$R \cap S \subseteq T \;\equiv\; R \subseteq T \cup \overline{S}, \tag{2.9}$$

subtraction and implication can be equivalently defined by $R - S := R \cap \overline{S}$ and $R \Rightarrow S := \overline{R} \cup S$. Property (2.9) will be referred to as *shunting*.

In the point-free style, equations are often proved via the so-called rules of *indirect equality*:

$$R = S \;\equiv\; \langle \forall W :: R \subseteq W \equiv S \subseteq W \rangle \;, \tag{2.10}$$
$$R = S \;\equiv\; \langle \forall W :: W \subseteq R \equiv W \subseteq S \rangle \;. \tag{2.11}$$

These rules, combined with universal properties, benefit the construction of calculational proofs. We will illustrate the point by proving the first of the two following distribution properties of subtraction:

$$(R \cup S) - T \;=\; (R - T) \cup (S - T) \;, \tag{2.12}$$
$$R - (S \cup T) \;=\; R - S - T \;. \tag{2.13}$$

Proof of (2.12)*:*

Appealing to indirect equality (2.10) we reason, for any relation W, that:

$$\begin{array}{ll} & (R \cup S) - T \subseteq W \\ \equiv & \quad \{ \text{ universal property of subtraction (2.7) } \} \\ & R \cup S \subseteq W \cup T \\ \equiv & \quad \{ \text{ universal property of union (2.5) } \} \\ & R \subseteq W \cup T \;\wedge\; S \subseteq W \cup T \\ \equiv & \quad \{ \text{ universal property of subtraction (2.7), twice } \} \\ & R - T \subseteq W \;\wedge\; S - T \subseteq W \\ \equiv & \quad \{ \text{ universal property of union (2.5) } \} \\ & (R - T) \cup (S - T) \subseteq W \;. \end{array}$$

□

We have already seen all we need of the lattice structure on its own. Let us now turn to the interaction of the lattice and the category structure. Composition distributes over union, but only weakly distributes over intersection. For a relation R and a bag of relations $\mathcal{S}$, we have:

$$R \cdot \langle \cup S : S \in \mathcal{S} : S \rangle \;=\; \langle \cup S : S \in \mathcal{S} : R \cdot S \rangle \;, \tag{2.14}$$

$$R \cdot (\cap S : S \in \mathcal{S} : S) \subseteq (\cap S : S \in \mathcal{S} : R \cdot S) \ . \tag{2.15}$$

Analogous rules hold for right-composition. It follows that the empty relation is a zero of composition and that composition is monotonic with respect to inclusion.

It is assumed that composition binds more tightly than union and intersection. Therefore, the above laws for binary union and intersection can be written thus:

$$\begin{aligned} R \cdot (S \cup T) &= R \cdot S \cup R \cdot T \ , \\ R \cdot (S \cap T) &\subseteq R \cdot S \cap R \cdot T \ , \end{aligned}$$

where, e.g., $R \cdot S \cup R \cdot T$ must be read as $(R \cdot S) \cup (R \cdot T)$.

Converse The last basic relational operator is the converse operator. For a relation $R : X \leftarrow Y$, its converse R° is of type $Y \leftarrow X$ and satisfies the following:

$$y \, (R^\circ) \, x \equiv x \, (R) \, y \ .$$

The converse operator is its own inverse, and it interacts with the category structure by preserving identities and distributing contravariantly over composition:

$$R^{\circ\circ} = R, \quad id^\circ = id, \quad (R \cdot S)^\circ = S^\circ \cdot R^\circ \ . \tag{2.16}$$

Converse distributes over all the operators –constants regarded as nullary operators– of the lattice structure:

$$\left.\begin{aligned} &(\cup R : R \in \mathcal{R} : R)^\circ = (\cup R : R \in \mathcal{R} : R^\circ) \ , \\ &(\cap R : R \in \mathcal{R} : R)^\circ = (\cap R : R \in \mathcal{R} : R^\circ) \ , \\ &(\overline{S})^\circ = \overline{(S^\circ)} \ , \quad \emptyset^\circ = \emptyset \ , \quad \Pi^\circ = \Pi \ ; \end{aligned}\right\} \tag{2.17}$$

and it preserves the inclusion order:

$$R \subseteq S \equiv R^\circ \subseteq S^\circ \ . \tag{2.18}$$

Any use of (2.16), (2.17) and (2.18) in what follows will be indicated as "converse".

The category structure, the lattice structure and converse interact all together through Dedekind's rule and Schröder's rules. *Dedekind's rule*, also known as the *modular law*, comes in a left- and a right- version which read as follows:

$$R \cdot S \cap T \subseteq R \cdot (S \cap R^\circ \cdot T) \ , \tag{2.19}$$

$$R \cdot S \cap T \subseteq (R \cap T \cdot S^\circ) \cdot S \ . \tag{2.20}$$

Schröder's rules, also called the *left-exchange* rule and the *right-exchange* rule, are:

$$R \cdot S \subseteq T \equiv \overline{T} \cdot S^\circ \subseteq \overline{R} \ , \tag{2.21}$$

$$R \cdot S \subseteq T \equiv R^\circ \cdot \overline{T} \subseteq \overline{S} \ . \tag{2.22}$$

The conjunction of the left-exchange rule and the right-exchange rule is equivalent to the –arguably more compact and more easily memorised– *middle-exchange* rule:

$$R \cdot S \cdot T \subseteq U \equiv R^\circ \cdot \overline{U} \cdot T^\circ \subseteq \overline{S} \ .$$

This rule is due to Jaap van der Woude.

Functions A relation R is said to be *entire* if $id \subseteq R^\circ \cdot R$, and it is said to be *simple* if $R \cdot R^\circ \subseteq id$. Recall the operational interpretation of relations associated with their type-arrows: relations take inputs on the right and deliver outputs on the left. Under this interpretation an entire relation is such that delivers at least one output for each possible input, while a simple relation delivers at most one output for each input. A *(total) function* is an entire and simple relation. The collection of all functions to X from Y will be denoted by $\mathsf{Fun}(X, Y)$.

Useful properties of functions include the fact that "functionhood" is preserved by composition and the so-called *shunting* rules, which now follow. For every function f and every pair of relations R and S:

$$f \cdot R \subseteq S \equiv R \subseteq f^\circ \cdot S \ , \tag{2.23}$$

$$R \cdot f^\circ \subseteq S \equiv R \subseteq S \cdot f \ . \tag{2.24}$$

The shunting rules can be used to show that equality of functions is equivalent to inclusion, i.e. for every pair of functions f and g:

$$f = g \equiv f \subseteq g \ . \tag{2.25}$$

Extensionality Recall the discussion in page 8 about the pointwise and point-free styles of manipulating relations. The pointwise style was also referred to as the set-theoretical style, while the point-free style is that advocated by an axiomatic approach to relations like Tarski's or the categorical/allegorical one. These two approaches can be unified by including a notion of "points" or "elements" into the point-free style. To introduce such a notion we need the so-called *unit type*, denoted by $\mathbf{1}$ and which represents

a singleton set. We choose to denote the unique element of 1 by $*$. Now, given a set X, a *point* or *element* of X is defined to be a function to X from 1, i.e. a member of the collection $\mathsf{Fun}(X,1)$. Function $\{(x,*)\}$ in $\mathsf{Fun}(X,1)$ represents the element x of X.

An important feature of biuary relations, being defined as subsets of cartesian products in set theory, is that of being built up from pairs of elements. These pairs of elements are, in terms of lattice theory, *atoms*: immediate successors of bottom. And a lattice is said to be *atomic* if every lattice member can be built up from atoms. For every pair of sets X and Y, the lattice $\mathsf{Rel}(X,Y)$ is indeed atomic, with atoms of the form $x \cdot y^\circ$ for x a point of X and y a point of Y. This comes down to the fact that, for every $R : X \leftarrow Y$, we have:

$$R = \langle \cup x,y : x \cdot y^\circ \subseteq R : x \cdot y^\circ \rangle \ , \tag{2.26}$$

$$R \neq \emptyset \equiv \langle \exists x,y :: x \cdot y^\circ \subseteq R \rangle \ , \tag{2.27}$$

where x and y are dummies rauging over points of, respectively, X and Y. -For axiomatic treatments of extensionality within the calculus of binary relations, see e.g. [130, 135]. For a graphical representation of the calculus of relations with points, see [41].-

Note that $x \cdot y^\circ \subseteq R$ corresponds to the pointwise statement $x \,\langle R \rangle\, y$. Also note that, by shunting of functions (2.24) and properties of converse -(2.16), (2.18)-, it can be expressed in several equivalent ways:

$$x \cdot y^\circ \subseteq R \equiv x \subseteq R \cdot y \equiv y \cdot x^\circ \subseteq R^\circ \equiv y \subseteq R^\circ \cdot x \ . \tag{2.28}$$

In particular, the expression $x \subseteq R \cdot y$ can be read, uuder the right-to-left operational interpretation of relations, as "x is a possible output of R for input y".

We will write $x : X$, and will say that x is of type X, to indicate that x is a point of X. Also, lower-case will be conventionally used to name points.

2.2 Sets within the Calculus of Relations

There are several ways in which the notion of subsets can be incorporated into the calculus of biuary relations. Two of them identify sets with certain relations, allowing us to mix sets and relations and, therefore, allowing us to reason about sets as a particular instance of reasoning about relations. A third allows powersets to be used within the typing of relations, and exploits the isomorphism between binary relations and set-valued functions.

Vectors The first view of sets as relations makes use of the unit type 1, the singleton set $\{*\}$, and it is closely related to the way points or elements are modelled as functions. Relations of type $X \leftarrow 1$ can be put in a one-to-one correspondence with subsets of X by identifying relation $A : X \leftarrow 1$ with set $\{ x \mid x \langle A \rangle * \}$. Furthermore, the lattice structure of $\mathbf{Rel}(X, 1)$ is order-isomorphic to the power-lattice of subsets of X. Following [136], we call these relations *vectors*, for their representation as $n \times 1$ boolean matrices if X is a set of size n. Others call them *left-conditions* [2, 49, 130], though their treatment, as well as that of [136], does not make use of 1 and is thus sligthly different. We will write $A : \mathrm{Vec}\, X$ to indicate that A is a relation of type $X \leftarrow 1$.

The universal relation Π of type $X \leftarrow 1$ models the whole set X. Accordingly, we will use X to denote such a relation. Context will always clearly determine whether X denotes the set X, as in the type of a relation, or the universal relation Π of appropriate type.

Since elements of a set X were represented as functions of type $X \leftarrow 1$, there is a natural embedding of elements of X into subsets of X, viz. the embedding of the function space of a certain type into the relation space of the same type, which corresponds to the formation of singletons.

Extensionality properties (2.26) and (2.27) can be applied to vectors, noting that the only function of type $1 \leftarrow 1$, i.e. the only point of 1, is id. Hence, for a vector A over X the following holds:

$$A = \langle \cup x : x \subseteq A : x \rangle , \tag{2.29}$$

$$A \neq \emptyset \equiv \langle \exists x :: x \subseteq A \rangle , \tag{2.30}$$

where dummy x ranges over points of X. Somewhat trivial, (2.29) can be read as "a set is the union of all its elements".

We will later use the following property of vectors. If x and y are points, and A and B are vectors, then:

$$x \cdot y^\circ \subseteq A \cdot B^\circ \equiv x \subseteq A \ \wedge \ y \subseteq B \ . \tag{2.31}$$

Coreflexives A second way of modelling subsets as relations uses so-called coreflexive relations. A relation C of type $X \leftarrow X$ is *coreflexive* if $C \subseteq id$. A coreflexive relation $C : X \leftarrow X$ can be interpreted as representing subset $\{ x \mid x \langle C \rangle x \}$ of X. Under this interpretation, relational union, relational intersection and the empty relation correspond to their respective counterparts in the power-lattice of subsets of X. To obtain an order-isomorphism between coreflexives and the power-lattice of subsets of X, complementation

has to be defined as $\tilde{C} := id - C$ and the top element as id. We will write $C : \mathrm{Cor}\, X$ to indicate that C is a coreflexive of type $X \leftarrow X$.

Coreflexive relations enjoy many properties that do not hold for general relations. For instance, every coreflexive C duplicates and is its own converse, and its composition to another coreflexive D equals their intersection:

$$C = C \cdot C\,, \quad C^\circ = C\,, \quad C \cdot D = C \cap D\,. \tag{2.32}$$

In the sequel, we will also make use of the following property:

$$R \cdot C \subseteq S \;\equiv\; R \cdot C \subseteq S \cdot C\,. \tag{2.33}$$

Isomorphism between Vectors and Coreflexives Both the representation of sets as vectors and the representation of sets as coreflexive relations are useful. It depends on their intended use which is more convenient.

For example, the existential image under a relation $R : X \leftarrow Y$ can be more readily expressed using vectors. Given a set $A : \mathrm{Vec}\, Y$, its existential image under R is just $R \cdot A : \mathrm{Vec}\, X$. Using coreflexives, the image of $A : \mathrm{Cor}\, Y$ under R would be $id \cap R \cdot A \cdot \Pi : \mathrm{Cor}\, X$.

On the other hand, given relations $R : X \leftarrow Y$ and $S : Y \leftarrow Z$, to restrict the "middle points" in composition $R \cdot S$ to a certain subset of Y, the corresponding coreflexive $C : \mathrm{Cor}\, Y$ can be just plugged in by means of composition: $R \cdot C \cdot S$. If we choose to use a vector $C : \mathrm{Vec}\, Y$, we need a more elaborate expression, like $R \cdot (id \cap C \cdot \Pi) \cdot S$ or $R \cdot (id \cap C \cdot C^\circ) \cdot S$.

However, there are occasions in which we have to choose one of the two representations and have to use it in both kinds of context, i.e. contexts in which vectors would be more appropriate as well as contexts in which coreflexives would be the best choice. It is then very useful to have means of transforming one representation into the other and vice versa. In fact, an isomorphism between the two representations is witnessed by the operator $\natural$, defined as follows:

$$\natural R \;:=\; id \cap R \cdot R^\circ\,. \tag{2.34}$$

This operator, when given a relation $R : X \leftarrow Y$, produces a coreflexive $\natural R : \mathrm{Cor}\, X$ that corresponds to the so-called range or left-domain of R. But $\natural$ has an inverse only when restricted to vectors. Such an inverse is given by what would be a general vector-range operator: given relation R typed as above, its vector-range is $R \cdot \Pi : \mathrm{Vec}\, X$. The restriction of this last operation to coreflexives is the inverse of the restriction of $\natural$ to vectors.

Formally, for every $A : \mathsf{Vec}\, X$ and every $C : \mathsf{Cor}\, X$, the following holds:

$$A = C \cdot X \quad \equiv \quad \mathbin{\mathfrak{c}} A = C \ . \tag{2.35}$$

–Recall that we have overloaded name X to also denote the universal relation of $\mathsf{Rel}(X, 1)$.– From (2.35), taking the instances $A, C := \emptyset, \emptyset$ and $A, C := X, id$, we obtain the expected connections between the bottom and top elements of the lattice of subsets of X in each representation:

$$\mathbin{\mathfrak{c}} \emptyset = \emptyset \ , \quad \mathbin{\mathfrak{c}} X = id \ . \tag{2.36}$$

Aware of the fact that this decision is somewhat arbitrary, we will favour vectors over coreflexives. This is due to the fact that, for a couple of reasons, expressions over vectors often look more natural. First, the lattice operators of $\mathsf{Rel}(X, 1)$ are in one-to-one correspondence with the –conventionally denoted by the same symbols– operators of the standard power-lattice of subsets of X. Second, the natural embedding of points as singleton sets into vectors facilitates a more natural phrasing of membership. –The reader might be readier to accept this last claim after looking at the definition of a "non-deterministic selection" statement presented later in this section.–

Having made the decision of favouring vectors, our use of coreflexives will always involve operator $\mathfrak{c}$ as well as some properties of it. Such properties are used as rules that aid the translation between expressions involving vectors and expression involving coreflexives. We now list them. For every pair of vectors A and B and every relation R:

$$A \subseteq B \quad \equiv \quad \mathbin{\mathfrak{c}} A \subseteq \mathbin{\mathfrak{c}} B \ , \tag{2.37}$$

$$\mathbin{\mathfrak{c}}(A \cup B) \quad = \quad \mathbin{\mathfrak{c}} A \cup \mathbin{\mathfrak{c}} B \ , \tag{2.38}$$

$$\mathbin{\mathfrak{c}}(A \cap B) \quad = \quad \mathbin{\mathfrak{c}} A \cap \mathbin{\mathfrak{c}} B \ , \tag{2.39}$$

$$R \cdot A \subseteq B \quad \equiv \quad R \cdot \mathbin{\mathfrak{c}} A \subseteq \mathbin{\mathfrak{c}} B \cdot R \ . \tag{2.40}$$

Note that in the expressions above $\mathfrak{c}$ has been given a higher precedence than those of union, intersection and composition. We will later make use of the more general fact that $\mathfrak{c}$ distributes over *arbitrary* unions –from which (2.38) follows as a particular case–, and we thus state this property explicitly. For every bag $\mathcal{A}$ of vectors, the following holds:

$$\mathbin{\mathfrak{c}}\langle \cup A : A \in \mathcal{A} : A \rangle \quad = \quad \langle \cup A : A \in \mathcal{A} : \mathbin{\mathfrak{c}} A \rangle \ . \tag{2.41}$$

To finalise, we present the coreflexive representation of singleton sets. Since points are simple relations, for every point a we have:

$$\mathbin{\mathfrak{c}} a \quad = \quad a \cdot a^{\circ} \ . \tag{2.42}$$

Non-Deterministic Selection Statement When programming at a level of abstraction that makes use of sets, picking out an arbitrary element from a given non-empty set is often needed. Formally, given $A : \mathrm{Vec}\, X$ such that $A \neq \emptyset$, a point $a : X$ such that $a \subseteq A$ is required. Using Morgan's specification statement –see Section 1.3–, the *non-deterministic selection statement* is denoted by $a :\subseteq A$ and defined as $a : [\, A \neq \emptyset \,,\, a \subseteq A \,]$, where a and A must be program variables typed as above. This statement is often called *generalised assignment* [118] and denoted, in set-theoretical contexts, using the symbol $:\in$.

Maximal and Minimal Sets We will later make use of the notion of maximal and minimal sets, represented as vectors, satisfying a certain given property.

Let P he a predicate on subsets of a given universal set X. Then a subset A of X is a *maximal* set satisfying P if for uo superset of A, except itself, P holds. The definition of *minimal* sets follows from duality. Formally, with B a dummy ranging over subsets of X, we define:

$$mxl\,(P, A) \;:=\; \langle \forall B : A \subseteq B : P\,B \equiv B = A \rangle \;, \tag{2.43}$$

$$mnl\,(P, A) \;:=\; \langle \forall B : B \subseteq A : P\,B \equiv B = A \rangle \;. \tag{2.44}$$

For certain predicates the above formalisations can be simplified. For example, a predicate P on sets is said to be *subset-closed* if

$$A1 \subseteq A2 \;\Rightarrow\; (P\,A2 \Rightarrow P\,A1) \;. \tag{2.45}$$

In such a case, to see whether A is a maximal P-set not all proper supersets of A must be checked for non-satisfaction of P. It suffices to check its "frontier supersets", i.e. the supersets of A that differ from it only by one element. Formally, with dummy a ranging over elements of X:

$$\left.\begin{array}{l} mxl\,(P, A) \;\equiv\; P\,A \;\wedge\; \langle \forall a : a \subseteq \overline{A} : \neg P\,(A \cup a) \rangle \\ \qquad\qquad\qquad\qquad \text{provided } P \text{ is subset-closed.} \end{array}\right\} \tag{2.46}$$

Dually, a predicate P is *superset-closed* if

$$A1 \subseteq A2 \;\Rightarrow\; (P\,A1 \Rightarrow P\,A2) \;. \tag{2.47}$$

And the following holds:

$$\left.\begin{array}{l} mnl\,(P, A) \;\equiv\; P\,A \;\wedge\; \langle \forall a : a \subseteq A : \neg P\,(A - a) \rangle \\ \qquad\qquad\qquad\qquad \text{provided } P \text{ is superset-closed.} \end{array}\right\} \tag{2.48}$$

It is worth noting that we could have formalised the more general notion of maximal and minimal relations, the above then following as a particular instance. We chose not to do so since we would have found no use for it.

Power Transpose It was briefly mentioned at the beginning of this section that there is a third way of incorporating the notion of subsets into the calculus of binary relations. Unlike vectors and coreflexives, representations that embed the corresponding powersets –or indeed power-lattices– within the collections $\mathsf{Rel}(\_,\_)$, the notion of powerset will now be included at the level of the typing of relations. In category-theoretical terms: powersets as objects rather than as collections of arrows.

We start with a pointwise set-theoretical presentation. There is a one-to-one correspondence between binary relations and set-valued functions. It associates each relation $R : X \leftarrow Y$ to a function $\Lambda R : \mathsf{P}X \leftarrow Y$, where $\mathsf{P}X$ is the powerset of X and ΛR is defined as $(\Lambda R)\, y := \{\, x \mid x\, (R)\, y \,\}$.

A formalisation in the point-free style now follows. It is first assumed that for every set X –a set as used in the types of relations and *not* in the sense of vectors or coreflexives– there exists a set $\mathsf{P}X$, called the *powerset* of X, and a relation $\in : X \leftarrow \mathsf{P}X$, called the *membership* relation on X. Also, it is assumed that for every relation $R : X \leftarrow Y$ there exists a function $\Lambda R : \mathsf{P}X \leftarrow Y$, called the *power transpose* of R, such that:

$$f = \Lambda R \quad \equiv \quad \in \cdot f = R \ , \tag{2.49}$$

for every function $f : \mathsf{P}X \leftarrow Y$. The isomorphism between binary relations and set-valued functions is witnessed by the power-transpose operator Λ. –The category-theoretical definition of *power allegories*, roughly categories of relations with power transpose, can be found in e.g. [28, Section 4.6].– We will assume that operator Λ binds more tightly than composition and all lattice operators.

Two consequences of (2.49) are the so-called *cancellation* and *fusion* properties of power transpose. Explicitly spelled out, they read as follows:

$$\in \cdot \Lambda R \ = \ R \ , \tag{2.50}$$

$$\Lambda R \cdot f \ = \ \Lambda(R \cdot f) \ , \tag{2.51}$$

where R is an arbitrary relation and f is a function.

We will not make as much use of power transpose as of vectors or coreflexives. Indeed, its only use has to do with a rather clean way of modelling quotient sets of equivalence relations within the calculus, presented in Section 2.4 and used in Chapter 7.

Power Transpose in relation to Vectors and Coreflexives First and foremost, we define a *power-point* or *power-element* of X to be a point of $\mathsf{P}X$. The collection of power-points of any X is, on account of the power-transpose isomorphism (2.49), isomorphic to the collection of vectors on X. For every $a : \mathsf{P}X$ and every $A : \mathsf{Vec}\,X$, we have:

$$a = \Lambda A \quad \equiv \quad \in \cdot a = A \ . \tag{2.52}$$

To finally bring this section to an end, we present one single property involving all three representations of sets, which will be used in the sequel. For arbitrary relations R and S, and vector A, the following holds:

$$\Lambda R \cdot A = \Lambda S \cdot A \quad \Leftarrow \quad R \cdot \mathbin{¢} A = S \cdot \mathbin{¢} A \ . \tag{2.53}$$

2.3 Fixed Points

It was pointed out in Section 2.1 that relations form complete lattices, viz. $\mathsf{Rel}(X, Y)$ for each pair of sets X and Y. Hence, the theorem of Knaster and Tarski on the existence of extreme fixed points [88, 143] can be applied. This theorem states that for every monotonic endofunction $\mathcal{F}$ on a complete lattice the equation $\mathcal{F}W = W$ has both a least and a greatest solution on W. Any solution of the equation above is called a *fixed point* of $\mathcal{F}$, while solutions of $\mathcal{F}W \subseteq W$ are called *prefix points* of $\mathcal{F}$ and solutions of $\mathcal{F}W \supseteq W$ are called *postfix points* of $\mathcal{F}$. The Knaster-Tarski theorem further states that the least fixed point of $\mathcal{F}$ coincides with its least prefix point and, dually, that its greatest fixed point coincides with its greatest postfix point.

Thus, the least fixed point of monotonic function $\mathcal{F}$ exists and is uniquely determined by stating: first, that it is a fixed point of $\mathcal{F}$ and, second, that it is least among the prefix points of $\mathcal{F}$ –a more useful statement than just saying it is least among the *fixed* points of $\mathcal{F}$–. We will denote the least fixed point of $\mathcal{F}$ by $\langle \mu W : \mathcal{F}W \rangle$, which is characterised by:

$$\langle \mu W : \mathcal{F}W \rangle \ = \ \mathcal{F}\langle \mu W : \mathcal{F}W \rangle \ , \tag{2.54}$$

$$\langle \mu W : \mathcal{F}W \rangle \ \subseteq \ V \quad \Leftarrow \quad \mathcal{F}V \subseteq V \ . \tag{2.55}$$

In a proof presented in Appendix A.1, we will make use of these characterising properties of least fixed points. Following [99], we will refer to (2.54) as "fixed-point computation" and to (2.55) as "fixed-point induction". A similar characterisation of the greatest fixed point of $\mathcal{F}$ can be obtained by dualisation, but we will have no use for greatest fixed points.

Fixed points can be nicely used in a calculational style [99]. We will later make use of some fixed-point rules that now follow. First, monotonicity of the μ operator:

$$\langle \mu W : \mathcal{F}\, W \rangle \subseteq \langle \mu W : \mathcal{G}\, W \rangle \;\Leftarrow\; \langle \forall W :: \mathcal{F}\, W \subseteq \mathcal{G}\, W \rangle \;. \quad (2.56)$$

Second, the *rolling* rule:

$$\langle \mu W : \mathcal{F}\,(\mathcal{G}\, W) \rangle \;=\; \mathcal{F}\,\langle \mu W : \mathcal{G}\,(\mathcal{F}\, W) \rangle \;. \quad (2.57)$$

Third, the *fixed-point exchange* rule, which has two variants. We spell out the first:

$$\left.\begin{array}{l} \langle \mu W : \mathcal{F}\,(\mathcal{G}\, W) \rangle \;=\; \langle \mu W : \mathcal{F}\,(\mathcal{H}\, W) \rangle \\ \qquad \Leftarrow \quad \langle \forall W :: \mathcal{G}\,(\mathcal{F}\,(\mathcal{H}\, W)) \;=\; \mathcal{H}\,(\mathcal{F}\,(\mathcal{G}\, W)) \rangle \;, \end{array}\right\} \quad (2.58)$$

provided functions $\mathcal{G}$ and $\mathcal{H}$ distribute over union. The second variant is obtained by replacing equality $=$ with inclusion $\subseteq$, and it only requires that function $\mathcal{G}$ distributes over union.

2.4 Orderings, Equivalences and Closure

The notion of order is pervasive in mathematics and computer science –and is present also in the social sciences–. Properties of orderings and equivalence relations –as a particular kind of ordering– can be concisely expressed in the calculus of binary relations. Some of such characterisations of orderings and equivalence relations are presented in this section. Besides, we present well-known mechanisms that produce specific kinds of orderings from a given relation, viz. the reflexive-transitive closure and the transitive closure.

Orderings A relation $R : X \leftarrow X$ is said to be *reflexive* if $id \subseteq R$, and it is said to be *transitive* if $R \cdot R \subseteq R$. A relation which is both reflexive and transitive is a *preorder*. It can be easily shown, using properties of converse, that a relation R is a preorder if and only if R° is a preorder. It can also be shown that if R and S are preorders, then so is their intersection $R \cap S$. Preorder R is *connected* if every two elements of X are related either by R or by R°, i.e. if $\Pi \subseteq R \cup R^\circ$.

Among well-known graph algorithms are a good many that solve optimisation problems, e.g. the computation of shortest paths and the computation of minimum spanning trees. We will thus need a formalisation of the notion of maximum and minimum elements in later chapters. Let $R : X \leftarrow X$ be a preorder and take a set $A : \mathsf{Vec}\, X$. An *R-maximum* of A is an element of A that is at least as good, according to R, as every other element of A.

An *R-minimum* element is an R°-maximum element. Formally, we define the set of R-maxima and the set of R-minima of A as follows:

$$max\,(R,A) \;:=\; \langle \cup x \,:\, x \subseteq A \;\wedge\; A \cdot x^\circ \subseteq R \,:\, x \rangle \;, \tag{2.59}$$

$$min\,(R,A) \;:=\; \langle \cup x \,:\, x \subseteq A \;\wedge\; x \cdot A^\circ \subseteq R \,:\, x \rangle \;, \tag{2.60}$$

where x is a dummy ranging over points of X. Maximum and minimum elements might not exist for certain combinations of R and A. However, our use of *max* and *min* will be limited to instances where R is a connected preorder and A is a finite non-empty set. In such instances, the sets $max\,(R,A)$ and $min\,(R,A)$ are non-empty.

Equivalence Relations A preorder $Q : X \leftarrow X$ is an *equivalence relation* if it is also symmetric. Relation Q is *symmetric* if $Q^\circ = Q$, which is equivalent to $Q^\circ \subseteq Q$. By properties of converse and the aforementioned fact that both the converse and the intersection of preorders are also preorders, every preorder R gives rise to an equivalence relation: $R \cap R^\circ$.

Let $Q : X \leftarrow X$ be an equivalence relation. The *Q-equivalence class* of an element $x : X$ is the set of all the elements related to x by Q. We can model it by the vector $Q \cdot x : \mathsf{Vec}\, X$ and, on account of the power-transpose isomorphism (2.52) and fusion law (2.51), also by the power-element $\Lambda Q \cdot x : \mathsf{P} X$. The *quotient set* of Q is the partition induced by Q on X that comprises all the Q-equivalence classes. We model it by the powerset vector $\Lambda Q \cdot X : \mathsf{Vec}\,(\mathsf{P} X)$ –recall, once more, that we have overloaded name X to also denote the universal relation of $\mathsf{Rel}(X,1)$–. Note that the quotient set of Q is a set of sets and we have chosen to model such a family by a vector on the powerset of X, i.e. the "outer-level" set is modelled by a vector whereas the "inner-level" sets are modelled by powerset objects. Such representation of quotient sets will be used in Chapter 7, when dealing with the problem of computing the strongly connected components of a directed graph.

In the same manner that X is partitioned by Q to form the quotient set of Q, subsets of X can also be partitioned. The *Q-quotient* of a set $A : \mathsf{Vec}\, X$ is the partition that comprises the Q-equivalence classes of all the elements in A, i.e. $\Lambda Q \cdot A : \mathsf{Vec}\,(\mathsf{P} X)$. Note the subtlety: the *quotient set* of Q is the Q-*quotient* of the whole set X. The set of all the elements in such Q-quotients contains all elements in the source set and probably more. Let us formalise this. First, the whole set of elements in a partition of type $\mathsf{Vec}\,(\mathsf{P} X)$ is obtained by composing the membership relation on its left, i.e. by applying $(\in \cdot)$ to get a vector $\mathsf{Vec}\, X$. Now, we can show in one line that the set of elements in the Q-quotient of A always contains A:

$$\in \cdot \Lambda Q \cdot A \;=\; Q \cdot A \;\supseteq\; A \;.$$

The two steps are justified, respectively, by power-transpose cancellation (2.50) and reflexivity of equivalence relations. The other inclusion does not hold in general, but the above one-line proof also shows that the set of elements in the Q-quotient of A is equal to A if and only if $Q \cdot A \subseteq A$ holds. In such a case, we will say that A *fits* Q.

Finally, a couple of facts we will make use of in Chapter 7. First, every power-element in a Q-quotient is the Q-equivalence class of some element and, consequently, corresponds to a non-empty set:

$$
\begin{array}{ll}
 & \Lambda B \subseteq \Lambda Q \cdot A \\
\equiv & \quad \{ \text{ extensionality (2.29) } \} \\
 & \Lambda B \subseteq \Lambda Q \cdot \langle \cup a : a \subseteq A : a \rangle \\
\equiv & \quad \left\{ \begin{array}{l} \text{distribution of composition over union (2.14),} \\ \text{power-transpose fusion (2.51)} \end{array} \right\} \\
 & \Lambda B \subseteq \langle \cup a : a \subseteq A : \Lambda(Q \cdot a) \rangle \\
\equiv & \quad \{ \ \Lambda B \text{ is an atom and, hence, it is irreducible } \} \\
 & \langle \exists a : a \subseteq A : \Lambda B \subseteq \Lambda(Q \cdot a) \rangle \\
\equiv & \quad \left\{ \begin{array}{l} \text{inclusion/equality of functions (2.25),} \\ \text{operator } \Lambda \text{ is an isomorphism} \end{array} \right\} \\
 & \langle \exists a : a \subseteq A : B = Q \cdot a \rangle \\
\Rightarrow & \quad \{ \text{ reflexivity of } Q \text{, predicate calculus } \} \\
 & \langle \exists a :: B \supseteq a \rangle \\
\equiv & \quad \{ \text{ extensionality (2.30) } \} \\
 & B \neq \emptyset \ .
\end{array}
$$

Furthermore, if A fits Q, every power-element in the Q-quotient of A corresponds to a subset of A. We reason in one line thus:

$$\Lambda B \subseteq \Lambda Q \cdot A \ \Rightarrow \ \in \cdot \Lambda B \subseteq \in \cdot \Lambda Q \cdot A \ \Rightarrow \ B \subseteq A \ ,$$

where the first implication is justified by Leibniz, and power-transpose cancellation (2.50), twice, plus the assumption that A fits Q justify the second implication. For future reference, we label these facts:

$$\Lambda B \subseteq \Lambda Q \cdot A \ \Rightarrow \ B \neq \emptyset \ \wedge \ B \subseteq A \quad \text{provided } A \text{ fits } Q. \quad (2.61)$$

Reflexive-Transitive Closure For every relation $R : X \leftarrow X$ there is a smallest preorder that contains R, called the *reflexive-transitive closure* of R and denoted by R^*. The universal property that characterises R^* follows:

for every preorder W,

$$R \subseteq W \quad \equiv \quad R^* \subseteq W \ . \tag{2.62}$$

There are other equivalent characterisations of R^* in terms of least fixed points. The first one arises rather naturally: R^* must be a relation W such that it is reflexive $-id \subseteq W-$, it is transitive $-W \cdot W \subseteq W-$, it contains R $-R \subseteq W-$, and it is the least relation satisfying such requirements. By universal property of union (2.3), R^* must then be the least relation W satisfying $id \cup R \cup W \cdot W \subseteq W$ and, by Knaster-Tarski, such a least prefix point coincides with the corresponding least fixed point. Hence,

$$R^* \quad = \quad \langle \mu W : id \cup R \cup W \cdot W \rangle \ .$$

The other μ-characterisations of R^* are:

$$\begin{array}{lcl} R^* & = & \langle \mu W : id \cup R \cdot W \rangle \ , \\ R^* & = & \langle \mu W : id \cup W \cdot R \rangle \ , \end{array}$$

which can be generalised as follows:

$$R^* \cdot S \quad = \quad \langle \mu W : S \cup R \cdot W \rangle \ , \tag{2.63}$$

$$S \cdot R^* \quad = \quad \langle \mu W : S \cup W \cdot R \rangle \ . \tag{2.64}$$

Yet another characterisation of R^*, in terms of pointwise statements, is the following: $x \langle R^* \rangle y$ is equivalent to the existence of a natural number n such that there exist elements $x_n, x_{n-1}, \ldots x_1, x_0$ that satisfy $x = x_n$, $\langle \forall i : n \geq i > 0 : x_i \langle R \rangle x_{i-1} \rangle$ and $x_0 = y$. This is a particular instance of a more general and well-known theorem due to Kleene [87], which we choose to omit. We will later use this characterisation only as an aid to intuition.

From the above μ-characterisations of closure and monotonicity of μ (2.56), we obtain the fact that the closure operator is monotonic: $R^* \subseteq S^*$ follows from $R \subseteq S$. From the universal property of closure and properties of converse, it can be shown that closure commutes with the converse operator: $R^{*\circ} = R^{\circ *}$. And from this it follows that R^* is an equivalence relation if R is a symmetric relation.

Closure can be "jumped over". The following three propositions, which are collectively called the *leap-frog over closure* rules, formalise this:

$$R^* \cdot S \ = \ S \cdot T^* \quad \Leftarrow \quad R \cdot S \ = \ S \cdot T \ , \tag{2.65}$$

$$R^* \cdot S \ \subseteq \ S \cdot T^* \quad \Leftarrow \quad R \cdot S \ \subseteq \ S \cdot T \ , \tag{2.66}$$

$$R^* \cdot S \ \supseteq \ S \cdot T^* \quad \Leftarrow \quad R \cdot S \ \supseteq \ S \cdot T \ . \tag{2.67}$$

To serve as an example of calculations with fixed-point rules we prove the first variant.

Proof of (2.65)*:*

$$
\begin{array}{ll}
 & R^* \cdot S \;=\; S \cdot T^* \\
\equiv & \quad \{ \text{ closure } -(2.63),\ (2.64)- \ \} \\
 & \langle \mu W : S \cup R \cdot W \rangle \;=\; \langle \mu W : S \cup W \cdot T \rangle \\
\Leftarrow & \quad \left\{ \begin{array}{l} \text{fixed-point exchange (2.58) with} \\ \quad \mathcal{F}\, W := S \cup W,\ \ \mathcal{G}\, W := R \cdot W,\ \ \mathcal{H}\, W := W \cdot T\,; \\ \text{proviso on } \mathcal{G}, \mathcal{H} \text{ holds since composition distributes} \\ \text{over union } -(2.14) \text{ and its right-analogue}- \end{array} \right\} \\
 & \langle \forall W :: R \cdot (S \cup W \cdot T) \;=\; (S \cup R \cdot W) \cdot T \rangle \\
\Leftarrow & \quad \{ \text{ distribution of composition over union, Leibniz } \} \\
 & R \cdot S \;=\; S \cdot T \ .
\end{array}
$$

□

The second version of the fixed-point exchange rule (2.58), which deals with inclusion instead of equality, can be used to give a similar proof of (2.66) and (2.67). The instance $T := id$ of the middle leap-frog variant (2.66) can be strengthened to an equivalence using the fact that $id^* = id$ and $R \subseteq R^*$. This will prove to be useful in future chapters:

$$R^* \cdot S \subseteq S \quad \equiv \quad R \cdot S \subseteq S \ . \tag{2.68}$$

Other properties of closure to be used in the sequel, whose proofs we omit, now follow. Let R and S be arbitrary relations, and a and b be points. Then:

$$(R \cup S)^* \;=\; (R^* \cdot S)^* \cdot R^* \ , \tag{2.69}$$

$$(R \cup S)^* \;=\; (R^* \cup S)^* \ , \tag{2.70}$$

$$(R \cdot a \cdot b^\circ)^* \;=\; id \cup R \cdot a \cdot b^\circ \ . \tag{2.71}$$

Finally, if R is a preorder, and a and b are points:

$$(R \cup a \cdot b^\circ \cup b \cdot a^\circ)^* \;=\; R \cdot (id \cup a \cdot b^\circ \cup b \cdot a^\circ) \cdot R \ . \tag{2.72}$$

Transitive Closure For every relation $R : X \leftarrow X$ there is also a smallest transitive relation that contains R, called the *transitive closure* of R and denoted by R^+. Characterisations of R^+ are analogous to those of R^*. Its

universal property reads as follows: for every transitive relation W,

$$R \subseteq W \quad \equiv \quad R^+ \subseteq W \ .$$

The transitive closure relates to the reflexive-transitive closure by the following two properties:

$$R^* \ = \ id \cup R^+ \ , \qquad R \cdot R^* \ = \ R^+ \ = \ R^* \cdot R \ . \tag{2.73}$$

And relation R^+ can also be given a Kleene-based pointwise characterisation similar to the one given above for R^*, except that the number n of "intermediate points" must not be zero: $x \langle R^+ \rangle y$ is equivalent to the existence of a *positive* natural number n such that there exist elements $x_n, x_{n-1}, \ldots x_1, x_0$ that satisfy $x = x_n$, $\langle \forall i : n \geq i > 0 : x_i \langle R \rangle x_{i-1} \rangle$ and $x_0 = y$.

2.5 Basic Graph Concepts

This section presents basic concepts of graph theory using the calculus of binary relations as working tool. There is a close connection between graphs and binary relations since every graph induces a relation between its vertices as well as a relation between its edges and its vertices. However, the phrasing of even the most elementary graph-theoretical notions in a calculational framework like ours is by no means standard. And the terminology used within graph theory is not standard either. Hence, every book or article devoted to the topic initially sets the terminology preferred by the authors since not doing so is likely to confuse readers. This section, as well as the rest of this chapter, establishes the basics of graph theory as will be used in the rest of this thesis.

Directed and Undirected Graphs A *graph* is defined to be a 4-tuple $(Vert, Edge, x1, x2)$ where $Vert$ and $Edge$ are sets, and $x1$ and $x2$ are functions of type $Vert \leftarrow Edge$ that determine the extreme vertices of an edge. This definition does not limit the graph to be directed or undirected, that being determined by what relations are built upon the extreme functions $x1$ and $x2$.

In a *directed graph* the extreme function $x1$ is interpreted as determining the vertex that an edge leads to whilst $x2$ determines the vertex an edge comes from. To a directed graph we associate a successor relation of type $Vert \leftarrow Vert$. If there is an edge leading to a vertex v from another –possibly the same– vertex w, then v is said to be a successor of w. The succesor relation is thus formally defined to be:

$$Succ \ := \ x1 \cdot x2^\circ \ . \tag{2.74}$$

On the other hand, in an *undirected graph* the fuuctions $x1$ and $x2$ are interpreted symmetrically as determining both extremes of an edge. No order is imposed upon the two vertices joined by an edge and, so, an undirected graph is equipped with a symmetric *adjacency* relation of type $Vert \leftarrow Vert$. Vertices v and w are adjacent if they are the two extremes of an edge, being irrelevant whether v is obtained through the function $x1$ and w through $x2$ or vice versa. Hence, we define the adjacency relation to be:

$$Adj \;:=\; x1 \cdot x2^{\circ} \cup x2 \cdot x1^{\circ} \;. \tag{2.75}$$

By properties of converse –(2.16), (2.17)–, Adj is indeed a symmetric relation, as expected. A directed graph can be converted into an undirected one by taking $Adj = Succ \cup Succ^{\circ}$.

Subgraphs Let G be a graph $(Vert, Edge, x1, x2)$. A *subgraph* of G is given by a subset V of its vertices and a subset E of its edges such that every edge in E has its extremes in V. This means that a subgraph is given by a set $V : \mathsf{Vec}\, Vert$ and a set $E : \mathsf{Vec}\, Edge$ such that:

$$x1 \cdot E \subseteq V \;\;\wedge\;\; x2 \cdot E \subseteq V \;. \tag{2.76}$$

When a subgraph is formed by the whole set of original vertices, i.e. $Vert$, and any subset E of edges it is called a *spanning subgraph*. In such a case (2.76) is trivially satisfied since $Vert$ is the universal relation of $\mathsf{Rel}(Vert, 1)$ and, hence, the subgraph is uniquely determined by E. We will therefore identify spanning subgraphs with their set of edges.

Consider G to be a directed graph. Then, given $E : \mathsf{Vec}\, Edge$, the successor relation of the spanning subgraph of G determined by E is:

$$succ\, E \;:=\; x1 \cdot \mathbin{\mathfrak{c}} E \cdot x2^{\circ} \;. \tag{2.77}$$

If G is considered to be undirected, the adjacency relation of its spanning subgraph determined by E is:

$$adj\, E \;:=\; x1 \cdot \mathbin{\mathfrak{c}} E \cdot x2^{\circ} \cup x2 \cdot \mathbin{\mathfrak{c}} E \cdot x1^{\circ} \;. \tag{2.78}$$

Coreflexives are symmetric (2.32) and, thus, properties of converse –(2.16), (2.17)- again imply, as with Adj, that relation $adj\, E$ is symmetric for every E.

Every graph is a spanning subgraph of itself. Consistently, the relations associated with a graph G can be expressed as the relations of the trivial spanning subgraph of it, viz. the subgraph determined by the whole set of edges $Edge$. The formal link are the equations $Succ = succ\, Edge$ and $Adj = adj\, Edge$, straightforward consequences of (2.36).

Given that $\mathbin{\not\vdash}$ is monotonic, by (2.37), and that so is composition, we also have that *succ* and *adj* are monotonic. This means that $succ\,E1 \subseteq succ\,E2$ follows from $E1 \subseteq E2$; in particular, we have $succ\,E \subseteq Succ$ for every E. And the same goes for *adj*. Finally, since $\mathbin{\not\vdash}$ distributes over union (2.41) and so does composition, *succ* and *adj* do distribute over union as well.

Parallel Edges and Loops The formalisation of the successor and adjacency relations of spanning subgraphs can aid the characterisation of other graph features. Let G be, again, a graph $(Vert, Edge, x1, x2)$, take an edge e –i.e. a function in $\mathsf{Fun}(Edge, 1)$– and let v and w be, respectively, the extreme vertices $x1 \cdot e$ and $x2 \cdot e$ of e. Then, using a pointwise style, relation $succ\,e$ is the set $\{(v, w)\}$ and relation $adj\,e$ is the set $\{(v, w), (w, v)\}$. In the point-free style, this corresponds to the following two equalities, which we call the *atomic successor* and *atomic adjacency* equalities:

$$succ\,e = (x1 \cdot e) \cdot (x2 \cdot e)^\circ \;, \tag{2.79}$$

$$adj\,e = (x1 \cdot e) \cdot (x2 \cdot e)^\circ \cup (x2 \cdot e) \cdot (x1 \cdot e)^\circ \;. \tag{2.80}$$

These "atomic" equalities follow from the definitions of *succ* (2.77) and *adj* (2.78), the coreflexive representation of singletons (2.42), and property (2.16) of converse.

Two edges $d, e : Edge$ of G are *parallel* exactly when $succ\,d = succ\,e$, if G is considered to be directed, and exactly when $adj\,d = adj\,e$, if G is considered to be undirected.

A *loop* is an edge that connects a vertex with itself. Formally, $e : Edge$ is a loop exactly when $succ\,e \subseteq id$, if G is directed, and exactly when $adj\,e \subseteq id$, if G is undirected. The directed/undirected distinction is rather artificial in this case since:

$$\begin{array}{ll} & adj\,e \subseteq id \\ \equiv & \left\{ \begin{array}{l} \text{atomic adjacency (2.80);} \\ \text{atomic successor (2.79), converse (2.16)} \end{array} \right\} \\ & succ\,e \cup (succ\,e)^\circ \subseteq id \\ \equiv & \left\{ \begin{array}{l} \text{universal property of union (2.5),} \\ \text{converse –(2.16), (2.18)–} \end{array} \right\} \\ & succ\,e \subseteq id \;. \end{array}$$

Simple Graphs Books on graph theory, as mentioned earlier, differ widely in terminology, even in what is called a graph. For instance, it is sometimes understood that the plain term "graph" is only applied to undirected graphs,

directed graphs thus having to be explicitly qualified as such; the opposite is not uncommon, i.e. that "graph" is implicitly understood to refer to a directed graph. From our definition of graph at the beginning of this section, it is understood that graphs need always be qualified as either directed or undirected.

Another common assumption is that parallel edges are not allowed. Our definition of graph naturally allows the existence of parallel edges. However, when there are no parallel edges or their existence is considered irrelevant, the graph can be described only by its set of vertices and the relationship between them induced by the edges. We call such a graph a *simple graph*.

Thus, a *simple directed graph* is a pair $(Vert, Succ)$ where $Succ$ is any relation of type $Vert \leftarrow Vert$. It is straightforward to construct the unique simple directed graph associated with a given directed graph, taking $Succ$ as defined above in (2.74). But several directed graphs could have the same successor relation and hence the same simple directed graph, due to the presence of parallel edges. Likewise, a *simple undirected graph* is a pair $(Vert, Adj)$ where Adj is a relation of type $Vert \leftarrow Vert$ that must be symmetric. As before, a given undirected graph induces a unique simple undirected graph, taking Adj as in (2.75), but a simple undirected graph can be obtained from several undirected graphs.

Incidence Relation and Hypergraphs The *incidence* relation of a graph $(Vert, Edge, x1, x2)$ is the relation of type $Vert \leftarrow Edge$ defined thus:

$$Inc \;:=\; x1 \cup x2 \;. \tag{2.81}$$

This definition does not make a distinction between directed and undirected graphs.

A *hypergraph* is a 3-tuple $(Vert, Edge, Inc)$ where Inc is any relation of type $Vert \leftarrow Edge$. Every graph induces a unique hypergraph, taking Inc as defined in (2.81). But there are hypergraphs that are not induced by any graph whatsoever. This is due to the fact that an edge of a hypergraph could be related to more than two vertices or to no vertices at all.

Let H be a hypergraph $(Vert, Edge, Inc)$. The *adjacency* relation of H is defined to be the following relation of type $Vert \leftarrow Vert$:

$$HAdj \;:=\; Inc \cdot Inc^{\circ} \;. \tag{2.82}$$

The *cograph* of H is the hypergraph $(Edge, Vert, Inc^{\circ})$.

Let G be a graph. Note that the adjacency relation of G is *not* equal to the adjacency relation of the hypergraph induced by G, reason for which

HAdj was given a different name. The *cograph* of G is defined to be the cograph of the hypergraph induced by G.

2.6 Connectedness and Acyclicity

The task of presenting graph-theoretical concepts using the calculus of binary relations is continued in this section with two important notions. First, we present the notion of connectedness, which deals with whether or not a graph can be traversed from a certain vertex or edge to another. Second, we present the notion of acyclicity, which refers to the absence of cycles: traversals of a graph that start from a certain vertex and arrive back at the same one. Both notions rely on the reachability and joinability relations of a graph, which we start the section with, and, also, both notions are related to the concept of covering, explored towards the end of the section.

Reachability and Joinability Let G be a directed graph with successor relation *Succ*. Since *Succ* relates two vertices when there is an edge between them, $Succ^*$ relates vertices for which one can traverse the graph using zero or more edges to arrive at one of the vertices having started from the other. If the vertices of a graph are thought to be railway stations and the edges to be railway lines, $Succ^*$ models possible destination/origin pairs one could travel by train. We will say that vertex v is *reachable* from vertex w whenever $v \langle Succ^* \rangle w$ holds. The *reachability* relation of graph G is defined accordingly:

$$Reach \;:=\; Succ^* \;. \tag{2.83}$$

Analogously, if G is an undirected graph with adjacency relation *Adj*, we will say that vertices v and w are *joinable* whenever $v \langle Adj^* \rangle w$ holds. The *joinability* relation of G is then defined to be:

$$Join \;:=\; Adj^* \;. \tag{2.84}$$

These relations can also be defined for spanning subgraphs. The reachability and joinability relations of the spanning subgraph determined by set E of edges are, respectively:

$$reach\, E \;:=\; (succ\, E)^* \;, \tag{2.85}$$

$$join\, E \;:=\; (adj\, E)^* \;. \tag{2.86}$$

Since *succ*, *adj* and closure are monotonic, so are *reach* and *join*. Also, $Reach = reach\ Edge$ and thus $reach\, E \subseteq Reach$ for every E. Analogous properties hold for *join*.

Strong Connectedness and Connectedness Let G be a directed graph with reachability relation *Reach*. Two vertices of G are *strongly connected* or *mutually reachable* if each of them is reachable from the other, i.e. if they are related by the following *strong connectedness* relation:

$$Str \;:=\; Reach \cap Reach^{\circ} \;. \tag{2.87}$$

A *strongly connected graph* is a graph in which every pair of vertices are strongly connected. And a maximal strongly connected subgraph of G is called a *strongly connected component* of G. Reachability relation *Reach* is a preorder and, therefore, *Str* is an equivalence relation –see remark on preorders and equivalence relations in page 21–. The *Str*-equivalence classes correspond to the sets of vertices of the strongly connected components of G. As customary by now, we also define the strong connectedness relation induced by a subset E of the set of edges of G:

$$str\, E \;:=\; (reach\, E) \cap (reach\, E)^{\circ} \;. \tag{2.88}$$

For the same reasons that *Str* is an equivalence relation, *str* always produces equivalence relations. Also, *str* is monotonic and the following holds: $Str = str\, Edge$. Hence, $str\, E \subseteq Str$ for every E.

Analogously, let G be an undirected graph with joinability relation *Join*. Two vertices of G are said to be *connected* if they are joinable, i.e. related by *Join*. A graph in which every two elements are connected is a *connected graph*, and a maximal connected subgraph of G is a *connected component* of G. Since *Join* is the reflexive-transitive closure of a symmetric relation, viz. *Adj*, it is an equivalence relation –see remark on closure commuting with converse in page 23–. The *Join*-equivalence classes are the sets of vertices of the connected components of G.

Formally, G is connected exactly when $\Pi = Join$, which is equivalent to $\Pi \subseteq Join$. The spanning subgraph of G determined by a set E of its edges is connected exactly when $\Pi = join\, E$ or, equivalently, $\Pi \subseteq join\, E$.

The fact that $join\, E \subseteq Join$ holds for every E implies, by transitivity of inclusion, that if the spanning subgraph of G determined by E is connected then so is G. Hence, a non-connected graph has no connected spanning subgraphs whatsoever. However, one can ask a spanning subgraph of a non-connected graph to "do its best", demanding that it preserves the original connected components, i.e. demanding that it keeps connected every two vertices which are connected in the whole graph. If the graph is connected, this requirement amounts to demanding the subgraph to be connected as well. We call such a spanning subgraph a *connectedness-preserving* subgraph. For

later use, we give a name to the formal phrasing of this property:

$$connpre\, E \;:=\; Join = join\, E \;. \tag{2.89}$$

Note that this is equivalent to $Join \subseteq join\, E$.

Biconnectedness In an undirected graph G, two vertices are *biconnected* if they are connected and remain connected after the elimination of any other vertex. A *biconnected graph* is a graph in which every pair of vertices are biconnected. And a *biconnected component* of G is a maximal biconnected subgraph of G. Unlike connectedness and strong connectedness, which partition the set of vertices according to the equivalences Str and $Join$, respectively, biconnectedness does not induce a partition on the vertices. However, it does partition the set of edges. For a small example, take a graph with 3 vertices u, v, w and 2 edges d, e joining, respectively, u with v, and v with w. Its two biconnected components are determined by the following pairs of vertex/edge sets: $(\{u, v\}, \{d\})$ and $(\{v, w\}, \{e\})$. Vertex v lies in two of the biconnected components. Such vertices are called *cut-vertices* because their removal disconnects vertices previously connected. In the example, the removal of v disconnects u and w. The edge set is indeed partitioned into $\{\{d\}, \{e\}\}$.

An equivalence relation on the edges then seems a more attractive way of formalising biconnectedness and we dedicate ourselves to such a task. Let G be the graph $(Vert, Edge, x1, x2)$ with incidence relation Inc –recall (2.81)–. Then take the *edge-adjacency* relation of G to be the (vertex-)adjacency relation (2.82) of the cograph of G, and the *edge-joinability* relation to be its closure:

$$\begin{aligned} EAdj &:= Inc^\circ \cdot Inc \;, \\ EJoin &:= EAdj^* \;. \end{aligned}$$

Now, two edges lie on the same biconnected component if they are connected, i.e. related via $EJoin$, and if they remain connected after the removal of any vertex. For the latter condition we need to formalise what it is to be (edge-)joinable using only some of the vertices. We proceed just as we did for spanning subgraphs. Given a set $V : \mathrm{Vec}\ Vert$, we define the edge-adjacency and edge-joinability relations under vertex set V to be:

$$\begin{aligned} eadj\, V &:= Inc^\circ \cdot \not\in V \cdot Inc \;, \\ ejoin\, V &:= (eadj\, V)^* \;. \end{aligned}$$

For a vertex $v : Vert$, its removal from the whole of the vertex set results in set $\overline{v}$. Hence, the biconnectedness equivalence, of type $Edge \leftarrow Edge$, is

$\langle \cap v :: ejoin\ \overline{v} \rangle$ with v ranging over points of $Vert$.

Acyclicity A *cycle* in a graph G corresponds to a traversal of G that, having started at a vertex v, say, ends at v again. An empty traversal, which trivially starts and ends at the same point, is not considered to be a cycle. This notion, as all the others, is formalised differently in the context of a directed graph than in the context of an undirected graph. The former is simpler than the latter.

In a directed graph G with successor relation $Succ$, the existence of a traversal between any two given vertices is determined by the reachability relation $Reach$, i.e. $Succ^*$. Similarly, the existence of a non-empty traversal is determined by the transitive closure of the successor relation: $Succ^+$. Hence, the existence of a cycle corresponds to $Succ^+$ having a non-empty intersection with the identity function. This amounts to saying that G is *acyclic* exactly when $Succ^+ \cap id = \emptyset$. Analogously, the absence of cycles in a spanning subgraph of G is formalised by means of the restricted successor relation: the spanning subgraph determined by set E of edges is acyclic exactly when $(succ\ E)^+ \cap id = \emptyset$.

Formalising acyclicity in undirected graphs is rather more challenging. The problem is that any traversal in an undirected graph could just be "walked" back to the starting vertex by using the same edges in reversed order. For example, take the graph formed ouly by vertices u, v and edge d joining u and v. The graph can be traversed using path $[u, d, v, d, u]$ which, in spite of the fact that it starts and ends in u, does not comply with our intuitive understanding of a cycle. In an undirected graph G with adjacency relation Adj, traversals of G correspond to relation $Join$, i.e. Adj^*. From the remarks and example above, we can then deduce that, unlike the case for directed graphs, neither $Join$ nor Adj^+ will be of much help in capturing the existence of cycles in G. In every non-trivial undirected graph, non-trivial meaning with at least one edge, relation $Adj^+ \cap id$ is non-empty irrespective of the existence of cycles.

But there is a way out. Suppose there is a cycle in G. Since empty traversals do not count as cycles, the set of edges of G involved in the cycle must be non-empty. Draw an edge e from such a set. Given that e forms part of a cycle, it must be the case that G can be traversed from one of the extremes of e to the other *without* using e. Therefore, the extremes of e must be joinable through $join\ \overline{e}$ and, so, $adj\ e \subseteq join\ \overline{e}$ must hold. Conversely, if $adj\ e \subseteq join\ \overline{e}$ holds for any edge e in G, then e must form a cycle with some other edges in $\overline{e}$ and, hence, there is a cycle in G. We conclude that an undirected graph G is *cyclic* if and only if $\langle \exists e :: adj\ e \subseteq join\ \overline{e} \rangle$, with e ranging over points of $Edge$, holds.

The use we have just made of *adj* and *join* will turn out to be of more general, and very profitable, use. We will thus give a name to such a relationship now and will explore some of its properties in the next subsection. We define a set $E1$ of edges to be *covered* by set $E2$ of edges if, given any pair of vertices adjacent via an edge in $E1$, a traversal that connects them via edges in $E2$ can be found. The *covering* relation, denoted from now on by $\preceq$, is then formally defined as follows:

$$E1 \preceq E2 \quad := \quad adj\, E1 \subseteq join\, E2 \ . \tag{2.90}$$

The above formalisation of the existence of cycles in undirected graphs can then be rephrased as: G is cyclic if and only if $\langle \exists e :: e \preceq \overline{e} \rangle$ holds.

We will later use the notion of acyclicity in spanning undirected subgraphs. In such a case, all the edges involved in the formalisation of the existence of cycles must be drawn from the set E, say, of edges of the subgraph. We define:

$$cyclic\, E \quad := \quad \langle \exists e : e \subseteq E : e \preceq E - e \rangle \ , \tag{2.91}$$

$$acyclic\, E \quad := \quad \neg\, cyclic\, E \ . \tag{2.92}$$

Covering We now present a collection of properties of covering that will be of much use in the sequel. We start with two general properties that will allow us to show that the covering relation $\preceq$ is a preorder. The first property corresponds to the relationship between inclusion and covering. Let $E1$ and $E2$ be arbitrary sets of edges and assume that $E1 \subseteq E2$ holds. We then have:

$$adj\, E1 \quad \subseteq \quad adj\, E2 \quad \subseteq \quad join\, E2 \ ,$$

where the first inclusion corresponds to *adj*-monotonicity, and the second follows from definition of *join* (2.86) and property $R \subseteq R^*$ of closure. Hence, inclusion implies covering:

$$E1 \subseteq E2 \quad \Rightarrow \quad E1 \preceq E2 \ . \tag{2.93}$$

The second property concerns the transformation of *adj* into *join* in the formal phrasing of covering. Every joinability relation, i.e. $join\, E$ for any E, having been constructed by an application of the reflexive-transitive closure operator, is a preorder. It then follows from the universal property of closure (2.62) plus the definition of $\preceq$ (2.90) that:

$$E1 \preceq E2 \quad \equiv \quad join\, E1 \subseteq join\, E2 \ . \tag{2.94}$$

We can now prove that the covering relation is a preorder: reflexivity of inclusion and (2.93) implies reflexivity of $\preceq$, transitivity of inclusion and (2.94) implies transitivity of $\preceq$.

The formalisation of acyclicity in undirected graphs gave rise to the formalisation of the covering relation. The relationship between the two notions is clear in their definitions –(2.90), (2.91)–. But covering is also related to the notion of preservation of connectedness in undirected graphs, as formalised by *connpre* (2.89). We now make explicit the relationship of covering with the connectedness-preserving predicate. Take the definition of *connpre* (2.89), replace *Join* with *join Edge* and replace the equality sign, on account of *join*-monotonicity, with inclusion. Property (2.94) above then implies the following:

$$connpre\, E \quad \equiv \quad Edge \preceq E \ . \tag{2.95}$$

Let us now analyse how the covering relation interacts with union. This is worth exploring due to the fact that *adj* distributes over union: it allows exploitation of the universal property of union since *adj* is placed on the left-hand side of $\subseteq$ in the definition of $\preceq$. Let $\mathcal{E}$ be a bag of edge sets and F be an edge set. We then manipulate as follows:

$$\begin{array}{ll} & \langle \cup E : E \in \mathcal{E} : E \rangle \preceq F \\ \equiv & \quad \{ \text{ definition of } \preceq \ (2.90) \ \} \\ & adj\, \langle \cup E : E \in \mathcal{E} : E \rangle \subseteq join\, F \\ \equiv & \quad \{ \text{ distribution of } adj \text{ over union } \} \\ & \langle \cup E : E \in \mathcal{E} : adj\, E \rangle \subseteq join\, F \\ \equiv & \quad \{ \text{ universal property of union (2.3) } \} \\ & \langle \forall E : E \in \mathcal{E} : adj\, E \subseteq join\, F \rangle \\ \equiv & \quad \{ \text{ definition of } \preceq \ (2.90) \ \} \\ & \langle \forall E : E \in \mathcal{E} : E \preceq F \rangle \ . \end{array}$$

Hence, union interacts with covering in the same way it does with inclusion:

$$\langle \cup E : E \in \mathcal{E} : E \rangle \preceq F \quad \equiv \quad \langle \forall E : E \in \mathcal{E} : E \preceq F \rangle \ , \tag{2.96}$$

for any bag $\mathcal{E}$ of edge sets and any edge set F.

Two Gates We bring this section to an end with a crucial property of the covering relation $\preceq$ which we call the *Two Gates* rule. It concerns, as when the formal definition of $\preceq$ arose, cycles in undirected graphs. As previously remarked, cycles are *non-empty* traversals that start and end at the same vertex: they must involve at least one edge. Cycles with only one

edge correspond to loops –see page 27–. The Two Gates rule is about cycles with at least two edges. We first give an informal explanation. Let d and e be edges and let E be a set of edges. Suppose that both $d \not\preceq E$ and $e \not\preceq E$ hold. This means that adding d to E will not make d part of a cycle, and the same goes for e. Now suppose that the addition of d to E covers e, i.e. that $e \preceq E \cup d$ holds. This means that, after having added d to E, subsequent addition of e does create a cycle. And both d and e are involved in this cycle. It must then also be the case that the addition of e to E covers d, i.e. we can conclude that $d \preceq E \cup e$ must also hold. By symmetry, the reverse implication is valid as well. Visualising the cycle that d and e form when added to E as the fence of an enclosed field inspired the name *Two Gates* for the rule: d alone or e alone can be "opened" to let the sheep break free.

We now state the Two Gates rule formally:

$$d \preceq E \cup e \;\equiv\; e \preceq E \cup d \qquad \text{provided } d \not\preceq E \text{ and } e \not\preceq E \;. \tag{2.97}$$

Since union appears on the right-hand side of $\preceq$ in the demonstrandum, it will be necessary to deal with expressions of the form $join\,(F \cup f)$ with F an edge set and f an edge. Already formulated properties of closure provide a handle for manipulating such expressions:

$$\begin{array}{ll}
 & join\ (F \cup f) \\
= & \quad \{\ \text{definition of } join\ (2.86)\ \} \\
 & (adj\ (F \cup f))^* \\
= & \quad \{\ adj \text{ distributes over union}\ \} \\
 & (adj\ F\ \cup\ adj\ f)^* \\
= & \quad \left\{ \begin{array}{l} \text{closure (2.70) with } R, S := adj\ F,\ adj\ f\ ; \\ \text{definition of } join\ (2.86) \end{array} \right\} \\
 & (join\ F\ \cup\ adj\ f)^* \\
= & \quad \{\ \text{atomic adjacency (2.80)}\ \} \\
 & (join\ F\ \cup\ (x1 \cdot f) \cdot (x2 \cdot f)^\circ\ \cup\ (x2 \cdot f) \cdot (x1 \cdot f)^\circ)^* \\
= & \quad \left\{ \begin{array}{l} \text{closure (2.72) with } R, a, b := join\ F,\ x1 \cdot f,\ x2 \cdot f\ ; \\ \text{atomic adjacency (2.80) again} \end{array} \right\} \\
 & join\ F \cdot (id\ \cup\ adj\ f) \cdot join\ F\ .
\end{array}$$

This equality we have just proved will be referred to as the *incrementality* property of $join$. For later reference, we state it explicitly:

$$join\,(F \cup f)\ =\ join\,F \cdot (id\ \cup\ adj\,f) \cdot join\,F\ . \tag{2.98}$$

We will also make use of the following fact:

$$f \preceq F \quad \equiv \quad (x1 \cdot f) \cdot (x2 \cdot f)^\circ \ \subseteq \ join\, F \ , \tag{2.99}$$

which follows from the definition of $\preceq$ (2.90), atomic adjacency (2.80), universal property of union (2.5), properties of converse and symmetry of joinability relations.

Now, finally, we proceed to prove the Two Gates rule.

Proof of (2.97)*:*

Take $v1$, $v2$, $w1$ and $w2$ to be the extreme vertices of edges d and e, i.e. $(x1 \cdot d)$, $(x2 \cdot d)$, $(x1 \cdot e)$ and $(x2 \cdot e)$, respectively.

We then argue thus:

$$\begin{array}{ll}
 & d \preceq E \cup e \\
\equiv & \quad \{\ (2.99),\ \text{extremes of } d\ \} \\
 & v1 \cdot v2^\circ \ \subseteq \ join\,(E \cup e) \\
\equiv & \quad \{\ (2.98),\ \text{atomic adjacency (2.80)},\ \text{extremes of } e\ \} \\
 & v1 \cdot v2^\circ \ \subseteq \ join\, E \cdot (id \ \cup \ w1 \cdot w2^\circ \ \cup \ w2 \cdot w1^\circ) \cdot join\, E \\
\equiv & \quad \{\ \text{distribution of composition over union}\ \} \\
 & v1 \cdot v2^\circ \ \subseteq \ join\, E \cdot join\, E \ \cup \ join\, E \cdot w1 \cdot w2^\circ \cdot join\, E \\
 & \qquad\qquad \cup \ join\, E \cdot w2 \cdot w1^\circ \cdot join\, E \\
\equiv & \quad \left\{ \begin{array}{l} \text{atoms are irreducible;\ joinability relations duplicate,} \\ \text{i.e. } join\, E \cdot join\, E = join\, E\text{,\ for being preorders} \end{array} \right\} \\
 & v1 \cdot v2^\circ \subseteq join\, E \quad \vee \quad v1 \cdot v2^\circ \subseteq join\, E \cdot w1 \cdot w2^\circ \cdot join\, E \\
 & \quad \vee \quad v1 \cdot v2^\circ \subseteq join\, E \cdot w2 \cdot w1^\circ \cdot join\, E \\
\equiv & \quad \{\ \text{assumption } d \not\preceq E \text{ cancels first disjunct by (2.99)}\ \} \\
 & v1 \cdot v2^\circ \subseteq join\, E \cdot w1 \cdot w2^\circ \cdot join\, E \\
 & \quad \vee \quad v1 \cdot v2^\circ \subseteq join\, E \cdot w2 \cdot w1^\circ \cdot join\, E \\
\equiv & \quad \left\{ \begin{array}{l} \text{property (2.31) of points and vectors, twice;} \\ \text{converse, symmetry of joinability relations} \end{array} \right\} \\
 & (v1 \subseteq join\, E \cdot w1 \quad \wedge \quad v2 \subseteq join\, E \cdot w2) \\
 & \quad \vee \quad (v1 \subseteq join\, E \cdot w2 \quad \wedge \quad v2 \subseteq join\, E \cdot w1) \\
\equiv & \quad \left\{ \begin{array}{l} \text{property (2.28) of points, four times;} \\ \text{symmetry of joinability relations again} \end{array} \right\} \\
 & (w1 \subseteq join\, E \cdot v1 \quad \wedge \quad w2 \subseteq join\, E \cdot v2) \\
 & \quad \vee \quad (w2 \subseteq join\, E \cdot v1 \quad \wedge \quad w1 \subseteq join\, E \cdot v2) \\
\equiv & \quad \{\ \text{as all the above, but using assumption } e \not\preceq E\ \}
\end{array}$$

$$e \preceq E \cup d \ .$$

□

2.7 Spanning Trees

Let G be a connected undirected graph. A *spanning tree* of G is a spanning subgraph of G which is both acyclic and connected. A well-known fact of graph theory is that a spanning subgraph of a given graph G is a spanning tree exactly when it is maximally acyclic, and also exactly when it is minimally connected. What it is not well-known is a proof of this proposition in a calculational style. We will present such a proof in this section.

Many books on graph theory often restrict propositions and algorithms as applicable only to certain kinds of graphs. Spanning trees and their properties are an example of this since only connected graphs can have spanning trees –see remark on connected spanning subgraphs in page 30–. Such restrictions are usually disregarded by stating that propositions and algorithms that apply only to, for example, connected graphs can be still applied to a non-connected graph by treating each connected component of it separately. However satisfactory such a treatment of restrictions might be, it is often the case that a good formalism allows the avoidance of such provisos without burdening the formalisation of the –then unconditional– statements. This happens when the formal phrasing of a restricted statement and its corresponding unrestricted one do not differ much and are, thus, equally manageable. Spanning trees illustrate this phenomenon: the restriction to connected graphs is unnecessary.

The natural generalisation of spanning trees, aiming at applying the concept to any graph, is that of *connectedness-preserving forests*. A forest is an acyclic graph, and a connectedness-preserving subgraph is one that preserves connected components. This notion coincides with the original concept when applied to a connected graph: a connected-preserving forest of a connected graph G is a spanning tree of G, and vice versa.

Formally, given an undirected graph G, the spanning subgraph induced by a subset E of edges of G is a *connectedness-preserving forest* of G if and only if both $acyclic\, E$ (2.92) and $connpre\, E$ (2.89) hold. Hence, we define the corresponding predicate as follows:

$$cpf\, E \ := \ acyclic\, E \ \wedge \ connpre\, E \ . \tag{2.100}$$

We will now state and prove the aforementioned property of spanning trees

but generalised to deal with connectedness-preserving forests. In proving it, we will try our best to follow advice from the literature on presentation of mathematical arguments [8, 14, 60]. At several stages in the proof we will show that purely syntactic considerations suggest the path to follow.

Proposition 2.101 For an undirected graph G and a subset E of its edges, the following three statements are equivalent:

(i) E is a connectedness-preserving forest of G,
(ii) E is a maximal acyclic subgraph of G,
(iii) E is a minimal connectedness-preserving snbgraph of G.

Proof:

First, let us use (2.100), (2.43) and (2.44) to make a formal note of the three statements that are to be proved equivalent:

(i) $acyclic\, E \ \wedge \ connpre\, E$,
(ii) $mxl\,(acyclic, E)$,
(iii) $mnl\,(connpre, E)$.

Recall the definitions of subset-closed (2.45) and superset-closed (2.47) predicates. Both *cyclic* and *connpre* are superset-closed predicates: superset-closedness of *cyclic* follows from transitivity of inclusion, the fact that inclusion implies covering (2.93), transitivity of covering and predicate calculus; superset-closedness of *connpre* follows from monotonicity of *join* and transitivity of inclusion. Therefore, *acyclic* is a subset-closed predicate, since it negates *cyclic*, and *connpre* is a superset-closed predicate, facts which allow us to rephrase the second and third statements, on account of (2.46) and (2.48), as follows:

(ii') $acyclic\, E \ \wedge \ \langle \forall e : e \subseteq \overline{E} : \neg\, acyclic\,(E \cup e) \rangle$,
(iii') $connpre\, E \ \wedge \ \langle \forall e : e \subseteq E : \neg\, connpre\,(E - e) \rangle$.

Now, the equivalences (i) $\equiv$ (ii') and (i) $\equiv$ (iii') can, by propositional calculus, be respectively simplified to:

(a) $acyclic\, E \ \Rightarrow \ (connpre\, E \ \equiv \ \langle \forall e : e \subseteq \overline{E} : \neg\, acyclic\,(E \cup e) \rangle)$,
(b) $connpre\, E \ \Rightarrow \ (acyclic\, E \ \equiv \ \langle \forall e : e \subseteq E : \neg\, connpre\,(E - e) \rangle)$.

Hence, a proof of (a) and (b) will do.

Since the formulation of *cyclic* is simpler than that of *acyclic*, in the sense that negation does not enter the formulation of the former, we can simplify

our demonstranda by cancelling negations and using De Morgan's laws to:

$$\text{(a')} \quad \mathit{acyclic}\, E \;\Rightarrow\; (\mathit{connpre}\, E \equiv \langle \forall e : e \subseteq \overline{E} : \mathit{cyclic}\,(E \cup e)\rangle)\ ,$$
$$\text{(b')} \quad \mathit{connpre}\, E \;\Rightarrow\; (\mathit{cyclic}\, E \equiv \langle \exists e : e \subseteq E : \mathit{connpre}\,(E - e)\rangle)\ .$$

Note that, so far, the proof has been guided mostly by syntactic considerations. Statements (ii') and (iii') popped up from the educated guess of analysing whether the predicates involved were subset/superset-closed, thus getting formulae syntactically closer to (i). This made (a) and (b) emerge, which were then transformed into (a') and (b') motivated by syntactic simplification –the elimination of a few ocurrences of negation–.

It is also worth remarking that we have not strengthened the demonstrandum: all expressions have been transformed into equivalent ones. So, no risks have been taken so far.

We now present proofs of (a') and (b'), which will rely on two claims. Both proofs will make use of the covering relation $\preceq$, as linking concept between acyclicity and connectedness. First, for (a'), assume $\mathit{acyclic}\, E$ and manipulate thus:

$$\begin{array}{ll} & \mathit{connpre}\, E \\ \equiv & \quad \{\ \text{property of } \mathit{connpre}\ (2.95)\ \} \\ & \mathit{Edge} \preceq E \\ \equiv & \quad \{\ \text{complementation: } \mathit{Edge} = E \cup \overline{E}\ \} \\ & E \cup \overline{E} \preceq E \\ \equiv & \quad \{\ \text{union}/\preceq\ (2.96), \text{reflexivity of } \preceq\ \} \\ & \overline{E} \preceq E \\ \equiv & \quad \{\ \text{extensionality (2.29), union}/\preceq\ (2.96) \text{ again}\ \} \\ & \langle \forall e\ :\ e \subseteq \overline{E}\ :\ e \preceq E \rangle \\ \equiv & \quad \{\ \text{claim, see (c1) below}\ \} \\ & \langle \forall e\ :\ e \subseteq \overline{E}\ :\ \mathit{cyclic}\,(E \cup e) \rangle\ . \end{array}$$

Second, for (b'), assume $\mathit{connpre}\, E$ and then:

$$\begin{array}{ll} & \mathit{cyclic}\, E \\ \equiv & \quad \{\ \text{definition of } \mathit{cyclic}\ (2.91)\ \} \\ & \langle \exists e\ :\ e \subseteq E\ :\ e \preceq E - e \rangle \\ \equiv & \quad \{\ \text{claim, see (c2) below}\ \} \\ & \langle \exists e\ :\ e \subseteq E\ :\ \mathit{connpre}\,(E - e) \rangle\ . \end{array}$$

Let us briefly analyse the calculations above. The first step in the proof of

(a') arises from the willingness to use the covering relation $\preceq$, as formal link between *connpre* and *cyclic*. The following two steps are motivated by the need of bringing $\overline{E}$ into the picture, and the next by the need of getting a universal quantification as well. At this stage the syntactic shape of the formula suggest, by predicate calculus, that the following suffices to complete the proof:

$$\text{(c1)} \quad \begin{cases} \mathit{cyclic}\,(E \cup e) \;\equiv\; e \preceq E \\ \qquad \text{provided} \;\; \mathit{acyclic}\,E \;\text{ and }\; e \subseteq \overline{E} \;. \end{cases}$$

The proof of (b') is much shorter. The definition of *cyclic* is unfolded and the second claim immediately pops up:

$$\text{(c2)} \quad \begin{cases} \mathit{connpre}\,(E - e) \;\equiv\; e \preceq E - e \\ \qquad \text{provided} \;\; \mathit{connpre}\,E \;\text{ and }\; e \subseteq E \;. \end{cases}$$

It is now only (c1) and (c2) that are left to be proved. We start with (c2) since its proof is simpler:

$$\begin{array}{ll} & \mathit{connpre}\,(E - e) \\ \equiv & \quad \{ \text{ definition of } \mathit{connpre}\ (2.89) \ \text{–weaker version–} \ \} \\ & \mathit{Join} \subseteq \mathit{join}\,(E - e) \\ \equiv & \quad \{ \text{ assumption } \mathit{connpre}\,E\ (2.89) \ \} \\ & \mathit{join}\,E \subseteq \mathit{join}\,(E - e) \\ \equiv & \quad \{ \ \mathit{join}/\preceq \ (2.94) \ \} \\ & E \preceq E - e \\ \equiv & \quad \{ \text{ assumption } e \subseteq E \text{ implies } E = (E - e) \cup e \ \} \\ & (E - e) \cup e \preceq E - e \\ \equiv & \quad \{ \text{ union}/\preceq \ (2.96), \text{ reflexivity of } \preceq \ \} \\ & e \preceq E - e \;. \end{array}$$

Only (c1) to go. We argue:

$$\begin{array}{ll} & \mathit{cyclic}\,(E \cup e) \\ \equiv & \quad \{ \text{ definition of } \mathit{cyclic}\ (2.91) \ \} \\ & \langle \exists d : d \subseteq E \cup e : d \preceq (E \cup e) - d \rangle \\ \equiv & \quad \{ \text{ split range, one-point rule } \} \\ & \langle \exists d : d \subseteq E : d \preceq (E \cup e) - d \rangle \;\; \vee \;\; e \preceq (E \cup e) - e \\ \equiv & \quad \{ \text{ assumption } e \subseteq \overline{E} \text{ implies } (E \cup e) - e = E \ \} \\ & \langle \exists d : d \subseteq E : d \preceq (E \cup e) - d \rangle \;\; \vee \;\; e \preceq E \end{array}$$

$$\equiv \quad \{ \text{ see below } \}$$
$$e \preceq E \ .$$

The last step of the calculation above is equivalent to:

$$(\exists d : d \subseteq E : d \preceq (E \cup e) - d) \quad \Rightarrow \quad e \preceq E \ .$$

This can be proved by assuming $d \subseteq E$ for arbitrary d and then showing that the implication

$$d \preceq (E \cup e) - d \quad \Rightarrow \quad e \preceq E$$

holds. This statement can be moulded into a more symmetric shape. The assumptions on d and e, viz. $d \subseteq E$ and $e \subseteq \overline{E}$, imply the following equalities:

$$(E \cup e) - d \ = \ (E - d) \cup e \quad \text{and}$$
$$E \ = \ (E - d) \cup d \ .$$

Hence, the last implication above can be rewritten as follows:

$$(*) \quad d \preceq (E - d) \cup e \quad \Rightarrow \quad e \preceq (E - d) \cup d \ .$$

Now, this looks like a job for the Two Gates rule (2.97) with $E := E - d$, but we would then need its provisos: $d \not\preceq E - d$ and $e \not\preceq E - d$.

On account of the definitions of *cyclic* (2.91) and *acyclic* (2.92), the proviso on d follows from the assumptions *acyclic* E, which we had not used yet, and $d \subseteq E$.

The proviso on e does not necessarily hold but, in such a case, the consequent of implication $(*)$ would hold –and thus so would $(*)$ as well– on account of $E - d \subseteq (E - d) \cup d$, the fact that inclusion implies covering (2.93), and transitivity of covering. If the proviso on e happens to hold then Two Gates does the job.

And this concludes our proof!

□

2.8 Paths

A basic concept of graph theory we have not dealt with yet is that of paths in a graph. It was mentioned before that if the vertices of a directed graph with successor relation *Succ* are thought to be railway stations and the edges to be railway lines, then reachability relation *Reach*, i.e. $Succ^*$, models possible

destination/origin pairs one could travel by train. In such a case, paths model possible *routes* one could travel by train rather than just destination/origin pairs.

An analogous situation occurs in undirected graphs with joinability relation *Join*, i.e. Adj^*, and their paths. We will formalise and deal with all the corresponding details only for the case of directed graphs.

Let G be a directed graph $(Vert, Edge, x1, x2)$ and let u, v be two elements of $Vert$. A *path* to vertex u from vertex v is an alternating sequence of vertices and edges $[v_n, e_n, v_{n-1}, e_{n-1}, \ldots v_1, e_1, v_0]$ such that $u = v_n$, $v = v_0$ and $\langle \forall i : n \geq i \geq 1 : x1 \cdot e_i = v_i \ \wedge \ x2 \cdot e_i = v_{i-1} \rangle$. A path to u from v exists exactly when u is reachable from v, i.e. when $u \langle Reach \rangle v$ holds. To be able to formally link the existence of paths to reachability relation *Reach* we need to manipulate sequences in our mathematical framework. Fortunately, there is a well-established theory of datatypes within relational frameworks like ours which we will now shortly review.

The approach to datatypes we will present is based on the theory of categories and allegories. For a thorough presentation of the relational theory of datatypes, including the relevant theory of categories and allegories along with applications in the program derivation area, we recommend book [28] by Bird and de Moor. Some of the history of this approach to datatypes can be roughly summarised as follows: In the 1980s, Bird and Meertens worked on a calculus of functions for program derivation which included a theory for manipulating lists [21, 22, 100]. At the same time, category theory had been gaining interest from the computer science community –see e.g. [15, 125]–, in particular from the functional programming advocates. These two trends met by a proof that the theory of lists of the calculus of Bird and Meertens was governed by more general categorical concepts [140]. Gibbons then moved on to generalise the theory of lists to cater for various kinds of trees [62, 63], supported by a general categorical treatment of datatypes that had been worked out by Malcolm [97, 98]. Such a categorical treatment was proved to be extendable to the allegorical, or relational, setting by de Moor [108, 109]. Backhouse and his colleagues at Eindhoven were at the time also exploring an equivalent relational approach to datatypes, yet not relying on categorical but on lattice-theoretical grounds [2]. –Also, Malcolm's thesis [97], though developed within the categorical framework, contains beginnings of the work of Eindhoven on relations.– In both settings, i.e. de Moor's and Backhouse's, the level of abstraction obtained is high enough to allow concise reasoning about programs parameterised by datatype constructors, a feature that has been named *polytypism*. So-called *polytypic programming* has been receiving a lot of attention in recent years, see e.g. [11, 29, 65, 78, 110, 120].

We will only apply the general theory to the datatype of paths in a graph. Therefore, it is not justified to introduce all the general machinery and we will limit the presentation of the theory to its application to the datatype of paths.

The Datatype *Path* We present the datatype *Path* and some operations on it using notation of a Gofer/Haskell-like functional programming language [23, 81, 144]. Let G be a graph $(Vert, Edge, x1, x2)$. The datatype *Path* of G is declared thus:

$$Path \;::=\; wrap\ Vert \mid cons\,(Vert, Edge, Path) \;.$$

This declares *wrap* and *cons* to be the constructor functions of *Path*, of type $Path \leftarrow Vert$ and $Path \leftarrow Vert \times Edge \times Path$, respectively. Using the above definition, the path $[v_2, e_2, v_1, e_1, v_0]$ is represented by the expression $cons\,(v_2, e_2, cons\,(v_1, e_1, wrap\,v_0))$.

Functions to retrieve the starting and ending vertex of a path can be defined by pattern-matching as follows:

$$\begin{aligned} start\,(wrap\,v) &= v\;, & end\,(wrap\,v) &= v\;,\\ start\,(cons\,(v,e,p)) &= start\,p\;; & end\,(cons\,(v,e,p)) &= v\;. \end{aligned}$$

The declaration of *Path* does not guarantee that every value of the datatype actually corresponds to a path in the given graph G. A predicate to check when this is the case follows:

$$\begin{aligned} isPath\,(wrap\,v) &\equiv True\;,\\ isPath\,(cons\,(v,e,p)) &\equiv (x1\;e = v) \wedge (x2\;e = end\,p) \wedge isPath\,p\;. \end{aligned}$$

Products Products are used in the definition of *Path* above and, thus, we need to formalise what they are. Binary products are usually taken as the basic notion on which ternary products are built. We cut short our way by formalising ternary products directly. Given sets *X1*, *X2* and *X3*, we can form their product $X1 \times X2 \times X3$, which comes equipped with projection functions *out1*, *out2* and *out3*. Set-theoretically speaking, these are defined as follows:

$$\begin{aligned} X1 \times X2 \times X3 &= \{\,(x_1,x_2,x_3) \mid x_1 \in X1,\, x_2 \in X2,\, x_3 \in X3\,\}\;,\\ out1\,(x_1,x_2,x_3) &= x_1\;,\\ out2\,(x_1,x_2,x_3) &= x_2\;,\\ out3\,(x_1,x_2,x_3) &= x_3\;. \end{aligned}$$

Also, given relations $R : X1 \leftarrow Y1$, $S : X2 \leftarrow Y2$ and $T : X3 \leftarrow Y3$, their product is a relation of type $X1 \times X2 \times X3 \leftarrow Y1 \times Y2 \times Y3$ defined as:

$$R \times S \times T \;=\; out1^\circ \cdot R \cdot out1 \;\cap\; out2^\circ \cdot S \cdot out2 \;\cap\; out3^\circ \cdot T \cdot out3 \;.$$

This means that $(x_1, x_2, x_3)\ (R \times S \times T)\ (y_1, y_2, y_3)$ is equivalent to the conjunction of $x_1\ (R)\ y_1$, $x_2\ (S)\ y_2$ and $x_3\ (T)\ y_3$.

Among the properties of relational product we have:

$$\begin{aligned} out1 \cdot (R \times S \times T) &\subseteq R \cdot out1 \;, \\ out2 \cdot (R \times S \times T) &\subseteq S \cdot out2 \;, \\ out3 \cdot (R \times S \times T) &\subseteq T \cdot out3 \;. \end{aligned}$$

These can be strengthened to equalities when the relations that appear only in the left-hand side of the inequation are functions. That is:

$$out1 \cdot (R \times f \times g) = R \cdot out1 \;, \tag{2.102}$$
$$out2 \cdot (f \times S \times g) = S \cdot out2 \;, \tag{2.103}$$
$$out3 \cdot (f \times g \times T) = T \cdot out3 \;, \tag{2.104}$$

for functions f and g –of the appropriate type in each case– . Also, converse distribute through product:

$$(R \times S \times T)^\circ \;=\; R^\circ \times S^\circ \times T^\circ \;. \tag{2.105}$$

Folds Functions *start* and *end* and predicate *isPath* above are defined by *structural recursion*, a recursion scheme determined by the definition, i.e. the structure, of the datatype. One can provide snch a recursion scheme once and for all by using higher-order functions. Functions providing this kind of "canned" structural recursion are known as *folds* or *catamorphisms* [28, 101]. The functional fold for the datatype *Path* is parameterised by two functions $f : X \leftarrow Vert$ and $g : X \leftarrow Vert \times Edge \times X$, for some set X , giving back a function of type $X \leftarrow Path$ defined as follows:

$$\begin{aligned} foldp\ f\ g\ (wrap\ v) &= f\ v \;, \\ foldp\ f\ g\ (cons\ (v, e, p)) &= g\ (v, e, foldp\ f\ g\ p) \;. \end{aligned}$$

Using this new tool, functions *start* and *end* can be defined more concisely:

$$start = foldp\ id\ out3 \;, \tag{2.106}$$
$$end = foldp\ id\ out1 \;. \tag{2.107}$$

Predicate *isPath* could be defined in a similar way, i.e. by means of functions only. However, we will make a better use of it if defined as a coreflexive relation –recall that coreflexive relations model subsets, which in turn are in a one-to-one correspondence with unitary predicates– and, thus, we need to generalise the above definition of fold to cater for relations. A point-free description of the functional fold above gives a clear signal of what the relational fold might look like:

$$\begin{aligned} foldp\ f\ g\ \cdot\ wrap &= f\ , \\ foldp\ f\ g\ \cdot\ cons &= g \cdot (id \times id \times foldp\ f\ g)\ . \end{aligned}$$

These equations suggest that, for relations R and S of appropriate types, we define:

$$foldp\ R\ S\ \cdot\ wrap = R\ , \tag{2.108}$$

$$foldp\ R\ S\ \cdot\ cons = S \cdot (id \times id \times foldp\ R\ S)\ . \tag{2.109}$$

Armed with this new tool we proceed to define a coreflexive relation that corresponds to *isPath*. In general, we can link an arhitrary predicate p to coreflexive $p?$ by the following equivalence: $x\ \langle p? \rangle\ y \equiv x = y \wedge p\,x$. We thus define:

$$isPath? = foldp\ wrap\ (cons \cdot ok?)\ , \tag{2.110}$$

where predicate *ok* sees to the consistence of the last edge of the path:

$$ok\ (v, e, p) \equiv (x1\ e = v) \wedge (x2\ e = end\ p)\ . \tag{2.111}$$

Before finally proceeding to establish the formal connection between the existence of paths and the reachability relation, we need one more result regarding folds. This key result relates the composition of a fold after the converse of a fold to a least fixed point. Using the "morphism" terminology, such a construction, i.e. the composition of a catamorphism after the converse of a catamorphism, is called a *hylomorphism* [101]; under the "fold" terminology, it is sometimes referred to as a *mould* –a term somewhat jokingly coined by Oege de Moor–. For the particular case of the datatype *Path*, the key result relating moulds to least fixed points reads as follows:

$$\left.\begin{aligned} (foldp\ R\ S) \cdot (foldp\ T\ U)^\circ &= \\ \langle \mu W : R \cdot T^\circ \cup S \cdot (id \times id \times W) \cdot U^\circ \rangle\ . & \end{aligned}\right\} \tag{2.112}$$

Paths and Reachability Finally, we are ready to prove that two vertices in a directed graph are linked through a path if and only if they are related

by the reachability relation. That is:

$$end \cdot isPath? \cdot start^\circ \;=\; Reach \;. \tag{2.113}$$

Proof:

Let us start manipulating the left-hand side since, this side being the most complex, it provides more opportunities for manipulation. We first notice that there is a mould we can simplify:

$$\begin{array}{ll}
 & isPath? \cdot start^\circ \\
= & \quad \{ \text{ definitions of } isPath? \text{ (2.110) and } start \text{ (2.106) } \} \\
 & (foldp\ wrap\ (cons \cdot ok?)) \cdot (foldp\ id\ out3)^\circ \\
= & \quad \{ \text{ moulds (2.112) } \} \\
 & \langle \mu W : wrap \cup cons \cdot ok? \cdot (id \times id \times W) \cdot out3^\circ \rangle \\
= & \quad \left\{ \begin{array}{l} \text{by converse (2.16) and products } -(2.104), (2.105)-: \\ \quad (id \times id \times W) \cdot out3^\circ \;=\; (out3 \cdot (id \times id \times W)^\circ)^\circ \\ \quad\quad = (out3 \cdot (id \times id \times W^\circ))^\circ \;=\; (W^\circ \cdot out3)^\circ \\ \quad\quad = out3^\circ \cdot W \end{array} \right\} \\
 & \langle \mu W : wrap \cup cons \cdot ok? \cdot out3^\circ \cdot W \rangle \\
= & \quad \{ \text{ closure (2.63) } \} \\
 & (cons \cdot ok? \cdot out3^\circ)^* \cdot wrap \;.
\end{array}$$

We record this result for later reference:

$$isPath? \cdot start^\circ \;=\; (cons \cdot ok? \cdot out3^\circ)^* \cdot wrap \;. \tag{2.114}$$

The left-hand side of our demonstrandum (2.113) has thus been simplified to:

$$end \cdot (cons \cdot ok? \cdot out3^\circ)^* \cdot wrap \;.$$

Now, it is worth trying to use the leap-frog over closure rule (2.65). We could use it either to make *end* jump to the right over the closure or to make *wrap* jump to the left over the closure. The choice of making *end* jump is appealing: given that *end* is a fold, its composition with *cons* can be manipulated using (2.109) in an attempt to get the proviso for (2.65); once *end* has leap-frogged, it will, again for being a fold but by means of (2.108), cancel with *wrap*. We proceed:

$$\begin{array}{ll}
 & end \cdot cons \cdot ok? \cdot out3^\circ \\
= & \quad \{ \text{ definition of } end \text{ (2.107), fold computation (2.109) } \} \\
 & out1 \cdot (id \times id \times end) \cdot ok? \cdot out3^\circ \\
= & \quad \{ \text{ products (2.102), } end \text{ is a function } \}
\end{array}$$

$$
\begin{array}{ll}
 & out1 \cdot ok? \cdot out3^{\circ} \\
= & \left\{ \begin{array}{l} \text{by definition of } ok? \text{ (2.111) and projection functions:} \\ \quad v \langle out1 \cdot ok? \cdot out3^{\circ} \rangle p \\ \qquad \equiv \langle \exists e :: (x1\, e = v) \wedge (x2\, e = end\, p) \rangle \\ \qquad \equiv v \langle x1 \cdot x2^{\circ} \cdot end \rangle p \end{array} \right\} \\
 & x1 \cdot x2^{\circ} \cdot end \\
= & \quad \{ \text{ definition of } Succ \text{ (2.74) } \} \\
 & Succ \cdot end \ .
\end{array}
$$

The main calculation can then be summarised and completed thus:

$$
\begin{array}{ll}
 & end \cdot isPath? \cdot start^{\circ} \\
= & \quad \{ \text{ mould into closure above (2.114) } \} \\
 & end \cdot (cons \cdot ok? \cdot out3^{\circ})^{*} \cdot wrap \\
= & \quad \{ \text{ leap-frog over closure (2.65), proviso proved above } \} \\
 & Succ^{*} \cdot end \cdot wrap \\
= & \left\{ \begin{array}{l} \text{definition of } Reach \text{ (2.83)}; \\ \text{definition of } end \text{ (2.107), fold computation (2.108)} \end{array} \right\} \\
 & Reach \ .
\end{array}
$$

And we are done!

□

However simple this result might seem to be, it nicely demonstrates how our calculational abilities have grown in the last few years. In the most comprehensive account of graph theory within the framework of the calculus of binary relations –book [136] by Schmidt and Ströhlein –, this property is not proved solely by calculational means. Rather, it is proved by a mixture of calculations, "verbal formality" and induction [136, pages 106-107]. The key new ingredient is the introduction of recursive datatypes into the realm of relational calculations.

A similar fact, i.e. similar to our (2.113), which shows that the reflexive-transitive closure operator can be defined as a mould on non-empty lists, can be found in [27, Section 2.4.1].

Chapter 3
Computing Closure

We will now proceed with our first steps into calculating graph algorithms. The graph problems to be treated in this chapter are specified by means of the reflexive-transitive closure and transitive closure operators –Section 2.4–. As mentioned in the introductory Chapter 1, our developments will combine conventional techniques for the derivation of imperative programs with using the calculus of binary relations for the expression and manipulation of graph concepts and properties.

The problems of this chapter can be posed both for directed and undirected graphs. We will initially pose the problems for directed graphs and, subsequently, phrase them in terms of a "given relation" that might correspond either to the successor relation *Succ* of a directed graph or to the adjacency relation *Adj* of an undirected graph. However, even when dealing with the abstract "given relation" we will keep the bias towards directed graphs in the sense that, e.g., we will say "vertex reachable from vertex" instead of "vertex reachable from, or joinable with, vertex". All such references to directed graphs must be understood as applicable to undirected graphs as well.

Section 3.1 deals with the problem of computing the transitive closure of a given relation, which models the all-pairs reachability problem of graphs. The algorithmic solution obtained corresponds to a well-known algorithm due to Stephen Warshall [146]. Section 3.2 presents another reachability problem: the fixed-source reachability problem, which requires the computation of the set of vertices that can be reached from a given initial set of vertices. Section 3.3 reviews some work related to the contents of this chapter.

3.1 All-Pairs Reachability

The *All-Pairs Reachability* problem for directed graphs can be posed as follows: given a graph with successor relation $Succ : Vert \leftarrow Vert$, compute its *proper* reachability relation $Succ^+ : Vert \leftarrow Vert$. The statement $v \, (Succ^+) \, w$ holds when one can arrive at v starting from w via one or more edges, as opposed to $v \, (Succ^*) \, w$, which accepts traversing zero edges. Thus, $Succ^+$ is called the proper reachability relation, as opposed to the -plain- reachability relation $Succ^*$. Our problem should then be called the All-Pairs Proper Reachability problem. However, we choose to drop the prefix "proper" and state that all references to "reachability" in this section must be understood as referring to "proper reachability".

Our problem can be abstracted from the specific realm of algorithmic graph theory and be posed simply as: given a relation $R : X \leftarrow X$, compute its transitive closure $R^+ : X \leftarrow X$. However, our informal remarks will keep referring to the graph model of the problem.

Specification The formal specification reads:

$$
\begin{array}{l}
[\![\ \textbf{var} \ S : X \leftarrow X ; \\
\quad S : [\ true \ , \ S = R^+ \ ] \\
]\!]
\end{array}
\tag{3.1}
$$

where $R : X \leftarrow X$ is a given relation.

Setting Up an Iteration We will develop an iteration, as algorithmic solution to (3.1), based on a design due to Stephen Warshall. In the early 1960s, Warshall reported his solution [146] to the problem of computing the transitive closure of an input binary relation -but the presentation used boolean matrices instead of relations-. He did so by directly presenting the algorithm followed by its proof of correctness. We will present Warshall's conception of a solution, not as a straightforwardly presented algorithm, but as a *design*. A design that can nowadays be spelled out using calculi for formal algorithm development.

Warshall's design is based on the successive computation of constrained reachability relations, working towards smaller constraints until the desired unconstrained reachability relation is obtained. Such constraints are determined by sets of vertices allowed to be visited as intermediate vertices while traversing the graph.

Let us first express such constrained reachability relations in a semi-formal fashion. Recall the discussion on transitive closure in Section 2.4 and then

note that relation R^+ can thus be semi-formally described as follows:

$$R^+ \;=\; R \;\cup\; R \cdot R \;\cup\; R \cdot R \cdot R \;\cup\; \cdots \;. \tag{3.2}$$

We want to restrict the traversing of the graph modelled by R in such a way that intermediate vertices are drawn from a certain given set. Let $A : \mathsf{Vec}\, X$ be such a set. The restriction of intermediate vertices to A must be reflected in (3.2) at the places where the composition operator is used. Recall from Section 2.2 the discussion on the convenience of modelling sets as vectors versus modelling sets as coreflexive relations. The restriction of "middle points" in compositions to a certain set is the standard example of situations where coreflexives are better suited than vectors. Hence, we use operator $\mathbin{\mathfrak{c}}$ to describe the restriction of the proper reachability relation R^+ to intermediaries drawn from A thus:

$$R^+ \text{ restricted to } A \;=\; R \;\cup\; R \cdot \mathfrak{c}A \cdot R \;\cup\; R \cdot \mathfrak{c}A \cdot R \cdot \mathfrak{c}A \cdot R \;\cup\; \cdots \;.$$

If A includes all the vertices of the graph, i.e. if $A = X$, then $\mathfrak{c}$-isomorphism (2.36) gives us $\mathfrak{c}A = id$ and we thus get relation R^+ as described in (3.2) back, as indeed expected.

Let us now abandon semi-formality. For setting up an iteration that refines (3.1), we need to propose a reasonable invariant and corresponding guard. One technique for doing so is that of replacing a constant in the postcondition by a freshly introduced variable –see e.g. [68, Section 16.3] or [82, Section 4.2]–. Turning the semi-formal description of the restricted R^+ above into a fully formal one will help us apply such a technique. We need to manipulate expression R^+ in such a way that a hidden id, which plays the role of $\mathfrak{c}A$ in the unrestricted R^+, is brought to light. We use property (2.73) of transitive closure plus the fundamental property (2.2) of identity relations to make id pop up, and then we introduce operator $\mathfrak{c}$ via property (2.36):

$$R^+ \;=\; R^* \cdot R \;=\; (R \cdot id)^* \cdot R \;=\; (R \cdot \mathfrak{c}X)^* \cdot R \;. \tag{3.3}$$

This rephrases the postcondition in such a way that the aforementioned technique for proposing invariants can be successfully applied. Replace constant X by fresh variable A to postulate the following invariant:

$$Inv \;:=\; S \;=\; (R \cdot \mathfrak{c}A)^* \cdot R \;,$$

and use $A \neq X$ as guard.

To establish the invariant initially, the informal description of restrictions to R^+ above suggests that restriction to $\emptyset$ is just R. Formally, we reason in

one line thus:

$$(R \cdot \not\in \emptyset)^* \cdot R \;=\; (R \cdot \emptyset)^* \cdot R \;=\; \emptyset^* \cdot R \;=\; R \;.$$

The first step appeals to property (2.36) of the $\not\in$-isomorphism, the second corresponds to $\emptyset$ being a zero of composition (2.14), and the third and last uses the fact that $\emptyset^* = id$. Hence, statement $S, A := R, \emptyset$ establishes the invariant and can thus be used as initial statement.

We still need to work out a variant. Since variable A is initially set to $\emptyset$ and must be equal to the whole set X on exit of the iteration, it seems reasonable to postulate A itself to be the variant expression and to postulate that progress will be guaranteed by its increase. We introduce a shorthand for the boolean expression that states that progress has been made:

$$Prg \;:=\; (A \supset A_0) \;.$$

In Morgan's refinement calculus, 0-subscripted variables in the postcondition of specification statements refer to the values of such variables in the *initial* state of the computation, as opposed to the values in the final state referred to by the unsubscripted plain variables [115, Chapter 8]. This feature facilitates expressing the requirement that an iteration must make progress, needed without exception when setting iterations up. In fact, that is the only use we will make of 0-subscripted variables.

Note that, in *Prg* above, we have implicitly assumed that X is a finite set since this is necessary for relation $\supset$ to be well-founded –or, more specifically, as some authors prefer, left-well-founded–.

A summary of the refinement of (3.1) follows:

$$\begin{array}{l} S : [\ true\ ,\ S = R^+\ ] \\ \sqsubseteq \quad \left\{ \begin{array}{l} \text{introduce local block and initialised iteration} \\ \text{according to discussion above} \end{array} \right\} \\ \quad |[\ \mathbf{var}\ A : \mathrm{Vec}\, X\ ; \\ \quad\ S, A := R, \emptyset\ ; \\ \quad\ \mathbf{do}\ A \neq X\ \rightarrow\ S, A : [\ A \neq X\ \wedge\ Inv\ ,\ Inv\ \wedge\ Prg\ ]\ \mathbf{od} \\ \quad]| \end{array}$$

Developing the Iteration Body We now need to refine the specification statement left above as body of the iteration. The postcondition states, in *Prg*, that progress must be guaranteed by adding to A elements which are not members of it. We thus explore the statement $A := A \cup x$, with x an element such that $x \subseteq \overline{A}$, analysing how variable S must be simultaneously

updated in order to preserve the invariant. Note that the guard is equivalent to $\overline{A}$ being non-empty and, hence, it guarantees the existence of the required x. Assume that *Inv* holds and then manipulate thus:

$$
\begin{array}{ll}
 & ((R \cdot \not\in A)^* \cdot R)\,[A := A \cup x] \\
= & \quad \{ \text{ substitution } \} \\
 & (R \cdot \not\in (A \cup x))^* \cdot R \\
= & \quad \left\{ \begin{array}{l} \text{distribution of } \not\in \text{ over union (2.38)} \\ \text{and of composition over union (2.14)} \end{array} \right\} \\
 & (R \cdot \not\in A \ \cup \ R \cdot \not\in x)^* \cdot R \\
= & \quad \{ \text{ closure over union (2.69) with } R, S := R \cdot \not\in A,\ R \cdot \not\in x \ \} \\
 & ((R \cdot \not\in A)^* \cdot R \cdot \not\in x)^* \cdot (R \cdot \not\in A)^* \cdot R \\
= & \quad \{ \ Inv \ \} \\
 & (S \cdot \not\in x)^* \cdot S \\
= & \quad \{ \text{ singleton coreflexives (2.42), closure (2.71) } \} \\
 & (id \ \cup \ S \cdot x \cdot x^\circ) \cdot S \\
= & \quad \{ \text{ distributiou of composition over union (2.14) } \} \\
 & S \ \cup \ S \cdot x \cdot x^\circ \cdot S \ .
\end{array}
$$

This calculation shows that the following statement maintains the invariant while guaranteeing progress, provided x is not a member of A:

$$S, A \ := \ S \ \cup \ S \cdot x \cdot x^\circ \cdot S \,,\ A \cup x \ .$$

Therefore, we have refined the iteration body as follows:

$$
\begin{array}{ll}
 & S, A : [\ A \neq X \ \wedge \ Inv \ ,\ Inv \ \wedge \ Prg \ ] \\
\sqsubseteq & \quad \left\{ \begin{array}{l} \text{introduce local block and sequential composition} \\ \text{according to discussion above} \end{array} \right\} \\
 & [\![\ \textbf{var} \ x : X ; \\
 & \quad x :\subseteq \overline{A} \, ; \\
 & \quad S, A \ := \ S \ \cup \ S \cdot x \cdot x^\circ \cdot S \,,\ A \cup x \\
 &]\!]
\end{array}
$$

This completes the algorithmic refinement of specification (3.1). After a development like the one we have carried out, assembling the whole code is not necessary for human understanding of the algorithmic solution. The *documented development*, as e.g. the contents of this section, is what we need for that purpose. –See Morgan's remarks on documentation, testing and debugging in [115, Chapter 19]. We also want to point out that, of course, assembling the code is nevertheless needed, if only for the need to submit it to a computer for execution.–

```
|[ var S : X ← X ;
   |[ var A : Vec X ;
      S, A := R, ∅ ;
      do A ≠ X →
           |[ var x : X ;
              x :⊆ Ā ;
              S, A := S ∪ S · x · x° · S , A ∪ x
           ]|
      od
   ]|
]|
```

Figure 3.4: First Algorithmic Solution of (3.1)

In the next subsection, our program will be further refined due to some efficiency considerations. Such refinement will be more readily visualised using the collected whole program. For that reason –and maybe also out of the bad habit of wanting to see the whole program?–, we present the program in Figure 3.4.

Data Refinement The program we have developed requires computation of the complement of A in the body of the iteration time and time again. This can be avoided by introducing a fresh variable $B : \mathsf{Vec}\, X$ and keeping it updated in such a way that it holds the value of $\overline{A}$ at every point of the program. Such an incorporation of new variables can be carried out by means of data refinement using so-called coupling invariants: predicates that relate new variables to old existing variables –see e.g. [115, Chapter 17] or [116]–. In our case, variable B is introduced via the following coupling invariant:

$$CI \;:=\; B = \overline{A} \;.$$

Having defined CI, the transformation of the program can be conducted in a fairly mechanical way. We explain it briefly.

Assignments to B are attached to every assignment of the program in such a way that CI is maintained. Inspecting Figure 3.4, it can be seen that we only need to add $B := X$ to the initialisation statement and $B := B - x$ to the assignment within the iteration. Having enforced the validity of CI in this way, the non-deterministic selection statement can be correctly replaced by $x :\subseteq B$, thereby eliminating the unwanted repeated evaluation of the complement of A. Efficiency can also be further improved by replacing the guard with $B \neq \emptyset$, again a correct transformation due to the validity of CI

```
|[ var S : X ← X ;
   |[ var B : Vec X ;
      S, B := R, X ;
      do B ≠ ∅ →
         |[ var x : X ;
            x :⊆ B ;
            S, B := S ∪ S · x · x° · S , B − x
         ]|
      od
   ]|
]|
```

Figure 3.5: Second Algorithmic Solution of (3.1), after Data Refinement

at every point of the program. We claim this new guard is more efficient since most implementations of sets perform comparisons to the empty set more rapidly than comparisons to the universal set. After all these transformations, variable A ends up being *useless*, in the sense that it is only used for computing new values of itself. It can thus be eliminated. The resulting program is shown in Figure 3.5.

–This second program could have been obtained directly had we initially rephrased the postcondition as $S = (R \cdot \overline{\phi\emptyset})^* \cdot R$, instead of as indicated by (3.3), and replaced constant $\emptyset$ by a variable. The choice of replacing constant X seemed more natural at the time. Also, it provided us with a good opportunity to show a simple example of data refinement.–

The particular kind of data refinement we have carried out is also known as "finite differencing" [122] or "formal differentiation" [139], a program refinement technique that originated from optimising transformations used in compilers. This technique is also presented in [68, Section 19.2], where it is proposed as a transformatiou that can improve the efficiency of an already correctly developed program –the use we have made of it–, thereby promoting the principle of separating the correctness and efficiency concerns. We will use this technique again in subsequent chapters.

Further Refinement and Warshall's Algorithm RELVIEW, briefly mentioned in Section 1.2, is a programming system in which imperative programs that manipulate variables of type "relation" can be directly executed. Thus, the program in Figure 3.5 can be straightforwardly translated into an executable RELVIEW program. But its translation into a conventional imperative programming language would not be as direct, since this would

require data-refining relation S and set B to the sort of data structures traditionally provided by such languages. This kind of data refinement would allow us to obtain Warshall's algorithm as it is commonly known. We will not get into all the details involved in the transformation of our abstract program into Warshall's algorithm, but only comment ou the most crucial aspect of it.

Relation S must be data-refined to a boolean matrix M indexed by elements of $X \times X$, where $M[i,j]$ is *true* if and only if $i \langle S \rangle j$ holds. In such a case, the assignment to S in the iteration body becomes a *simultaneous* assignment to all the elements of M. However, it can be proved that all the atomic assignments involved in such a simultaneous assignment are *independent*, in the sense that, should such atomic assignments be arbitrarily serialised, the whole outcome remains the same. A proof of this fact, developed under a more general algebraic framework of which our reachability problem is just a particular instance, can be found in [12, Section 6.2]. –Reference [12] is further discussed in the final section of this chapter.– A particular serialisation of the assignments to the elements of M corresponds to Warshall's algorithm, which can be obtained from our abstract program on account of the crucial aforementioned "independence of atomic assignments" fact.

3.2 Fixed-Source Reachability

The Fixed-Source Reachability problem is phrased, for directed graphs, as follows: given a graph with successor relation $Succ : Vert \leftarrow Vert$ and given a set of vertices $V : \mathsf{Vec}\ Vert$, compute the set of vertices reached from V. This means computing $Reach \cdot V : \mathsf{Vec}\ Vert$ where $Reach$, as defined by (2.83) in Section 2.6, is $Succ^*$.

Again, the problem can be posed only in terms of closure: given a relation $R : X \leftarrow X$ and a set $A : \mathsf{Vec}\, X$, compute $R^* \cdot A : \mathsf{Vec}\, X$.

Specification We formalise the specification thus:

$$
\begin{array}{l}
[\![\ \mathbf{var}\ B : \mathsf{Vec}\, X ; \\
\quad B : [\, true\ ,\ B = R^* \cdot A \,] \\
]\!]
\end{array}
\tag{3.6}
$$

where $R : X \leftarrow X$ and $A : \mathsf{Vec}\, X$ are given relations.

Playing with Fixed Points To proceed with the algorithmic refinement of (3.6), we will manipulate a good deal the fixed-point expression of $R^* \cdot A$, i.e. the expression $\langle \mu W : A \cup R \cdot W \rangle$.

Let us take relation R as an implicit parameter and define function f as corresponding to the above expression when applied to argument A:

$$f\,C \;:=\; \langle \mu\, W : C \cup R \cdot W \rangle \;. \tag{3.7}$$

Now, our postcondition reads $B = f\,A$.

Some properties of f can now be deduced using the calculus of fixed points presented in Section 2.3. First, we have:

$$f\,\emptyset \;=\; \langle \mu\, W : R \cdot W \rangle \;=\; \emptyset \;,$$

which follows from the fact that $W := \emptyset$ solves the equation $R \cdot W = W$ and, since $\emptyset$ is the least relation, it must then be the least solution of the equation. Also, we have:

$$\begin{array}{ll} & f\,C \\ = & \quad \{\ \text{definition of } f\ (3.7)\ \} \\ & \langle \mu\, W : C \cup R \cdot W \rangle \\ = & \quad \left\{ \begin{array}{l} \text{rolling rule (2.57) with} \\ \quad \mathcal{F}\,W := C \cup W\,,\ \mathcal{G}\,W := R \cdot W \end{array} \right\} \\ & C \;\cup\; \langle \mu\, W : R \cdot (C \cup W) \rangle \\ = & \quad \left\{ \begin{array}{l} \text{distribution of composition over union (2.14),} \\ \text{definition of } f\ (3.7) \end{array} \right\} \\ & C \;\cup\; f\,(R \cdot C) \;. \end{array}$$

Hence, we have proved the following two properties:

$$f\,\emptyset \;=\; \emptyset \;, \tag{3.8}$$

$$f\,C \;=\; C \cup f\,(R \cdot C) \;. \tag{3.9}$$

We could then propose as invariant:

$$B \cup f\,C \;=\; f\,A \;,$$

using a freshly introduced program variable C. Such an invariant would be, first, coupled with guard $C \neq \emptyset$ because of (3.8), second, established initially by the statement $B, C := \emptyset, A$ and, third, maintained by the statement $B, C := B \cup C,\ R \cdot C$ because of (3.9).

This is all too nice but, if there are cycles in the graph modelled by R, we have a non-terminating iteration. Take, e.g., R and A to be, respectively, $\{(x,x)\}$ and $\{x\}$, in which case variable C will invariantly hold the value $\{x\}$ and the guard will always be true. It is not surprising that we ended up with a non-terminating iteration since no variant was considered at all. In fact, it is very often useful to take, as first hint for the construction

of the body of an iteration, statements that decrease –according to some well-founded relation– the variant in order to guarantee progress. We have nevertheless showed this first attempt at a solution to learn from it, as well as to illustrate the kind of manipulation of fixed points we will use to arrive at a correct solution.

Above, variable B was meant to *increasingly* accumulate the elements of $f\,A$. Hence, a reasonable candidate for variant could have been B, with progress guaranteed by its increase. But, above, we had no guarantee that the elements added to B in each iteration were actually increasing the size of B. The problem is that function f does not know about elements that, having already been considered, do not need to be taken into account anymore. Let us define a function g that suitably generalises function f in such a manner:

$$g(B,C) \;:=\; \langle \mu W : C \cup (R \cdot W - B) \rangle \;. \tag{3.10}$$

We will refer to the first argument of g as the set of *"already generated"* elements, and to the second argument as the set of *"seed"* elements of the R-generative process.

Let us now play the game of analysing interesting properties of function g and, in particular, its relation to function f. To start with, an empty set of already generated elements takes us back to function f:

$$g(\emptyset,C) \;=\; f\,C \;, \tag{3.11}$$

since $V - \emptyset = V$ for any relation V. Now, an empty set of seed elements gives us the following:

$$g(B,\emptyset) \;=\; \langle \mu W : R \cdot W - B \rangle \;=\; \emptyset \;,$$

where, as before when analysing function f, we have that $W := \emptyset$ is a solution of $R \cdot W - B = W$ and it must then he its least solution. After considering an empty set of seeds, let us analyse the case when we can take some of the seeds out of the R-generative process and leave the rest for later consideration:

$$\begin{array}{ll} & g\,(B\,,\; C1 \cup C2) \\ = & \quad \{ \text{ definition of } g \text{ (3.10) } \} \\ & \langle \mu W : C1 \cup C2 \cup (R \cdot W - B) \rangle \\ = & \quad \left\{ \begin{array}{l} \text{–discharge } C1 \text{ from the rest of the generative process–} \\ \text{union/subtraction: } U \cup V = U \cup (V - U) \\ \qquad\qquad \text{with } U,V := C1\,,\; C2 \cup (R \cdot W - B) \end{array} \right\} \\ & \langle \mu W : C1 \cup ((C2 \cup (R \cdot W - B)) - C1) \rangle \end{array}$$

$$
\begin{array}{ll}
= & \left\{ \begin{array}{l} \text{fixed-point rolling rule (2.57) with} \\ \quad \mathcal{F}\,W \;:=\; C1 \cup W\,, \\ \quad \mathcal{G}\,W \;:=\; (C2 \cup (R \cdot W - B)) - C1 \end{array} \right\} \\
 & C1 \;\cup\; \langle \mu W : (C2 \cup (R \cdot (C1 \cup W) - B)) - C1 \rangle \\
= & \quad \{ \text{ distribution of subtraction over union (2.12) } \} \\
 & C1 \;\cup\; \langle \mu W : (C2 - C1) \;\cup\; (R \cdot (C1 \cup W) - B - C1) \rangle \\
= & \quad \{ \text{ subtraction/union (2.13) } \} \\
 & C1 \;\cup\; \langle \mu W : (C2 - C1) \;\cup\; (R \cdot (C1 \cup W) - (B \cup C1)) \rangle \\
= & \left\{ \begin{array}{l} \text{distribution of composition over union (2.14)} \\ \text{and of subtraction over union (2.12)} \end{array} \right\} \\
 & C1 \;\cup\; \langle \mu W : (C2 - C1) \;\cup\; (R \cdot C1 - (B \cup C1)) \\
 & \qquad\qquad\qquad \cup\; (R \cdot W \;- (B \cup C1)) \rangle \\
= & \quad \{ \text{ definition of } g \text{ (3.10) } \} \\
 & C1 \;\cup\; g(B \cup C1\,,\, (C2 - C1) \cup (R \cdot C1 - (B \cup C1)))\;.
\end{array}
$$

In summary, we have:

$$g(B,\emptyset) \;=\; \emptyset\;, \tag{3.12}$$

$$g(B,\, C1 \cup C2) \;= \quad C1 \;\cup\; g(B \cup C1\,,\, (C2 - C1) \cup (R \cdot C1 - (B \cup C1)))\;. \tag{3.13}$$

Setting Up an Iteration Having analysed some of the properties of function g, we start the refinement of specification (3.6) to an iteration. From postcondition $B = f\,A$, we now postulate an invariant that uses a fresh program variable C and that also uses function g to control elements already generated and accumulated in variable B. We take:

$$Inv \;:=\; B \cup g(B,C) \;=\; f\,A\;.$$

From property (3.12) it can be seen that an adequate guard is $C \neq \emptyset$. From property (3.11) it follows that statement $B, C := \emptyset, A$ can be used to establish the invariant initially. Since the contents of variable B should grow, starting from $\emptyset$, until it keeps the whole set $f\,A$, we take as variant expression B with progress guaranteed by its augmentation. As in Section 3.1, we use a shorthand for the proposition that states that the iteration is making progress:

$$Prg \;:=\; (B \supset B_0)\;.$$

Note that set X has again been assumed to be finite, which means we are only considering graphs with a finite amount of vertices. This makes $\supset$ in Prg above to be well-founded.

Let us summarise the refinement of statement (3.6) so far:

$$
\begin{array}{ll}
 & B : [\ \mathit{true}\ ,\ B = R^* \cdot A\ ] \\
\sqsubseteq & \left\{ \begin{array}{l} \text{introduce local block and initialised iteration} \\ \text{according to discussion above} \end{array} \right\} \\
 & |[\ \ \mathbf{var}\ C : \mathsf{Vec}\, X\ ; \\
 & \quad B, C := \emptyset, A\ ; \\
 & \quad \mathbf{do}\ C \neq \emptyset\ \rightarrow\ B, C : [\ C \neq \emptyset\ \wedge\ \mathit{Inv}\ ,\ \mathit{Inv}\ \wedge\ \mathit{Prg}\ ]\ \mathbf{od} \\
 &]|
\end{array}
$$

Developing the Iteration Body To refine the body of the iteration, progress must be achieved by adding elements to B which are not in it. Instead of using the addition of one element to B, we will go for the addition of a non-empty set of elements to B and we will later refine such a general statement to more specific ones. Thus, we will consider the statement $B := B \cup D$ where set D is not empty and does not share elements with B. Let us name these assumptions: (i) $D \neq \emptyset$ and (ii) $B \cap D = \emptyset$. To analyse the maintenance of the invariant under such an assignment, we assume a simultaneous assignment to variables B and C will do the work. The expression to be assigned to C will be calculated in the process of proving maintenance of the invariant. Assume Inv holds and then proceed thus:

$$
\begin{array}{ll}
 & \mathit{Inv}\,[\,B, C := B \cup D, \mathit{Exp}\,] \\
\equiv & \quad \{\ \text{substitution}\ \} \\
 & B\ \cup\ D\ \cup\ g\,(B \cup D, \mathit{Exp})\ =\ f\,A \\
\equiv & \quad \{\ \mathit{Inv}\ \} \\
 & B\ \cup\ D\ \cup\ g\,(B \cup D, \mathit{Exp})\ =\ B\ \cup\ g\,(B, C) \\
\Leftarrow & \quad \{\ \text{Leibniz}\ \} \\
 & D\ \cup\ g\,(B \cup D, \mathit{Exp})\ =\ g\,(B, C) \\
\equiv & \left\{ \begin{array}{l} \text{assume (iii) } D \subseteq C\text{; hence, } C = D \cup C\ , \\ \text{interaction of function } g \text{ with union (3.13)} \\ \text{with } B, C1, C2 := B, D, C \end{array} \right\} \\
 & D\ \cup\ g\,(B \cup D, \mathit{Exp})\ = \\
 & \quad D\ \cup\ g\,(B \cup D, (C - D) \cup (R \cdot D - (B \cup D))) \\
\Leftarrow & \quad \{\ \text{Leibniz}\ \} \\
 & \mathit{Exp}\ =\ (C - D) \cup (R \cdot D - (B \cup D))\ .
\end{array}
$$

In the process of calculating the body of the iteration, three assumptions were imposed on D, viz. (i), (ii) and (iii). Assumption (ii) follows from assumption (iii) since it can be shown that $B \cap C = \emptyset$ is an additional invariant of the obtained iteration –proof omitted–.

```
|[ var B : Vec X ;
   |[ var C : Vec X ;
      B, C := ∅, A ;
      do C ≠ ∅ →
           |[ var D : Vec X ;
              D : [ C ≠ ∅ , D ≠ ∅ ∧ D ⊆ C ] ;
              B, C := B ∪ D , (C − D) ∪ (R · D − (B ∪ D))
           ]|
      od
   ]|
]|
```

Figure 3.14: General Algorithmic Solution of (3.6)

Summarising, the body of the iteration has been refined as follows:

$$
\begin{array}{ll}
 & B, C : [\, C \neq \emptyset \wedge \mathit{Inv} \;,\; \mathit{Inv} \wedge \mathit{Prg} \,] \\
\sqsubseteq & \left\{ \begin{array}{l} \text{introduce local block and sequential composition} \\ \text{according to discussion above} \end{array} \right\} \\
 & \quad |[\;\; \textbf{var}\;\; D : \mathsf{Vec}\, X \,; \\
 & \quad\;\; D : [\, C \neq \emptyset \,,\; D \neq \emptyset \wedge D \subseteq C \,] \,; \\
 & \quad\;\; B, C := B \cup D \,,\, (C - D) \cup (R \cdot D - (B \cup D)) \\
 & \quad]|
\end{array}
$$

We collect the whole code in Figure 3.14. This general algorithmic solution can be further refined by choosing specific ways of drawing set D from set C. We will soon proceed to do so.

Other Alternatives Similar algorithms can be obtained from slightly different generalisations of function f. The reader is invited to reproduce the above development using the following function h instead of g:

$$h\,(B, C) \;:=\; \langle\, \mu W : (C \cup R \cdot W) - B \,\rangle \;.$$

Refinement to Singleton-Selection Version One way of further refining the general algorithmic solution is by choosing D to be a singleton set, i.e. a point. An elementary data refinement can be carried out using coupling invariant $x = D$ to replace variable D with a variable x of type X. Requirement $x \neq \emptyset$ is then guaranteed to hold by the type of x since points are non-empty. The specification statement in the iteration body of

```
|[ var B : Vec X ;
   |[ var C : Vec X ;
      B, C := ∅, A ;
      do C ≠ ∅ →
           |[ var x : X ;
              x :⊆ C ;
              B, C := B ∪ x, (C − x) ∪ (R · x − (B ∪ x))
           ]|
      od
   ]|
]|
```

Figure 3.15: Singleton-Selection Solution of (3.6)

the general solution then becomes the non-deterministic selection statement $x :\subseteq C$. The result is presented in Figure 3.15.

Note that the only use the program makes of the successor relation R of the graph is through the expression $R \cdot x$, which denotes the set of successor vertices of vertex x. This suggests the use of an adjacency-lists –or successor lists– representation for the graph.

The only thing still left to do in order to implement our algorithm in a conventional programming language is to data-refine the sets to some representation. Lists can be used for this purpose, assuming all the sets are finite. In such a case, the order in which vertices are kept in variables B and C can determine whether the graph is traversed in a depth-first fashion, a breadth-first fashion, or some other kind.

Refinement to Whole-Selection Version Another possible refinement of the general solution is obtained by choosing set D to be the whole of set C, i.e. by refining the specification statement affecting D to $D := C$. In this case variable D can be rendered useless by using C where D is used. D can thus be eliminated. The solution obtained in this case is shown in Figure 3.16; it corresponds to traversing the graph in a layered breadth-first fashion.

3.3 Related Work

The closure problems presented in this chapter have been extensively treated by others before. Our treatment has been included mainly as a warming-

```
|[ var B : Vec X ;
   |[ var C : Vec X ;
      B, C := ∅, A ;
      do C ≠ ∅ → B, C := B ∪ C, R · C − (B ∪ C) od
   ]|
]|
```

Figure 3.16: Whole-Selection Solution of (3.6)

up exercise that familiarises the reader with our style of development. We dedicate this section to review previous presentations by other authors.

The all-pairs reachability problem is treated in [12, 16, 54, 136]. Backhouse et al. [12] make use of a calculational framework based on regular algebra within which graphs are modelled by –adjacency– matrices. The more general setting of regular algebra allows them to treat all-pairs reachability as an instance of a general problem that also covers the computation of minimum paths between all pairs of vertices in a weighted graph. Moreover, our whole derivation is nearly a step-by-step instance of the more general derivation they present. The key properties of the closure operator of regular algebra they exploit were first used by Backhouse and Carré in a much broader study of path-finding problems [10]. In fact, the essence of the derivation of Backhouse et al. in [12] is the same as that of Backhouse and Carré in [10], except for the use made in the former of more modern calculational techniques for constructing imperative programs. Regarding other authors, Berghammer's construction [16] of Warshall's algorithm is very similar to ours, and so is the presentation of Feijs and van Ommering [54] except for the fact that they do not restrict progress of the iteration to singletons. The treatment of Feijs and van Ommering of all-pairs reachability is then, in that respect, similar to our treatment of fixed-source reachability, where a general algorithmic solution is first arrived at and then further refined to more specific solutions. Finally, in [136, Section 3.2], Schmidt and Ströhlein present a correctness proof of Warshall's algorithm using the calculational framework of binary relations, but their presentation does not make explicit use of standard techniques for the development of imperative programs.

The fixed-source reachability problem is treated in [12, 16, 105, 133]. As it was the case for all-pairs reachability, Backhouse et al. [12] treat, within their setting of regular algebra, fixed-source reachability as an instance of a more general problem that also covers the computation of minimum paths between a single source vertex and all other vertices of a weighted graph. We will achieve the same level of generality and a little more in the next chapter,

where we deal with a generic problem of computing so-called representatives, of which fixed-source reachability is an instance. Backhouse et al. obtain both depth-first and breadth-first traversal algorithms that correspond to our singleton-selection solution, but no layered breadth-first traversal like our whole-selection solution since they restrict their derivation to selection of singletons. Berghammer [16], on the other hand, uses the framework of binary relations but restricts the algorithmic solution to a layered breadth-first traversal. Möller [105] and Russling [133] also derive only a layered breadth-first traversal algorithm, but within a calculational framework of n-ary relations. Russling's treatment is more general since it covers a "class of layer-oriented graph algorithms" that includes the fixed-source reachability problem and the fixed-source shortest paths problem, both of which will be accounted for in our general treatment of computing representatives in the next chapter.

Out of the realm of imperative programming, Bird and de Moor derive in [28, Section 6.7] a functional-style algorithm for computing closure that corresponds to fixed-source reachability. Our use of fixed points is based on their treatment of closure. Also, building on the *fold-unfold* categorical approach to datatypes within functional programming –which we briefly met in Section 2.8, see e.g. [98, 101]–, Gibbons and Jones present in [66, 80] calculational derivations of functional algorithms that compute breadth-first traversals of trees. Such traversals could be adaptable to cater for graphs under a representation of graphs as infinite trees, an idea functional programming researchers seem to like to play with –see e.g. [53, Section 2.2.2] and [102, Section 4.4]–.

Chapter 4
Computing Representatives

The contents of this chapter deal with a general problem of computing sets of representative elements selected from among a given set of candidates. Our interest in the problem of computing representatives stems from the fact that a few graph problems can be specified as instances of it. This chapter presents a development of general algorithmic solutions to the problem of computing representatives, and the instantiation of such general solutions to specific graph algorithms. The results contained in this chapter were partially reported in [30].

Selection of representative elements is aided by two relations Q and R on the set of candidates. The first, Q, is an equivalence relation that partitions the set of candidates in equivalence classes. We will be interested in choosing one representative element from each of such classes. The second relation, R, is a preorder that determines which candidates can represent their class. A representative element must be a maximum under R of its class. The set of candidates is specified in terms of a third relation S. The whole set of candidates is generated with the reflexive-transitive closure of S from a given initial set of candidates.

Section 4.1 presents a formalisation of the notion of representatives, as well as a formal specification of the general computational problem we will be dealing with. Section 4.2 prepares the ground for the refinement of our problem to an algorithm by exploring some properties of the formal notions involved, viz. the selection of representatives and the generation of candidates by closure. Section 4.3 then presents the actnal derivation of a general algorithmic solution and, mimicking the obtention of the two final solutions to the fixed-source reachability problem in pages 61-62 of Section 3.2, this general solution is further refined in Section 4.4 to a singleton-selection version and a whole-selection version. We then dedicate ourselves to the application of the obtained algorithmic solutions to graph problems. Section

4.5 treats the problem of computing minimum paths in a weighted graph; the well-known algorithm due to Dijkstra [43] is obtained as an instance of the singleton-selection solution. Section 4.6 treats the problem of computing shortest paths in an unweighted graph; unlike the analogue problem of weighted paths, both the singleton-selection and the whole-selection solutions are applicable in this case. Section 4.7 deals again with the fixed-source reachability problem, showing that the treatment of this problem presented in Section 3.2 can be obtained as an instance of our treatment of the representatives problem. Section 4.8 closes the chapter by reviewing related work.

4.1 Specification

This section presents a formalisation of the notion of representatives, and a specification of the problem of computing representatives drawn from a set of candidates generated by closure.

As mentioned earlier, selection of representatives is defined in terms of an equivalence relation Q and a preorder R, both of type $X \leftarrow X$ with X the type of candidates. Let B and C be sets of elements of X represented by vectors $-B, C : \mathsf{Vec}\, X-$. We say that B is a set of representatives for set C if the following holds:

$$B \subseteq C \quad \wedge \quad C \subseteq (Q \cap R) \cdot B \quad \wedge \quad B \cdot B^{\circ} \cap Q \subseteq id \ . \qquad (4.1)$$

In words, the first conjunct says that all representatives, i.e. all the elements in B, are drawn from C. The second conjunct says that every element in C is represented by some element in B, with representation of an element c by an element b meaning that they are equivalent under Q and that c is at most b under R. Finally, the third conjunct says that the set of representatives must have at most one element per Q-equivalence class. Combining all, set B contains exactly one element per Q-class of C and such elements are R-maxima of their respective classes.

Henceforth we will use $B \trianglelefteq C$ to stand for the first two conjuncts of (4.1); it may be read as "B is a *thinning* of C". Both Q and R are preorders and, since intersection preserves preorders, so is $Q \cap R$. This implies that $\trianglelefteq$ is reflexive and transitive as well. We also introduce predicate *uniq*, defined such that *uniq* B stands for the third conjunct of (4.1); we call it the *uniqueness* predicate.

The set of candidates from which representatives are to be drawn must also be specified. We take it to be generated by repeated iteration of a relation $S : X \leftarrow X$ from a given initial set $A : \mathsf{Vec}\, X$. In other words, the set of

candidates is the existential image of A under S^*, just like the set that had to be computed in Section 3.2.

Specification Finally, we formalise the problem of computing representatives of a set generated by closure. The specification reads thus:

$$
\begin{array}{l}
[\![\ \textbf{var}\ B : \mathsf{Vec}\, X\,; \\
\quad B : [\ \mathit{true}\ ,\ \mathit{uniq}\, B\ \wedge\ B \trianglelefteq S^* \cdot A\ ] \\
]\!]
\end{array}
\tag{4.2}
$$

where $Q, R, S : X \leftarrow X$ –noting that Q and R are implicitly used in *uniq* and $\trianglelefteq$– and $A : \mathsf{Vec}\, X$ are given relations.

4.2 Exploring Some Properties

Construction of an algorithm that solves our problem requires manipulation of the formal notions involved. In this section we explore the properties of such notions and then proceed with the actual derivation of an algorithm in the next section.

The Uniqueness Predicate First, let us recall that *uniq* was defined as follows:

$$\mathit{uniq}\, B \ :=\ B \cdot B^\circ \cap Q \subseteq \mathit{id}\ . \tag{4.3}$$

Thus, a set satisfies *uniq* if and only if it contains at most one element per Q-equivalence class. Among the sets that can be guaranteed to satisfy *uniq* are included the empty set and singleton sets, the latter being represented by points, i.e. functions:

$$\mathit{uniq}\, \emptyset\ , \tag{4.4}$$

$$\mathit{uniq}\, B \quad \text{provided } B \text{ is a function}\ . \tag{4.5}$$

The empty relation and all functions are simple relations, i.e. if W is either $\emptyset$ or a function then $W \cdot W^\circ \subseteq \mathit{id}$. Both (4.4) and (4.5) follow from this fact.

Knowing that *uniq* holds for small sets, one could explore under what conditions bigger sets satisfying the uniqueness property can be built. Let us analyse the case of the union of two sets:

$$
\begin{array}{ll}
 & \mathit{uniq}\, (B \cup C) \\
\equiv & \quad \{\ \text{definition of } \mathit{uniq}\ (4.3)\ \} \\
 & (B \cup C) \cdot (B \cup C)^\circ \cap Q \subseteq \mathit{id}
\end{array}
$$

$$
\begin{array}{ll}
 & (B \cdot B^\circ \cap Q) \cup (B \cdot C^\circ \cap Q) \\
 & \quad \cup (C \cdot B^\circ \cap Q) \cup (C \cdot C^\circ \cap Q) \subseteq id \\
\equiv & \quad \{ \text{ universal property of union (2.3) } \} \\
 & (B \cdot B^\circ \cap Q \subseteq id) \ \wedge \ (B \cdot C^\circ \cap Q \subseteq id) \\
 & \quad \wedge \ (C \cdot B^\circ \cap Q \subseteq id) \ \wedge \ (C \cdot C^\circ \cap Q \subseteq id) \\
\equiv & \quad \left\{ \begin{array}{l} \text{definition of } uniq \text{ (4.3), twice; second and third} \\ \text{conjuncts are equivalent by properties of converse} \\ \text{–(2.16), (2.17), (2.18)– and by symmetry of } Q \end{array} \right\} \\
 & uniq\, B \ \wedge \ uniq\, C \ \wedge \ (B \cdot C^\circ \cap Q \subseteq id) \ .
\end{array}
$$

Thus, we have derived the rule:

$$uniq\,(B \cup C) \ \equiv \ uniq\, B \ \wedge \ uniq\, C \ \wedge \ (B \cdot C^\circ \cap Q \subseteq id) \ . \tag{4.6}$$

In words, union preserves uniqueness if the Q-classes present in both sets are represented by the same element.

The Thinning Relation Thinning, denoted by $\trianglelefteq$, was defined as:

$$B \trianglelefteq C \ := \ B \subseteq C \ \wedge \ C \subseteq (Q \cap R) \cdot B \ . \tag{4.7}$$

As was done for the uniqueness predicate, we analyse interaction of the thinning relation with union. Specifically, we analyse monotonicity of union with respect to $\trianglelefteq$. We proceed thus:

$$
\begin{array}{ll}
 & D \cup B \trianglelefteq D \cup C \\
\equiv & \quad \{ \text{ definition of } \trianglelefteq \text{ (4.7) } \} \\
 & D \cup B \subseteq D \cup C \ \wedge \ D \cup C \subseteq (Q \cap R) \cdot (D \cup B) \\
\equiv & \quad \{ \text{ distribution of composition over union (2.14) } \} \\
 & D \cup B \subseteq D \cup C \ \wedge \ D \cup C \subseteq (Q \cap R) \cdot D \cup (Q \cap R) \cdot B \\
\Leftarrow & \quad \left\{ \begin{array}{l} \text{monotonicity of union with respect to} \\ \text{inclusion, relation } Q \cap R \text{ is reflexive} \end{array} \right\} \\
 & B \subseteq C \ \wedge \ C \subseteq (Q \cap R) \cdot B \\
\equiv & \quad \{ \text{ definition of } \trianglelefteq \text{ (4.7) } \} \\
 & B \trianglelefteq C \ .
\end{array}
$$

Hence, union is indeed monotonic with respect to thinning:

$$D \cup B \trianglelefteq D \cup C \ \Leftarrow \ B \trianglelefteq C \ . \tag{4.8}$$

Property (4.8) is all too nice, but we can do better than that. To explain what we mean by "better", let us do a little operational reading of (4.8): to take a thinning of $D \cup C$, it suffices to thin C while leaving D intact. Now, if we are to leave D intact, one would want to be able to thin C ignoring those Q-classes in C which are present in D. In other words, if $B \trianglelefteq C - Q \cdot D$, under what conditions $D \cup B \trianglelefteq D \cup C$ does hold? Let us explore this:

$$
\begin{array}{ll}
 & D \cup B \trianglelefteq D \cup C \\
\equiv & \quad \{ \text{ repeat of the first two steps of the proof of (4.8) above } \} \\
 & D \cup B \subseteq D \cup C \ \wedge \ D \cup C \subseteq (Q \cap R) \cdot D \cup (Q \cap R) \cdot B \\
\equiv & \quad \{ \text{ universal property of union (2.5), twice } \} \\
 & D \subseteq D \cup C \ \wedge \ B \subseteq D \cup C \\
 & \wedge \ D \subseteq (Q \cap R) \cdot D \cup (Q \cap R) \cdot B \\
 & \wedge \ C \subseteq (Q \cap R) \cdot D \cup (Q \cap R) \cdot B \\
\equiv & \quad \{ \text{ union, } Q \cap R \text{ reflexive: first and third conjunct hold } \} \\
 & B \subseteq D \cup C \ \wedge \ C \subseteq (Q \cap R) \cdot D \cup (Q \cap R) \cdot B \\
\equiv & \quad \left\{ \begin{array}{l} \text{–expression } C - Q \cdot D \text{ needs to be included–} \\ \text{complementation: } W = (W \cap V) \cup (W - V) \\ \qquad\qquad \text{with } W, V := C, Q \cdot D \end{array} \right\} \\
 & B \subseteq D \cup (C \cap Q \cdot D) \cup (C - Q \cdot D) \\
 & \wedge \ (C \cap Q \cdot D) \cup (C - Q \cdot D) \subseteq (Q \cap R) \cdot D \cup (Q \cap R) \cdot B \\
\Leftarrow & \quad \left\{ \begin{array}{l} \text{first conjunct: union; for the second conjunct} \\ \text{we claim: } C \cap Q \cdot D \subseteq (Q \cap R) \cdot D \end{array} \right\} \\
 & B \subseteq C - Q \cdot D \ \wedge \ C - Q \cdot D \subseteq (Q \cap R) \cdot B \\
\equiv & \quad \{ \text{ definition of } \trianglelefteq \text{ (4.7) } \} \\
 & B \trianglelefteq C - Q \cdot D \ .
\end{array}
$$

And we can obtain a proviso on C, D from the claim:

$$
\begin{array}{ll}
 & C \cap Q \cdot D \\
\subseteq & \quad \{ \text{ Dedekind's rule (2.20) } \} \\
 & (C \cdot D^\circ \cap Q) \cdot D \\
\subseteq & \quad \{ \text{ assume } C \cdot D^\circ \subseteq R \ \} \\
 & (Q \cap R) \cdot D \ .
\end{array}
$$

Hence, the following rule holds:

$$D \cup B \trianglelefteq D \cup C \ \Leftarrow \ B \trianglelefteq C - Q \cdot D \quad \text{provided } C \cdot D^\circ \subseteq R \, . \tag{4.9}$$

Generalising Closure The properties of the closure operator we will use in relation to the representatives problem require a suitable generalisation of closure. This generalisation closely resembles the one used for the fixed-source reachability problem in Section 3.2 –see page 58–. As in that section, if function f is defined as follows:

$$f\,C \;:=\; (\mu W : C \cup S \cdot W) \;, \tag{4.10}$$

it then holds that $f\,A = S^* \cdot A$. In Section 3.2, function f was generalised to take into account that some elements, having already been considered, did not need to be generated anymore. For the representatives problem, the generalisation of f we will make use of takes into account, not that some elements have already been considered, but that representative elements for some Q-equivalence classes have already been obtained. We will see that, under some conditions, the S-generative process does not need to consider those classes at all anymore. We will now proceed to define and analyse function g, the generalisation of f; we will, however, abandon the use of f, preferring the expression $S^* \cdot C$ to $f\,C$.

Function g can be seen as modelling a pruning of the generative process:

$$g(B,C) \;:=\; (\mu W : C \cup (S \cdot W - Q \cdot B)) \;. \tag{4.11}$$

As before, we call the second argument of g the set of *"seed"* elements of the generative process, whilst the first argument corresponds to the set of *"already generated representatives"*. As before, we record some interesting properties of g. First:

$$g(\emptyset,C) \;=\; S^* \cdot C \;, \tag{4.12}$$

$$g(B,\emptyset) \;=\; \emptyset \;. \tag{4.13}$$

Their proofs are as in Section 3.2. Rule (4.12) has a weaker form when generalised to any set of already generated representatives:

$$g(B,C) \;\subseteq\; S^* \cdot C \;. \tag{4.14}$$

It follows from property $V - U \subseteq V$ of subtraction and monotonicity of the least fixed-point operator μ (2.56). To complement rule (4.13), which deals with the case of an empty set of seeds, let us seek a rule that deals with the union of two sets of seeds. Unlike the analogue rule that was used in Section 3.2, no seeds are taken out of the generative process. This will be accounted for in the key *Thinning the Closure* rule presented in the next snbsection. We manipulate thus:

$$\begin{array}{ll} & g(B,\, C1 \cup C2) \\ = & \quad \{ \text{ definition of } g \text{ (4.11) } \} \end{array}$$

$$
\begin{array}{ll}
 & (\mu W : C1 \cup C2 \cup (S \cdot W - Q \cdot B)) \\
= & \left\{ \begin{array}{l} \text{fixed-point rolling rule (2.57) with} \\ \quad \mathcal{F} W := C1 \cup W , \\ \quad \mathcal{G} W := C2 \cup (S \cdot W - Q \cdot B) \end{array} \right\} \\
 & C1 \cup (\mu W : C2 \cup (S \cdot (C1 \cup W) - Q \cdot B)) \\
= & \left\{ \begin{array}{l} \text{distribution of composition over union (2.14)} \\ \text{and of subtraction over union (2.12)} \end{array} \right\} \\
 & C1 \cup (\mu W : C2 \cup (S \cdot C1 - Q \cdot B) \cup (S \cdot W - Q \cdot B)) \\
= & \{ \text{ definition of } g \text{ (4.11) } \} \\
 & C1 \cup g(B, C2 \cup (S \cdot C1 - Q \cdot B)) \ .
\end{array}
$$

In summary:

$$g(B, C1 \cup C2) \ = \ C1 \cup g(B, C2 \cup (S \cdot C1 - Q \cdot B)) \ . \qquad (4.15)$$

Thinning the Closure A crucial property of function g in connection to the thinning relation $\trianglelefteq$ is presented in this subsection. We call it the *Thinning the Closure* rule. The validity of this rule relies on the parameters Q, R and S of the representatives problem satisfying the following two requirements:

$$S \subseteq R \ , \qquad (4.16)$$

$$S \cdot (Q \cap R) \subseteq (Q \cap R) \cdot S \ . \qquad (4.17)$$

The first requirement says that the generating relation S gives rise to candidates that are no greater under R. Therefore, representative elements, being R-maxima of their classes, are likely to be produced sooner rather than later. The second requirement states that relation S is monotonic on the thinning mediator $Q \cap R$. This is better explained using points. Suppose there are elements x and y such that $x \langle Q \cap R \rangle y$, i.e. they are members of the same class and x is at most as good as y. This means that y can represent x. Now suppose that an element x' is generated from x, i.e. $x' \langle S \rangle x$ holds. Then there must be an element y' such that both $x' \langle Q \cap R \rangle y'$ and $y' \langle S \rangle y$ hold. This means that y can generate elements that represent every element generated from x. Therefore, x can be safely disposed of in the presence of y. These two requirements allow the S-generative process to be pruned accordingly.

The Thinning the Closure rule reads:

$$\left. \begin{array}{lll} g(B \cup D, C - Q \cdot D) & \trianglelefteq & g(B, C) - Q \cdot D \\ & & \text{provided } C \cdot D^{\circ} \subseteq R \ , \\ & & \quad S \cdot D - Q \cdot B \subseteq C \ . \end{array} \right\} \qquad (4.18)$$

An informal explanation follows. Suppose that, out of the generative process with set C as seeds and set B as already selected representatives, we have chosen a set D that includes some more representatives. We thus want to ignore the classes represented in D from now on; this corresponds to the expression on the right-hand side of the $\trianglelefteq$-equation. Under some conditions this can be done by eliminating every class represented in D from the seeds and including D among the already chosen representatives; this corresponds to the left-hand side of the $\trianglelefteq$-equation. Note that the set on the left-hand side is considerably smaller than the set on the right-hand side. The provisos for this thinning to be valid are (a) that any element in D is no smaller under R than any element in C, and (b) that the immediate successors under S of elements in D, with the exception of those in classes represented in B, are already included among the seeds. Requirements (4.16) and (4.17) on relations Q, R and S imply that the elements that are being eliminated from the generative process, either for being in a Q-class of D or for being generated from elements in a Q-class of D, will be represented either by elements in D or by elements generated from the immediate S-successors of D that are included in C.

The proof of this rule is somewhat long and tedious. It is presented in Appendix A.1.

4.3 Developing an Iteration

We now proceed with the actual derivation of an algorithm that computes representatives. The specification statement of (4.2) will be refined to an iteration.

Setting Up the Iteration For the refinement of (4.2) to an iteration, we take the conjunct $uniq\,B$ from the postcondition as part of the invariant. Properties (4.4) and (4.6) suggest that B is initialised to $\emptyset$ and then repeatedly augmented. To deal with the rest of the postcondition, we will use function g to repeatedly cut down the set of candidates according to the classes already represented in B. All told, we propose the following invariant, which requires the introduction of a new program variable C for the record of the changing set of seeds:

$$
\begin{aligned}
Inv1 &:= uniq\,B\ , \\
Inv2 &:= B \cup g(B,C) \trianglelefteq S^* \cdot A\ , \\
Inv &:= Inv1 \wedge Inv2\ .
\end{aligned}
$$

Assignment $B, C := \emptyset, A$ establishes the invariant due to (4.4), (4.12) and reflexivity of $\trianglelefteq$. Property (4.13) suggests $C \neq \emptyset$ as guard. The iteration will progressively accumulate representatives in variable B. Assuming there is a finite number of Q-equivalence classes, a reasonable choice for variant expression is B, with progress guaranteed by its increase. As customary by now, we introduce a shorthand for this:

$$Prg \;:=\; (B \supset B_0) \;.$$

Note that, in this case, we do not assume that X is a finite set. In fact, in some of the instances of the representatives problem we will treat in later sections X will indeed be an infinite set. The guarantee that the increase of B is bounded comes from assuming that the number of Q-equivalence classes is finite and from *Inv1*, which guarantees that B holds at most one member per Q-class.

What we have discussed so far corresponds to the following refinement of the specification statement of (4.2):

$$\begin{array}{ll} & B : [\; true \;,\; uniq\, B \;\wedge\; B \trianglelefteq S^{\circ} \cdot A \;] \\ \sqsubseteq & \left\{ \begin{array}{l} \text{introduce local block and initialised iteration} \\ \text{according to the discussion above} \end{array} \right\} \\ & |[\;\; \textbf{var}\; C : \mathrm{Vec}\, X \,; \\ & \quad B, C := \emptyset, A \,; \\ & \quad \textbf{do}\; C \neq \emptyset \;\rightarrow\; B, C : [\; C \neq \emptyset \;\wedge\; Inv \;,\; Inv \;\wedge\; Prg \;] \;\textbf{od} \\ &]| \end{array}$$

Developing the Iteration Body We now go for the body of the iteration. In order to guarantee progress, variable B should be augmented with elements of new Q-equivalence classes, i.e. classes not yet represented in it. This is achieved by assignment $B := B \cup D$ provided the following holds: (i) $D \neq \emptyset$ and (ii) $B \cdot D^{\circ} \cap Q = \emptyset$. Assuming that variable C will also require updating, we explore maintenance of the invariant by the statement $B, C := B \cup D, Exp$.

For the first half of the invariant, we reason thus:

$$\begin{array}{ll} & Inv1\,[\,B, C := B \cup D, Exp\,] \\ \equiv & \quad \{\; \text{substitution} \;\} \\ & uniq\,(B \cup D) \\ \equiv & \quad \{\; \text{preservation of uniqueness under union (4.6)} \;\} \\ & uniq\, B \;\wedge\; uniq\, D \;\wedge\; (B \cdot D^{\circ} \cap Q \subseteq id) \\ \equiv & \quad \{\; Inv1 \text{, assumption (ii) above implies the third conjunct} \;\} \end{array}$$

$uniq\, D$.

Hence, we also need (iii) $uniq\, D$. Maintenance of the second half of the invariant will provide us with the right value to assign to C . We reason as follows:

$$
\begin{array}{ll}
 & Inv2\,[\,B,C := B \cup D, Exp\,] \\
\equiv & \quad \{\ \text{substitution}\ \} \\
 & B \cup D \cup g\,(B \cup D, Exp) \ \trianglelefteq\ S^* \cdot A \\
\Leftarrow & \quad \{\ Inv2\ ,\ \text{transitivity of}\ \trianglelefteq\ \} \\
 & B \cup D \cup g\,(B \cup D, Exp) \ \trianglelefteq\ B \cup g\,(B,C) \\
\Leftarrow & \quad \{\ \text{monotonicity of union with respect to thinning (4.8)}\ \} \\
 & D \cup g\,(B \cup D, Exp) \ \trianglelefteq\ g\,(B,C) \\
\equiv & \quad \left\{ \begin{array}{l} \text{assume (iv)}\ D \subseteq C\ ;\ \text{hence,}\ C = D \cup C\ , \\ \text{interaction of function}\ g\ \text{with union (4.15)} \end{array} \right\} \\
 & D \cup g\,(B \cup D, Exp) \ \trianglelefteq\ D \cup g\,(B\,,\, C \cup (S \cdot D - Q \cdot B)) \\
\Leftarrow & \quad \left\{ \begin{array}{l} \text{refined monotonicity of union with respect to} \\ \text{thinning (4.9) -assume proviso of the rule-} \end{array} \right\} \\
 & g\,(B \cup D, Exp) \ \trianglelefteq\ g\,(B\,,\, C \cup (S \cdot D - Q \cdot B)) \ - \ Q \cdot D \\
\Leftarrow & \quad \left\{ \begin{array}{l} \text{Thinning the Closure rule (4.18) -assume} \\ \text{its two provisos-, transitivity of}\ \trianglelefteq \end{array} \right\} \\
 & g\,(B \cup D, Exp) \ \trianglelefteq\ g\,(B \cup D\,,\, (C \cup (S \cdot D - Q \cdot B)) - Q \cdot D) \\
\Leftarrow & \quad \{\ \text{reflexivity of}\ \trianglelefteq\ \} \\
 & Exp \ = \ (C \cup (S \cdot D - Q \cdot B)) - Q \cdot D\ .
\end{array}
$$

Two sets of provisos were assumed in the fifth and sixth steps, which will impose one more restriction on D. We start with the provisos of (4.18), assumed in the sixth step. The second of them,

$$S \cdot D - Q \cdot B \ \subseteq\ C \cup (S \cdot D - Q \cdot B)\ ,$$

holds without further restrictions while the first,

$$(C \cup (S \cdot D - Q \cdot B)) \cdot D^\circ \ \subseteq\ R\ , \tag{4.19}$$

does impose one more requirement on D . We manipulate thus:

$$
\begin{array}{ll}
 & (C \cup (S \cdot D - Q \cdot B)) \cdot D^\circ \\
\subseteq & \quad \{\ \text{subtraction}\ \} \\
 & (C \cup S \cdot D) \cdot D^\circ \\
\subseteq & \quad \{\ R\ \text{reflexive, requirement (4.16)}\ \} \\
 & (R \cdot C \cup R \cdot D) \cdot D^\circ
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ distribution of composition over union (2.14) } \} \\
& R \cdot (C \cup D) \cdot D^\circ \\
= & \{ \text{ assumption (iv) above } \} \\
& R \cdot C \cdot D^\circ \\
\subseteq & \{ \text{ assume (v) } C \cdot D^\circ \subseteq R \,;\; R \text{ transitive } \} \\
& R \ .
\end{array}
$$

We have only one proviso left to check, the one of (4.9):

$$g\,(B, C \cup (S \cdot D - Q \cdot B)) \cdot D^\circ \ \subseteq \ R \ .$$

It follows from (4.19), which we show as follows:

$$
\begin{array}{ll}
& g\,(B\,,\ C \cup (S \cdot D - Q \cdot B)) \cdot D^\circ \\
\subseteq & \{ \text{ from function } g \text{ to closure (4.14) } \} \\
& S^* \cdot \ (C \cup (S \cdot D - Q \cdot B)) \cdot D^\circ \\
\subseteq & \{ \text{ (4.19) proved above } \} \\
& S^* \cdot R \\
\subseteq & \left\{ \begin{array}{l} \text{since } R \text{ is a preorder, requirement (4.16) and universal} \\ \text{property of closure (2.62) imply that } S^* \subseteq R \text{ holds} \end{array} \right\} \\
& R \cdot R \\
\subseteq & \{ \ R \text{ transitive } \} \\
& R \ .
\end{array}
$$

In summary, we have collected five requirements (i)-(v) on D to make statement $B, C := B \cup D, \ldots$ maintain the invariant and guarantee progress:

$$
\begin{array}{lll}
\text{(i)} \ D \neq \emptyset \ , & \text{(ii)} \ B \cdot D^\circ \cap Q = \emptyset \ , & \text{(iii)} \ \mathit{uniq}\, D \ , \\
\text{(iv)} \ D \subseteq C \ , & \text{(v)} \ C \cdot D^\circ \subseteq R \ . &
\end{array}
$$

Requirement (ii) is implied by (iv) due to a third invariant of the developed iteration: $B \cdot C^\circ \cap Q = \emptyset$, i.e. B and C do not share equivalence classes. The proof that this is indeed an invariant of the developed iteration is presented in Appendix A.2. All told, we have refined the body of the iteration as follows:

$$
\begin{array}{ll}
& B, C : [\ C \neq \emptyset \ \wedge \ \mathit{Inv} \ , \ \mathit{Inv} \ \wedge \ \mathit{Prg} \ ] \\
\sqsubseteq & \left\{ \begin{array}{l} \text{introduce local block and sequential composition} \\ \text{according to discussion above} \end{array} \right\}
\end{array}
$$

```
|[ var B : Vec X ;
   |[ var C : Vec X ;
      B, C := ∅, A ;
      do C ≠ ∅ →
           |[ var D : Vec X ;
              D : [ C ≠ ∅ , (i) ∧ (iii) ∧ (iv) ∧ (v) ] ;
              B, C := B ∪ D, (C ∪ (S · D − Q · B)) − Q · D
           ]|
      od
   ]|
]|
```

Figure 4.20: General Algorithmic Solution of (4.2)

```
|[ var D : Vec X ;
   D : [ C ≠ ∅ , (i) ∧ (iii) ∧ (iv) ∧ (v) ] ;
   B, C := B ∪ D, (C ∪ (S · D − Q · B)) − Q · D
]|
```

The whole code, our general algorithmic solution to the representatives problem, is collected in Figure 4.20. In the following section, the specification statement left in the body of the iteration will be further refined, thus obtaining more specific solutions.

Feasibility Before moving on to the next section, note that the specification statement yet to be refined might not be *feasible*: there might be no values for D that comply with the postcondition, even if Q and R satisfy the requirements imposed on them so far and C satisfies the precondition. However, we will see that, in the process of refining the general solution to more specific ones, some additional requirements will be imposed on the parameters of our problem, which will make the developed programs feasible.

4.4 Further Refinement

This section continues the development of algorithmic solutions to the representatives problem. As in the treatment of the fixed-source reachability problem in Section 3.2, we will construct singleton-selection and whole-selection algorithms by refining the specification statement left in Figure 4.20.

```
|[ var B : Vec X ;
   |[ var C : Vec X ;
      B, C := ∅, A ;
      do C ≠ ∅ →
           |[ var x : X ;
              x :⊆ max (R, C) ;
              B, C := B ∪ x, (C ∪ (S · x − Q · B)) − Q · x
           ]|
      od
   ]|
]|
```

Figure 4.22: Singleton-Selection Solution of (4.2)

Refinement to Singleton-Selection Version Set D, to be drawn from C according to condition (iv), can be chosen to be a singleton set. A singleton set would then trivially satisfy (i), and would also satisfy (iii) on account of singletons being functions and property (4.5) of *uniq*. Therefore, only conditions (iv) and (v) would be required of such a singleton.

This refinement can be formalised by introducing a fresh variable x of type X to replace D, i.e. by applying data refinement with coupling invariant $x = D$. The specification statement in Figure 4.20 would then become:

$$x : [\ C \neq \emptyset\ ,\ x \subseteq C\ \wedge\ C \cdot x^{\circ} \subseteq R\ ]\ . \tag{4.21}$$

This corresponds to choosing x to be an R-maximum element of C. To guarantee the existence of such a maximum, which amounts to guaranteeing that statement (4.21) is feasible, it suffices to require that R is a connected preorder and to make sure that C is always finite. Finiteness of C as an invariant of the iteration can easily be shown to follow from two further requirements: that A is finite, and that S is finitary, i.e. that $S \cdot x$ is finite for every element x.

Under the new requirements, and using definition (2.59), statement (4.21) is equivalent to:

$$x :\subseteq max\,(R, C)\ .$$

The resulting algorithm is presented in Figure 4.22.

Refinement to Whole-Selection Version Choosing set D to be a singleton is the extreme where D is chosen to be as small as possible. At

the other extreme, there is the option of choosing D to be as big as possible. First thing that comes to mind is taking D to be the whole of set C. This would not necessarily work though, since C might have more than one element from some Q-equivalence class and D is required to satisfy *uniq*.

The biggest that D could be is the result of choosing one element per Q-class present in C. If such elements are chosen arbitrarily, D would be a thinning with $R := \Pi$ of C that also satisfies *uniq*; i.e. D would be a subset of C with exactly one element per Q-equivalence class not preferring any element to any other in each of the classes.

We thus explore conditions under which the following refinement is valid:

$$\left.\begin{array}{l} D : [\ C \neq \emptyset\ ,\ \text{(i)} \wedge \text{(iii)} \wedge \text{(iv)} \wedge \text{(v)}\ ] \\ \quad \sqsubseteq \quad D : [\ C \neq \emptyset\ ,\ \mathit{uniq}\, D \ \wedge\ D \subseteq C \ \wedge\ C \subseteq Q \cdot D\ ]\ . \end{array}\right\} \tag{4.23}$$

The postcondition of the last specification statement requires D to satisfy *uniq* and to be such that $D \trianglelefteq C$ with $R := \Pi$. Refinement (4.23), as a strengthening of the postcondition, follows from:

$$\left.\begin{array}{r} C \neq \emptyset \ \wedge\ \mathit{uniq}\, D \ \wedge\ D \subseteq C \ \wedge\ C \subseteq Q \cdot D \\ \Rightarrow \quad \text{(i)} \wedge \text{(iii)} \wedge \text{(iv)} \wedge \text{(v)}\ , \end{array}\right\} \tag{4.24}$$

which we now proceed to prove. Conditions (iii) and (iv), viz. $\mathit{uniq}\, D$ and $D \subseteq C$, follow trivially from the antecedent. It remains to show that (i) and (v), viz. $D \neq \emptyset$ and $C \cdot D^{\circ} \subseteq R$, also do so.

Let us start with (i) $D \neq \emptyset$. Assume the antecedent of (4.24) and then, using conjuncts $C \subseteq Q \cdot D$ and $C \neq \emptyset$, we have that:

$$D = \emptyset \quad \Rightarrow \quad C \subseteq \emptyset \quad \equiv \quad \mathit{false}\ .$$

Hence, (i) indeed holds.

Showing (v) $C \cdot D^{\circ} \subseteq R$ will impose additional requirements on the parameters R, S and A of our problem. Using conjunct $D \subseteq C$ of the antecedent of (4.24), we have that (v) follows from:

$$C \cdot C^{\circ} \quad \subseteq \quad R\ . \tag{4.25}$$

This amounts to saying that all the elements of C have the same R-cost. We call this condition, i.e. (4.25), the *R-homogeneity* of C. We claim that, under certain conditions we will derive shortly, R-homogeneity of C is an invariant of the iteration of our general solution in Figure 4.20. Therefore, (v) holds. Let us now explore such a claim.

It must be the case that A is R-homogeneous for C to be R-homogeneous

initially, i.e. for condition (4.25) to be established by the initialisation statement $B, C := \ldots, A$. Hence, we require:

$$A \cdot A^\circ \subseteq R \ . \tag{4.26}$$

If it is assumed that C is R-homogeneous at the beginning of the iteration body, it still holds after the first statement since such statement does not affect C. It remains to show that the second, and last, statement of the iteration body preserves R-homogeneity of C. Assuming that the antecedent of (4.24) holds, and that so does (4.25), we argue thus:

$$\begin{array}{ll} & (4.25)\ [\ B, C := \ldots, (C \cup (S \cdot D - Q \cdot B)) - Q \cdot D\ ] \\ \equiv & \left\{ \begin{array}{l} \text{substitution; distribute subtraction over union (2.12)} \\ \text{taking } W, V := C - Q \cdot D\ ,\ S \cdot D - Q \cdot B - Q \cdot D \end{array} \right\} \\ & (W \cup V) \cdot (W \cup V)^\circ \subseteq R \\ \Leftarrow & \left\{ \begin{array}{l} \text{hypothesis } C \subseteq Q \cdot D \text{ in (4.24) implies that } W = \emptyset\ ; \\ \text{by subtraction we have } V \subseteq S \cdot D\ , \text{converse (2.16)} \end{array} \right\} \\ & S \cdot D \cdot D^\circ \cdot S^\circ \subseteq R \\ \Leftarrow & \{\ \text{hypothesis } D \subseteq C \text{ in (4.24), and (4.25)}\ \} \\ & S \cdot R \cdot S^\circ \subseteq R \ . \end{array}$$

Hence, R-homogeneity of C is maintained by the iteration body provided we impose the following as a new requirement on R and S:

$$S \cdot R \cdot S^\circ \subseteq R \ . \tag{4.27}$$

In words, this new requirement ensures that, if C is R-homogeneous –i.e. all the elements of C have the same R-cost–, then the set of all S-descendants of the whole of C is also R-homogeneous –i.e. all such descendants also share a, possibly different, R-cost–.

We have thus proved implication (4.24) and, therefore, the validity of refinement (4.23), under new requirements (4.26) and (4.27). The assignment statement in the iteration body, which follows the specification statement, can also be refined. The expression used to assign a new value to C can be simplified using the new postcondition of the specification statement:

$$\begin{array}{ll} & (C \cup (S \cdot D - Q \cdot B)) - Q \cdot D \\ = & \{\ \text{distribution of subtraction over union (2.12)}\ \} \\ & (C - Q \cdot D) \cup (S \cdot D - Q \cdot B - Q \cdot D) \\ = & \{\ \text{since } C \subseteq Q \cdot D \text{ we have } C - Q \cdot D = \emptyset\ \} \\ & S \cdot D - Q \cdot B - Q \cdot D \end{array}$$

$$= \left\{ \begin{array}{l} \text{subtraction/union (2.13) and} \\ \text{distribution of composition over union (2.14)} \end{array} \right\}$$
$$S \cdot D - Q \cdot (B \cup D) \ .$$

Summarising, the iteration body of the general solution in Figure 4.20 has been refined to:

$$\begin{array}{l} [\!| \ \textbf{var} \ D : \mathsf{Vec}\, X \ ; \\ \quad D : [\ C \neq \emptyset \ , \ \mathit{uniq}\, D \ \wedge \ D \subseteq C \ \wedge \ C \subseteq Q \cdot D \ ] \ ; \\ \quad B, C \ := \ B \cup D \, , \, S \cdot D - Q \cdot (B \cup D) \\ |\!] \end{array}$$

provided the input set A is R-homogeneous (4.26) and the generator S preserves R-homogeneity (4.27).

Now note the interesting fact that our last specification statement above can be refined, by weakening the precondition, to:

$$D : [\ \mathit{true} \ , \ \mathit{uniq}\, D \ \wedge \ D \subseteq C \ \wedge \ C \subseteq Q \cdot D \ ] \ , \tag{4.28}$$

which is an instance of the very initial specification of the representatives problem (4.2) using $Q, R, S, A, B := Q, \Pi, \emptyset, C, D$. We choose to refine (4.28) using the singleton-selection solution of Figure 4.22; for this, note that requirements (4.16) and (4.17) hold if $S = \emptyset$.

We thus obtain our final whole-selection version as shown in Figure 4.29, where the local singleton-selection program of Figure 4.22 has been inserted after renaming some of the local variables and simplifying some of the expressions involving R and S, now Π and $\emptyset$. Specifically, local variable C has been renamed as E, expression $\mathit{max}\,(\Pi, E)$ –see (2.59)– has been simplified to E, and expression $E \cup (\emptyset \cdot x - Q \cdot D)$ has been simplified to E.

4.5 Fixed-Source Minimum Paths

This section shows that the fixed-source minimum paths problem is an instance of the representatives problem, and that it satisfies the required conditions for the singleton-selection algorithmic solution to apply.

In the *fixed-source minimum paths* problem, we are provided with a directed graph $(\mathit{Vert}, \mathit{Edge}, x1, x2)$ and a function $\mathit{weight} : \mathsf{R} \leftarrow \mathit{Edge}$ that assigns a weight, or cost, to each edge. We are also given a set $V : \mathsf{Vec}\ \mathit{Vert}$ called the source. For every vertex w reachable from V, we are required to compute a path of minimum cost among all the paths to w that start from a vertex in

```
|[ var B : Vec X ;
   |[ var C : Vec X ;
      B, C := ∅, A ;
      do C ≠ ∅ →
           |[ var D : Vec X ;
              |[ var E : Vec X ;
                 D, E := ∅, C ;
                 do E ≠ ∅ →
                      |[ var x : X ;
                         x :⊆ E ;
                         D, E := D ∪ x , E − Q · x
                      ]|
                 od
              ]| ;
              B, C := B ∪ D , S · D − Q · (B ∪ D)
           ]|
      od
   ]|
]|
```

Figure 4.29: Whole-Selection Solution of (4.2)

V . We formalise the problem making use of the datatype $Path$ and related functions defined in Section 2.8.

Assume we have the set of all the paths that start in vertices of the source set V . Partitioning such a set according to the ending vertices of the paths and drawing one of minimum cost from each class provides the required output. This corresponds to the instantiation:

$$X \;:=\; Path \;,$$

and to choosing representatives using the following relations:

$$p1 \,\langle Q \rangle\, p2 \;:=\; end\, p1 \;=\; end\, p2 \;, \tag{4.30}$$

$$p1 \,\langle R \rangle\, p2 \;:=\; cost\, p1 \;\geq\; cost\, p2 \;, \tag{4.31}$$

where *end* is, as in Section 2.8, a function that returns the ending vertex of a path and *cost* is a function that returns the sum of the weights of the edges of a path. For completeness, we spell out the definition of *cost* :

$$\left.\begin{array}{rcl} cost\,(wrap\,v) & = & 0 \;, \\ cost\,(cons\,(v,e,p)) & = & weight\,e \;+\; cost\,p \;. \end{array}\right\} \tag{4.32}$$

Note that a *maximum* under R is a path of *minimum* cost. We still need to instantiate parameters S and A of the general representatives problem. The set of paths that start in elements of V is $isPath? \cdot start^{\circ} \cdot V$ which, according to property (2.114), equals $(cons \cdot ok? \cdot out2^{\circ})^{*} \cdot wrap \cdot V$. This suggests instantiating:

$$\begin{array}{rcl} S & := & cons \cdot ok? \cdot out2^{\circ} \;, \\ A & := & wrap \cdot V \;. \end{array}$$

Using pointwise statements, these instantiations correspond to:

$$p1 \,\langle S \rangle\, p2 \;\equiv\; \langle \exists\, v,e \;:\; ok\,(v,e,p2) \;:\; p1 = cons\,(v,e,p2) \rangle \;, \tag{4.33}$$

$$p\,\langle A \rangle\, * \;\equiv\; \langle \exists\, v \;:\; v\,\langle V \rangle\, * \;:\; p = wrap\,v \rangle \;. \tag{4.34}$$

Armed with these relations Q , R and S , we now need to check that requirements (4.16) and (4.17) hold. This will allow us to use the singleton-selection solution of the representatives problem presented in Figure 4.22.

We analyse (4.16) thus:

$$\begin{array}{ll} & p1 \,\langle S \rangle\, p2 \\ \equiv & \quad \{ \text{ definition of } S \text{ (4.33) } \} \\ & \langle \exists\, v,e \;:\; ok\,(v,e,p2) \;:\; p1 = cons\,(v,e,p2) \rangle \end{array}$$

$$
\begin{array}{ll}
\Rightarrow & \{ \text{ definition of } \mathit{cost}\ (4.32)\ \} \\
& \langle\, \exists\, e ::\ \mathit{cost}\ p1\ =\ \mathit{weight}\ e\ +\ \mathit{cost}\ p2\,\rangle \\
\Rightarrow & \{ \text{ assume weights are non-negative } \} \\
& \mathit{cost}\ p1\ \geq\ \mathit{cost}\ p2 \\
\equiv & \{ \text{ definition of } R\ (4.31)\ \} \\
& p1\ \langle R \rangle\ p2\ .
\end{array}
$$

Hence, (4.16) holds provided weights are non-negative.

Now for (4.17):

$$
\begin{array}{ll}
& p1'\ \langle S \rangle\ p1\ \ \wedge\ \ p1\ \langle Q \cap R \rangle\ p2 \\
\equiv & \{ \text{ definitions of } Q\ (4.30),\ R\ (4.31) \text{ and } S\ (4.33)\ \} \\
& \langle\, \exists\, v, e\ :\ \mathit{ok}\ (v, e, p1)\ :\ p1' = \mathit{cons}\ (v, e, p1)\,\rangle \\
& \quad \wedge\ \ \mathit{end}\ p1\ =\ \mathit{end}\ p2\ \ \wedge\ \ \mathit{cost}\ p1\ \geq\ \mathit{cost}\ p2 \\
\Rightarrow & \left\{ \begin{array}{l} \text{second conjunct and definition of } \mathit{ok}\ (2.111) \text{ imply that} \\ \mathit{ok}\ (v, e, p1)\ \equiv\ \mathit{ok}\ (v, e, p2)\text{, take } p2' := \mathit{cons}\ (v, e, p2) \end{array} \right\} \\
& \langle\, \exists\, v, e\ :\ \mathit{ok}\ (v, e, p1)\ \wedge\ \mathit{ok}\ (v, e, p2)\ : \\
& \quad p1' = \mathit{cons}\ (v, e, p1)\ \ \wedge\ \ p2' = \mathit{cons}\ (v, e, p2)\,\rangle \\
& \quad \wedge\ \ \mathit{cost}\ p1\ \geq\ \mathit{cost}\ p2 \\
\Rightarrow & \{ \text{ definitions of } \mathit{end}\ (2.107) \text{ and } \mathit{cost}\ (4.32)\ \} \\
& \langle\, \exists\, v, e\ :\ \mathit{ok}\ (v, e, p2)\ :\ p2' = \mathit{cons}\ (v, e, p2)\,\rangle \\
& \quad \wedge\ \ \mathit{end}\ p1'\ =\ \mathit{end}\ p2'\ \ \wedge\ \ \mathit{cost}\ p1'\ \geq\ \mathit{cost}\ p2' \\
\equiv & \{ \text{ definitions of } Q\ (4.30),\ R\ (4.31) \text{ and } S\ (4.33)\ \} \\
& p1'\ \langle Q \cap R \rangle\ p2'\ \ \wedge\ \ p2'\ \langle S \rangle\ p2\ .
\end{array}
$$

We have thus proved that this instance of the representatives problem has as solution the singleton-selection algorithm of Figure 4.22. The result is Dijkstra's algorithm for the computation of minimum paths [43]. Note that the requirements used to guarantee that the singleton-selection algorithm is feasible, viz. that R is connected, A is finite and S is finitary, also hold. The last two ones do so on account of the assumption that we are only dealing with graphs whose vertex set and edge set are botb finite.

The whole-selection algorithm cannot be used since requirement (4.27) does not hold.

4.6 Fixed-Source Shortest Paths

A simpler version of the fixed-source minimum paths problem is obtained by uniformly assigning a weight of one to each edge. This means we only care for the length, i.e. the number of edges, of the paths and we thus speak of *shortest* paths rather than *minimum* paths.

As a particular case of the minimum paths problem, the shortest paths problem is also an instance of the representatives problem. Requirements (4.16) and (4.17) hold, as proved in the previous section, and, therefore, the singleton-selection algorithm can be used to compute shortest paths. Requirements (4.26) and (4.27) also hold in this case, as we prove below, which makes the whole-selection algorithm of Figure 4.29 also applicable for the computation of shortest paths.

We prove (4.26) thus:

$$
\begin{array}{ll}
 & p1\,\langle A\rangle * \ \wedge\ *\langle A^\circ\rangle\, p2 \\
\equiv & \quad \{\ \text{definition of } A\ (4.34)\text{, converse}\ \} \\
 & \langle \exists v : v\,\langle V\rangle * : p1 = wrap\ v\rangle \\
 & \quad \wedge\ \langle \exists v : v\,\langle V\rangle * : p2 = wrap\ v\rangle \\
\Rightarrow & \quad \{\ \text{definition of } cost\ (4.32)\ \} \\
 & cost\ p1\ =\ 0\ \wedge\ cost\ p2\ =\ 0 \\
\Rightarrow & \quad \{\ \text{definition of } R\ (4.31)\ \} \\
 & p1\,\langle R\rangle\, p2\ .
\end{array}
$$

For (4.27), we argue as follows:

$$
\begin{array}{ll}
 & p1'\,\langle S\rangle\, p1\ \wedge\ p1\,\langle R\rangle\, p2\ \wedge\ p2\,\langle S^\circ\rangle\, p2' \\
\equiv & \quad \{\ \text{definitions of } R\ (4.31)\ \text{and } S\ (4.33)\text{, converse}\ \} \\
 & cost\ p1\ \geq\ cost\ p2 \\
 & \quad \wedge\ \langle \exists v,e\ :\ ok\ (v,e,p1)\ :\ p1' = cons\ (v,e,p1)\ \rangle \\
 & \quad \wedge\ \langle \exists v,e\ :\ ok\ (v,e,p2)\ :\ p2' = cons\ (v,e,p2)\ \rangle \\
\Rightarrow & \quad \left\{ \begin{array}{l} \text{cost of the paths is just their length} \\ \text{–i.e. } weight\ e\ =\ 1 \text{ for any edge } e\text{–} \end{array} \right\} \\
 & cost\ p1\ \geq\ cost\ p2 \\
 & \quad \wedge\ cost\ p1' = cost\ p1\ +\ 1\ \wedge\ cost\ p2' = cost\ p2\ +\ 1 \\
\Rightarrow & \quad \{\ \text{arithmetic}\ \} \\
 & cost\ p1' \geq\ cost\ p2' \\
\equiv & \quad \{\ \text{definition of } R\ (4.31)\ \} \\
 & p1'\,\langle R\rangle\, p2'\ .
\end{array}
$$

Note that edges are not significant for this problem. The existence of parallel edges, for instance, is completely irrelevant. This was not the case for the minimum paths problem, since a pair of parallel edges might have had different weights and thus determine paths of different costs in spite of "walking" over the same vertices. Given that edges are not significant for the computation of shortest paths, this problem could have been posed in terms of simple graphs –see page 28 of Section 2.5– with edges not entering the picture at all. This would have involved redefining the datatype *Path* and all its related functions and we thus chose not to do so. Such a redefinition can be found in [127, Section 3.2]

4.7 Fixed-Source Reachability

The fixed-source reachability problem, as stated in Section 3.2, is that of computing the set of vertices that can be reached in a given graph from a given source set of vertices. Formally, the aim of the problem is the computation of $Succ^* \cdot V : \mathrm{Vec}\ Vert$ where $Succ : Vert \leftarrow Vert$ is the successor relation of the graph and $V : \mathrm{Vec}\ Vert$ is the given source.

We now show that the fixed-source reachability problem is an instance of the problem of computing representatives (4.2). First of all, let us make the straightforward decision to take $X, S, A := Vert, Succ, V$. The instantiations of S and A give us a set of candidates $S^* \cdot A$ that corresponds to the whole set we aim to compute, i.e. the whole set of vertices reachable from V. We then need to instantiate Q and R accordingly. Since every vertex is relevant on its own, no pair of different elements should be set to be equivalent, which leads to choosing $Q := id$. Such a selection for the equivalence relation Q makes the uniqueness predicate (4.3) be the constant predicate that always return *true*. Also, it makes the thinning relation (4.7) be the equality relation, irrespective of what preorder R is chosen to be:

$$
\begin{array}{ll}
 & B \trianglelefteq C \\
\equiv & \quad \{ \text{ definition of } \trianglelefteq \ (4.7) \text{ with } Q := id \ \} \\
 & B \subseteq C \ \wedge \ C \subseteq (id \cap R) \cdot B \\
\equiv & \quad \{ \ R \text{ is reflexive: } id \subseteq R \text{; hence, } id \cap R = id \ \} \\
 & B \subseteq C \ \wedge \ C \subseteq B \\
\equiv & \quad \{ \text{ inclusion/equality of relations } \} \\
 & B = C \ .
\end{array}
$$

Hence, the instance $X, Q, S, A := Vert, id, Succ, V$ of the representatives problem (4.2) corresponds to the fixed-source reachability problem (3.6) –applying substitution $X, R, A := Vert, Succ, V$ since in Section 3.2 relation

$R : X \leftarrow X$ was used as the successor relation of the graph and $A : \mathrm{Vec}\, X$ as the source vertex set–.

As proved in the calculation above, taking $Q := id$ renders preorder R useless as far as the specification of the problem is concerned. However, for using the algorithmic solutions we derived before, we need requirements (4.16) and (4.17) to hold. Also, R must be connected to guarantee that the singleton-selection solution of Figure 4.22 is feasible. Finally, we also need additional requirements (4.26) and (4.27) to hold if we want to use the whole-selection solution of Figure 4.29. Choosing $R := \Pi$ makes all the aforementioned conditions valid and, therefore, both the singleton- and the whole-selection algorithms can be used.

It can be proved that the three algorithmic solutions, i.e. the general one in Section 4.3 and the two more specific ones in Section 4.4, for this instance $X, Q, R, S, A := Vert, id, \Pi, Succ, V$ correspond to the three algorithmic solutions in Section 3.2 for the fixed-source reachability problem –with $X, R, A := Vert, Succ, V$–. However, some formal manipulation is needed to obtain the exact correspondence. General solution of Figure 4.20 only requires simplification of (iii) and (v) to *true* and distribution of subtraction over union to be transformed into Figure 3.14. Singleton-selection solution of Figure 4.22 is similarly transformed into Figure 3.15, simplifying $max\ (\Pi, C)$ to C. Simplification of the whole-selection solution of Figure 4.29 to obtain 3.16 is somewhat more laborious: the block that declares variable E can be proved equivalent to assignment $D := C$, which then allows discarding variable D by making direct use of C instead.

4.8 Related Work

The class of graph problems we have treated as instances of the representatives problem has been tackled by others using different calculational frameworks. We commented on the work of Backhouse et al. [12], Möller [105] and Russling [133] before in Section 3.3.

Backhouse et al. [12] cover our singleton-selection algorithm for the minimum/shortest paths problem and the reachability problem. They weight the edges with elements of an arbitrary regular algebra, thereby achieving a higher level of generality. However, we believe our instantiation to real numbers in Section 4.5 and Section 4.6 can be also proved correct using the carrier set of a regular algebra. They fail to provide our whole-selection algorithm since they restrict their treatment to singleton sets: "The algorithm we develop is based on an iterative process in which at each iteration Theorem 7.1 is used to 'process' *one* node" [12, page 13] –our emphasis–. However,

we believe their "Key Theorem 7.1" can be generalised to cater for arbitrary sets. Möller first treated the reachability problem in [105], then with Russling the shortest paths problem in [107] and, later, Russling presented a "class of layer-oriented graph algorithms" [133] where he treats both problems in a uniform way –see also [33]–. Their solutions correspond to our whole-selection algorithm. They do not deal with the minimum paths problem. Clenaghan has shown that Backhouse et al.'s and Möller and Russling's approach to this class of graph algorithmic problems can be formally unified using dynamic algebra [34]; it would be interesting to see how our approach relates to this work.

A related reference is [50], where van den Eijnde treats, in a calculational style similar to ours though using non-conventional control structures, a class of graph problems that includes a so-called ascending reachability problem.

Chapter 5
Computing Maximal Sets

In this chapter we deal with the general problem of computing maximal sets satisfying a certain given predicate. As with the problem of computing representatives in Chapter 4, our interest in the problem of computing maximal sets comes from the existence of some instances of it that correspond to graph problems. The contents of this chapter were first partially reported in [128] and later fully reported in [129].

Section 5.1 presents a formal specification of the general problem of computing maximal sets satisfying a certain predicate P, as well as some properties that will be required of P for the development of algorithmic solutions. Section 5.2 then presents the derivation of an algorithmic solution of the general problem, and such a solution is further refined in Section 5.3. The properties of P on which the derivation is based rely on the existence of some auxiliary predicates and functions that form part of the final algorithm. When calculating graph instances of the general problem in the rest of the chapter, the calculus of relations is used to prove such assumed properties of P and to *calculate* the auxiliary components for each instance. Section 5.4 presents the instance of computing maximal independent sets of vertices and Section 5.5 presents the instance of computing maximal sets of edges without cycles, which corresponds to computing connectedness-preserving forests. As usual, we close the chapter by reviewing related work in Section 5.6.

5.1 Specification

This section presents a formal specification of the general problem of computing maximal sets satisfying a given predicate P, and the conditions required of P for the refinement of such a specification to a program.

Let X be some universe, i.e. a set, and let P be a predicate on subsets of

it. A subset of X, maximal among the subsets of X for which predicate P holds, is to be computed. Using the formalisation (2.43) of maximal sets presented in page 17 of Section 2.2, the aim is to compute a set A that satisfies $mxl\,(P, A)$.

Specification Our problem is specified as follows:

$$\begin{array}{l} [\!| \ \textbf{var}\ A : \mathsf{Vec}\, X\ ; \\ \quad A : [\ \mathit{true}\ ,\ mxl\,(P, A)\ ] \\ |\!] \end{array} \tag{5.1}$$

Requirements Some properties will be required of predicate P to refine (5.1) to a program. First, P is assumed to be *subset-closed*. As stated in (2.45), and repeated here for convenience, this means:

$$A1 \subseteq A2 \ \Rightarrow\ (P\,A2 \Rightarrow P\,A1)\ , \tag{5.2}$$

where $A1$ and $A2$ are dummies ranging over subsets of X. Since the empty set is a subset of every set, a subset-closed predicate that does not hold on the empty set must be the trivial everywhere-*false* predicate, in which case our computational problem would have no solution. Our second asumption does away with such a possibility by stating that P must hold on the empty set:

$$P\,\emptyset\ . \tag{5.3}$$

Finally, it is assumed that P is *incremental* with respect to a second predicate Q in the following sense:

$$P\,(A \cup a)\ \equiv\ P\,A\ \wedge\ Q\,(A, a) \qquad \text{provided}\ \ a \subseteq \overline{A}\ , \tag{5.4}$$

where A and a range, respectively, over subsets of X and elements of X. Provided P is subset-closed, incrementality is no restriction since such a Q always exists: take $Q\,(A, a)$ to be $P\,(A \cup a)$. However, we are not interested in arbitrary Q satisfying (5.4) since the efficiency of the program will depend on such a selection.

Non-trivial subset-closedness, i.e. (5.2) and (5.3), is related to the notion of *matroids* used in combinatorial optimisation. Matroids are defined as families of sets over a certain given universe –or, equivalently, predicates on such sets, as our P– that satisfy some properties, non-trivial subset-closedness among them. Matroids are used to develop greedy algorithms –see e.g. [36, Chapter 17] or [92, Chapter 7]–, very much in the same way that we will use non-trivial subset-closedness of P to refine (5.1) to a program. Hence, the formal development we present in this chapter can be seen as a derivational

presentation of a generic matroid-based greedy algorithm. A full matroid would also guarantee that all maximal sets are of maximum size, but we do not need this and, in fact, the instance of independent sets of vertices we will present in Section 5.4 does not fit the full matroid model.

5.2 Developing an Iteration

We now proceed to derive a program that refines (5.1). The development is carried out in two stages. First, in this section, the specification is algorithmically refined to an iteration. Second, guided by some efficiency considerations, we will proceed in the next section to transform the state space of the program by data refinement.

Setting Up the Iteration Heading towards an iteration, we want to propose a reasonable invariant and a corresponding guard from the given postcondition. One technique to do so, which we have already used in previous chapters, is that of replacing a constant in the postcondition by a fresh variable. Rewriting our postcondition according to property (2.46) of *mxl*, which depends on predicate P being subset-closed, will help us to apply such a technique. We manipulate the postcondition of (5.1) thus:

$$
\begin{array}{ll}
 & mxl\,(P, A) \\
\equiv & \quad \{\ \text{property of } mxl\ (2.46),\ P \text{ is subset-closed } (5.2)\ \} \\
 & P\,A\ \wedge\ \langle \forall a : a \subseteq \overline{A} : \neg P(A \cup a) \rangle \\
\equiv & \quad \left\{ \begin{array}{l} \text{contrapositive -this simplifies the expression since it} \\ \text{eliminates the use of complementation and negation-} \end{array} \right\} \\
 & P\,A\ \wedge\ \langle \forall a : P(A \cup a) : a \subseteq A \rangle\ .
\end{array}
$$

With the above rephrasing of the postcondition, introducing a fresh variable B to replace the third ocurrence of A gives an invariant with $B \neq A$ as the corresponding guard. Let us introduce the following shorthands for the invariants:

$$
\begin{array}{rcl}
\mathit{Inv1} & := & P\,A\ , \\
\mathit{Inv2} & := & \langle \forall a : P(A \cup a) : a \subseteq B \rangle\ , \\
\mathit{Inv} & := & \mathit{Inv1} \wedge \mathit{Inv2}\ .
\end{array}
$$

We now need to work out a variant. For that, we first remark that the invariant implies $A \subseteq B$, since for any element x we have:

$$
x \subseteq A \quad \equiv \quad A \cup x = A \quad \Rightarrow \quad P(A \cup x) \quad \Rightarrow \quad x \subseteq B\ .
$$

The two implications are justified by *Inv1* and *Inv2*, respectively. Therefore, provided the invariant holds, the negation of the guard, $B = A$, is equivalent to $B \subseteq A$ and it thus satisfies:

$$B = A \quad \equiv \quad B - A = \emptyset \ . \tag{5.5}$$

It then seems reasonable to choose $B - A$ as variant, with progress guaranteed by its decrease. We introduce, as customary by now, a shorthand for progress:

$$Prg \ := \ (B - A \subset B_0 - A_0) \ .$$

Set X is assumed to be finite to guarantee well-foundedness.

We will have completed the global structure of the iteration after working out a way to establish the invariant initially. Recall that P was assumed to hold on the empty set (5.3); hence, an assignment of the empty set to A will do for *Inv1*. Assigning the whole universe X to B will make the consequent of *Inv2* equivalent to *true* and, hence, will make *Inv2* hold.

The statement of (5.1) is then formally refined thus:

$$\begin{array}{ll} & A : [\ \mathit{true}\ ,\ \mathit{mxl}\,(P, A)\ ] \\ \sqsubseteq & \quad \left\{ \begin{array}{l} \text{introduce local block and initialised iteration} \\ \text{according to discussion above} \end{array} \right\} \\ & |[\ \ \mathbf{var}\ B : \mathsf{Vec}\, X\ ; \\ & \quad A, B := \emptyset, X\ ; \\ & \quad \mathbf{do}\ B \neq A \ \rightarrow\ A, B : [\ B \neq A \ \wedge\ \mathit{Inv}\ ,\ \mathit{Inv}\ \wedge\ \mathit{Prg}\ ]\ \mathbf{od} \\ &]| \end{array}$$

Developing the Iteration Body We now proceed to refine the iteration body. The variant must be decreased by suhtracting elements from $B - A$. This can be done by taking an element x in $B - A$ and either subtracting it from B or adding it to A. It is therefore promising to explore the effect of assignments $A := A \cup x$ and $B := B - x$ on the invariant. Thus, assume the invariant holds and also assume that $x \subseteq B - A$ –recall (5.5), which makes the existence of such an x feasible if the guard holds–.

Let us first study augmentation of A:

$$\begin{array}{ll} & \mathit{Inv1}\ [\, A := A \cup x\,] \\ \equiv & \quad \{\ \text{substitution}\ \} \\ & P\,(A \cup x) \end{array}$$

$$
\begin{array}{ll}
\equiv & \left\{ \begin{array}{l} P \text{ is } Q\text{-incremental (5.4)}, \\ \text{assumption on } x \text{ implies } x \subseteq \overline{A} \end{array} \right\} \\
& P\,A \;\wedge\; Q\,(A,x) \\
\equiv & \quad \{\ \textit{Inv1}\ \} \\
& Q\,(A,x)\ ,
\end{array}
$$

and

$$
\begin{array}{ll}
& \textit{Inv2}\,[\,A := A \cup x\,] \\
\equiv & \quad \{\ \text{substitution}\ \} \\
& \langle \forall a\ :\ P\,(A \cup x \cup a)\ :\ a \subseteq B \rangle \\
\Leftarrow & \quad \{\ \text{union, } P \text{ is subset-closed (5.2), predicate calculus}\ \} \\
& \langle \forall a\ :\ P\,(A \cup a)\ :\ a \subseteq B \rangle \\
\equiv & \quad \{\ \textit{Inv2}\ \} \\
& \textit{true}\ .
\end{array}
$$

Now the same analysis for the diminishing of B: the first part of the invariant *Inv1* is not affected, and

$$
\begin{array}{ll}
& \textit{Inv2}\,[\,B := B - x\,] \\
\equiv & \quad \{\ \text{substitution}\ \} \\
& \langle \forall a\ :\ P\,(A \cup a)\ :\ a \subseteq B - x \rangle \\
\equiv & \quad \{\ \text{subtraction, atoms}\ \} \\
& \langle \forall a\ :\ P\,(A \cup a)\ :\ a \subseteq B \wedge a \neq x \rangle \\
\equiv & \quad \{\ \text{distribution of universal quantification over conjunction}\ \} \\
& \langle \forall a\ :\ P\,(A \cup a)\ :\ a \subseteq B \rangle \;\wedge\; \langle \forall a\ :\ P\,(A \cup a)\ :\ a \neq x \rangle \\
\equiv & \quad \{\ \textit{Inv2}\text{; contrapositive}\ \} \\
& \langle \forall a\ :\ a = x\ :\ \neg P\,(A \cup a) \rangle \\
\equiv & \quad \{\ \text{one-point rule}\ \} \\
& \neg P\,(A \cup x) \\
\equiv & \left\{ \begin{array}{l} P \text{ is } Q\text{-incremental (5.4)}, \\ \text{assumption on } x \text{ implies } x \subseteq \overline{A} \end{array} \right\} \\
& \neg P\,A \;\vee\; \neg Q\,(A,x) \\
\equiv & \quad \{\ \textit{Inv1}\ \} \\
& \neg Q\,(A,x)\ .
\end{array}
$$

Nicely symmetric! According to whether $Q\,(A,x)$ holds, the invariant is maintained by one assignment or the other. This suggests the use of an alternation. Accordingly, we refine the statement of the iteration body as

```
|[ var A : Vec X ;
   |[ var B : Vec X ;
      A, B := ∅, X ;
      do B ≠ A →
           |[ var x : X ;
              x :⊆ B − A ;
              if   Q(A, x) → A := A ∪ x
              [] ¬ Q(A, x) → B := B − x
              fi
           ]|
      od
   ]|
]|
```

Figure 5.6: First Algorithmic Solution of (5.1)

follows:

$$
\begin{array}{ll}
 & A, B : [\ B \neq A \ \wedge\ Inv\ ,\ Inv \ \wedge\ Prg\ ] \\
\sqsubseteq & \left\{ \begin{array}{l} \text{introduce local block and alternation} \\ \text{according to discussion above} \end{array} \right\} \\
 & |[\ \textbf{var}\ x : X ; \\
 & \quad x :\subseteq B - A ; \\
 & \quad \textbf{if}\ \ Q(A, x) \rightarrow A := A \cup x \\
 & \quad [] \ \neg Q(A, x) \rightarrow B := B - x \\
 & \quad \textbf{fi} \\
 &]|
\end{array}
$$

This completes the algorithmic refinement. Figure 5.6 shows the collected code.

5.3 Further Refinement

Some efficiency considerations lead us to further develop the program of Figure 5.6 in this section. Two new variables will be incorporated by means of data refinement.

More Requirements At this stage we will need further assumptions on predicate Q. These assumptions will seem to come out of the blne, but we remark that they arose from analysing instances of the general program.

In the same way, the given restrictions on P –non-trivial subset-closedness and incrementality– were abstracted from an initial attempt to carry out the above general development for an instance rather than for the general problem. Thus, these added constraints on Q must be taken in the same spirit in which initial constraints on P were taken.

It is assumed that predicate Q can be expressed in terms of another predicate Q' and a function f. Additionally, f is assumed to be *incremental* with respect to an operator $\oplus$. Formally:

$$Q(A,a) \equiv Q'(f\,A,a)\ , \tag{5.7}$$

$$f(A \cup a) = f\,A \oplus a\ . \tag{5.8}$$

Function f takes a set in $\mathrm{Vec}\,X$ to an element of a new type Y. Hence, Q' is a predicate on $Y \times X$ and operator $\oplus$ takes a pair in $Y \times X$ to an element in Y. Function application is assumed to bind more tightly than operator $\oplus$, so that $f\,A \oplus a$ above must be read as $(f\,A) \oplus a$.

Data Refinement Let us now motivate the two new variables by analysing the program of Figure 5.6. We will then define a suitable coupling invariant for their introduction.

First, reevaluation of $B - A$ on every iteration seems expensive. This can be avoided by introducing a fresh program variable C to hold its value. Such a variable will also make the guard become less costly. Thus, the first half of our coupling invariant is:

$$CI1 \ := \ C = B - A\ .$$

Second, due to (5.7), one can safely assume that the cost of evaluating Q is shared by the cost of Q' and the cost of f. This prompts us to introduce a program variable y to hold the value of $f\,A$, which reduces evaluation of Q in the alternation to that of Q' solely. Updates of y will be made by means of $\oplus$, thanks to incrementality of f (5.8). Hence, the second conjunct of the coupling invariant is:

$$CI2 \ := \ y = f\,A\ .$$

Armed with coupling invariant $CI1 \wedge CI2$, we now give an overview of how to conduct the transformation of the program. Assignments to C, y are attached to every assignment of the program in such a way that the coupling invariant is maintained. Both branches of the alternation induce the same assignment on C, viz. $C := C - x$, which can then be moved out of the alternation. By (5.5), the guard can become $C \neq \emptyset$. The non-

```
|[ var A : Vec X ;
   |[ var C : Vec X ; y : Y ;
      A, C, y := ∅, X, f ∅ ;
      do C ≠ ∅ →
           |[ var x : X ;
              x :⊆ C ; C := C − x ;
              if  Q'(y, x) → A, y := A ∪ x, y ⊕ x
              [] ¬ Q'(y, x) → skip
              fi
           ]|
      od
   ]|
]|
```

Figure 5.9: Second Algorithmic Solution of (5.1), after Data Refinement

deterministic assignment to variable x becomes just $x :\subseteq C$. The main role of y is its use in the guards of the alternation. All the above transformations render variable B useless, and it is thus eliminated.

The data-refined solution is shown in Figure 5.9.

5.4 Maximal Independent Vertex Sets

In this section and the next, we will instantiate the general algorithmic solution of Figure 5.9 to graph problems. The properties imposed on P must be shown to hold for the specific predicate in each case. Following the calculational spirit of algorithmics we aim to promote, components $Q, Q', f, \oplus$ for each instance will be *calculated* instead of given a priori.

This section deals with the problem of computing maximal independent vertex sets of undirected graphs. Two vertices of an undirected graph are said to be *independent* if they are not adjacent. An *independent set* of vertices is one in which any two elements are independent. Formally, for graph $(Vert, Edge, x1, x2)$ with adjacency relation $Adj : Vert \leftarrow Vert$, and set V : Vec *Vert* , we define:

$$indep\ V \quad := \quad V \cdot V^{\circ} \cap Adj \ \subseteq \ \emptyset \ . \tag{5.10}$$

It is easy to show that the empty set is independent. The same goes for showing that a singleton set v is independent if there are no loops –as defined in page 27 of Section 2.5– incident on the vertex v . Thus, small independent

sets are easy to find. A more interesting problem is that of finding maximal independent sets, which corresponds to the instance $X, P := Vert, indep$ of (5.1).

Checking the Requirements First, it must be shown that *indep* is non-trivially subset-closed. Property (5.2) follows from the monotonicity properties of all the operators involved –converse, composition and intersection– with respect to inclusion. The empty relation $\emptyset$ is a zero of both composition and intersection; hence, (5.3) also holds.

We check the incrementality property, whilst also calculating an adequate instance for the auxiliary predicate Q, thus:

$$
\begin{array}{ll}
 & indep\ (V \cup v) \\
\equiv & \quad \{\ \text{definition of } indep\ (5.10)\ \} \\
 & (V \cup v) \cdot (V \cup v)^\circ \cap Adj \subseteq \emptyset \\
\equiv & \quad \left\{ \begin{array}{l} \text{distribution of converse, composition} \\ \text{and intersection over union} \end{array} \right\} \\
 & (V \cdot V^\circ \cap Adj) \cup (V \cdot v^\circ \cap Adj) \\
 & \quad \cup\ (v \cdot V^\circ \cap Adj) \cup (v \cdot v^\circ \cap Adj) \subseteq \emptyset \\
\equiv & \quad \{\ \text{universal property of union (2.3)}\ \} \\
 & (V \cdot V^\circ \cap Adj \subseteq \emptyset) \ \wedge\ (V \cdot v^\circ \cap Adj \subseteq \emptyset) \\
 & \quad \wedge\ (v \cdot V^\circ \cap Adj \subseteq \emptyset) \ \wedge\ (v \cdot v^\circ \cap Adj \subseteq \emptyset) \\
\equiv & \quad \left\{ \begin{array}{l} \text{definition of } indep\ (5.10);\ \text{second and third} \\ \text{conjuncts are equivalent by properties of converse} \\ \text{–(2.16), (2.17), (2.18)– and by symmetry of } Adj \end{array} \right\} \\
 & indep\ V \ \wedge\ (v \cdot V^\circ \cap Adj \subseteq \emptyset) \ \wedge\ (v \cdot v^\circ \cap Adj \subseteq \emptyset)\ .
\end{array}
$$

This provides us with a predicate Q that fulfills requirement (5.4) for the instance $P := indep$, viz. the one given by the last two conjuncts above. To search for Q' and f as required by (5.7) the middle conjunct is further manipulated; such manipulation applies to the last conjunct as well:

$$
\begin{array}{ll}
 & v \cdot V^\circ \cap Adj \subseteq \emptyset \\
\equiv & \quad \{\ \text{complementation shnnting (2.9)}\ \} \\
 & v \cdot V^\circ \subseteq \overline{Adj} \\
\equiv & \quad \{\ \text{Schröder's left-exchange rule (2.21)}\ \} \\
 & Adj \cdot V \subseteq \overline{v} \\
\equiv & \quad \{\ \text{complementation}\ \} \\
 & v \subseteq \overline{Adj \cdot V}\ .
\end{array}
$$

Thus, to comply with (5.7) we take:

$$\begin{aligned} f\,V &:= \overline{Adj \cdot V} \qquad \text{(and, therefore, } f\,\emptyset = Vert\text{)} \ , \\ Q'(W,v) &:= v \subseteq W \ \wedge \ v \not\subseteq Adj \cdot v \ . \end{aligned}$$

Operator $\oplus$ can also be derived, given the above definition of f, to make (5.8) hold. Using distributivity of composition over union and De Morgan's rule we obtain:

$$W \oplus v \ := \ W - Adj \cdot v \ .$$

Note that the form of Q' and $\oplus$ suggest the use of adjacency lists to represent the graph. The resulting program could be further improved but we will not go into such detail.

5.5 Connectedness-Preserving Forests

This section treats the problem of computing maximal acyclic edge sets in an undirected graph, which corresponds to computing connectedness-preserving forests, as an instance of our general problem. Again, we calculate components $Q, Q', f, \oplus$ instead of first giving them and then proving them correct according to the requirements imposed on P.

Let G be an undirected graph. Recall from Section 2.7 that connectedness-preserving forests generalise spanning trees and that, as stated by Proposition 2.101, a spanning subgraph of G is a connectedness-preserving forest if and only if it is a maximal acyclic subgraph. Since spanning subgraphs are uniquely determined by their sets of edges, we can formally state that a subset E of the edge set of G is a connectedness-preserving forest of G if and only if $mxl\,(acyclic, E)$ holds. Predicate *acyclic* was defined by (2.92) in terms of predicate *cyclic* (2.91) in page 33 of Section 2.6. For convenience, we repeat such definitions here:

$$acyclic\,E \ \equiv \ \neg\, cyclic\,E \ , \tag{5.11}$$

$$cyclic\,E \ \equiv \ \langle \exists\, e \,:\, e \subseteq E \,:\, e \preceq E - e \rangle \ . \tag{5.12}$$

As it was the case for independent sets of vertices in the previous section, small acyclic sets of edges are easily found. The empty set is acyclic. A singleton e is acyclic unless it is a loop. Finding a maximal acyclic edge set is a more challenging problem which, computationally, corresponds to the instance $X, P := Edge, acyclic$ of specification (5.1).

Checking the Requirements We now proceed to verify that the algorithmic solution presented in Figure 5.9 applies to this instance by verifying that predicate *acyclic* adequately meets the requirements. We will capitalise on facts used in Section 2.6 and in the proof of Proposition 2.101, in pages 38-41 of Section 2.7.

Subset-closedness of *acyclic* was shown and used in page 38. That does away with (5.2). Definitions of *acyclic* (5.11) and *cyclic* (5.12) and the empty range rule of predicate calculus imply (5.3). To show incrementality of *acyclic*, we will make use of fact (c1) in page 40. Assume $e \subseteq \overline{E}$ and then manipulate thus:

$$\begin{array}{ll} & \mathit{acyclic}\,(E \cup e) \\ \equiv & \quad \{\ \mathit{acyclic} \text{ is subset-closed (5.2)},\ E \subseteq E \cup e\ \} \\ & \mathit{acyclic}\,E\ \wedge\ \mathit{acyclic}\,(E \cup e) \\ \equiv & \quad \left\{ \begin{array}{l} \text{(c1), the proviso is given by the first conjunct and} \\ \text{assumption } e \subseteq \overline{E},\ \mathit{acyclic} \text{ negates } \mathit{cyclic} \text{ (5.11)} \end{array} \right\} \\ & \mathit{acyclic}\,E\ \wedge\ e \not\preceq E\ . \end{array}$$

Therefore, taking $Q\,(E, e) := e \not\preceq E$ makes the incrementality requirement (5.4) hold for $P := \mathit{acyclic}$. Unfolding the definition of $\preceq$ (2.90) expands $Q\,(E, e)$ as $\mathit{adj}\,e \not\subseteq \mathit{join}\,E$ and, then, the following definitions suggest themselves to make property (5.7) hold:

$$\begin{array}{rcl} f & := & \mathit{join} \qquad \text{(and, therefore, } f\,\emptyset = \mathit{id}\text{)}\ , \\ Q'\,(Pt, e) & := & \mathit{adj}\,e \not\subseteq Pt\ . \end{array}$$

For defining operator $\oplus$ in such a way that (5.8) holds, we borrow fact (2.98) from Section 2.6 where it was used to prove the Two Gates rule:

$$\mathit{join}\,(E \cup e) \ = \ \mathit{join}\,E \cdot (\mathit{id} \cup \mathit{adj}\,e) \cdot \mathit{join}\,E\ .$$

Hence, we define:

$$Pt \oplus e \ := \ Pt \cdot (\mathit{id} \cup \mathit{adj}\,e) \cdot Pt\ .$$

Variable y of the general program in Figure 5.9 gets to hold an equivalence relation in this instance. A common way of implementing equivalence relations is through the partitions that correspond to their quotient sets. The well-known *Union-Find* problem of manipulating partitions has been thoroughly studied and efficient implementations are available –see [36, Chapter 22] for a review–. Its operations can be expressed, using the related equivalence relations instead of the partitions themselves, as follows:

$$\mathit{Same}\,(Pt, a, b) \ := \ a \cdot b^{\circ} \subseteq Pt\ ,$$

$$Union\,(Pt, a, b) \quad := \quad Pt \cdot (id \cup a \cdot b^{\circ} \cup b \cdot a^{\circ}) \cdot Pt \ .$$

where operation *Same* is built on, and corresponds to the use given to, *Find*. Due to atomic adjacency (2.80), components Q' and $\oplus$ calculated above can be defined in terms of, respectively, *Same* and *Union* using the extreme vertices $(x1 \cdot e)$ and $(x2 \cdot e)$ of edge e as parameters.

5.6 Related Work

First, we remark again on the relationship of the general development presented in this chapter to the field of combinatorial optimisation, as commented on at the end of Section 5.1. Our algorithmic solutions in Figure 5.6 and Figure 5.9 are general matroid-based greedy algorithms which we have formally develop using a calculational style, a style of presentation not commonly found in the optimisation or algorithmics literature –see e.g. [36, Chapter 17] or [92, Chapter 7]–. The contents of this thesis are only slightly connected to the broad area of optimisation algorithms. Specifically, only the maximisation aspect of the problem of computing representatives in Chapter 4, the relationship to matroids and greedy algorithms of this chapter, and the treatment of the problem of computing minimum spanning trees in Chapter 6 witness such a connection. Bird, Curtis and de Moor have treated optimisation algorithms in a derivational style, using the calculus of binary relations and the categorical approach to datatypes, extensively [24, 25, 26, 28, 37, 38, 39, 111].

Russling treats the problem of computing maximal independent vertex sets in [131]. Unlike our computation of one of such sets, Rnssling deals with the computation of the family of *all* maximal independent sets of a given graph. At the end of the development, he remarks that "for practical use the algorithm should be implemented by standard backtracking techniques". Such an implementation could be realised using a logic programming language like Prolog [35]. Regarding this possibility, there is an interesting remark in [114, Section 8] about guarded commands, backtracking and Prolog, which could be elaborated upon to formally link our development to an implementation that can prodnce all maximal independent sets or that can backtrack over the production of one maximal independent set until it fulfills some other required condition. This is a nice thought in relation to C.A.R. Hoare's promoted ideal of unifying different theories and paradigms of programming [74, 76].

Berghammer treats the problem of computing spanning trees in [16], for a connected input graph. In relation to our generalisation of spanning trees as connectedness-preserving forests, the connectedness restriction of the input

graph is in our opinion unnecessary, as no greater manipulability is gained in general by such an assumption –see our remarks on this kind of restrictions in page 37–. Berghammer further restricts the graph, not only to be connected, but also to have a non-empty set of edges. Both decisions seem to be guided by the willingness to obtain a particular kind of algorithmic solution, viz. one akin to Prim's algorithm for the computation of minimum spanning trees [126], which was originally designed to cater only for connected graphs with non-empty vertex sets. Our solution is, on the other hand, akin to Kruskal's algorithm for the computation of minimum spanning trees [90]. We tackle both such minimisation algorithms, for unrestricted input graphs, in Chapter 6. A final difference between [16] and our treatment is that Berghammer presents the problem in terms of simple graphs –under the terminology used in this document; see page 28 of Section 2.5–, thus not dealing with edges as separate entities in the graph, while we take edges to be the relevant component of graphs in the definition of spanning trees.

Chapter 6

Computing (more than) Minimum Spanning Trees

We have already looked at spanning trees of undirected graphs and their formalisation within the calculus of relations –Section 2.7–. If a weight, or cost, is assigned to each edge of an undirected graph, then every spanning tree of it also gets a cost: the sum of the weights of all the edges it comprises. An interesting problem with many practical applications is that of computing spanning trees of minimum cost. This problem is tackled in this chapter under our relational-calculational approach.

Most, if not all, introductions to graph algorithmics include a section on the problem of computing minimum spanning trees, usually presenting two well-known algorithms that solve this problem: one due to Kruskal [90], and another commonly attributed to Prim [126] though actually invented earlier by Jarník [79]. We will present the construction of both algorithms.

When the notion of spanning trees was formalised in Section 2.7, we actually did so for the more general notion of connectedness-preserving forests. The former were then indirectly treated as an instance of the latter and the restriction of dealing only with connected graphs was sent away. Likewise, this chapter only makes use of the formal concept of connectedness-preserving forests and, therefore, we could have more accurately titled it as "Computing Minimum Connectedness-Preserving Forests". We chose not to do so for marketing reasons.

Section 6.1 presents a formal specification of the problem. Section 6.2 postulates an appropriate combination of invariant, guard and variant that altogether set up an iteration that refines the initial specification. Section 6.3 then presents an initial exploration of properties of the postulated invariant. Such properties allow us to construct Kruskal's algorithm in Section 6.4. The construction of Prim's algorithm requires some more graph concepts

and further properties of the invariant. These are presented in Section 6.5, which offers a little calculational theory of cuts in a graph, and in Section 6.6, which explores connections between the invariant and the existence of certain cuts. Section 6.7 then presents the construction of Prim's algorithm. Finally, Section 6.8 reviews related work.

6.1 Specification

This section presents the formal specification of our computational problem. We are given an undirected graph $(Vert, Edge, x1, x2)$ along with a function $weight : \mathbf{R} \leftarrow Edge$, which is assumed to provide non-negative weights. What the computation must deliver is a subset $E : \mathsf{Vec}\, Edge$ that comprises a connectedness-preserving forest, i.e. such that $cpf\, E$ (2.100) holds, and that has minimum cost, as determined by function $weight$, among all other such forests.

The function assigning a cost to each set of edges is:

$$cost\, E \;:=\; \langle +e : e \subseteq E : weight\, e \rangle \;, \tag{6.1}$$

where e ranges over elements of type $Edge$. The predicate that determines whether a set of edges is a connectedness-preserving forest of minimum cost is defined as follows:

$$mincpf\, E \;:=\; cpf\, E \;\wedge\; \langle \forall F : cpf\, F : cost\, E \leq cost\, F \rangle \;, \tag{6.2}$$

where F ranges over vectors of type $\mathsf{Vec}\, Edge$.

The formal specification of our problem then simply reads thus:

$$\begin{array}{l} [\![\ \mathbf{var}\ E : \mathsf{Vec}\, Edge\ ; \\ \quad E : [\ true\ ,\ mincpf\, E\ ] \\ ]\!] \end{array} \tag{6.3}$$

6.2 Setting Up an Iteration

Initial steps to refine specification (6.3) are taken in this section by providing the necessary ingredients to set up a correct iteration: invariant, guard and variant.

To start with, we observe the similarity of the problem that occupies us in this chapter and the problem in Section 5.5 of computing plain –as opposed to minimum-cost– connectedness-preserving forests. The latter was solved as an instance of the generic problem of computing maximal sets: a maximal

acyclic set of edges is a connectedness-preserving forest. The output of the program obtained in Section 5.5 then satisfies the first conjunct of *mincpf* in the postcondition of (6.3) and, therefore, it seems reasonable to attempt a reuse of the generic development presented in Chapter 5. We will indeed reuse the invariants therein to cater for the first conjunct of *mincpf*.

We thus postulate appropriate instances of the invariants of Chapter 5 as given by the substitution $P, A, B := acyclic, E, D$:

$$\begin{array}{lcl} Inv1 & := & acyclic\, E \ , \\ Inv2 & := & \langle \forall e : acyclic\,(E \cup e) : e \subseteq D \rangle \ . \end{array}$$

Fresh program variable D is also used in the guard $D \neq E$. This takes care of the first half of the postcondition, i.e. invariants *Inv1* and *Inv2* along with the negation of the guard imply $mxl\,(acyclic, E)$ and, by Proposition 2.101, this is equivalent to $cpf\,E$.

The second conjunct of *mincpf* in the postcondition adds the minimum-cost quality to the maximal forest that must be delivered as output. –From now on, we will call an acyclic set of edges a *forest*, whether it is maximal or not.– As dictated by the development of Chapter 5, forest E is initialised as empty and is then gradually augmented until no more additions are possible, which event signals that we have computed a maximal forest. We now need this maximal forest to be of minimum cost. First thought that might come to mind: maintain a minimum-cost forest all along, i.e. state as a third invariant that E must be of minimum-cost. Second thought: minimum-cost among what other forests? A minimum-cost forest among all forests is certainly not what we want since, having assumed that edge weights are non-negative, such a forest could only be the empty one or forests consisting exclusively of edges with weight zero. A minimum-cost forest compared to the maximal forests does not make sense either, since a non-maximal forest is likely to be of lesser cost than a maximal forest irrespectively of this leading to the obtention of a finally minimum-cost maximal forest or not. This last remark provides the key insight: we want to gradually grow a forest in such a way that it *leads* to a maximal forest of minimum-cost, i.e. to a forest for which *mincpf* holds.

We then formalise our third invariant as follows:

$$Inv3 \ := \ \langle \exists M : E \subseteq M : mincpf\, M \rangle \ ,$$

and give the usual name to the whole invariant:

$$Inv \ := \ Inv1 \wedge Inv2 \wedge Inv3 \ .$$

Now we must prove that the whole postcondition will hold at the end of the

iteration. We argue thus:

$$
\begin{array}{ll}
 & \mathit{Inv} \ \wedge \ D = E \\
\Rightarrow & \quad \{ \ \mathit{Inv1} \ , \ \mathit{Inv2} \ \text{and guard as in Chapter 5} \ \} \\
 & \mathit{mxl}\,(\mathit{acyclic}, E) \ \wedge \ \mathit{Inv3} \\
\equiv & \quad \{ \ \text{definition of } \mathit{Inv3} \ , \ \text{predicate calculus} \ \} \\
 & \langle \exists M \ : \ E \subseteq M \ : \ \mathit{mxl}\,(\mathit{acyclic}, E) \ \wedge \ \mathit{mincpf}\, M \ \rangle \\
\equiv & \quad \left\{ \begin{array}{l} \text{by definitions of } \mathit{mincpf} \ (6.2) \ \text{and } \mathit{cpf} \ (2.100) \ , \\ \text{we have that } \ \mathit{mincpf}\, M \ \Rightarrow \ \mathit{acyclic}\, M \end{array} \right\} \\
 & \langle \exists M \ : \ E \subseteq M \ : \ \mathit{mxl}\,(\mathit{acyclic}, E) \ \wedge \ \mathit{acyclic}\, M \ \wedge \ \mathit{mincpf}\, M \ \rangle \\
\Rightarrow & \quad \left\{ \begin{array}{l} \text{definition of } \mathit{mxl} \ (2.43) \ \text{with } \ P, A := \mathit{acyclic}, E \\ \text{and instantiation with } \ B := M \end{array} \right\} \\
 & \langle \exists M \ : \ E \subseteq M \ : \ M = E \ \wedge \ \mathit{mincpf}\, M \ \rangle \\
\equiv & \quad \{ \ \text{one-point rule} \ \} \\
 & \mathit{mincpf}\, E \ .
\end{array}
$$

As for the rest of the set-up of the iteration, we do not fix the initialisation statement yet since it will vary from Kruskal's to Prim's algorithm. However, we remark that the initialisation statement from Chapter 5 works for the third invariant as well. We borrow the variant from Chapter 5 and define:

$$\mathit{Prg} \ := \ (D - E \ \subset \ D_0 - E_0) \ .$$

The first refinement step has been completed:

$$
\begin{array}{ll}
 & E : [\ \mathit{true} \ , \ \mathit{mincpf}\, E \ ] \\
\sqsubseteq & \quad \left\{ \begin{array}{l} \text{introduce local block and iteration} \\ \text{according to discussion above} \end{array} \right\} \\
 & [\!| \ \textbf{var} \ D : \mathsf{Vec}\, \mathit{Edge} \ ; \\
 & \quad E, D : [\ \mathit{true} \ , \ \mathit{Inv} \ ] \ ; \\
 & \quad \textbf{do} \ D \neq E \ \rightarrow \ E, D : [\ D \neq E \ \wedge \ \mathit{Inv} \ , \ \mathit{Inv} \ \wedge \ \mathit{Prg} \ ] \ \textbf{od} \\
 & |\!]
\end{array}
$$

6.3 Exploring Some Properties

This section investigates properties that will allow us to refine the specification statements left in the set-up of the iteration. After this section, we will be able to construct Kruskal's algorithm, but Prim's algorithm will require further elaboration in later sections.

As expected, the investigation starts with an analysis of what can be taken from the development carried out in Chapter 5. The relevant properties that gave rise to the general iteration body there were proved in pages 92-93. The appropriate instances, using substitution $P, A, B := acyclic, E, D$ as before plus $x := e$ and $Q\,(E, e) := e \not\preceq E$, read as follows:

$$(Inv1 \wedge Inv2)\,[\,E := E \cup e\,] \quad \Leftarrow \quad Inv1 \wedge Inv2 \wedge e \not\preceq E \ , \tag{6.4}$$

$$(Inv1 \wedge Inv2)\,[\,D := D - e\,] \quad \Leftarrow \quad Inv1 \wedge Inv2 \wedge e \preceq E \ , \tag{6.5}$$

provided that $e \subseteq D - E$, which also guarantess that the iteration makes progress according to Prg.

Our current problem uses one more invariant, viz. $Inv3$. Hence, we need to explore how the assignments involved in properties (6.4) and (6.5) above interact with this third invariant.

Extending the Diminishing of D We start out with (6.5) since it is much simpler to deal with. This is due to the fact that assignments to variable D do not affect $Inv3$. Therefore, we can straightaway state that:

$$Inv\,[\,D := D - e\,] \quad \Leftarrow \quad Inv \ \wedge \ e \preceq E \ \wedge \ e \subseteq D - E \ . \tag{6.6}$$

Extending the Augmentation of E Tackling (6.4) is not as trivial. It will occupy us for all the rest of this section. We need to investigate under what conditions $Inv3\,[\,E := E \cup e\,]$ follows from $Inv3$. Let us proceed to do so.

Assume $Inv3$ taking M' as witness and, therefore, once the definition of $mincpf$ (6.2) has been expanded, we have that the following holds:

$$E \subseteq M' \ \wedge \ cpf\,M' \ \wedge \ (\forall\, F \,:\, cpf\,F \,:\, cost\,M' \leq cost\,F) \ . \tag{6.7}$$

Since we are aiming to extend proposition (6.4), we also assume its provisos, in particular:

$$e \not\preceq E \ \wedge \ e \subseteq D - E \ . \tag{6.8}$$

We now have to prove $Inv3\,[\,E := E \cup e\,]$, which, again unfolding definition of $mincpf$ (6.2), reads thus:

$$\left.\begin{array}{l} (\exists\, M \,:\, E \cup e \subseteq M \,:\, cpf\,M \\ \qquad\qquad\qquad\qquad \wedge \ (\forall\, F \,:\, cpf\,F \,:\, cost\,M \leq cost\,F)) \ . \end{array}\right\} \tag{6.9}$$

Assumption (6.7) says that E is included in a minimum-cost connectedness-preserving forest M'. If new edge e is also included in M', we are done.

But, if e is not in M', we need a different witness to make (6.9) hold. Somewhat naively, one could first think of $M' \cup e$, but this is clearly not a forest due to the acyclicity property of M' being maximal. It is more reasonable to try to exchange edge e for some other edge m included in M', i.e. to analyse under what circumstances $(M' - m) \cup e$ serves as a witness for (6.9), with m being some edge in M'.

We will now proceed with such an aualysis. Hence, from now on assume that edges e and m are such that:

$$e \subseteq \overline{M'} \quad \wedge \quad m \subseteq M' \ , \tag{6.10}$$

which implies the following:

$$e \subseteq \overline{M' - m} \quad \wedge \quad m \subseteq M' \cup e \ , \tag{6.11}$$

$$(M' - m) \cup e \quad = \quad (M' \cup e) - m \ . \tag{6.12}$$

Conditions for $(M' - m) \cup e$ to satisfy each of the three conjuncts in (6.9), i.e. the range and the two conjuncts in the body, will be calculated.

Inclusion in New Witness Regarding the range of (6.9), we have that:

$$E \cup e \subseteq (M' - m) \cup e \quad \equiv \quad m \subseteq \overline{E} \ . \tag{6.13}$$

This can be proved using basic properties of the lattice structure of the calculus of relations, and using also the facts that $E \subseteq M'$ and $e \subseteq \overline{E}$ as given by, respectively, (6.7) and (6.8).

New Witness as a CP Forest For the first conjunct in the body of (6.9), we manipulate as follows:

$$\begin{array}{ll}
 & \mathit{cpf}\ ((M' - m) \cup e) \\
\equiv & \quad \{ \text{ definition of } \mathit{cpf}\ (2.100)\,,\ (6.12)\ \} \\
 & \mathit{acyclic}\ ((M' - m) \cup e) \quad \wedge \quad \mathit{connpre}\ ((M' \cup e) - m) \\
\equiv & \left\{ \begin{array}{l} \text{property (c1) in page 40 with } E, e := M' - m, e \text{ and} \\ \text{property (c2) in page 40 with } E, e := M' \cup e, m\,; \\ \text{provisos given by (6.11) and by the middle conjunct of} \\ \text{(6.7) since, by subset- and superset-closedness:} \\ \quad \mathit{cpf}\ M' \ \equiv \ \mathit{acyclic}\ M' \ \wedge \ \mathit{connpre}\ M' \\ \quad\quad\quad \Rightarrow \ \mathit{acyclic}\ (M' - m) \ \wedge \ \mathit{connpre}\ (M' \cup e) \end{array} \right\} \\
 & e \not\preceq M' - m \quad \wedge \quad m \preceq (M' \cup e) - m \\
\equiv & \quad \{ \ (6.12) \ \} \\
 & e \not\preceq M' - m \quad \wedge \quad m \preceq (M' - m) \cup e
\end{array}$$

$$
\begin{array}{ll}
 & \left\{ \begin{array}{l} \text{by (6.7) and (6.10) we have } \mathit{acyclic}\, M' \text{ and } m \subseteq M', \\ \text{definition of } \mathit{acyclic} \text{ (2.92) then gives } m \not\preceq M' - m\,; \\ \text{this and the first conjunct give the provisos for the} \\ \text{Two Gates rule (2.97) with } d, e, E := m, e, M' - m \end{array} \right\} \\
\equiv & \\
 & e \not\preceq M' - m \;\wedge\; e \preceq (M' - m) \cup m \\
\equiv & \quad \{ \text{ by (6.10) we have } m \subseteq M', \text{ so } (M' - m) \cup m = M' \ \} \\
 & e \not\preceq M' - m \;\wedge\; e \preceq M' \\
\equiv & \left\{ \begin{array}{l} \text{since } e \subseteq \mathit{Edge} \text{ and inclusion implies covering (2.93)} \\ \text{we have } e \preceq \mathit{Edge}\,; \text{ by (6.7) we have } \mathit{connpre}\, M' \\ \text{and by (2.95) we then have } \mathit{Edge} \preceq M'\,; \text{ the second} \\ \text{conjunct then follows from transitivity of } \preceq \end{array} \right\} \\
 & e \not\preceq M' - m \;. \qquad (6.14)
\end{array}
$$

New Witness with Minimum-Cost Finally, the second conjunct in the body of (6.9) is manipulated thus:

$$
\begin{array}{ll}
 & (\forall F \;:\; \mathit{cpf}\, F \;:\; \mathit{cost}\, ((M' - m) \cup e) \;\leq\; \mathit{cost}\, F) \\
\equiv & \left\{ \begin{array}{l} (\Rightarrow) \text{ middle conjunct of (6.7), instantiation;} \\ (\Leftarrow) \text{ third conjunct of (6.7), transitivity of } \leq \end{array} \right\} \\
 & \mathit{cost}\, ((M' - m) \cup e) \;\leq\; \mathit{cost}\, M' \\
\equiv & \quad \{ \text{ definition of } \mathit{cost} \text{ (6.1), assumptions (6.10) } \} \\
 & \mathit{cost}\, M' - \mathit{weight}\, m + \mathit{weight}\, e \;\leq\; \mathit{cost}\, M' \\
\equiv & \quad \{ \text{ arithmetic } \} \\
 & \mathit{weight}\, e \;\leq\; \mathit{weight}\, m \;. \qquad (6.15)
\end{array}
$$

Does There Exist a New Witness? We have calculated sufficient and necessary conditions for $(M' - m) \cup e$ to be a witness of existential quantification (6.9). These conditions are the right-hand side of equivalence (6.13), and propositions (6.14) and (6.15). We collect these three requirements in a single formula:

$$
m \subseteq \overline{E} \;\wedge\; e \not\preceq M' - m \;\wedge\; \mathit{weight}\, e \;\leq\; \mathit{weight}\, m \;. \qquad (6.16)
$$

But the question still arises whether such a wituess actually exists. More specifically, we need to prove that, if e is not included in M', there exists an edge m in M' that satisfies (6.16).

Let us first appeal to some intuitive reasoning in order to find our way towards a formal proof regarding the existence of such an edge m. Forest M' is a connectedness-preserving one. Therefore, the extremes of e must be

connected through a path in M', a path which would become a cycle if we added e to M'. Hence, the edge m we want to pick out from M' must come from such a path in order to avoid the creation of a cycle. That is what the second conjunct of (6.16) states: that once m is out of the way, the presence of e does not endanger the absence of cycles. But, does such an m exist? Yes, the path at issue must be non-empty and some of its edges must not be included in E since, if all the path in M' connecting the extremes of e were in E, then e should be creating a cycle when added to E. And this is not the case, as guaranteed by $e \not\preceq E$ in (6.8).

Let us now phrase the above reasoning in a fully formal fashion. First, we have that $e \preceq M'$. This fact was used in the last step of the calculation leading to (6.14), and its proof is in the justification of that step. This statement, $e \preceq M'$, is the formal counterpart of the existence of a path in M' connecting the extremes of e. But there might be more to just such a path in M', i.e. there might be several other edges in M' not included in the path connecting e. Let us get rid of such edges. We reason as follows: There must be a minimal set N included in M' such that $e \preceq N$. Its existence follows from the fact that we have assumed our graphs to be finite and they thus comprise finite sets of edges. Hence, edges can be drawn from M' while preserving property $(e \preceq)$ until no more edges can be taken out without violating this property.

Therefore, we know that there exists a set N such that:

$$N \subseteq M' \;\wedge\; e \preceq N \;\wedge\; \langle \forall n : n \subseteq N : e \not\preceq N - n \rangle \;. \tag{6.17}$$

–Note that N must comprise *exactly* the set of edges in the path connecting the extremes of e, and that N is actually the unique *minimum* set included in M' satisfying $(e \preceq)$ rather than just a minimal one, but we do not need to use such a fact.–

When reasoning informally, we said that not all the path connecting the extremes of e could be included in E, since this would contradict assumption $e \not\preceq E$ given by (6.8). We now prove that claim formally:

$$\begin{array}{ll} & N - E = \emptyset \\ \equiv & \quad \{\ \emptyset \text{ least relation, universal property of subtraction (2.7)}\ \} \\ & N \subseteq E \\ \Rightarrow & \quad \{\ \text{inclusion implies covering (2.93)}\ \} \\ & N \preceq E \\ \Rightarrow & \quad \{\ \text{middle conjunct of (6.17), transitivity of covering}\ \} \\ & e \preceq E \\ \equiv & \quad \{\ \text{first conjunct of assumption (6.8)}\ \} \end{array}$$

false .

Hence,

$$N - E \neq \emptyset \ . \tag{6.18}$$

And we claim that any edge m drawn from $N - E$ will fulfil our needs, i.e. (6.16). For the time being, we only claim that:

$$\left.\begin{array}{l} (\forall m : m \subseteq N - E : m \subseteq M' \ \wedge \\ \qquad\qquad m \text{ satisfies the first two conjuncts of (6.16)}) \ . \end{array}\right\} \tag{6.19}$$

We will deal with the third and last conjunct of (6.16) later.

Proof of (6.19):

Assume $m \subseteq N - E$. Hence, m is included both in N and in $\overline{E}$ and, also using the first conjunct of (6.17), we have thus obtained both $m \subseteq M'$ and the first conjunct of (6.16): $m \subseteq \overline{E}$.

We now observe that *acyclic* N must hold, due to the fact that *acyclic* M' holds by the middle conjunct of (6.7), that $N \subseteq M'$ holds by (6.17) and that *acyclic* is subset-closed. Hence, since m is included in N, on account of the third conjunct of (6.17) and of *acyclic* N, we have:

$$e \not\preceq N - m \ \wedge \ m \not\preceq N - m \ . \tag{6.20}$$

Also, we have $N = (N - m) \cup m$ which, by substitution in the middle conjunct of (6.17), gives:

$$e \preceq (N - m) \cup m \ .$$

The Two Gates rule (2.97) can now be applied with $d, e, E := e, m, N - m$ and (6.20) as provisos, to obtain:

$$m \preceq (N - m) \cup e \ .$$

Inclusion of N in M' by (6.17), plus the fact that inclusion implies covering (2.93) and transitivity of covering, then implies:

$$m \preceq (M' - m) \cup e \ . \tag{6.21}$$

We are now ready to prove that m satisfies the second conjunct of (6.16):

$$\begin{array}{ll} & e \preceq M' - m \\ \equiv & \quad \{ \text{ reflexivity of } \preceq, \text{ union/}\preceq \text{ (2.96) } \} \\ & (M' - m) \cup e \preceq M' - m \\ \Rightarrow & \quad \{ \text{ (6.21), transitivity of } \preceq \ \} \end{array}$$

$$
\begin{array}{ll}
 & m \preceq M' - m \\
\equiv & \left\{ \begin{array}{l} \text{second conjunct of (6.7) implies } \mathit{acyclic}\, M' , \\ m \text{ included in } M' , \text{ definition of } \mathit{acyclic}\ (2.92) \end{array} \right\} \\
 & \mathit{false} \ .
\end{array}
$$

□

Finishing Off the Augmentation of E To finalise, we deal with the third and last requirement on m in (6.16): $\mathit{weight}\, e \leq \mathit{weight}\, m$. We will now proceed to postulate conditions that will guarantee its validity and, with them, finish off the rule for the maintenance of the whole invariant under augmentation of E.

As indicated by proviso (6.8) on edge e, set $D - E$ must be the source from where e is drawn. If we can prove that m must also belong to $D - E$, then requiring e to be of minimum weight in $D - E$ guarantees the satisfaction of the third requirement on m.

Up to this point, reasoning about the maintenance of *Inv3* has been based solely upon the assumption that *Inv3* holds initially. Now, we will also need the assumption that *Inv1* and *Inv2* hold. The key property still left to be proved is:

$$\langle \forall\, m \ :\ m \subseteq M' - E \ :\ m \subseteq D \rangle \ . \tag{6.22}$$

For ease of reference, we write out here the incrementality property (5.4) of Chapter 5 as instantiated in Section 5.5:

$$\mathit{acyclic}\,(F \cup f) \ \equiv \ \mathit{acyclic}\, F \ \wedge \ f \not\preceq F \quad \text{provided } f \subseteq \overline{F} \ . \tag{6.23}$$

Proof of (6.22)*:*

Assume $m \subseteq M' - E$. Then,

$$
\begin{array}{ll}
 & m \subseteq D \\
\Leftarrow & \quad \{\ \mathit{Inv2}\ \} \\
 & \mathit{acyclic}\,(E \cup m) \\
\equiv & \quad \left\{ \text{(6.23) with } F, f := E, m \text{, assumption } m \subseteq \overline{E} \text{, } \mathit{Inv1} \right\} \\
 & m \not\preceq E \ .
\end{array}
$$

And this last proposition is shown to hold thus:

$$m \preceq E$$

$$\Rightarrow \quad \left\{ \begin{array}{l} \text{first conjunct of } \mathit{Inv3}\ (6.7) \text{ and assumption} \\ m \subseteq \overline{E} \text{ imply } E \subseteq M' - m \text{, inclusion implies} \\ \text{covering (2.93), transitivity of covering} \end{array} \right\}$$

$$m \preceq M' - m$$

$$\equiv \quad \left\{ \begin{array}{l} \text{second conjunct of } \mathit{Inv3}\ (6.7) \text{ implies } \mathit{acyclic}\, M' \text{,} \\ \text{assumption } m \subseteq M' \text{, definition of } \mathit{acyclic}\ (2.92) \end{array} \right\}$$

$$\mathit{false} \quad .$$

□

We then have, by (6.19) and (6.22), that any edge in $N - E$ also belongs to $D - E$. Hence, if e is of minimum weight in $D - E$ and m is drawn from $N - E$, it must be the case that m satisfies the third conjunct of (6.16).

We conclude that, if preorder $R : \mathit{Edge} \leftarrow \mathit{Edge}$ is defined as

$$e1\ (R)\ e2 \quad := \quad \mathit{weight}\ e1 \ \leq\ \mathit{weight}\ e2 \quad ,$$

and edge e is drawn from the set $\mathit{min}\,(R, D - E)$ then, on account of (6.19) and (6.22), the following holds:

$$\langle\, \forall\, m \ :\ m \subseteq N - E \ :\ m \subseteq M' \ \wedge \ m \text{ satisfies all of (6.16)} \,\rangle\,. \quad (6.24)$$

Finally, we can state the sought after counterpart of (6.6):

$$\mathit{Inv}\,[\,E := E \cup e\,] \ \Leftarrow \ \mathit{Inv} \ \wedge \ e \not\preceq E \ \wedge \ e \subseteq \mathit{min}\,(R, D - E)\,. \quad (6.25)$$

It follows from all the discussion above but, crucially, from the statements that guarantee the existence of a witness for *Inv3*: (6.18) and (6.24).

6.4 Kruskal's Algorithm

The exploration carried out in the previous section is good enough for the construction of Kruskal's algorithm, which we proceed to do in this section. We do so by refining the specification statements left at the end of Section 6.2 as initialisation and iteration body.

Since the initialisation statement of Chapter 5 works for the new invariant *Inv3* as well, we reuse it:

$$E, D : [\ \mathit{true}\ ,\ \mathit{Inv}\ ] \ \sqsubseteq \ \ E, D := \emptyset, \mathit{Edge}\ .$$

The key properties spelt out in Section 6.3 that will be put to use in refining

the specification statement of the iteration body are (6.6) and (6.25):

$$Inv\,[\,D := D - e\,] \;\Leftarrow\; Inv \;\wedge\; e \preceq E \;\wedge\; e \subseteq D - E \;,$$
$$Inv\,[\,E := E \cup e\,] \;\Leftarrow\; Inv \;\wedge\; e \not\preceq E \;\wedge\; e \subseteq min\,(R, D - E) \;.$$

Only two remarks before developing the iteration body:

(i) In Chapter 5 it was proved that the guard $D \neq E$ is equivalent to $D - E \neq \emptyset$, provided the invariant holds. -See (5.5) and apply substitution $A, B := E, D$.- Hence, the precondition of the iteration body guarantees the existence of elements in $D - E$.

(ii) Set $min\,(R, D - E)$ is included in set $D - E$. Hence, if an edge belongs to the former, it also belongs to the latter. Furthermore, given that R is a connected preorder, $min\,(R, D - E)$ is non-empty whenever $D - E$ is non-empty.

Therefore, the following is a valid refinement:

$$\begin{array}{l} E, D : [\; D \neq E \;\wedge\; Inv \;,\; Inv \;\wedge\; Prg \;] \\ \sqsubseteq \quad \left\{ \begin{array}{l} \text{introduce local block and alternation} \\ \text{according to discussion and properties above} \end{array} \right\} \\ \quad |[\;\; \textbf{var}\;\; e : Edge \;; \\ \quad\quad e :\subseteq min\,(R, D - E) \;; \\ \quad\quad \textbf{if}\;\; e \preceq E \;\rightarrow\; D := D - e \\ \quad\quad [\!] \;\; e \not\preceq E \;\rightarrow\; E := E \cup e \\ \quad\quad \textbf{fi} \\ \quad]| \end{array}$$

This completes our first approximation to Kruskal's algorithm as a refinement of specification (6.3). Figure 6.26 shows the collected code. Note how similar it is to the program in Figure 5.6.

Data Refinement The similarity between the approximation to Kruskal's algorithm in Figure 6.26 and the first algorithmic solution to the general problem of Chapter 5 in Figure 5.6 allows us to apply to the former the same data refinement that was applied to the latter in Section 5.3. We use suitable instances of the coupling invariants *CI1* and *CI2* in page 95, viz. those obtained via $A, B := E, D$ plus $C, y := F, Pt$ and, from Section 5.5, $f := join$. Hence, the coupling invariants that introduce new program variables F and Pt are:

$$CI1 \;:=\; F \;=\; D - E \;,$$
$$CI2 \;:=\; Pt \;=\; join\,E \;.$$

```
|[ var E : Vec Edge ;
   |[ var D : Vec Edge ;
      E, D := ∅, Edge ;
      do D ≠ E →
           |[ var e : Edge ;
              e :⊆ min (R, D − E) ;
              if e ⪯ E → D := D − e
              [] e ⋠ E → E := E ∪ e
              fi
           ]|
      od
   ]|
]|
```

Figure 6.26: First Approximation to Kruskal's Algorithm, as Algorithmic Solution of (6.3)

The transformation of the program is carried out just as explained in Section 5.3, using the instances of auxiliary predicate Q' and operator $\oplus$ that were calculated in Section 5.5:

$$\begin{aligned} Q'(Pt, e) &:= adj\, e \not\subseteq Pt \ , \\ Pt \oplus e &:= Pt \cdot (id \cup adj\, e) \cdot Pt \ . \end{aligned}$$

Also, as suggested at the end of Section 5.5, operations *Same* and *Union* of the well-known *Union-Find* problem are used to implement Q' and $\oplus$.

The resulting program, our version of Krnskal's algorithm, is shown in Figure 6.27.

6.5 A Little Theory of Cuts

In this section we develop a little theory of cuts: partitions of size two of the vertex set of a graph. This will aid an extension of the rules of maintenance of the iuvariant in Section 6.3 in a way that will help us to construct Prim's algorithm.

Cuts, Respecting and Crossing Let G be a graph $(Vert, Edge, x1, x2)$. A *cut* of G is just a partitioning of *Vert* into two disjoint sets: $(V, \overline{V})$ for some V : Vec *Vert* . We identify a cut with either one of its halves, the other being uniquely determined through complementation.

```
|[ var E : Vec Edge ;
   |[ var F : Vec Edge ; Pt : Vert ← Vert ;
      E, F, Pt := ∅, Edge, id ;
      do F ≠ ∅ →
           |[ var e : Edge ;
              e :⊆ min (R, F) ; F := F − e ;
              if   Same (Pt , x1·e , x2·e) → skip
              [] ¬ Same (Pt , x1·e , x2·e) →
                              E, Pt := E ∪ e , Union (Pt , x1·e , x2·e)
              fi
           ||
      od
   ||
||]
```

Figure 6.27: Kruskal's Algorithm, after Data Refinement, as Algorithmic Solution of (6.3)

A set of edges can either respect or cross a cut V. It *respects* the cut if all of its edges have both extremes either in V or in $\overline{V}$. An alternative phrasing of this is saying that if one extreme of an edge is in V, so must be the other extreme. A set of edges *crosses* the cut if it does not respect it, which means that some of its edges have one extreme in V and the other in $\overline{V}$. Formally, given $E : \mathsf{Vec}\, Edge$ and a cut $V : \mathsf{Vec}\, Vert$, we define:

$$respect\,(E,V) \;:=\; adj\,E \cdot V \;\subseteq\; V \;, \tag{6.28}$$

$$cross\,(E,V) \;:=\; \neg\; respect\,(E,V) \;. \tag{6.29}$$

These definitions are consistent with the freedom of identifying cuts with either one of its components. Schröder's right-exchange rule (2.22) and symmetry of adjacency relations entails such consistency for *respect*:

$$respect\,(E,V) \;\equiv\; respect\,(E,\overline{V}) \;.$$

Consistency for *cross* then follows from (6.29) and propositional calculus.

A set of edges respects a cut if all its edges respect it, while a set of edges crosses a cut if at least one of its edges crosses it. Extensionality and distributivity over union allows a formal proof of this fact:

$$\begin{array}{ll} & respect\,(E,V) \\ \equiv & \quad \{ \text{ definition of } respect\ (6.28)\text{, extensionality } (2.29) \ \} \\ & adj\,(\cup e \;:\; e \subseteq E \;:\; e) \cdot V \;\subseteq\; V \end{array}$$

$$
\begin{array}{ll}
\equiv & \{ \text{ distribution of } adj \text{ and composition over union } \} \\
& (\cup e \ : \ e \subseteq E \ : \ adj\, e \cdot V \,) \ \subseteq \ V \\
\equiv & \left\{ \begin{array}{l} \text{universal property of union (2.3),} \\ \text{definition of } respect \text{ (6.28)} \end{array} \right\} \\
& \langle \forall e \ : \ e \subseteq E \ : \ respect\,(e, V) \rangle \ .
\end{array}
$$

Using de Morgan's laws of predicate calculus we also get the corresponding statement for *cross*:

$$respect\,(E, V) \ \equiv \ \langle \forall e \ : \ e \subseteq E \ : \ respect\,(e, V) \rangle \ , \tag{6.30}$$

$$cross\,(E, V) \ \equiv \ \langle \exists e \ : \ e \subseteq E \ : \ cross\,(e, V) \rangle \ . \tag{6.31}$$

Paths also Respect Cuts By definition, $respect\,(E, V)$ tells us that both extremes of each edge in E lie on the same side of the cut V. This implies that each path built from edges in E should also remain on only one side of the cnt. And vice versa: if each E-path stays in only one half of the cut then, in particular, so do E-paths of unitary length, i.e. those given by $adj\, E$. Formally:

$$respect\,(E, V) \ \equiv \ join\, E \cdot V \ \subseteq \ V \ . \tag{6.32}$$

It follows from property (2.68) of closure with $R, S := adj\, E, V$.

A cut built from a base set V through E-paths is respected by E:

$$respect\,(E \ , \ join\, E \cdot V) \ . \tag{6.33}$$

This follows from (6.32) and transitivity of joinability relations.

Monotonicity Properties For any cut V, predicate $respect\,(\_, V)$ is subset-closed. This fact follows from monotonicity of adj and transitivity of inclusion. The contrapositive rule of propositional calculus then implies that $cross\,(\_, V)$ is superset-closed. Formally,

$$
\begin{array}{lll}
E1 \subseteq E2 & \Rightarrow & \langle \forall V :: respect\,(E2, V) \ \Rightarrow \ respect\,(E1, V) \rangle \ , \\
E1 \subseteq E2 & \Rightarrow & \langle \forall V :: cross\,(E1, V) \ \Rightarrow \ cross\,(E2, V) \rangle \ .
\end{array}
$$

These statements, which can be seen as "$\subseteq$ to $\Rightarrow$" antimonotonicity and monotonicity properties, can be extended to "$\preceq$ to $\Rightarrow$" statements. Furthermore, the extension allows a strengthening to equivalences:

$$E1 \preceq E2 \ \equiv \ \langle \forall V :: respect\,(E2, V) \ \Rightarrow \ respect\,(E1, V) \rangle \ , \tag{6.34}$$

$$E1 \preceq E2 \ \equiv \ \langle \forall V :: cross\,(E1, V) \ \Rightarrow \ cross\,(E2, V) \rangle \ . \tag{6.35}$$

These two properties are equivalent by the contrapositive rule of propositional calculus. It then suffices to prove only one.

Proof of (6.34):

For $(\Rightarrow)$, we argue thus:

$$
\begin{array}{ll}
 & E1 \preceq E2 \\
\equiv & \quad \{ \text{ definition of } \preceq \ (2.90) \ \} \\
 & adj\ E1 \subseteq join\ E2 \\
\Rightarrow & \quad \{ \text{ monotonicity of composition } \} \\
 & \langle \forall V :: adj\ E1 \cdot V \subseteq join\ E2 \cdot V \rangle \\
\Rightarrow & \quad \{ \text{ transitivity of inclusion } \} \\
 & \langle \forall V :: join\ E2 \cdot V \subseteq V \ \Rightarrow \ adj\ E1 \cdot V \subseteq V \rangle \\
\equiv & \quad \{ (6.32), \text{ definition of } respect \ (6.28) \ \} \\
 & \langle \forall V :: respect\,(E2, V) \ \Rightarrow \ respect\,(E1, V) \rangle \ .
\end{array}
$$

For $(\Leftarrow)$, a more elaborate manipulation is needed. Assume the right-hand side holds and then proceed as follows:

$$
\begin{array}{ll}
 & E1 \preceq E2 \\
\equiv & \quad \{ \text{ extensionality } (2.29), \text{ union}/\preceq \ (2.96) \ \} \\
 & \langle \forall e : e \subseteq E1 : e \preceq E2 \rangle \\
\equiv & \quad \{ \text{ one-edge covering } (2.99) \ \} \\
 & \langle \forall e : e \subseteq E1 : (x1 \cdot e) \cdot (x2 \cdot e)^{\circ} \subseteq join\ E2 \rangle \\
\equiv & \quad \{ \text{ shunting of functions } (2.24) \ \} \\
 & \langle \forall e : e \subseteq E1 : x1 \cdot e \subseteq join\ E2 \cdot x2 \cdot e \rangle \\
\Leftarrow & \quad \left\{ \begin{array}{l} \text{by } (6.33),\ E2 \text{ respects } join\ E2 \cdot x2 \cdot e, \text{ by assumption,} \\ E1 \text{ then respects it as well; definition of } respect \ (6.28) \end{array} \right\} \\
 & \langle \forall e : e \subseteq E1 : x1 \cdot e \subseteq adj\ E1 \cdot join\ E2 \cdot x2 \cdot e \rangle \\
\Leftarrow & \quad \{ \text{ monotonicity of } adj \text{, reflexivity of joinability relations } \} \\
 & \langle \forall e : e \subseteq E1 : x1 \cdot e \subseteq adj\ e \cdot x2 \cdot e \rangle \\
\equiv & \quad \{ \text{ shunting of functions } (2.24) \ \} \\
 & \langle \forall e : e \subseteq E1 : (x1 \cdot e) \cdot (x2 \cdot e)^{\circ} \subseteq adj\ e \rangle \\
\equiv & \quad \{ \text{ atomic adjacency } (2.80), \text{ union } \} \\
 & true \ .
\end{array}
$$

□

Crossing For later use, we give a name to the set of all edges that cross a given cut:

$$\mathit{crossing}\, V \;:=\; \langle \cup e \,:\, \mathit{cross}\,(e, V) \,:\, e \rangle \;. \tag{6.36}$$

6.6 Exploring Some More Properties

As a follow-up to Section 6.3, this section presents a deeper exploration of the invariants on which the construction of Kruskal's algorithm was based. Such an exploration results on the obtention of a fourth invariant, essential for the construction of Prim's algorithm.

Augmentation of E and Cuts Recall rule (6.25):

$$\mathit{Inv}\,[\,E := E \cup e\,] \;\Leftarrow\; \mathit{Inv} \;\wedge\; e \not\preceq E \;\wedge\; e \subseteq \mathit{min}\,(R, D - E) \;.$$

Kruskal's algorithm, as stated in this rule, augments E with a *global* minimum e, i.e. minimum within $D - E$. Prim's algorithm makes use of the fact that the new edge e can be selected as a *local* minimum, i.e. minimum within a subset of $D - E$, provided such a subset complies with certain restrictions.

This is the point where the theory of cuts is put to use. We know that in order to safely add e to E it must be the case that $e \not\preceq E$. By cut-monotonicity (6.34), this is equivalent to the existence of a cut V, say, respected by E and crossed by e. The set of edges that cross such a cut V is a *safe* set, in the sense that selecting e to be of minimum weight within it guarantees we are heading towards a minimum-cost connectedness-preserving forest in the same way that selecting a global minimum edge does. Formally:

$$\left.\begin{array}{l} \mathit{Inv}\,[\,E := E \cup e\,] \;\Leftarrow \\ \quad \mathit{Inv} \;\wedge\; \langle \exists\, V \,:\, \mathit{respect}\,(E, V) \,:\, e \subseteq \mathit{min}\,(R, \mathit{crossing}\, V) \rangle \;. \end{array}\right\} \tag{6.37}$$

Also, it is the case that the crossing of such a cut is a subset of $D - E$, i.e. it is the case that:

$$\langle \forall\, V \,:\, \mathit{respect}\,(E, V) \,:\, \mathit{crossing}\, V \subseteq D - E \rangle \;\Leftarrow\; \mathit{Inv} \;. \tag{6.38}$$

We first prove this last proposition because we will need it in the proof of rule (6.37).

Proof of (6.38)*:*

Assume Inv. Then, for any cut V, argue thus:

$$
\begin{array}{ll}
 & \mathit{crossing}\, V \subseteq D - E \\
\equiv & \quad \{ \text{ extensionality } \} \\
 & \langle \forall e : e \subseteq \mathit{crossing}\, V : e \subseteq D - E \rangle \\
\equiv & \quad \{ \text{ definition of } \mathit{crossing}\ (6.36), \text{ subtraction } \} \\
 & \langle \forall e : \mathit{cross}\,(e, V) : e \subseteq D \ \wedge\ e \subseteq \overline{E} \rangle \\
\Leftarrow & \quad \{\ \mathit{Inv2}\ \} \\
 & \langle \forall e : \mathit{cross}\,(e, V) : \mathit{acyclic}\,(E \cup e) \ \wedge\ e \subseteq \overline{E} \rangle \\
\equiv & \quad \{ \text{ incrementality of } \mathit{acyclic}\ (6.23),\ \mathit{Inv1}\ \} \\
 & \langle \forall e : \mathit{cross}\,(e, V) : e \not\preceq E \ \wedge\ e \subseteq \overline{E} \rangle \\
\equiv & \quad \{ \text{ distribution of universal quantification over conjunction } \} \\
 & \langle \forall e : \mathit{cross}\,(e, V) : e \not\preceq E \rangle \ \wedge\ \langle \forall e : \mathit{cross}\,(e, V) : e \subseteq \overline{E} \rangle \\
\equiv & \quad \left\{ \begin{array}{l} \text{contrapositive, twice; } \mathit{respect} \text{ is the negation of} \\ \mathit{cross}\ (6.29);\ \text{extensionality of } \mathit{respect}\ (6.30) \end{array} \right\} \\
 & \langle \forall e : e \preceq E : \mathit{respect}\,(e, V) \rangle \ \wedge\ \mathit{respect}\,(E, V) \\
\equiv & \quad \{ \text{ cut-monotonicity } (6.34) \} \\
 & \mathit{respect}\,(E, V)\ .
\end{array}
$$

□

We now proceed to prove rule (6.37). The proof will reuse a good deal of the reasoning presented in Section 6.3.

Proof of (6.37)*:*

Assume the antecedent, i.e.

$$\mathit{Inv} \ \wedge\ \langle \exists V : \mathit{respect}\,(E, V) : e \subseteq \mathit{min}\,(R, \mathit{crossing}\, V) \rangle\ .$$

The assumption on e, by definition of min (2.60), entails $e \subseteq \mathit{crossing}\, V$. The rest of the assumptions, viz. Inv and the assumption on E, then give us, by proposition (6.38) proved above, that $e \subseteq D - E$. Also, by definition of $\mathit{crossing}$ (6.36), we have $\mathit{cross}\,(e, V)$ and, then, the assumption on E plus cut-monotonicity (6.34) imply $e \not\preceq E$. All of this allow us to use property (6.4) to conclude $(\mathit{Inv1} \ \wedge\ \mathit{Inv2})\,[E := E \cup e]$.

Proving $\mathit{Inv3}\,[E := E \cup e]$ is, as in Section 6.3, the tricky part. Recall that the big issue was proving the existence of a witness for existential quantification (6.9) in the case that e was not included in M'. And also recall that the existence of such a witness followed from the existence of an edge m in

M' satisfying (6.16). Such an edge m was proved to exist as a consequence of several facts. First, (6.18):

$$N - E \neq \emptyset \ ;$$

second, (6.19):

$$\langle \forall m : m \subseteq N - E : m \subseteq M' \ \wedge \\ m \text{ satisfies the first two conjuncts of (6.16)} \rangle \ ;$$

and, finally, (6.22):

$$\langle \forall m : m \subseteq M' - E : m \subseteq D \rangle \ .$$

The first two properties show the existence of edges in M' satisfying the first two conjuncts of (6.16). The last property show that such edges must also belong to $D - E$, which implies that, were e selected to be of minimum weight in $D - E$, each edge m in $N - E$ would satisfy the third and last conjunct of (6.16): $\mathit{weight}\, e \leq \mathit{weight}\, m$.

In our new circumstances, all we need to prove is:

$$(N - E) \cap \mathit{crossing}\, V \neq \emptyset \ . \tag{6.39}$$

This is snfficient since, from property (6.19) above, edges that are drawn from $(N - E) \cap \mathit{crossing}\, V$ belong to M' and satisfy the first two conjuncts of (6.16). Since edge e has now been chosen to be of minimum weight in $\mathit{crossing}\, V$, such edges would also satisfy the third conjunct of (6.16).

We now prove (6.39). The only other property of N we will need to borrow from Section 6.3 is the following: $e \preceq N$, the middle conjunct of (6.17). We manipulate thus:

$$\begin{array}{ll} & (N - E) \cap \mathit{crossing}\, V \neq \emptyset \\ \equiv & \quad \{ \text{ extensionality (2.30) } \} \\ & \langle \exists n :: n \subseteq (N - E) \cap \mathit{crossing}\, V \rangle \\ \equiv & \quad \{ \text{ intersection, subtraction, definition of } \mathit{crossing}\ (6.36) \ \} \\ & \langle \exists n :: n \subseteq N \ \wedge \ n \subseteq \overline{E} \ \wedge \ \mathit{cross}\,(n, V) \rangle \\ \equiv & \quad \left\{ \begin{array}{l} \text{assumption } \mathit{respect}\,(E, V), \text{ for extensionality} \\ \text{of } \mathit{respect}\ (6.30) \text{ and contrapositive rule, is} \\ \text{equivalent to } \langle \forall e : \mathit{cross}\,(e, V) : e \subseteq \overline{E} \rangle ; \\ \text{hence, the third conjunct implies the second} \end{array} \right\} \\ & \langle \exists n :: n \subseteq N \ \wedge \ \mathit{cross}\,(n, V) \rangle \\ \equiv & \quad \{ \text{ extensionality of } \mathit{cross}\ (6.31) \ \} \\ & \mathit{cross}\,(N, V) \end{array}$$

$$
\begin{array}{ll}
\Leftarrow & \{ \text{ assumption } \mathit{cross}\,(e,V)\text{, cut-monotonicity (6.35) } \} \\
& e \preceq N \\
\equiv & \{ \text{ property of } N \text{ referred to above, from (6.17) } \} \\
& \mathit{true} \ .
\end{array}
$$

□

We are now ready to use new rule (6.37) for augmentation of E. Note that the assignment statement involved not only maintains the invariant, but also guarantees that the iteration makes progress. Progress, as defined by Prg, is achieved if E is augmented with edges drawn from $D - E$. Due to proposition (6.38), this is indeed the case.

A New Invariant Old rule (6.25) –repeated at the beginning of this section– required set $D - E$ to be non-empty, which is guaranteed by the guard $D \neq E$ at the entry point of the iteration. New rule (6.37) requires a stronger precondition, the existence of a cut respected by E and with a non-empty crossing:

$$
\begin{array}{ll}
& \langle\, \exists V \,:\, \mathit{respect}\,(E,V) \,:\, \mathit{crossing}\, V \neq \emptyset \,\rangle \\
\equiv & \{ \text{ (6.38), intersection } \} \\
& \langle\, \exists V \,:\, \mathit{respect}\,(E,V) \,:\, (D-E) \cap \mathit{crossing}\, V \neq \emptyset \,\rangle \\
\equiv & \{ \text{ extensionality, intersection, definition of } \mathit{crossing} \text{ (6.36) } \} \\
& \langle\, \exists V \,:\, \mathit{respect}\,(E,V) \,:\, \langle\, \exists e \,:\, e \subseteq D-E \,:\, \mathit{cross}\,(e,V) \,\rangle \,\rangle \\
\equiv & \{ \text{ quantifications exchange } \} \\
& \langle\, \exists e \,:\, e \subseteq D-E \,:\, \langle\, \exists V \,:\, \mathit{respect}\,(E,V) \,:\, \mathit{cross}\,(e,V) \,\rangle \,\rangle \\
\equiv & \{ \text{ cut-monotonicity (6.34) } \} \\
& \langle\, \exists e \,:\, e \subseteq D-E \,:\, e \not\preceq E \,\rangle \ . \qquad (6.40)
\end{array}
$$

Since the guard already guarantees $D - E \neq \emptyset$, it would suffice to require the following new invariant $Inv4$ to make (6.40) hold at the entry point of the iteration:

$$
Inv4 \ := \ \langle\, \forall e \,:\, e \subseteq D-E \,:\, e \not\preceq E \,\rangle \ .
$$

We now dedicate ourselves to the task of working out a way to establish $Inv4$. In Kruskal's algorithm, each edge drawn from $D - E$ is added to E or taken out from D depending on whether it satisfies, respectively, $(\not\preceq E)$ or $(\preceq E)$. It seems reasonable to try to establish $Inv4$ by subtracting from D all edges that satisfy $(\preceq E)$ at once. We check up on our intuition by

the following formal manipulation:

$$
\begin{array}{ll}
 & \mathit{Inv4}\,[\,D := D - D'\,] \\
\equiv & \quad \{\ \textbf{definition of } \mathit{Inv4}\ \text{, substitution}\ \} \\
 & \langle \forall e\ :\ e \subseteq D - D' - E\ :\ e \not\preceq E \rangle \\
\equiv & \quad \{\ \textbf{subtraction}\ \} \\
 & \langle \forall e\ :\ e \subseteq D - E\ \wedge\ e \subseteq \overline{D'}\ :\ e \not\preceq E \rangle \\
\equiv & \quad \{\ \textbf{contrapositive}\ \} \\
 & \langle \forall e\ :\ e \subseteq D - E\ \wedge\ e \preceq E\ :\ e \subseteq D' \rangle \\
\equiv & \quad \{\ \textbf{universal property of union (2.3)}\ \} \\
 & \langle \cup e\ :\ e \subseteq D - E\ \wedge\ e \preceq E\ :\ e \rangle\ \subseteq\ D'\ . \qquad (6.41)
\end{array}
$$

Hence, *Inv4* can indeed be established by subtracting from D a set D' that satisfies (6.41). We take D' to be the smallest possible set:

$$D' \ :=\ \langle \cup e\ :\ e \subseteq D - E\ \wedge\ e \preceq E\ :\ e \rangle\ .$$

Define the new global invariant to be:

$$\mathit{Inv}' \ :=\ \mathit{Inv}\ \wedge\ \mathit{Inv4}\ .$$

We have already got at our disposal rules for the maintenance and establishment of *Inv*. Those rules can be used to establish *Inv*, and D' can then be subtracted from D to establish *Inv4*. Such a subtraction would not affect the validity of *Inv1* or *Inv3* since they do not mention D, but the validity of *Inv2* might be at risk. We will now show that such a subtraction from D maintains *Inv2*.

For showing that *Inv2* is maintained by the subtraction of D' from D, we will reuse rule (6.5) for the diminishing of D, which was borrowed from Chapter 5:

$$(\mathit{Inv1}\ \wedge\ \mathit{Inv2})\,[\,D := D - e\,]\ \Leftarrow\ \mathit{Inv1}\ \wedge\ \mathit{Inv2}\ \wedge\ e \preceq E\ ,$$

with the added proviso that $e \subseteq D - E$ holds. We are specifically interested in the core of this implication regarding *Inv2*, spelled out as follows, where the additional proviso has been incorporated into the antecedent of the implication:

$$\left.\begin{array}{l} \mathit{Inv2}\,[\,D := D - e\,]\ \Leftarrow \\ \qquad\qquad \mathit{Inv1}\ \wedge\ \mathit{Inv2}\ \wedge\ e \preceq E\ \wedge\ e \subseteq D - E\ . \end{array}\right\} \qquad (6.42)$$

We now argue as follows:

$$\mathit{Inv2}\,[\,D := D - D'\,]$$

$$
\begin{array}{ll}
\equiv & \{ \text{ definition of } \mathit{Inv2} \text{ , substitution } \} \\
& \langle \forall e : \mathit{acyclic}(E \cup e) : e \subseteq D - D' \rangle \\
\equiv & \{ \text{ subtraction, complementation } \} \\
& \langle \forall e : \mathit{acyclic}(E \cup e) : e \subseteq D \ \wedge \ D' \subseteq \overline{e} \rangle \\
\equiv & \{ \text{ extensionality (2.29), universal property of union (2.3) } \} \\
& \langle \forall e : \mathit{acyclic}(E \cup e) : e \subseteq D \ \wedge \ \langle \forall d : d \subseteq D' : d \subseteq \overline{e} \rangle \rangle \\
\equiv & \left\{ \begin{array}{l} \text{assume } D' \neq \emptyset \text{ to allow distribution of} \\ \text{conjunction over universal quantification} \end{array} \right\} \\
& \langle \forall e : \mathit{acyclic}(E \cup e) : \langle \forall d : d \subseteq D' : e \subseteq D \ \wedge \ d \subseteq \overline{e} \rangle \rangle \\
\equiv & \{ \text{ complementation, subtraction } \} \\
& \langle \forall e : \mathit{acyclic}(E \cup e) : \langle \forall d : d \subseteq D' : e \subseteq D - d \rangle \rangle \\
\equiv & \{ \text{ quantifications exchange, definition of } \mathit{Inv2} \ \} \\
& \langle \forall d : d \subseteq D' : \mathit{Inv2}[D := D - d] \rangle \\
\Leftarrow & \{ \text{ (6.42) with } e := d \ \} \\
& \langle \forall d : d \subseteq D' : \mathit{Inv1} \ \wedge \ \mathit{Inv2} \ \wedge \ d \preceq E \ \wedge \ d \subseteq D - E \rangle \\
\equiv & \left\{ \begin{array}{l} \text{assumption } D' \neq \emptyset \text{ again to distribute} \\ \text{conjunction over universal quantification} \end{array} \right\} \\
& \mathit{Inv1} \ \wedge \ \mathit{Inv2} \ \wedge \ \langle \forall d : d \subseteq D' : d \preceq E \ \wedge \ d \subseteq D - E \rangle \\
\equiv & \{ \text{ by definition of } D' \text{ the third conjunct is true } \} \\
& \mathit{Inv1} \ \wedge \ \mathit{Inv2} \ .
\end{array}
$$

Assumption $D' \neq \emptyset$ was needed a couple of times in the calculation above but, since $\mathit{Inv2}[D := D - \emptyset]$ is equivalent to $\mathit{Inv2}$, the rule is valid whether D' is empty or not:

$$\mathit{Inv2}[D := D - D'] \quad \Leftarrow \quad \mathit{Inv1} \ \wedge \ \mathit{Inv2} \ . \tag{6.43}$$

We conclude from (6.43), plus the manipulation on $\mathit{Inv4}[D := D - D']$ above, that subtracting D' from D establishes Inv' if Inv holds initially:

$$\mathit{Inv}'[D := D - \langle \cup e : e \subseteq D - E \ \wedge \ e \preceq E : e \rangle] \quad \Leftarrow \quad \mathit{Inv} . \tag{6.44}$$

6.7 Prim's Algorithm

Now, enough machinery is available to us for constructing Prim's algorithm. As with Kruskal's algorithm, we start off with the iteration that was set up in Section 6.2, but using invariant Inv' instead of Inv. Since Inv' is stronger than Inv, it is correct to keep the same guard. The refinement yet to be carried out is that of the specification statements left as initialisation and

The initialisation statement from Chapter 5 and rule (6.44) are combined thus:

$$
\begin{array}{ll}
 & E, D : [\ true\ ,\ Inv'\ ] \\
\sqsubseteq & \quad \{\ \text{introduce sequential composition}\ \} \\
 & E, D : [\ true\ ,\ Inv\ ]\ ;\ E, D : [\ Inv\ ,\ Inv'\ ] \\
\sqsubseteq & \quad \{\ \text{initialisation from Chapter 5, (6.44)}\ \} \\
 & E, D\ :=\ \emptyset, Edge\ ;\ D\ :=\ D - \langle \cup e\ :\ e \subseteq D - E\ \wedge\ e \preceq E\ :\ e \rangle \\
= & \quad \{\ \text{consecutive assignments}\ \} \\
 & E, D\ :=\ \emptyset,\ Edge - \langle \cup e\ :\ e \subseteq Edge\ \wedge\ e \preceq \emptyset\ :\ e \rangle \\
= & \quad \{\ \text{definition of}\ \preceq\ (2.90)\ \} \\
 & E, D\ :=\ \emptyset,\ Edge - \langle \cup e\ :\ adj\, e \subseteq id\ :\ e \rangle\ .
\end{array}
$$

In the last step, definition of $\preceq$ (2.90) was expanded to show that the initially disposable edges are the loops of the graph –as defined in Section 2.5–. It is often assumed that the input graph does not have loops.

The iteration body is constructed via rules (6.37) and (6.44):

$$
\begin{array}{ll}
 & E, D : [\ D \neq E\ \wedge\ Inv'\ ,\ Inv'\ \wedge\ Prg\ ] \\
\sqsubseteq & \quad \left\{ \begin{array}{l} \text{introduce sequential composition; if progress is achieved} \\ \text{by the first statement, the second must not spoil it} \end{array} \right\} \\
 & E, D : [\ D \neq E\ \wedge\ Inv'\ ,\ Inv\ \wedge\ Prg\ ]\ ; \\
 & E, D : [\ Inv\ ,\ Inv'\ \wedge\ (D - E \subseteq D_0 - E_0)\ ] \\
\sqsubseteq & \quad \left\{ \begin{array}{l} \text{(6.37), note that the calcnlation leading to (6.40)} \\ \text{shows that}\ \ D \neq E\ \wedge\ Inv'\ \ \text{implies the existence} \\ \text{of the required cut; (6.44), progress is not spoiled} \end{array} \right\}
\end{array}
$$

$$
\begin{array}{l}
|[\ \ \textbf{var}\ V : \mathsf{Vec}\ Vert\ ; \\
\quad V : [\ D \neq E\ \wedge\ Inv'\ ,\ respect\,(E, V)\ \wedge\ crossing\, V \neq \emptyset\ ]\ ; \\
\quad |[\ \ \textbf{var}\ e : Edge\ ; \\
\qquad e :\subseteq min\,(R, crossing\, V)\ ; \\
\qquad E\ :=\ E \cup e \\
\quad]| \\
]|\ ; \\
D\ :=\ D - \langle \cup e\ :\ e \subseteq D - E\ \wedge\ e \preceq E\ :\ e \rangle
\end{array}
$$

We have finally got our first approximation to Prim's algorithm as a refinement of specification (6.3). The whole code collected in Figure 6.45.

```
|[  var E : Vec Edge ;
   |[  var D : Vec Edge ;
     E, D := ∅ , Edge − ⟨∪e : adj e ⊆ id : e⟩ ;
     do D ≠ E →
          |[  var V : Vec Vert ;
              V : [ D ≠ E ∧ Inv' , respect (E, V) ∧ crossing V ≠ ∅ ] ;
              |[  var e : Edge ;
                  e :⊆ min (R, crossing V) ;
                  E := E ∪ e
              ]|
          ]| ;
          D := D − ⟨∪e : e ⊆ D − E ∧ e ≼ E : e⟩
     od
   ]|
]|
```

Figure 6.45: First Approximation to Prim's Algorithm, as Algorithmic Solution of (6.3)

Data Refinement There is a good deal of inefficiency in our first approximation to Prim's algorithm, which we now proceed to do away with via data refinement. We first put forward, as temporary propositions, desired conditions on the new program variables. These propositions are then used to postulate the definitive conjuncts of the coupling invariant.

The biggest efficiency problem in sight is the computation of an appropriate cut in the iteration body. Maintaing a cut with the right characteristics all the way through clears this up. New program variable W is introduced to serve as such a cut. Therefore, it must satisfy:

$$P1 \quad := \quad respect\,(E, W) \ .$$

Cut W can be thought of as the vertex set of the forest grown so far E. Below, when refining condition $P1$, it will indeed be defined to be the vertex set of E via the incidence relation Inc of the graph –as defined by (2.81) in Section 2.5–. We also need W to have a non-empty crossing for it to prove useful when the cut is required in the iteration body. However, imposing that the non-empty crossing condition must hold *all* the time would be too strong a requirement. For instance, after growing the whole required output we can reasonably expect W to contain all the vertices of the graph, i.e. expect $W = Vert$ to hold. In such a case the crossing of W is empty. There are also extreme cases, e.g. dealing with a graph that has an empty edge set, where every crossing is empty. We settle this issue by requiring the crossing

of W to be non-empty only if W does not comprise all the vertex set:

$$P2 \quad := \quad \mathit{crossing}\, W = \emptyset \quad \Rightarrow \quad W = \mathit{Vert} \ .$$

This is all we need of new variable W.

To comply with $P1$, we can maintain W as the vertex set of E, i.e. as $\mathit{Inc} \cdot E$. There might be isolated vertices in the graph, so we cater for them by postulating the weaker:

$$CI1 \quad := \quad \mathit{Inc} \cdot E \subseteq W \ .$$

It is not difficult to show that $P1$ follows from $CI1$. Using definition (6.28) of *respect*, we argue thus:

$$\begin{array}{ll} & \mathit{adj}\, E \cdot W \\ \subseteq & \quad \{ \text{ property of } \mathit{adj} \text{ and } \mathit{Inc} \ \} \\ & (\mathit{Inc} \cdot E) \cdot (\mathit{Inc} \cdot E)^\circ \cdot W \\ \subseteq & \quad \{ \text{ universal relation of type } 1 \leftarrow 1 \text{ is } \mathit{id} \ \} \\ & \mathit{Inc} \cdot E \\ \subseteq & \quad \{ \ CI1 \ \} \\ & W \ . \end{array}$$

But there is a problem: W is allowed too much freedom. For instance, $CI1$ –and $P1$ and $P2$– can be met by assigning Vert to W. So, we tighten W up by allowing into it only vertices connected through E, provided they are originally connected in the input graph:

$$CI2 \quad := \quad W \cdot W^\circ \cap \mathit{Join} \ \subseteq \ \mathit{join}\, E \ .$$

To avoid recomputations from scratch of the crossing of W, the second biggest source of inefficiency in the program of Figure 6.45, a new program variable F is introduced to maintain it. Requirement $P2$ is then proposed as definitive, but after expressing it in terms of F:

$$\begin{array}{lll} CI3 & := & F = \mathit{crossing}\, W \ , \\ CI4 & := & F = \emptyset \ \Rightarrow \ W = \mathit{Vert} \ . \end{array}$$

Finally, the whole coupling invariant:

$$CI \quad := \quad CI1 \ \wedge \ CI2 \ \wedge \ CI3 \ \wedge \ CI4 \ .$$

We now describe how the program in Figure 6.45 is transformed using CI as coupling invariant. We take a light approach under which the transformations are explained without providing the fully formal proofs.

The initialisation of E as the empty forest is extended with assignment

$W, F := \emptyset, \emptyset$ to establish $CI1$, $CI2$ and $CI3$. Immediately after, conditional non-emptiness of F is achieved with an iteration that grows W up to the establishment of $CI4$:

do $F = \emptyset \ \wedge \ W \neq Vert \ \rightarrow$
  ||[**var** v : $Vert$;
    $v :\subseteq \overline{W}$;
    $W, F := W \cup v$, *crossing* v
  ]||
od

The negation of $CI4$ was taken as guard, and the iteration body maintains the initially established other three conjuncts of CI. Finiteness of $Vert$ guarantees termination since W is augmented in each iteration.

The guard of the main iteration is replaced according to the following fact:

$$Inv' \ \wedge \ CI \quad \Rightarrow \quad (D = E \ \equiv \ W = Vert) \ .$$

Hence, $W \neq Vert$ becomes the new guard. We can then dispose of variable D as it is rendered useless by this transformation.

In the iteration body, the specification statement affecting V could be replaced by $V := W$, as $P1$ holds on account of $CI1$ and *crossing* $W \neq \emptyset$ also holds on account of the guard, $CI3$ and $CI4$. However, since variable V is only used to select the new edge e, and that selection statement can be replaced after assignment $V := W$ by $e :\subseteq min\,(R, F)$ on account of $CI3$, variable V is simply eliminated.

Drawing e from F implies that it crosses W, and predicate *cross* enjoys the following property:

$$cross\,(e, W) \ \equiv \ \langle \exists\, v, w \ : \ v \cdot w^{\circ} \subseteq adj\, e \ : \ v \subseteq \overline{W} \ \wedge \ w \subseteq W \rangle \ ,$$

where dummies v and w range over points of $Vert$. Therefore, when adding e to E, augmenting W with the $\overline{W}$-extreme of e maintains $CI1$ and $CI2$. We use an auxiliary variable v and assignment $v := \overline{W} \cap Inc \cdot e$ to get hold of the required $\overline{W}$-extreme of e. An implementation can improve on this by storing in F each edge paired up with its "outer" extreme. To maintain $CI3$, variable F is updated according to this *crossing*-rule:

$$\begin{aligned} &cross\,(W \cup v) \ = \\ &\quad (cross\, W \ - \ Inc^{\circ} \cdot v) \ \cup \ (Inc^{\circ} \cdot v \ \cap \ Inc^{\circ} \cdot (\overline{W} - v)) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \text{provided } v \subseteq \overline{W} \ . \end{aligned}$$

After such an updating, the crossing F might have become empty –if a connected component of the graph has just been completed–. The same

auxiliary iteration used in the initialisation must be used here to reestablish $CI4$.

The transformation of the program has been completed. But we now point out the fact that variable W has been used as $W \neq Vert$ in guards and as $\overline{W}$ in most other expressions. It is thus more convenient to have a program variable that holds the complement of W. This is so because comparisons to the empty set are often, if not always, implemented more efficiently than comparisons to the universal set, and because recomputations via complementation are then avoided. Hence, we introduce a new program variable W' using $W' = \overline{W}$ as coupling invariant. Variable W can be eliminated after replacing all its uses by expressions on W', and we then rename W' as W to avoid the prime symbol.

The resulting program, our version of Prim's algorithm, is shown in Figure 6.46.

Originally, Prim's algorithm was designed to deal with input graphs that were connected and had a non-empty vertex set. We set ourselves the task of "redesigning" Prim's algorithm to cater for any graph. The development, though admittedly long, is quite pleasant, as much of it is guided by hints given out by the formulae in manipulation. An instance of "syntactic hints" is the postcondition required of local variable V in the first solution: it gave rise to the proposal of $P1$ and $P2$ as means of achieving the elimination of V. It is even more pleasant to see that the complexity added by accepting unrestricted graphs is not significant. Suppose the input graph is connected and its vertex set is non-empty. Connectedness then implies that only $Vert$ and $\emptyset$, which actually identify the same cut, have empty crossings. There is thus no need for the procedure *GetCrossing*. The initial call to *GetCrossing* can be replaced by one execution of its iteration body: non-emptiness of $Vert$ guarantees at least one round of the iteration while connectedness guarantees that one round suffices. The second call to *GetCrossing*, within the main iteration body, can be plainly removed as the crossing will not become empty unless W has been emptied as well. Therefore, only the calls to *GetCrossing* would be affected by restricting the input graph. The rest of the program, to us the most complex part of it all, remains just the same.

6.8 Related Work

We only know of two derivations of minimum spanning tree algorithms: Bird and de Moor's [25], where Kruskal's algorithm is obtained as an instance of a generic algorithmic solution to a certain kind of optimisation problems; and Berghammer et al.'s [20], where Prim's algorithm is derived via calcu-

```
|[ var E : Vec Edge ;
  |[ var W : Vec Vert ; F : Vec Edge ;
     E, W, F := ∅, Vert, ∅ ;
     GetCrossing (W, F) ;
     do W ≠ ∅ →
          |[ var e : Edge ;
             e :⊆ min (R, F) ;
             |[ var v : Vert ;
                v := W ∩ Inc · e ;
                E, W := E ∪ e , W − v ;
                F := (F − Inc° · v) ∪ (Inc° · v ∩ Inc° · W)
             ]|
          ]| ;
          GetCrossing (W, F)
     od
  ]|
]|

with proc GetCrossing (in out W : Vec Vert , F : Vec Edge)
       do F = ∅ ∧ W ≠ ∅ →
            |[ var v : Vert ;
               v :⊆ W ;
               W, F := W − v, crossing v
            ]|
       od
```

Figure 6.46: Prim's Algorithm, after Data Refinement, as Algorithmic Solution of (6.3)

lations with binary relations and standard methods for the development of imperative programs.

Bird and de Moor, encouraged by the view that the structure of data can dictate the structure of programs, focus in [25] on the derivation of optimisation algorithms under the allegorical approach to datatypes. For this reason, the minimum spanning tree instance problem is specified in terms of lists right from the beginning, unlike our presentation with sets that could be further developed towards an implementation using lists at a later stage. Kruskal's algorithm interests Bird and de Moor due to its optimisation nature and not to its graph-theoretical character. The graph properties involved in the instantiation of their generic solution [25, Section 8] are not dealt with calculationally, but only semi-formally. Their full formalisation would correspond to parts of our Section 5.5 and this chapter.

Related to Bird and de Moor's work, in the matter of the categorical treatment of datatypes, Gibbons [64] has proposed a datatype that models a restricted kind of graphs, viz. directed acyclic graphs with ordered edges –i.e. the incoming and outgoing edges of a vertex form a list rather than a set–. Its application seems to be, as far as it is shown in [64], limited and we know of no further exploration of its applicability.

The derivation of Prim's algorithm by Berghammer and his colleagues in [20] is a follow-up to Berghammer's [16], reviewed in Chapter 5. As in the former treatment of unweighted spanning trees, the input graph is required once more to be connected and with a non-empty edge set. Also, the main derivation is again worked out in terms of simple graphs, hence not dealing with edges –except as represented by atomic symmetric adjacency relations–. But this time their first algorithmic solution is later data-refined to a program in terms of full graphs as we understand them, i.e. plain undirected graphs. Edges can thus be given a weight and the minimisation aspect of the problem can be tackled. The simple graphs and the normal graphs, in their terminology graphs and multigraphs, respectively, are linked by a Galois connection –see e.g. [1] or [2, Chapter 5]– between the lattice of edge sets and the lattice of symmetric adjacency relations. This is related to our adj, which produces adjacency relations from edge sets and which distributes through arbitrary unions, a sufficient and necessary condition for a function between lattices to be the lower adjoint of a Galois connection.

In relation to the contents of our Section 6.3, Berghammer et al. make further use of Galois connections when proving their "Edge Replacement" lemma [20, Section 6.4, Lemma 31]. The proof is carried out in a very compact fashion by exploiting the properties of a Galois connection involving the subgraphs of a fixed spanning tree. This corresponds to what took us so much time

proving the existence of a witness in Section 6.3. At the time of writing, we have not been able to get a handle on this second Galois connection, but it is certainly an attractive way of shortening the proofs of this chapter and very probably of getting a deeper understanding of other features of connectedness-preserving forests like e.g. those explored in Section 2.7. We thus intend to get down to it in future work.

Chapter 7

Computing Strongly Connected Components

This chapter presents the derivation of an algorithm that computes the strongly connected components of a directed graph. Tackling this problem in the context of the present thesis, which aims to show the applicability of the calculus of relations to the *derivation* of graph algorithms, was motivated by two previous exercises in dealing with this non-trivial algorithmic problem in a methodical fashion: Dijkstra's "exercise in orderly program composition" [46] –see also [45, Chapter 25]– and Kruseman Aretz's "exercise in program presentation" as a tribute to Dijkstra [89]. The programs therein presented are akin to Tarjan's famous algorithm for the computation of strongly connected components by means of a depth-first traversal of the graph [141].

Dijkstra's essay offers a derivation which, however, "contains a few 'surprises' ... without an elaborate heuristic justification" [46, page 22]. It has to be noted that at the time this essay was written, the mid 1970s, calculational methods had not yet found their way into the field of program development. It is thus not surprising that the graph properties involved are not treated calculationally at all and, without a formalism allowing uninterpreted manipulation of formulae representing graph concepts, "surprises" were likely to come up. Kruseman Aretz's essay, written about a decade and a half later, offers a clearer presentation of the algorithm that benefits from abstracting data-representation details, using sets and lists instead of arrays intricately representing the relevant data. Nevertheless, the algorithm is *not derived* but merely *presented*, albeit in an orderly and quite beautiful fashion.

We hope that the contents of this chapter live up to the expectations of state-of-the-art program development as a follow-up to Dijkstra's and Kruseman Aretz's work. In the introduction to his essay, Kruseman Aretz points out that he expects "that the *presentation* can be transformed into a *derivation*"

–our emphasis– and that he hopes this will "be done in the future" [89, page 251]. It pleases us to have moved on a little closer to such hopes, in spite of our derivation being admittedly rife with many long tricky calculations. We believe, however, that our derivation represents a stepping-stone towards a nice and compact derivational presentation.

In what follows, Section 7.1 presents a formal specification of the problem. Section 7.2 then sets the scene for the derivation of an iterative algorithm by proposing an invariant. The invariant is somewhat complex and comprises a good number of conjuncts; hence, emphasis will be laid on justifying how each of the conjuncts was deduced. Section 7.3 offers the rest of the set-up of the iteration: guard, initialisation statement, and variant. Means of making the iteration progress are dealt with in Section 7.4 by examining four different ways of decreasing the variant. Section 7.5 assembles the body of the iteration, thereby completing the program we offer as a refinement of the initial specification, and Section 7.6 provides some comments on further refinement and implementation details. Section 7.7 closes the chapter with a review of related work.

7.1 Specification

Strong connectedness, within the calculational framework of binary relations, was presented in page 30 of Chapter 2. For the sake of convenience, we recall and relabel some relevant definitions here. Let G be a directed graph $(Vert, Edge, x1, x2)$. Definitions (2.74) and (2.83) of the successor and reachability relations of G, which are used to construct the strong connectedness relation of G, as well as definition (2.87) of the strong connectedness relation itself state that:

$$Succ = x1 \cdot x2^{\circ} \ , \tag{7.1}$$

$$Reach = Succ^{\star} \ , \tag{7.2}$$

$$Str = Reach \cap Reach^{\circ} \ . \tag{7.3}$$

Relation Str is an equivalence relation on $Vert$ that relates mutually reachable, i.e. strongly connected, vertices.

In page 21 of Chapter 2, we showed a way of modelling quotient sets of equivalence relations as powerset vectors. The set of strongly connected components of G is precisely the quotient set of $Str : Vert \leftarrow Vert$ and, therefore, our problem can be formally specified as follows:

$$\begin{array}{l} [\![\ \mathbf{var} \ \ SC : \mathrm{Vec}\,(\mathsf{P}\,Vert) \ ; \\ \quad SC : [\ true \ , \ SC = \Lambda Str \cdot Vert \ ] \\ ]\!] \end{array} \tag{7.4}$$

We now proceed with its refinement to a program.

7.2 A Non-Trivial Invariant

This section proposes a reasonable invariant to develop an iteration that refines the specification of our problem. As mentioned in the introduction, the invariant we will end up with is rather complex, comprising several conjuncts. We embrace the task of justifying every conjunct by explicit reference to the underlying reasoning that puts them forward.

Before embarking on our main task, some definitions from Chapter 2 are exported and relabelled. The successor, reachability and strong connectedness relations of graph G can be restricted to make use of only a subset of edges. As defined in (2.77), (2.85) and (2.88), these relations are:

$$succ\, F \;=\; x1 \cdot \natural F \cdot x2^{\circ} \;, \tag{7.5}$$

$$reach\, F \;=\; (succ\, F)^{\star} \;, \tag{7.6}$$

$$str\, F \;=\; (reach\, F) \cap (reach\, F)^{\circ} \;, \tag{7.7}$$

where F is the vector on $Edge$ that models the subset of edges one is allowed to traverse.

We now go for the invariant.

The Subgraph Seen So Far To begin with, it is most reasonable to expect that the vertices and edges must be incrementally examined. Thus, at each execution of the iteration body only *some* of the vertices and *some* of the edges have been inspected. These vertices and edges determine the subgraph of G that has been seen so far. How much is known about the strongly connected components of G must be in accordance with such a subgraph, and that is what we lay bare on our first invariant. We get it by replacing constants $Vert$ and $Edge$ in the postcondition of (7.4) by freshly introduced variables $V : \mathsf{Vec}\, Vert$ and $E : \mathsf{Vec}\, Edge$ that reflect the subgraph seen so far:

$$P1 \;:=\; SC \;=\; \Lambda(str\, E) \cdot V \;,$$

i.e. SC is the $(str\, E)$-quotient of V. Note that constant $Edge$ is hidden in (7.4). The fact $Str = str\, Edge$ suffices to unveil it.

It is also reasonable to assume that V and E determine a consistent subgraph of G. As formalised in (2.76), this means that both extreme vertices

of edges in E must be in V. This provides us with two more invariants:

$$P2 \quad := \quad x1 \cdot E \subseteq V \ ,$$
$$P3 \quad := \quad x2 \cdot E \subseteq V \ .$$

Furthermore, we require the partition SC to be consistent in the sense that the vertices it contains should be exactly those contained in V and no more. This is formalised by the expression $\in \cdot SC = V$, which is equivalent to stating that V fits $str\, E$ –see remarks on partitioning sets by equivalence relations and the whole set of elements in such partitions in pages 21-22 of Section 2.4–. Hence, we ask V to fit $str\, E$ invariantly through our iteration-to-be:

$$P4 \quad := \quad str\, E \cdot V \subseteq V \ .$$

And this is the end of the consistency requirements on the subgraph given by V and E.

We want to point out that, by the end of this section, we will have collected invariants named $Inv1$, $Inv2$, et cetera. Those will be the definitive ones. We have named the above invariants $P1$, $P2$, $P3$ and $P4$ since there is quite some juggling ahead of us before we decide on the final invariants.

An adequate guard for our provisional invariants $P1$, $P2$, $P3$ and $P4$ could be $\neg (V = Vert \ \wedge \ E = Edge)$. But we will postpone this issue until the next section, after the final invariants have been decided on.

Final Components and Intermediate Components According to proposed invariant $P1$, program variable SC holds some "nearly strongly connected" components of G computed out of V and E. We now wonder whether some of these components are *final*, i.e. full strongly connected components of G, and under what conditions they are so. The remaining components in SC, i.e. the ones still in construction, we will call *intermediate*.

In order to analyse conditions that determine which components in SC are final and which are intermediate, we partition the set of inspected vertices V into two sets:

$$Q1 \quad := \quad V = Vf \cup Vi \quad \wedge \quad Vf \cap Vi = \emptyset \ .$$

Sets Vf and Vi are, respectively, meant to comprise the vertices of final components and intermediate components in SC. We will also make use of an analogous partition of set E:

$$Q2 \quad := \quad E = Ef \cup Ei \quad \wedge \quad Ef \cap Ei = \emptyset \ .$$

Set Ef is meant to comprise the edges that have determined the final components computed so far, whereas Ei contains the rest of the inspected edges.

Partitioning the vertices as in $Q1$, on account of $P1$ and distributivity of composition over union, implies that:

$$SC \;=\; \Lambda(str\,E)\cdot Vf \;\cup\; \Lambda(str\,E)\cdot Vi \;. \tag{7.8}$$

Each half of partition SC as stated in this equation should be consistent in the same sense as of condition $P4$, i.e. both Vf and Vi should fit $str\,E$. Hence, we require:

$$\begin{array}{lcl} Q3 & := & str\,E\cdot Vf \;\subseteq\; Vf \;, \\ Q4 & := & str\,E\cdot Vi \;\subseteq\; Vi \;. \end{array}$$

Note that $Q1$, $Q3$ and $Q4$ imply, and thus take care of, invariant $P4$.

Final Components We now want to examine under what conditions the first half of SC as stated in (7.8) corresponds to final components, i.e. under what conditions the following holds:

$$\Lambda(str\,E)\cdot Vf \;=\; \Lambda Str\cdot Vf \;. \tag{7.9}$$

This we manipulate as follows:

$$\begin{array}{ll} & (7.9) \\ \Leftarrow & \quad \{ \text{ power transpose, vectors and coreflexives (2.53) } \} \\ & str\,E\cdot \overline{\in}\, Vf \;=\; Str\cdot \overline{\in}\, Vf \\ \equiv & \quad \{ \; str\,E \subseteq Str \text{, monotonicity of composition } \} \\ & Str\cdot \overline{\in}\, Vf \;\subseteq\; str\,E\cdot \overline{\in}\, Vf \\ \equiv & \quad \{ \text{ coreflexives (2.33) } \} \\ & Str\cdot \overline{\in}\, Vf \;\subseteq\; str\,E \\ \Leftarrow & \quad \left\{ \begin{array}{l} \text{–set } Ef \text{ is meant to be the relevant subset of } E \\ \text{as regards the final components and we thus need} \\ \text{to introduce it–} \quad Ef \subseteq E \text{, monotonicity of } str \end{array} \right\} \\ & Str\cdot \overline{\in}\, Vf \;\subseteq\; str\,Ef \\ \equiv & \quad \{ \text{ definition of } str \text{ (7.7), intersection (2.6) } \} \\ & Str\cdot \overline{\in}\, Vf \;\subseteq\; reach\,Ef \quad \wedge \quad Str\cdot \overline{\in}\, Vf \;\subseteq\; (reach\,Ef)^\circ \;. \end{array} \tag{7.10}$$

We now deal with the first conjunct:

$$\begin{array}{ll} & Str\cdot \overline{\in}\, Vf \;\subseteq\; reach\,Ef \\ \Leftarrow & \quad \{ \text{ definition of } Str \text{ (7.3), intersection } \} \\ & Reach\cdot \overline{\in}\, Vf \;\subseteq\; reach\,Ef \end{array}$$

$$
\begin{array}{ll}
\Leftarrow & \left\{ \begin{array}{l} \text{coreflexives included in } id \text{ –this prepares us to apply} \\ \text{the leap-frog over closure rule without making an} \\ \text{unreasonable strengthening of the demonstrandum–} \end{array} \right\} \\
& \mathit{Reach} \cdot \mathbin{\not\in} \mathit{Vf} \;\subseteq\; \mathbin{\not\in} \mathit{Vf} \cdot \mathit{reach}\, \mathit{Ef} \qquad (7.11) \\
\Leftarrow & \left\{ \begin{array}{l} \text{definitions of } \mathit{Reach}\ (7.2) \text{ and } \mathit{reach}\ (7.6)\,, \\ \text{leap-frog over closure } (2.66) \end{array} \right\} \\
& \mathit{Succ} \cdot \mathbin{\not\in} \mathit{Vf} \;\subseteq\; \mathbin{\not\in} \mathit{Vf} \cdot \mathit{succ}\, \mathit{Ef} \\
\Leftarrow & \left\{ \begin{array}{l} \text{definitions of } \mathit{Succ}\ (7.1) \text{ and } \mathit{succ}\ (7.5)\,, \text{ monotonicity} \\ \text{of composition, coreflexives duplicate } (2.32) \end{array} \right\} \\
& x2^\circ \cdot \mathbin{\not\in} \mathit{Vf} \;\subseteq\; \mathbin{\not\in} \mathit{Ef} \cdot x2^\circ \;\wedge\; x1 \cdot \mathbin{\not\in} \mathit{Ef} \;\subseteq\; \mathbin{\not\in} \mathit{Vf} \cdot x1\ . \qquad (7.12)
\end{array}
$$

This last proposition will be taken as invariant of the loop. We now show that this suffices for Vf and Ef to determine the final components in SC, as the second conjunct of (7.10) also follows from it:

$$
\begin{array}{ll}
& \mathit{Str} \cdot \mathbin{\not\in} \mathit{Vf} \\
\subseteq & \left\{ \begin{array}{l} \text{definition of } \mathit{Str}\ (7.3)\,, \text{ distribution of composition} \\ \text{over intersection –right-analogue of } (2.15)– \end{array} \right\} \\
& \mathit{Reach} \cdot \mathbin{\not\in} \mathit{Vf} \;\cap\; \mathit{Reach}^\circ \cdot \mathbin{\not\in} \mathit{Vf} \\
\subseteq & \left\{ \begin{array}{l} (7.11) \text{ above follows from } (7.12), \\ \text{coreflexives included in } id \end{array} \right\} \\
& \mathbin{\not\in} \mathit{Vf} \cdot \mathit{reach}\, \mathit{Ef} \;\cap\; \mathit{Reach}^\circ \\
\subseteq & \quad \{ \text{Dedekind's rule } (2.19), \text{ symmetry of coreflexives } (2.32) \} \\
& \mathbin{\not\in} \mathit{Vf} \cdot (\mathit{reach}\, \mathit{Ef} \;\cap\; \mathbin{\not\in} \mathit{Vf} \cdot \mathit{Reach}^\circ) \\
\subseteq & \quad \{ \text{intersection, coreflexives duplicate } (2.32) \} \\
& \mathbin{\not\in} \mathit{Vf} \cdot \mathit{Reach}^\circ \\
= & \quad \{ \text{converse } (2.16), \text{ symmetry of coreflexives } (2.32) \} \\
& (\mathit{Reach} \cdot \mathbin{\not\in} \mathit{Vf})^\circ \\
\subseteq & \quad \{ (7.11) \text{ above again, converse } (2.18) \} \\
& (\mathbin{\not\in} \mathit{Vf} \cdot \mathit{reach}\, \mathit{Ef})^\circ \\
\subseteq & \quad \{ \text{coreflexives included in } id\,, \text{ converse } (2.18) \} \\
& (\mathit{reach}\, \mathit{Ef})^\circ\ .
\end{array}
$$

Hence, (7.12) does imply both conjuncts of (7.10). To postulate the two conditions in (7.12) as invariants, we use property (2.40) for the transformation of coreflexive expressions into vector expressions and, since they will not be further manipulated, we name them as definitive invariants:

$$
\begin{array}{lcl}
\mathit{Inv1} & := & x2^\circ \cdot \mathit{Vf} \;\subseteq\; \mathit{Ef}\ , \\
\mathit{Inv2} & := & x1 \cdot \mathit{Ef} \;\subseteq\; \mathit{Vf}\ .
\end{array}
$$

In the calculation that led to these two conditions, several design decisions were made. To start with, it was decided to deal with the first conjunct of (7.10) before the second; then, the following two steps strengthened the demonstrandum to arrive at (7.11). As mentioned in the hints, detecting a possible application of the leap-frog over closure rule gave rise to such a strengthening. Symmetric decisions could have been made by initially choosing the second conjunct of (7.10). Using properties of converse -(2.16), (2.18)- and symmetry of both Str and coreflexives, the subsequent weakening of Str to $Reach$ would have led to the closure being leap-frogged rightwards instead of leftwards. We would have obtained a mirror image of $Inv1$ and $Inv2$, with the roles of $x1$ and $x2$ interchanged. There are no obvious reasons to favour either of these two options over the other and we thus stick to our initial choice.

We finalise the issue of the final components by showing that $Q3$ is also implied by $Inv1$ and $Inv2$. First, definition of $Succ$ (7.1), $Inv1$ and $Inv2$ imply $Succ \cdot Vf \subseteq Vf$. Then, definition of $Reach$ (7.2) and property (2.68) of closure give us $Reach \cdot Vf \subseteq Vf$. Finally, definition of Str (7.3) and intersection imply $Str \cdot Vf \subseteq Vf$, and the fact that $str\, E \subseteq Str$ then implies $Q3$.

The final components have been dealt with: $Q3$ and (7.9) hold under conditions $Inv1$ and $Inv2$.

Intermediate Components We now deal with the intermediate components, i.e. the second half of SC as stated in equation (7.8). We have seen that edges in Ef determine the final components. On top of that, we would like them to play no role in the computation of the rest of the components. Hence, we now search for conditions that make the intermediate components dependent solely on edges in Ei, i.e. conditions under which the following holds:

$$\Lambda(str\, E) \cdot Vi \;=\; \Lambda(str\, Ei) \cdot Vi \;. \tag{7.13}$$

We proceed, with a manipulation quite similar to the one of (7.9), thus:

$$\begin{array}{ll} & (7.13) \\ \Leftarrow & \quad \{ \text{ power transpose, vectors and coreflexives (2.53) } \} \\ & str\, E \cdot \not\in Vi \;=\; str\, Ei \cdot \not\in Vi \\ \equiv & \quad \{ \; E \supseteq Ei \text{, monotonicity of } str \text{ and composition } \} \\ & str\, E \cdot \not\in Vi \;\subseteq\; str\, Ei \cdot \not\in Vi \\ \equiv & \quad \{ \text{ coreflexives (2.33) } \} \\ & str\, E \cdot \not\in Vi \;\subseteq\; str\, Ei \end{array}$$

$$
\begin{array}{ll}
\equiv & \{ \text{ definition of } str \ (7.7) \text{, intersection } (2.6) \ \} \\
 & str\ E \cdot \not\in Vi \subseteq reach\ Ei \ \wedge \ str\ E \cdot \not\in Vi \subseteq (reach\ Ei)^\circ . \quad (7.14)
\end{array}
$$

Unlike with (7.10), we now choose to continue the calculation with the second conjunct:

$$
\begin{array}{ll}
 & str\ E \cdot \not\in Vi \subseteq (reach\ Ei)^\circ \\
\equiv & \left\{ \begin{array}{l} \text{converse } -(2.16), (2.18)-, \\ \text{symmetry of } str\ E \text{ and coreflexives} \end{array} \right\} \\
 & \not\in Vi \cdot str\ E \subseteq reach\ Ei \\
\Leftarrow & \{ \text{ definition of } str \ (7.7) \text{, intersection } \} \\
 & \not\in Vi \cdot reach\ E \subseteq reach\ Ei \\
\Leftarrow & \left\{ \begin{array}{l} \text{coreflexives included in } id \text{ -as before, this} \\ \text{prepares us to apply the leap-frog rule-} \end{array} \right\} \\
 & \not\in Vi \cdot reach\ E \subseteq reach\ Ei \cdot \not\in Vi \qquad (7.15) \\
\Leftarrow & \{ \text{ definition of } reach \ (7.6) \text{, leap-frog over closure } (2.67) \ \} \\
 & \not\in Vi \cdot succ\ E \subseteq succ\ Ei \cdot \not\in Vi \\
\Leftarrow & \left\{ \begin{array}{l} \text{definition of } succ \ (7.5) \text{, monotonicity of} \\ \text{composition, coreflexives duplicate } (2.32) \end{array} \right\} \\
 & \not\in Vi \cdot x1 \cdot \not\in E \subseteq x1 \cdot \not\in Ei \ \wedge \ \not\in Ei \cdot x2^\circ \subseteq x2^\circ \cdot \not\in Vi \\
\equiv & \left\{ \begin{array}{l} \text{converse } -(2.16), (2.18)-, \\ \text{symmetry of coreflexives } (2.32) \end{array} \right\} \\
 & \not\in Vi \cdot x1 \cdot \not\in E \subseteq x1 \cdot \not\in Ei \ \wedge \ x2 \cdot \not\in Ei \subseteq \not\in Vi \cdot x2 \ . \quad (7.16)
\end{array}
$$

Again, this last proposition will be taken as invariant of the loop, but we still need to show that the first conjunct of (7.14) also follows from (7.16):

$$
\begin{array}{ll}
 & str\ E \cdot \not\in Vi \\
\subseteq & \left\{ \begin{array}{l} \text{definition of } str \ (7.7) \text{, distribution of} \\ \text{composition over intersection } (2.15) \end{array} \right\} \\
 & reach\ E \cdot \not\in Vi \ \cap \ (reach\ E)^\circ \cdot \not\in Vi \\
\subseteq & \left\{ \begin{array}{l} \text{coreflexives included in } id \text{; converse } (2.16), \\ \text{symmetry of coreflexives } (2.32) \end{array} \right\} \\
 & reach\ E \ \cap \ (\not\in Vi \cdot reach\ E)^\circ \\
\subseteq & \{ \ (7.15) \text{ above follows from } (7.16) \text{, converse } (2.18) \ \} \\
 & reach\ E \ \cap \ (reach\ Ei \cdot \not\in Vi)^\circ \\
= & \{ \text{ converse } (2.16) \text{, symmetry of coreflexives } (2.32) \ \} \\
 & reach\ E \ \cap \ \not\in Vi \cdot (reach\ Ei)^\circ \\
\subseteq & \{ \text{ Dedekind's rule } (2.19) \text{, symmetry of coreflexives } (2.32) \ \} \\
 & \not\in Vi \cdot (\not\in Vi \cdot reach\ E \ \cap \ (reach\ Ei)^\circ)
\end{array}
$$

$$
\begin{array}{ll}
\subseteq & \{ \text{ intersection, coreflexives duplicate (2.32) } \} \\
& \mathfrak{c}\, Vi \cdot reach\, E \\
\subseteq & \{ \text{ (7.15) above again } \} \\
& reach\, Ei \cdot \mathfrak{c}\, Vi \\
\subseteq & \{ \text{ coreflexives included in } id \ \} \\
& reach\, Ei \ .
\end{array}
$$

Therefore, (7.16), to be taken as invariant, suffices for both conjuncts of (7.14) to hold. However, we will postulate only the second conjunct of (7.16) as invariant since the first follows from *Inv2*, as we now show:

$$
\begin{array}{ll}
& \mathfrak{c}\, Vi \cdot x1 \cdot \mathfrak{c}\, E \ \subseteq \ x1 \cdot \mathfrak{c}\, Ei \\
\equiv & \left\{ \begin{array}{l} \text{split } E \text{ by } Q2 \text{, distribution of } \mathfrak{c} \text{ and composition over} \\ \text{union } -(2.14),\ (2.38)-\text{, universal property of union (2.5)} \end{array} \right\} \\
& \mathfrak{c}\, Vi \cdot x1 \cdot \mathfrak{c}\, Ef \ \subseteq \ x1 \cdot \mathfrak{c}\, Ei \quad \wedge \quad \mathfrak{c}\, Vi \cdot x1 \cdot \mathfrak{c}\, Ei \ \subseteq \ x1 \cdot \mathfrak{c}\, Ei \\
\Leftarrow & \left\{ \begin{array}{l} \text{coreflexive version of } Inv2 \text{; second conjunct} \\ \text{is true since coreflexives are included in } id \end{array} \right\} \\
& \mathfrak{c}\, Vi \cdot \mathfrak{c}\, Vf \cdot x1 \ \subseteq \ x1 \cdot \mathfrak{c}\, Ei \\
\equiv & \{ \text{ coreflexives versus vectors } -(2.32),\ (2.39)- \ \} \\
& \mathfrak{c}(Vi \cap Vf) \cdot x1 \ \subseteq \ x1 \cdot \mathfrak{c}\, Ei \\
\equiv & \{ \text{ left-hand side is empty by } Q1 \text{ and (2.36) } \} \\
& true \ .
\end{array}
$$

We thus only postulate the vector-version, via property (2.40), of the second conjunct of (7.16):

$$Inv3 \quad := \quad x2 \cdot Ei \ \subseteq \ Vi \ .$$

Note again the design decision of first dealing with the second conjunct of (7.14) by strengthening it to (7.15). It is the opposite decision to the one that was taken when dealing with the final components. Had we used for the intermediate components the same strategy that was used for the final components, we would have arrived at the *x1-x2* mirror images of the conditions that were actually obtained. We will not get into details, but point out that such mirror images would form a "difficult" invariant if conjoined to *Inv1* and *Inv2*, in the sense that the iteration would then not be able to explore edges in a one-by-one manner. This led to using the opposite strategy when dealing with the intermediate components and, thus, to *Inv3* as defined above.

As it happened with *Q3* before, assumption *Q4* can now be shown to follow from the new invariant. The argument is similar, albeit slightly more

elaborate, to the one that showed $Q3$ from $Inv1$ and $Inv2$. First, use definition of *succ* (7.5) and (7.16), which we have proved to follow from $Inv2$ and $Inv3$, to show that $(succ\ E)^\circ \cdot \phi\ Vi \subseteq \phi\ Vi \cdot (succ\ E)^\circ$. Translate this into a vector expression with the aid of (2.40) to obtain $(succ\ E)^\circ \cdot Vi \subseteq Vi$. Then, use definition of *reach* (7.6) and property (2.68) of closure to get $(reach\ E)^\circ \cdot Vi \subseteq Vi$. Definition of *str* (7.7) and intersection then imply $str\ E \cdot Vi \subseteq Vi$, i.e. $Q4$.

Finally, the intermediate components have also been dealt with: $Q4$ and (7.13) hold under condition $Inv3$ –and $Inv2$–.

The x-V-E Equations So far, our only definitive invariants are $Inv1$, $Inv2$ and $Inv3$, all of which express relationships between the extremes of the edges inspected so far and the vertices inspected so far. These are what we call *x-V-E equations*. Left behind were two other such equations, which constrained the subgraph seen so far to be consistent: $P2$ and $P3$. In the light of equations $Inv1$ to $Inv3$ these can be simplified and we dedicate ourselves to such a task now.

We first deal with $P2$ thus:

$$
\begin{array}{ll}
 & x1 \cdot E \subseteq V \\
\equiv & \quad \{ \text{ splits given by } Q1 \text{ and } Q2 \ \} \\
 & x1 \cdot (Ef \cup Ei) \subseteq Vf \cup Vi \\
\equiv & \quad \left\{ \begin{array}{l} \text{distribution of composition over union (2.14),} \\ \text{universal property of union (2.5)} \end{array} \right\} \\
 & x1 \cdot Ef \subseteq Vf \cup Vi \ \wedge \ x1 \cdot Ei \subseteq Vf \cup Vi \\
\equiv & \quad \{ \ Inv2 \text{ makes first conjunct true } \} \\
 & x1 \cdot Ei \subseteq Vf \cup Vi \qquad (7.17) \\
\Leftarrow & \quad \{ \text{ union } \} \\
 & x1 \cdot Ei \subseteq Vi \ .
\end{array}
$$

Hence, provisional invariant $P2$ can be disposed of by taking:

$$Inv4 \ := \ x1 \cdot Ei \subseteq Vi \ .$$

However, the question arises whether the last step of the calculation above, the only strengthening step, was sensible, in the sense of putting at risk the feasibility of the conjoined invariants. Discussion of this is delayed until after dealing with $P3$, which we proceed to do at once:

$$
\begin{array}{ll}
 & x2 \cdot E \subseteq V \\
\equiv & \quad \{ \text{ splits given by } Q1 \text{ and } Q2 \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & x2 \cdot (Ef \cup Ei) \subseteq Vf \cup Vi \\
\equiv & \left\{ \begin{array}{l} \text{distribution of composition over union (2.14),} \\ \text{universal property of union (2.5)} \end{array} \right\} \\
 & x2 \cdot Ef \subseteq Vf \cup Vi \quad \wedge \quad x2 \cdot Ei \subseteq Vf \cup Vi \\
\equiv & \quad \{ \; Inv3 \text{ makes second conjunct true } \} \\
 & x2 \cdot Ef \subseteq Vf \cup Vi \; .
\end{array}
$$

Hence, we can dispose of $P3$ by postulating:

$$Inv5 \quad := \quad x2 \cdot Ef \subseteq Vf \cup Vi \; .$$

Since $Inv5$ is equivalent to $P3$, no risk has been taken this time.

We now go back to questioning the strengthening of (7.17) into $Inv4$. Had we taken the former as invariant, no doubts would be cast on the replacement of $P2$ by it since both statements are equivalent. If we can show that from a computation state in which (7.17) holds a harmless assignment statement establishes $Inv4$, we will have shown that feasibility of (7.17) implies feasibility of $Inv4$ and, hence, that the strengthening in dispute is safe. We call such an assignment *harmless* if all other invariants are maintained.

The restriction imposed by $Inv4$ is that the $x1$-extremes of Ei must lie in Vi, instead of unrestrictedly in $Vf \cup Vi$. Were this not true, one could try a simple transferral of the offending edges to Ef to solve the problem. Consider the set F defined to be $Ei \cap x1^{\circ} \cdot Vf$, i.e. the set of edges in Ei with $x1$-extremes in Vf. We claim that assignment $Ef, Ei := Ef \cup F, Ei - F$ establishes $Inv4$ if (7.17) holds initially, and that it also maintains invariants $Inv1$ to $Inv3$ as well as $Inv5$. The proof of this claim boils down to some non-interesting juggling with the calculus of relations that the reader must already be familiar with –rules of the lattice structure and shunting of functions is all it takes–; we thus omit it. The only provisional invariant at risk under the assignment at issue is $Q2$, but this is trivially maintained as the assignment just shuffles the partition.

We must remark that any other strengthening, viz. reducing the right-hand side of (7.17) to Vf or reducing the right-hand side of $Inv5$ to either Vf or Vi, would have given rise to infeasible or to "difficult" invariants.

To summarise, we have done away with provisional invariants $P2$ and $P3$ by postulating $Inv4$ and $Inv5$, which complete the set of x-V-E equations.

Finishing Off Let us review what has happened to the provisional invariants up to this point, and then proceed to finish with the ones not yet taken care of. Out of the P-invariants only $P1$ still needs to be dealt with, since we

have just seen $P2$ and $P3$ leaving, and $P4$ follows from the Q-invariants. Out of the Q-invariants only $Q1$ and $Q2$ remain, since $Q3$ and $Q4$ were taken care of by the first three definitive invariants $Inv1$ to $Inv3$.

Due to the introduction of sets Vf, Vi, Ef and Ei, it seems reasonable to get rid of the sets we started out with: V and E. This calls, in order to sort $P1$ out, for partitioning SC as well into, say, SCf and SCi, which would hold each half of SC as stated by equation (7.8). But such halves are expressed in terms of E, which we want to get rid of. This problem is solved by appealing to (7.9) and (7.13), which hold on account of $Inv1$ to $Inv3$ and which give us $\Lambda Str \cdot Vf$ and $\Lambda(str\ Ei) \cdot Vi$ as the corresponding halves for SCf and SCi.

Variable SC was fixed in the specification as the program variable to hold the final result of the computation and, therefore, it cannot be sent away the way V and E were. Since it is clear that the SCf-half would be the one accumulating the final result, we can refrain from introducing SCf and keep SC to play its role. Hence, we take care of $P1$ by introducing program variable SCi and postulating:

$$\begin{aligned} Inv6 &:= SC = \Lambda Str \cdot Vf\ , \\ Inv7 &:= SCi = \Lambda(str\ Ei) \cdot Vi\ . \end{aligned}$$

As for $Q1$ and $Q2$, we only need to scrap the bits that refer to the discarded V and E and keep:

$$\begin{aligned} Inv8 &:= Vf \cap Vi = \emptyset\ , \\ Inv9 &:= Ef \cap Ei = \emptyset\ . \end{aligned}$$

We have finally completed our toy ninefold invariant. As usual, we will let Inv denote the conjunction of $Inv1$ to $Inv9$.

Before moving on to the next section, we need to remark that, in spite of having discharged variables V and E, we will still use them in the rest of the chapter. However, they will only represent abbreviations for the expressions $Vf \cup Vi$ and $Ef \cup Ei$, respectively, instead of being program variables. This makes all the recorded facts about V and E in this section still valid, with the exception of $P1$ and (7.8) due to the fact that the role of variable SC was changed later. In particular, we will make use of (7.9) and (7.13), which imply the following alternative phrasings of $Inv6$ and $Inv7$:

$$\begin{aligned} Inv6' &:= SC = \Lambda(str\ E) \cdot Vf\ , \\ Inv7' &:= SCi = \Lambda(str\ E) \cdot Vi\ . \end{aligned}$$

$$
\begin{array}{lcl}
Inv1 & - & x2^{\circ} \cdot Vf \subseteq Ef \\
Inv2 & - & x1 \cdot Ef \subseteq Vf \\
Inv3 & - & x2 \cdot Ei \subseteq Vi \\
Inv4 & - & x1 \cdot Ei \subseteq Vi \\
Inv5 & - & x2 \cdot Ef \subseteq Vf \cup Vi \\
Inv6 & - & SC = \Lambda Str \cdot Vf \\
Inv7 & - & SCi = \Lambda(str\ Ei) \cdot Vi \\
Inv8 & - & Vf \cap Vi = \emptyset \\
Inv9 & - & Ef \cap Ei = \emptyset \\
Inv6' & - & SC = \Lambda(str\ E) \cdot Vf \\
Inv7' & - & SCi = \Lambda(str\ E) \cdot Vi
\end{array}
$$

Figure 7.18: The Ninefold Invariant

A summary of the whole invariant is shown in Figure 7.18.

7.3 Setting Up the Rest of the Iteration

This section presents the other three important components of the design of an iteration: guard, initialisation statement and variant.

Conjunct *Inv6* of the invariant implies the postcondition if conjoined to $Vf = Vert$. Hence, we take $Vf \neq Vert$ as guard.

All nine conjuncts of the invariant are easily established by assigning the empty relation to every variable. This is consistent with the operational view that the variables hold values related to the subgraph seen so far since, initially, nothing has been seen.

According to the guard and the initialisation, variable Vf is empty at the beginning and must grow until it holds all the vertices of the graph. This suggests Vf itself as variant along with $\supset$ as well-founded relation. –As usual, we assume both the vertex set and the edge set of the input graph to be finite.– However, this is too naive a variant, since expecting Vf to increase on each iteration ignores the computation plan underlying the design of the invariant: the progressive construction of intermediate components until they can become final components. It is more sensible to expect that on each iteration one of the following occurs:

- the set of final vertices is augmented due to the realisation that the construction of an intermediate component has been finished and it

must then be passed to the set of final components –this would also involve changes in the sets of final edges, intermediate vertices and intermediate edges, but we focus on the augmentation of Vf as the central indication of progress in this case–;

- keeping the set of final vertices fixed, the set Ef is augmented as the result of detecting edges that do not and will not contribute to the make-up of non-final components, be they intermediate or not yet started;
- keeping the sets of final vertices and edges fixed, new vertices are added to Vi in order to make the construction of intermediate components progress; or
- keeping the sets of final vertices, final edges and intermediate vertices fixed, new edges are added to Ei also in order to make the construction of intermediate components progress.

We then postulate (Vf, Ef, Vi, Ei) as variant expression using the well-founded lexical ordering induced by $\supset$, i.e. progress will be guaranteed as follows:

$$\begin{array}{lll} Prg & := & (Vf \supset Vf_0) \\ & & \vee \quad (Vf = Vf_0 \quad \wedge \quad Ef \supset Ef_0) \\ & & \vee \quad ((Vf, Ef) = (Vf_0, Ef_0) \quad \wedge \quad Vi \supset Vi_0) \\ & & \vee \quad ((Vf, Ef, Vi) = (Vf_0, Ef_0, Vi_0) \quad \wedge \quad Ei \supset Ei_0) \ . \end{array}$$

As customary by now, after setting up the iteration, we present a summary of how the initial specification statement in (7.4) has been refined so far:

$$\begin{array}{l} SC : [\ true\ ,\ SC = \Lambda Str \cdot Vert\ ] \\ \sqsubseteq \quad \left\{ \begin{array}{l} \text{introduce local block and initialised iteration} \\ \text{according to discussion above} \end{array} \right\} \\ \quad |[\ \textbf{var}\ SCi : \mathsf{Vec}\,(\mathsf{P}\,Vert)\ ; \\ \qquad\quad Vf, Vi : \mathsf{Vec}\ Vert\ ;\ Ef, Ei : \mathsf{Vec}\ Edge\ ; \\ \qquad SC, SCi, Vf, Vi, Ef, Ei\ :=\ \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\ ; \\ \qquad \textbf{do}\ \ Vf \neq Vert\ \rightarrow\ SC, SCi, Vf, Vi, Ef, Ei\ : \\ \qquad\qquad\qquad\qquad [\ Vf \neq Vert\ \wedge\ Inv\ ,\ Inv\ \wedge\ Prg\ ] \\ \qquad \textbf{od} \\ \quad]| \end{array}$$

7.4 Making the Iteration Progress

Armed with invariant, guard and variant, we proceed in this section to develop preliminaries for the iteration body. These preliminaries correspond to the exploration of means of making progress, as determined by the variant, whilst maintaining the invariant.

Augmenting Vf Augmentation of the set Vf is the prime way to make progress, as it also entails augmentation of the set of completely computed strongly connected components of the graph. We assume Vf is augmented as the result of finalising the construction of one of the intermediate components, which is then transferred to the set of final components. Conditions under which this is feasible can be calculated.

An assignment of the form $Vf := Vf \cup W$ guarantees augmentation of Vf if W is such that:

$$W \neq \emptyset \quad \wedge \quad W \cap Vf = \emptyset \ . \tag{7.19}$$

We want W to be the set of vertices comprising one of the intermediate components, i.e. comprising an element of SCi. Such an element would be a power-element of type $\mathsf{P}\,Vert$ which, by the power-transpose isomorphism (2.52), should correspond to a unique vector over $Vert$: our sought after W. We thus take W to be such that $\Lambda W \subseteq SCi$. This gives us, on account of $Inv7'$, property (2.61) of quotient sets, and $Q4$, that $W \neq \emptyset$ and $W \subseteq Vi$ hold. Requirements (7.19) on W are then fulfilled because Vf and Vi are disjoint by $Inv8$.

Transferral of the selected component ΛW involves shifting, not only the vertices W, but also some edges as well. We name the set of such edges F and assume it must be a subset of Ei, i.e. $F \subseteq Ei$. An appropriate value for F can be calculated as part of the analysis of the invariants.

From all the above, we conclude that the assignment under which maintenance of the invariants must be analysed is:

$$\begin{aligned} &SC, SCi,\ Vf, Vi, Ef, Ei \\ &\quad := \ SC \cup \Lambda W\,,\ SCi - \Lambda W\,,\ Vf \cup W\,,\ Vi - W\,,\ Ef \cup F\,,\ Ei - F\ . \end{aligned}$$

For the sake of compactness, we will denote the result of applying the substitution above to $Inv1$ by $\partial Inv1$, and the same goes for $Inv2$, $Inv3$, et cetera.

We first proceed with the x-V-E invariants. We only show the results of

the calculations, carried out using rules of the lattice structure:

$$\begin{aligned}
\partial Inv1 &\equiv x2^\circ \cdot W \subseteq Ef \cup F \ , \\
\partial Inv2 &\equiv x1 \cdot F \subseteq Vf \cup W \ , \\
\partial Inv3 &\equiv x2 \cdot (Ei - F) \subseteq \overline{W} \ , \\
\partial Inv4 &\equiv x1 \cdot (Ei - F) \subseteq \overline{W} \ , \\
\partial Inv5 &\equiv x2 \cdot F \subseteq Vf \cup Vi \ .
\end{aligned}$$

Adding the aforementioned –reasonable– assumption $F \subseteq Ei$, we then obtain, first, that the last one holds:

$$\partial Inv5 \equiv true \ ;$$

and, second, that two of the others give the value F must have:

$$\partial Inv2 \wedge \partial Inv4 \equiv F = Ei \cap x1^\circ \cdot W \ .$$

Finally, with the value of F having been determined, the remaining two x-V-E equations provide conditions under which the transferral of W to the set of final components is viable:

$$\partial Inv1 \wedge \partial Inv3 \equiv x2^\circ \cdot W \subseteq E \wedge x2^\circ \cdot W \cap Ei \subseteq x1^\circ \cdot W \ .$$

In words, all the outgoing edges of W have been inspected. And, either these are in Ef, in which case they lead into final components by $Inv2$, or they lead back into W, which shows W is not connected to any other intermediate component.

As for the rest of the invariants, $Inv6$ and $Inv7$ are maintained without additional assumptions. A fully formal proof of this claim would call for a little calculus of equivalence classes in order to make the presentation attractive. We only argue informally as follows. ΛW was drawn from SCi. By $Inv7'$ this means W is a $(str\, E)$-class included in set Vi. After the assignment statement at issue, which shuffles Vf, Vi, Ef and Ei, the full set of inspected edges E remains invariant but W is then included in Vf. Hence, W is still a $(str\, E)$-class though not included in Vi but in Vf. Preservation of $Inv6'$ and $Inv7'$ requires, therefore, that ΛW is subtracted from SCi and added to SC. Assuming the conditions for the maintenance of the x-V-E invariants shown above, since such invariants imply the equivalence between $Inv6$ and $Inv6'$, and between $Inv7$ and $Inv7'$, we conclude that $Inv6$ and $Inv7$ are thus also maintained under such conditions.

Maintenance of the disjointness invariants $Inv8$ and $Inv9$ is straightforward.

Augmenting *Ef* According to the variant, if Vf cannot be augmented as above, we ought to try augmenting set Ef without altering Vf. An increase in Ef can be accomplished by an assignment of the form $Ef := Ef \cup e$, where e is an edge such that $e \subseteq \overline{Ef}$. Even though the variant allows Vi and Ei to be changed in this case, it is worth first exploring the effect of altering only Ef. It turns out that no other change is necessary.

Again, the effect of the assignment above on the invariants will be denoted by $\partial Inv1$, $\partial Inv2$, and so on. The x-V-E invariants impose conditions on the extremes of e. *Inv3* and *Inv4* are not affected by the assignment at issue. For the others, we obtain:

$$\begin{aligned} \partial Inv1 &\equiv true\ , \\ \partial Inv2 &\equiv x1 \cdot e \subseteq Vf\ , \\ \partial Inv5 &\equiv x2 \cdot e \subseteq Vf \cup Vi\ . \end{aligned}$$

The $x1$-extreme of e must then be in Vf, and its $x2$-extreme must be either in Vf or in Vi since atoms are irreducible. But, were the $x2$-extreme of e included in Vf, shunting function $x2$ by (2.23) would entail $e \subseteq x2^{\circ} \cdot Vf$ and, therefore, edge e should be in Ef by *Inv1*, contradicting our original assumption that, in order to make progress, e had been drawn from $\overline{Ef}$. We thus conclude that maintenance of the invariant requires:

$$x1 \cdot e \subseteq Vf \ \wedge \ x2 \cdot e \subseteq Vi\ .$$

Further information on the set from which e originates can be deduced. Since e must not belong to Ef, it must come either from Ei or from $\overline{E}$. Were e drawn from Ei, the fact that its $x1$-extreme should be in Vf would contradict *Inv4* since Vf and Vi are disjoint. Hence, it must be the case that $e \subseteq \overline{E}$.

Invariants *Inv6*, *Inv7* and *Inv8* are trivially maintained since the variables involved are not altered. *Inv9* is also maintained due to e being taken from $\overline{E}$.

Augmenting *Vi* We now explore augmentation of set Vi. This must be achieved via an assignment of the form $Vi := Vi \cup v$, where v is a vertex not in Vi. Since Vf cannot be altered and must be kept disjoint from Vi, vertex v must come from $\overline{V}$. Again, we first explore the possibility of not altering Ei in spite of such an alteration being allowed by the variant and, again, it turns out that such a change is not necessary.

Invariants *Inv1* and *Inv2* are unaffected by the assignment at issue. *Inv3*, *Inv4* and *Inv5* are easily shown to be maintained since their left-hand sides are unaltered as their right-hand sides increase. *Inv6* is unaffected, whereas

Inv7 requires a change on variable SCi that we derive below. *Inv8* is maintained since v comes from $\overline{V}$, and *Inv9* is unaffected.

For the maintenance of *Inv7*, the required change on SCi is worked out as follows:

$$
\begin{array}{ll}
 & Inv7\,[\,SCi,\,Vi\;:=\;SCi'\,,\,Vi\,\cup\,v\,] \\
\equiv & \quad \{\ \text{substitution}\ \} \\
 & SCi'\;=\;\Lambda(str\,Ei)\cdot(Vi\,\cup\,v) \\
\equiv & \quad \left\{ \begin{array}{l} \text{distribution of composition over union (2.14);} \\ \text{power-transpose fusion (2.51),}\ v\ \text{is a function} \end{array} \right\} \\
 & SCi'\;=\;\Lambda(str\,Ei)\cdot Vi\;\cup\;\Lambda(str\,Ei\cdot v) \\
\equiv & \quad \{\ Inv7\ \} \\
 & SCi'\;=\;SCi\;\cup\;\Lambda(str\,Ei\cdot v)\ .
\end{array}
$$

By *Inv3* and *Inv4*, both extremes of edges in Ei are explored vertices included in Vi. Since new vertex v comes from the set of unexplored vertices $\overline{V}$, its $(str\,Ei)$-class should be a singleton that comprises only v itself. This is the kind of manipulation we had in mind when we pointed out before that a little calculus of equivalence classes would enhance the quality of the presentation. As a modest example, we show the proof of this claim in full:

$$
\begin{array}{ll}
 & str\,Ei\cdot v\;=\;v \\
\equiv & \quad \{\ str\,Ei\ \text{is reflexive}\ \} \\
 & str\,Ei\cdot v\;\subseteq\;v \\
\Leftarrow & \quad \{\ \text{definition of}\ str\ (7.7)\,,\ \text{intersection}\ \} \\
 & reach\,Ei\cdot v\;\subseteq\;v \\
\equiv & \quad \{\ \text{definition of}\ reach\ (7.6)\,,\ \text{closure (2.68)}\ \} \\
 & succ\,Ei\cdot v\;\subseteq\;v \\
\Leftarrow & \quad \left\{ \begin{array}{l} \text{definitions of}\ succ\ (7.5)\ \text{and}\ \not\subseteq\ (2.34)\,, \\ \text{intersection,}\ v\ \text{included in}\ \overline{V} \end{array} \right\} \\
 & x1\cdot Ei\cdot Ei^{\circ}\cdot x2^{\circ}\cdot\overline{V}\;\subseteq\;v \\
\Leftarrow & \quad \{\ \emptyset\ \text{is a zero of composition and is the least relation}\ \} \\
 & Ei^{\circ}\cdot x2^{\circ}\cdot\overline{V}\;\subseteq\;\emptyset \\
\equiv & \quad \left\{ \begin{array}{l} \text{Schröder's right-exchange rule (2.22), converse (2.16),} \\ \text{the universal relation of type}\ \ 1\leftarrow 1\ \ \text{is the identity} \\ \text{relation and, therefore, it is the case that}\ \ \overline{\emptyset}=id \end{array} \right\} \\
 & x2\cdot Ei\;\subseteq\;V \\
\equiv & \quad \{\ Inv3\,,\ \text{union}\ \}
\end{array}
$$

$true$.

Hence, new vertex v should indeed form a new singleton intermediate component. We then finally conclude that, in this case, the appropriate assignment statement to achieve progress without spoiling the invariant is $SCi, Vi := SCi \cup \Lambda v, Vi \cup v$.

Augmenting *Ei* To close this section, we deal with the last choice for making the iteration progress: increasing Ei whilst keeping Vf, Ef and Vi fixed. Such an increase must be achieved with edges not in Ei or Ef, in order both to guarantee progress and to maintain disjointness of final and intermediate edges. Hence, assigment $Ei := Ei \cup e$ with $e \subseteq \overline{E}$ is explored.

The effect on the x-V-E invariants gives conditious on the extremes of e. *Inv1*, *Inv2* and *Inv5* do not mention Ei and are thus unaffected. For *Inv3* and *Inv4*, the following holds:

$$\begin{aligned} \partial Inv3 &\equiv x2 \cdot e \subseteq Vi \ , \\ \partial Inv4 &\equiv x1 \cdot e \subseteq Vi \ . \end{aligned}$$

As for the rest of the invariants, *Inv6* is not affected, *Inv7* requires reorganising the intermediate components as explained below, *Inv8* is not affected, and *Inv9* is maintained for having drawn e from $\overline{E}$.

Regarding *Inv7*, as equivalence relation $str\ Ei$ grows, some equivalence classes in the partition SCi might need to be merged. The fact that both extremes of new edge e must lie, as stated above, in Vi means that e connects, in one direction, two intermediate components in SCi. If these two components were already connected in the opposite direction, a cycle has been created and, therefore, the components in such a cycle must be merged into a single component. The calculations regarding this are very knotty and we only give an overview.

We start by spelling out the key property of str involved in this manipulation. If F is an edge set and f is an edge, then:

$$\left.\begin{aligned} str\,(F \cup f) \ &= \ str\,F \ \cup \ C \cdot C^{\circ} \\ &\text{where} \quad C \ = \ reach\,F \cdot x1 \cdot f \ \cap \ (reach\,F)^{\circ} \cdot x2 \cdot f \ . \end{aligned}\right\} \quad (7.20)$$

Set C comprises the union of all the $(str\ F)$-components chained in a cycle under $reach\,(F \cup f)$, if such a cycle exists, that is. If no such cycle has been created, C turns out to be empty.

Property (7.20) provides a handle on $\partial Inv7$. Let W be $C[F, f := Ei, e]$. This means W is the new intermediate SCi-component that should result

from merging those components chained into a cycle by new edge e. Hence, augmenting Ei with e requires the following updating of SCi:

$$\begin{array}{l} \textbf{if}\ \ W = \emptyset \ \rightarrow\ \textbf{skip} \\ [\!]\ \ W \neq \emptyset \ \rightarrow\ SCi := (SCi - \Lambda(str\ Ei) \cdot W) \cup \Lambda W \\ \textbf{fi} \end{array}$$

The expression $\Lambda(str\ Ei) \cdot W$ above denotes the sub-partition in SCi that must be merged into the single ΛW. The set of such components can be equivalently expressed in terms of only SCi and W thus:

$$(\cup c : c \subseteq SCi \ \wedge\ \in \cdot c \subseteq W : c)\ , \tag{7.21}$$

where c is a dummy that ranges over elements of $\mathsf{P}\,Vert$.

7.5 Assembling the Iteration Body

The previous section explored means of making the iteration progress, but we still need to assemble the odds and ends into a correct iteration body. Correct meaning that it refines the specification statement left in our last refinement step in page 146:

$$SC, SCi, Vf, Vi, Ef, Ei : [\ Vf \neq Vert \ \wedge\ Inv\ ,\ Inv \ \wedge\ Prg\ ]\ . \tag{7.22}$$

This will give shape to a program that correctly refines the initial specification (7.4). And we will consider this program our final program, yet further refinement and some implementation details will be briefly discussed in the next and last section.

Preconditions for Progress Each of the four ways to make progress was subject to some conditions on the various pieces of data used. We gather such conditions in the list that follows:

(i) Augmentation of Vf requires the existence of an intermediate component W, i.e. for which $\Lambda W \subseteq SCi$ holds, such that $x2^{\circ} \cdot W \subseteq E$ and $x2^{\circ} \cdot W \cap Ei \subseteq x1^{\circ} \cdot W$.

(ii) Augmentation of Ef requires the existence of an unexplored edge e, i.e. for which $e \subseteq \overline{E}$ holds, with extremes such that $x1 \cdot e \subseteq Vf$ and $x2 \cdot e \subseteq Vi$.

(iii) Augmentation of Vi requires the existence of an unexplored vertex v, i.e. for which $v \subseteq \overline{V}$ holds.

(iv) Augmentation of Ei requires the existence of an unexplored edge e, i.e. for which $e \subseteq \overline{E}$ holds, with extremes such that $x1 \cdot e \subseteq Vi$ and $x2 \cdot e \subseteq Vi$.

We now analyse this list to gain a better understanding on how to fit together the progress-making statements.

First, a connection is drawn between case (i) and cases (ii) and (iv). Case (i) requires the existence of a component W such that $x2^\circ \cdot W \subseteq E$, which is equivalent to $x2^\circ \cdot W - E$ being empty and means that all outgoing edges of W have been explored. Cases (ii) and (iv), on the other hand, require an unexplored outgoing edge of Vi. Since Vi comprises all the vertices in partition SCi, such an edge must go out from, i.e. have its $x2$-extreme in, an intermediate component. Formally:

$$
\begin{array}{ll}
 & x2 \cdot e \subseteq Vi \\
\equiv & \quad \{\ Q4\text{, reflexivity of } str\, E\ \} \\
 & x2 \cdot e \subseteq str\, E \cdot Vi \\
\equiv & \quad \{\ Inv7'\text{, power-transpose cancellation (2.50)}\ \} \\
 & x2 \cdot e \subseteq \in \cdot SCi \\
\equiv & \quad \{\ \text{extensionality (2.29)}\ \} \\
 & x2 \cdot e \subseteq \in \cdot (\cup c : c \subseteq SCi : c) \\
\equiv & \quad \left\{ \begin{array}{l} \text{distribution of composition over} \\ \text{union, atoms are irreducible} \end{array} \right\} \\
 & (\exists c : c \subseteq SCi : x2 \cdot e \subseteq \in \cdot c) \\
\equiv & \quad \left\{ \begin{array}{l} \text{change of quantification dummy } W := \in \cdot c\text{,} \\ \text{power-transpose isomorphism (2.52)} \end{array} \right\} \\
 & (\exists W : \Lambda W \subseteq SCi : x2 \cdot e \subseteq W) \\
\equiv & \quad \left\{ \begin{array}{l} \text{shunting of functions (2.23); both cases (ii) and (iv)} \\ \text{require } e \text{ to be unexplored –i.e. included in } \overline{E}\text{–,} \\ \text{universal property of intersection (2.6), subtraction} \end{array} \right\} \\
 & (\exists W : \Lambda W \subseteq SCi : e \subseteq x2^\circ \cdot W - E)\ . \qquad (7.23)
\end{array}
$$

Therefore, cases (ii) and (iv) also require, as case (i) does, the existence of an intermediate component W, with the difference that now $x2^\circ \cdot W - E$ should be non-empty and an edge drawn from it is needed.

After this observation, we proceed with the actual assembly of cases for the body of the iteration. Recall that we are in the process of dealing with the specification statement (7.22). Therefore, the guard $Vf \neq Vert$ can be counted in as a precondition.

No Intermediate Components Having seen that (i), (ii) and (iv) all require the existence of an intermediate component, only (iii) is applicable when SCi is empty. To apply (iii) it is required that $\overline{V}$ is non-empty, which indeed follows from SCi being empty as we now argue. First note that

SCi is empty if and only if Vi is empty; this can be formally shown using $Inv7$. Partition SCi being empty then implies V being equal to Vf, and Vf does not contain all the vertices of the graph on account of the guard. Hence, $V \neq Vert$ holds and, thus, so does $\overline{V} \neq \emptyset$.

We conclude that the following "naked" guarded statement does the job when there are no intermediate components at all:

$$
\begin{array}{lll}
SCi = \emptyset & \rightarrow & |[\ \ \textbf{var}\ v : Vert\ ; \\
& & \quad v :\subseteq \overline{V}\ ;\ \ SCi, Vi := \Lambda v, v \\
& &]|
\end{array}
$$

Note that we have made use of the fact that Vi is empty on account of SCi being empty.

An Intermediate-to-Final Component Case (i) is dealt with straightforwardly by assembling its precondition and progress-making statement in a guarded command of the following form:

$$
\begin{array}{l}
\langle \exists W\ :\ \Lambda W \subseteq SCi\ :\ x2^{\circ} \cdot W - E = \emptyset \\
\qquad\qquad\qquad\qquad\qquad \wedge\ \ x2^{\circ} \cdot W \cap Ei \subseteq x1^{\circ} \cdot W \rangle \\
\quad \rightarrow\ |[\ \ \textbf{var}\ W : \mathrm{Vec}\ Vert\ ; \\
\qquad\qquad F\ : \mathrm{Vec}\ Edge\ ; \\
\qquad\ \vdots \\
\quad]|
\end{array}
$$

Some Intermediate-not-Final Components To tackle cases (ii) and (iv) we use the fact that the requirement on the $x2$-extreme of new edge e is equivalent to (7.23). This means there must be an intermediate component W that is not final yet. Not final on account of its set of outgoing edges not having been fully explored, i.e. $x2^{\circ} \cdot W - E$ being non-empty. Picking such a component plus one of its unexplored outgoing edges gets us prepared to handle cases (ii) and (iv). However, such cases only apply when the other extreme, i.e. the $x1$-extreme, of the edge is in Vf or in Vi, and there is the third possibility of it being in $\overline{V}$. This third potential situation is handled via a combination of (iii), whereby the unexplored new vertex is added to the family of intermediate components, and (iv), regarding the new vertex as already a member of Vi.

All the above is formally arranged in a guarded statement as follows:

$$\langle \exists W\ :\ \Lambda W \subseteq SCi\ :\ x2^{\circ} \cdot W - E \neq \emptyset \rangle$$

$$
\begin{array}{l}
\rightarrow \ \|[\ \ \textbf{var}\ W : \mathrm{Vec}\ \mathit{Vert}\ ; \\
\qquad\qquad e : \mathit{Edge}\ ;\ v : \mathit{Vert}\ ; \\
\qquad W : [\ \mathit{true}\ ,\ \Lambda W \subseteq \mathit{SCi}\ \wedge\ x2^{\circ} \cdot W - E \neq \emptyset\ ]\ ; \\
\qquad e :\subseteq x2^{\circ} \cdot W - E\ ;\ v := x1 \cdot e\ ; \\
\qquad \textbf{if}\ v \subseteq \mathit{Vf}\ \rightarrow\ \cdots \\
\qquad [\!]\ v \subseteq \mathit{Vi}\ \rightarrow\ \cdots \\
\qquad [\!]\ v \subseteq \overline{V}\ \rightarrow\ \cdots \\
\qquad \textbf{fi} \\
\quad \|]
\end{array}
$$

In the third branch of the inner alternation, where (iii) and (iv) are combined, the rather complex progress-making statement of case (iv) can be simplified. This is due to the fact that, having just added via (iii) the $x1$-extreme of e to Vi, no need of merging components arises. Formally, this corresponds to W as used by case (iv) in pages 151-152 being empty. The result of this simplification is shown in Figure 7.24, where the body of the iteration is assembled.

Finishing Off The three guards of the "naked" guarded commands we have given are *complete*, this meaning that their disjunction is equivalent to *true*. The guarded commands at issue can therefore be joined in an alternation that correctly refines the specification statement of the body of our iteration. The whole alternation is presented in Figure 7.24.

For proving that the disjunction of the guards holds, we will make use of contracted *graphs*. Let G be a graph and Q be an equivalence relation on its vertex set. The *Q-contraction* of G is a graph whose vertices are the Q-equivalence classes and whose edges are those edges of G not incident on Q-equivalent vertices. The extremes of edges in the Q-contracted graph of G are the Q-classes to which their extremes belong in G. It is a fact that, for every equivalence relation Q, if G is finite then so is its Q-contraction.

The particular use we need of contractions is that determined by the equivalence relation $\mathit{str}\ \mathit{Ei}$ on the input graph. In such a contracted graph, there cannot be cycles linked by edges in Ei: the existence of a cycle of such a form would imply that the vertices in the cycle form a bigger $(\mathit{str}\ \mathit{Ei})$-class when united, contradicting the assumption that each vertex in the cycle was a $(\mathit{str}\ \mathit{Ei})$-class in the first place.

Let us now prove that the guards are complete. Assume neither the second guard nor the third guard holds. This means:

$$
\begin{array}{l}
(\forall W\ :\ \Lambda W \subseteq \mathit{SCi}\ :\ x2^{\circ} \cdot W - E \neq \emptyset \\
\qquad\qquad\qquad\qquad \vee\ \ x2^{\circ} \cdot W \cap \mathit{Ei} \not\subseteq x1^{\circ} \cdot W)\ ,
\end{array}
$$

```
if SCi = ∅ →
    ||[ var v : Vert ;
        v :⊆ V̄ ;  SCi, Vi := Λv, v
    ]||
▯ ⟨∃W : ΛW ⊆ SCi : x2° · W − E = ∅
                    ∧  x2° · W ∩ Ei ⊆ x1° · W ⟩ →
    ||[ var W : Vec Vert ;
            F : Vec Edge ;
        W : [ true , ΛW ⊆ SCi ∧ x2° · W − E = ∅
                              ∧ x2° · W ∩ Ei ⊆ x1° · W ] ;
        F := Ei ∩ x1° · W ;
        SC, SCi := SC ∪ ΛW , SCi − ΛW ;
        Vf, Vi, Ef, Ei := Vf ∪ W , Vi − W , Ef ∪ F , Ei − F
    ]||
▯ ⟨∃W : ΛW ⊆ SCi : x2° · W − E ≠ ∅ ⟩ →
    ||[ var W : Vec Vert ;
            e : Edge ; v : Vert ;
        W : [ true , ΛW ⊆ SCi ∧ x2° · W − E ≠ ∅ ] ;
        e :⊆ x2° · W − E ;  v := x1 · e ;
        if v ⊆ Vf → Ef := Ef ∪ e
        ▯ v ⊆ Vi → W := reach Ei · x1 · e
                              ∩ (reach Ei)° · x2 · e ;
                   if W = ∅ → skip
                   ▯ W ≠ ∅ →
                       SCi := (SCi − Λ(str Ei) · W) ∪ ΛW
                   fi ;
                   Ei := Ei ∪ e
        ▯ v ⊆ V̄ → SCi, Vi, Ei := SCi ∪ Λv , Vi ∪ v , Ei ∪ e ;
        fi
    ]||
fi
```

Figure 7.24: Alternation that refines the specification statement of the iteration body (7.22)

$$\langle\, \forall W \,:\, \Lambda\mathcal{W} \subseteq SCi \,:\, x2^{\circ} \cdot W - E = \emptyset \,\rangle \ .$$

By predicate calculus, this implies:

$$\langle\, \forall W \,:\, \Lambda\mathcal{W} \subseteq SCi \,:\, x2^{\circ} \cdot W \cap Ei \not\subseteq x1^{\circ} \cdot W \,\rangle \tag{7.25}$$

We will now show that the first guard must hold as follows: Were it the case that $SCi \neq \emptyset$ holds, (7.25) would entail the existence of an infinite chain of $(str\ Ei)$-classes linked by edges in Ei. This would contradict the fact that the $(str\ Ei)$-contracted graph is finite and acyclic. Hence, we will conclude that the first guard, i.e. $SCi = \emptyset$, must hold if the other two guards do not.

We now proceed to construct the above-referred infinite chain. Suppose $SCi \neq \emptyset$. Then take an element ΛW_0 in SCi. By *Inv7*, set W_0 must be a $(str\ Ei)$-class and, by (7.25), there must be an outgoing edge of W_0 in Ei not leading back into W_0. Invariant *Inv4* guarantees that such an edge must go to a vertex in Vi and, therefore, into an element ΛW_1, different to ΛW_0, in SCi. The reasoning carried out for ΛW_0 applies to ΛW_1 as well, i.e. it must be the case that W_1 is a $(str\ Ei)$-class for which some outgoing edge in Ei leads into a different ΛW_2 in SCi, et cetera. Hence, we can construct an infinite chain $[\Lambda W_0, \Lambda W_1, \Lambda W_2, \ldots]$ that implies the aforementioned contradiction and thus shows the completeness of the three guards in Figure 7.24.

7.6 Further Refinement

Up to this point we have arrived at an abstract program that we offer as final. The level of abstraction determined by its use of sets, be it in the form of powersets or vectors, is no different to that of the programs that were offered as final in previous chapters. However, some pieces of it are even "less executable" than the average abstract program dealt with in this thesis, e.g. the guards of the second and third branches of the alternation in Figure 7.24. This section puts forward means of further refining our final program towards an implementation, but without providing all the detailed technicalities.

Data-Refining *SCi* Variable SCi holds the partition of set Vi of vertices in conformity with the strong connectedness relation determined by set Ei of edges. Without further provisos, partition SCi can have any shape. We might say that it can be as "disorganised" as it wants, and this fact may become a burden for the efficient evaluation of the guards of the main, i.e. outermost, alternation. Imposing some order upon the shape of SCi solves this problem, the order being that its components must be linked sequentially

without rightward references. More specifically, data-refining SCi to a list in which all outgoing Ei-edges from any component must lead into the same component, except for one edge that must lead into the immediate component on its left, unless no such component exists.

Such a data refinement guarantees that the head of the "listified" SCi, the head being the leftmost component, absorbs all its own outgoing Ei-edges into itself. Formally, for the vector W that corresponds to the component at the head of the list, the following holds:

$$x2^\circ \cdot W \cap Ei \subseteq x1^\circ \cdot W \ .$$

Therefore, the second and third guards of the main alternation can be implemented just by inspecting whether all the outgoing edges of the head component have been explored. By the same token, the specification statements that open the branches of these same guards can be implemented just by picking the head of the list.

In the body of the third branch of the main alternation, another simplification is brought about by this data refinement. Specifically, in the middle branch of the alternation within it, i.e. the branch with $v \subseteq Vi$ as guard. The value therein assigned to W comes down to the set of vertices comprising the initial segment of the list up to the component where the $x1$-extreme of e, i.e. v, resides. Such a value is uever empty and the innermost alternation can thus be disposed of retaining only its second branch, which now should just contract the above-referred initial segment into one single component.

In all other assignments to SCi, it is simple to enforce the no-rightward-references list structure.

Avoiding Computation of Complements and More Throughout the program, repeated use is made of the expressions $\overline{V}$ and $\overline{E}$ –though the latter is disguised as $(-E)$, which corresponds to $(\cap \overline{E})-$. As the reader might recall, neither V nor E are variables of the program, but only abbreviations for $Vf \cup Vi$ and $Ef \cup Ei$, respectively, which means that computing the complemented expressions above can be quite expensive.

Introduction via data refinement of two new variables V' and E' with coupling invariant $(V' = \overline{Vf \cup Vi} \ \wedge \ E' = \overline{Ef \cup Ei})$ does away with the cost of computing complements. All assignments to Vf, Vi, Ef and Ei are extended with appropriate assignments to V' and E'.

Introduction of E' also brings even better news: variables Ef and Ei, as well as local variable F, can be eliminated since they become useless! The only place where Ei is used for computing values to be assigned to other

variables is in the middle branch of the inner alternation. As remarked above when data-refining SCi, the value assigned to W in the branch at issue can be computed without using Ei. The value assigned to SCi in the same branch can also be computed without the help of Ei by using expression (7.21).

Driving Back Home Recall the introduction to this chapter, which remarks that our motivation for tackling the problem of computing strongly connected components came from two previous pieces of work by Dijkstra and Kruseman Aretz. The program obtained after all the simplifications mentioned in this section is very similar to the programs offered in Dijkstra's [45, Chapter 25] and [46], before arrays are brought in, and in Kruseman Aretz's [89], except for not having data-refined SC along with SCi into a list.

7.7 Related Work

Tarjan's algorithm for the computation of the strongly connected components of a directed graph [141] is much better known than the algorithmic solutions offered by Dijkstra and Kruseman Aretz in their respective essays. The correctness of Tarjan's algorithm relies on structural properties of what he calls a *palm tree*: a subtree of the input graph created by depth-first traversal. Tarjan also numbers the vertices of the graph according to the order in which the palm tree is constructed, and attaches a second numbering to each vertex in conformity with some other structural properties of the tree.

Kruseman Aretz points out that in his solution "also a depth-first graph traversal is present" and supports his claim with "symptom[s] for it". Also, he considers that the vertex numbering is "overspecific" and "obscures" important characteristics of the algorithm [89, page 259]. We agree with his claim on the presence of a depth-first traversal in his solution, yet hold the opinion that it would be pleasant to see such a link being made "obvious" by some kind of formal structure. As regards the vertex numbering, we also side with Kruseman Aretz, yet note the fact that the properties of palm trees on which such a numbering is based have proved useful in the design of many other algorithms. These include Tarjan's algorithm for the computation of biconnected components [141], Kosaraju and Sharir's algorithm for the computation of strongly connected components [138], Hopcroft and Tarjan's algorithm for testing the planarity of a graph and building its planar embedding [77], and a few others. Since the correctness proofs of all these algorithms make use of properties of palm trees, it seems that a good deal of work must be done to support Kruseman Aretz's claim of the "over-

specificity" and "obscnrity" brought about by palm trees in a more general context. Again, we remark that we sympathise with his claim, bnt also note that there is a long way to go before adequate presentations of all the aforementioned algorithms are built via suitable abstractions that send away palm trees yet still connect the whole algorithmic family.

Apart from the already mentioned essays by Dijkstra, the only two other references we know of that offer *derivational* presentations of algorithms based on depth-first traversals are Gries and Jinynn Xue's [70] and Madhukar et al.'s [96]. Both presentations are, in spirit, similar. They use the same standard techniques for development of imperative programs we make use of, but no graph-oriented formal calcnli for the manipulation of the properties their algorithms rely on. Therefore, their manipulation of preconditions, postconditions, invariants and so on is carried out in a conventional fashion, or simply omitted by referencing standard books on graph theory. No calculations regarding graph properties are present at all. It seems that adding to their work the kind of manipulations with relations we put forward in this thesis is worth researching. We also remark that the derivations in both articles are based on Tarjan's palm trees.

A related reference is that of King and Launchbury's implementation of depth-first traversal grapb algorithms in a functional programming language [84, 91]. Derivations of the programs are not given, but compact and nice correctness proofs in a calculational style are. The possibility of tnrning such correctness proofs into derivations is worth exploring.

Chapter 8
Conclusions

This thesis has presented a small portion of graph theory and algorithmics using the calculus of binary of relations as working tool. The goals we initially set up have been, in our opinion, successfully achieved, yet nevertheless with some drawbacks. This final chapter offers a summary of the results obtained in this thesis and an appraisal of their worth. Hints on further research are given.

Relations and Graphs Chapter 2 presented the basics of graph theory within the calculational framework of binary relations. A considerable part of its contents is the result of previous research, most of it gathered in [136], but the treatment of a few concepts and properties, viz. the biconnectedness equivalence relation, the formalisation of acyclicity in undirected graphs and its linkage to connectedness through the covering relation, and the manipulation of paths under the allegorical approach to datatypes, appears to be original work. We are quite pleased with these results and we believe the exploration of graph theory with the calculus of relations is a task worth continuing.

We feel particularly contented with the covering relation defined in Section 2.6 and the exploitation of its properties thus far. We first came across it when dealing with the acyclicity instance of the general algorithm for the computation of maximal sets in Chapter 5. Its further application to linking acyclicity with connectedness in Section 2.7, and to reasoning about edge replacement in spanning trees in Section 6.3 and about cuts and crossings in Section 6.5 was positively satisfying. Still, our understanding of the link between all these concepts can benefit from a deeper exploration of the covering relation, striving for more compact presentations of its linking role. The covering relation makes use of function *adj* which, on account of its distributivity over arbitrary unions, as briefly mentioned in Section 6.8, is the lower adjoint of a Galois connection between the lattice of edge sets and

the lattice of symmetric relations on the vertex set of any given undirected graph. Most, if not all, of the properties of the covering relation are related to this fact. A good starting point for this suggested further exploration is the study of the covering relation in the context of pair algebras [9].

The study of paths under the allegorical approach to datatypes can also be extended by researching its connection to the modelling of paths within the calculus of n-ary relations of Möller [104, 105].

Calculational Graph Algorithmics Chapters 3 to 7, the bulk of this thesis, were concerned with the derivation of programs that solve graph computational problems. These derivations were carried out by combining predicate, refinement and relational calculi. This combination allowed the treatment of algorithmic principles as well as graph properties in a calculational fashion. We think we have succeded in proving the applicability of the framework of binary relations to the derivation of graph algorithms. Nonetheless, there is a good deal of room for improvement in the work we have presented, which we will comment on as we summarise the achievements in the contents of these chapters.

Chapter 3 presented, as a warming-up exercise, the derivation of graph algorithms that correspond to the computation of the reflexive-transitive closure of given input relations. These algorithms had been treated derivationally by others before [12, 133, 136], but we showed an innovative use of the fixed-point calculus for obtaining these algorithms first presented in [30].

The general problem of computing representatives in Chapter 4 dealt with a class of graph algorithms also related to closure that includes the minimum paths, shortest paths, and reachability problems –the last one was also treated in Chapter 3–. All the different algorithmic solutions to this class of graph problems previously derived by others [12, 133] were successfully covered once and for all. In Chapter 5, two graph algorithms were constructed as instances of a general problem of computing maximal sets. These two chapters are in tune with one of the main goals of the mathematics of program construction, viz. the successful abstraction of key concepts involved in the design of an algorithm that permits eliminating unnecessary detail and allows its derivation as an instance of a generic family of algorithms. Such a generic approach is highly beneficial since the doors remain open to the obtention of new instances. In the case of Chapter 5, as remarked in Section 5.1, every matroid provides a new instance of the general problem and its algorithmic solution. The discovery of more instances can be aided by other calculi, as regards proving that such new candidates satisfy the requirements of the generic family.

Chapters 6 and 7 are both the main source of satisfaction and of dissatisfaction of our tackling of calculational graph algorithmics. On the one hand, satisfaction comes from the fact that they present our biggest case-studies thus far, especially Chapter 7, and the derivations therein presented are successful in so far as algorithmic solutions are actually obtained by deduction and calculation. On the other hand, the length and knottiness of the calculations is dissatisfying. It is not fair to put this down to the complexity of the problems therein treated. Neither do we think this is evidence against the applicability of the calculus of relations to graph algorithmics. We believe this comes down to not having been able to come up yet with adequate abstractions of the key graph-theoretical and algorithmic features involved; such abstractions would aid a more compact presentation. Some thoughts on potential sources of improvement follow.

Chapter 6 made a good deal of use of the general derivation in the preceding chapter. Rather than reusing, adapting and extending the results of Chapter 5, it might well be the case that the general development can be carried out with such extra features already under consideration. This would allow the obtention of the algorithms of Chapter 6 as plain instances of the general solution. We believe this is true both of the minor adaptation that catered for Kruskal's algorithm and of the larger adaptation and extension that catered for Prim's algorithm. Clearly, this calls for further research. Besides, the graph properties involved could be dealt with more concisely by searching for better abstractions. This relates to the previous remarks on the covering relation since, for instance, as briefly commented on in Section 6.8, Galois connections can make edge replacement in spanning trees be treated in a more succint fashion.

Chapter 7 includes a number of lengthy calculations as well as several properties treated only by "verbal formality". Again, we think this can be improved by continuing the search for a more compact expression of the core of the graph properties and algorithmic principles the problem involves. We lay stress once more on our belief that this is not a drawback of the calculus of relations itself in relation to graph algorithmics. On the contrary, we are quite pleased with the fact that relational calculations allowed the "discovery" of the main features of Dijkstra's derivational treatment of the same problem in [46]. It seems to us that the framework of relations provides a solid foundation on which to build the required further abstractions. An instance of this –and a somewhat trivial one– is the little calculus of equivalence classes referred to in Section 7.4.

Some Final Remarks We close this chapter –and this thesis!– with a few concluding remarks and some other incidental observations.

It must be admitted that we needed some training in derivational programming and this thesis served such a purpose considerably. Consequently, we must confess that, in most cases, we developed the derivations with an algorithmic solution in mind. However, we had only roughly sketched ideas of such solutions and the final details were always calculated. Still, we exerted ourselves for deducing or explicitly presenting the reasons that gave rise to every design decision. We believe we have succeded in such an endeavour, even when we were obviously heading for a specific kind of final solution as, e.g., when deriving Kruskal's and Prim's algorithms for the computation of minimum spanning trees.

We think we have positively helped graph algorithmics to move towards a more modern, and clearer, presentation of its often intricate details. There is still an awful lot of work to be done, not only for the vast number of other graph algorithmic problems not tackled in this thesis or in the other references we have given, but also for the need of further improvement on the presentation of the few problems treated here, especially those in Chapters 6 and 7.

As a somewhat incidental remark, we comment on an interesting and somewhat amusing article by Harary and Read on "The Null-Graph", i.e. the graph with no vertices and, hence, no edges [72]. They show how many well-known authors on graph theory advocate for the admittance or rejection of this "paradoxical beast" as a graph. Our -very- limited knowledge of graph theory has not yet given us any signs of the null-graph being troublesome. Accordingly, we would "vote" for its acceptance as a graph. All the graph concepts, properties and algorithms presented in this thesis apply to the null-graph without this leading to any contradictions or paradoxes. Particularly interesting is the case of Prim's algorithm, whose first step requires drawing an arbitrary vertex of the input graph and, hence, cannot deal with the null-graph. As remarked in Section 6.7, we are pleased to have designed a version of Prim's algorithm applicable to the null-graph as well as to non-connected graphs, the other characteristic of input graphs most versions of Prim's algorithm reject.

Another incidental observation regards finiteness of graphs. In all our algorithmic chapters, the graphs were assumed to be finite in order to guarantee termination of the derived programs. Functional programming languages with lazy evaluation [23, 81, 144] are well-known for being able to manipulate infinite data structures as easily as finite ones. In many cases, the correctness of functional programs is independent of whether the data structures being manipulated are finite or infinite. The techniques used in functional programming for reasoning about such data structures could well help extending graph algorithms to deal with the infinity case.

Bibliography

[1] C.J. Aarts. Galois connections presented calculationally. Graduating Dissertation, Eindhoven University of Technology, 1992. Available from `http://www.win.tue.nl/win/cs/wp/`.

[2] C.J. Aarts, R.C. Backhouse, P.F. Hoogendijk, E. Voermans, and J.C.S.P. van der Woude. A relational theory of datatypes. Working document, 1992. Available from `http://www.win.tue.nl/win/cs/wp/`.

[3] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Series in Computer Science and Information Processing. Addison-Wesley, 1974.

[4] R.-J.R. Back. Correctness preserving program refinements: Proof theory and applications. Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[5] R.-J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[6] R.-J.R. Back and J. von Wright. *Refinement Calculus.* Graduate Texts in Computer Science. Springer-Verlag, 1998.

[7] R.C. Backhouse. *Program Construction and Verification.* International Series in Computer Science. Prentice Hall, 1986.

[8] R.C. Backhouse. Making formality work for us. *EATCS Bulletin*, 38:219–249, 1989.

[9] R.C. Backhouse. Pair algebras and Galois connections. *Information Processing Letters*, 67(4):169–175, 1998.

[10] R.C. Backhouse and B.A. Carré. Regular algebra applied to path-finding problems. *Journal of the Institute of Mathematics and its Applications*, 15:161–186, 1975.

[11] R.C. Backhouse, H. Doornbos, and P.F. Hoogendijk. A class of commuting relators. Department of Mathematics and Computing Science, Eindhoven University of Technology, 1992. Available from `http://www.win.tue.nl/win/cs/wp/`.

[12] R.C. Backhouse, J.P.H.W. van den Eijnde, and A.J.M. van Gasteren. Calculating path algorithms. *Science of Computer Programming*, 22(1-2):3–19, 1994. Earlier version in [13].

[13] R.C. Backhouse and A.J.M. van Gasteren. Calculating a path algorithm. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science 669, pages 32–44. Springer-Verlag, 1992.

[14] R.C. Backhouse, guest editor. Special issue on the calculational method. *Information Processing Letters*, 53(3), 1995.

[15] M. Barr and C. Wells. *Category Theory for Computing Science*. International Series in Computer Science. Prentice Hall, 2nd edition, 1995.

[16] R. Berghammer. Combining relation calculus and the Dijkstra-Gries method for deriving relational programs. To appear in *Journal for Information Sciences*, 1999.

[17] R. Berghammer, R. Behnke, and P. Schneider. Relational programs in the RELVIEW system, 1997. Available from `http://www.informatik.uni-kiel.de/~progsys/relview.html`.

[18] R. Berghammer, T.F. Gritzner, and G. Schmidt. Prototyping relational specifications using higher-order objects. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Higher Order Algebra, Logic and Term Rewriting*, Lecture Notes in Computer Science 816. Springer-Verlag, 1993.

[19] R. Berghammer and B. von Karger. Algorithms from relational specifications. Chapter 8 of [32], pages 131–149.

[20] R. Berghammer, B. von Karger, and A. Wolf. Relation-algebraic derivation of spanning tree algorithms. In J. Jeuring, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 1422, pages 23–43. Springer, 1998.

[21] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, NATO ASI Series F: Computer and Systems Sciences, Volume 36, pages 3–42. Springer-Verlag, 1987.

[22] R.S. Bird. A calculus of functions for program derivation. In D.A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, pages 287–308. Addison-Wesley, 1990.

[23] R.S. Bird. *Introduction to Functional Programming using Haskell.* International Series in Computer Science. Prentice Hall, 2nd edition, 1998.

[24] R.S. Bird and O. de Moor. From dynamic programming to greedy algorithms. In B. Möller, H. Partsch, and S. Schumann, editors, *Formal Program Development*, Lecture Notes in Computer Science 755, pages 43–61. Springer-Verlag, 1992.

[25] R.S. Bird and O. de Moor. Solving optimisation problems with catamorphisms. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science 669, pages 45–66. Springer-Verlag, 1992.

[26] R.S. Bird and O. de Moor. List partitions. *Formal Aspects of Computing*, 5(1):61–78, 1993.

[27] R.S. Bird and O. de Moor. Relational program derivation and context-free language recognition. In A.W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, International Series in Computer Science, pages 17–35. Prentice Hall, 1994.

[28] R.S. Bird and O. de Moor. *Algebra of Programming.* International Series in Computer Science, 100th Title. Prentice Hall, 1997.

[29] R.S. Bird, O. de Moor, and P.F. Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.

[30] R.S. Bird and J.N. Ravelo. On computing representatives. *Information Processing Letters*, 63(1):1–7, 1997.

[31] B. Bollobás. *Graph Theory.* Graduate Texts in Mathematics 63. Springer-Verlag, 1979.

[32] C. Brink, W. Kahl, and G. Schmidt, editors. *Relational Methods in Computer Science.* Advances in Computing Science. Springer, 1997.

[33] T. Brunn, B. Möller, and M. Russling. Layered graph traversals and hamiltonian path problems – An algebraic approach. In J. Jeuring, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 1422, pages 96–121. Springer, 1998.

[34] K. Clenaghan. Calculational graph algorithmics: Reconciling two approaches with dynamic algebra. Report CS-R9518, CWI, 1995.

[35] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 4th edition, 1994.

[36] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* The MIT Electrical Engineering and Computer Science Series. MIT Press, 1990.

[37] S. Curtis. Partitions revisited. Qualifying dissertation for transfer to DPhil status, Oxford University Computing Laboratory, 1993. Available from
`http://www.comlab.ox.ac.uk/oucl/publications/books/algebra/`.

[38] S. Curtis. *A Relational Approach to Optimization Problems.* DPhil thesis, Oxford University Computing Laboratory, Programming Research Group, 1996. Available as Technical Monograph PRG-122 and also from
`http://www.comlab.ox.ac.uk/oucl/publications/books/algebra/`.

[39] S. Curtis. Dynamic programming: A different perspective. In R.S. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 1–23. IFIP, Chapman & Hall, 1997.

[40] S. Curtis and G. Lowe. A graphical calculus. In B. Möller, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 947, pages 214–231. Springer-Verlag, 1995.

[41] S. Curtis and G. Lowe. Proofs with graphs. *Science of Computer Programming*, 26:197–216, 1996. Earlier version in [40].

[42] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge Mathematical Textbooks. Cambridge University Press, 1990.

[43] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1, 1959.

[44] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[45] E.W. Dijkstra. *A Discipline of Programming.* Series in Automatic Computation. Prentice Hall, 1976.

[46] E.W. Dijkstra. Finding the maximum strong components of a directed graph (EWD376). In *Selected Writings on Computing: A Personal Perspective*, Texts and Monographs in Computer Science, pages 22–30. Springer-Verlag, 1982.

[47] E.W. Dijkstra and W.H.J. Feijen. *A Method of Programming*. Addison-Wesley, 1988.

[48] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[49] R.M. Dijkstra. Relational calculus and relational program semantics. Report CS-R9408, University of Groningen, Department of Mathematics and Computing Science, 1994.

[50] J.P.H.W. van den Eijnde. Conservative fixpoint functions on a graph. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science 669, pages 80–99. Springer-Verlag, 1992.

[51] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiae Scientarum Imperialis Petropolitanae*, 8:128–140, 1736.

[52] L. Euler. The Königsberg bridges. *Scientific American*, 189(1):66–70, 1953.

[53] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, Programs from outer space). In *Principles of Programming Languages*. ACM Press, 1996. Available from `http://www-cse.uta.edu/~fegaras/fegaras.html`.

[54] L.M.G. Feijs and R.C. van Ommering. Abstract derivation of transitive closure algorithms. *Information Processing Letters*, 63(3):159–164, 1997.

[55] R.W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Mathematical Aspects of Computer Science*. American Mathematical Society, 1967.

[56] L.R. Ford and D.R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[57] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[58] L.R. Foulds. *Graph Theory Applications.* Springer-Verlag, 1992.

[59] P. Freyd and A. Ščedrov. *Categories, Allegories.* Mathematical Library, Volume 39. North-Holland, 1990.

[60] A.J.M. van Gasteren. *On the Shape of Mathematical Arguments.* Lecture Notes in Computer Science 445. Springer-Verlag, 1990.

[61] A. Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1985.

[62] J. Gibbons. *Algebras for Tree Algorithms.* DPhil thesis, Oxford University Computing Laboratory, Programming Research Group, 1991. Available as Technical Monograph PRG-94.

[63] J. Gibbons. Upwards and downwards accumulations on trees. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science 669, pages 122–138. Springer-Verlag, 1992.

[64] J. Gibbons. An initial-algebra approach to directed acyclic graphs. In B. Möller, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 947, pages 282–303. Springer-Verlag, 1995.

[65] J. Gibbons. Polytypic downwards accumulations. In J. Jeuring, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 1422, pages 207–233. Springer-Verlag, 1998.

[66] J. Gibbons and G. Jones. The under-appreciated unfold. To appear in *International Conference on Functional Programming*, 1998. Available from `http://www.brookes.ac.uk/~p0071749/home.html`.

[67] M. Gondran and M. Minoux. *Graphs and Algorithms.* John Wiley & Sons, 1984.

[68] D. Gries. *The Science of Programming.* Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[69] D. Gries and F.B. Schneider. *A Logical Approach to Discrete Math.* Texts and Monographs in Computer Science. Springer-Verlag, 1994.

[70] D. Gries and Jinyun Xue. The Hopcroft-Tarjan planarity algorithm, presentation and improvements. Technical Report TR88-906, Department of Computer Science, Cornell University, 1988. Available from `http://www.cs.cornell.edu/`.

[71] F. Harary. *Graph Theory.* Addison-Wesley, 1969.

[72] F. Harary and R.C. Read. Is the null-graph a pointless concept? In R.A. Bari and F. Harary, editors, *Graphs and Combinatorics*, Lecture Notes in Mathematics 406, pages 37–44. Springer-Verlag, 1973.

[73] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.

[74] C.A.R. Hoare. Unified theories of programming. In M. Broy and B. Schieder, editors, *Mathematical Methods in Program Development*, NATO AS1 Series F: Computer and Systems Sciences, Volume 158, pages 313–367. Springer, 1997.

[75] C.A.R. Hoare, I.J. Hayes, He Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorenson, J.M. Spivey, and B.A. Sufrin. Laws of programming. *Communications of the ACM*, 30:672–686, 1987.

[76] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. International Series in Computer Science. Prentice Hall, 1998.

[77] J.E. Hopcroft and R.E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.

[78] P. Jansson and J. Jeuring. PolyLib – A library of polytypic functions. Chalmers University of Technology, 1998. Available from `http://www.cs.chalmers.se/~patrikj/poly/`.

[79] V. Jarník. O jistem problemu minimalnim. *Praca Moravske Prirodovedecke Sploecnosti*, 6:57-63, 1930.

[80] G. Jones and J. Gibbons. Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Computer Science Report 71, University of Auckland, 1993. Available from `http://www.brookes.ac.uk/~p0071749/home.html`.

[81] M.P. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Yale University, 1994. Available from `http://www.cs.nott.ac.uk/Department/Staff/mpj/`.

[82] A. Kaldewaij. *Programming: The Derivation of Algorithms*. International Series in Computer Science. Prentice Hall, 1990.

[83] B. von Karger and R. Berghammer. Computing kernels in directed bichromatic graphs. *Information Processing Letters*, 62(1), 1997.

[84] D.J. King and J. Launchbury. Structuring depth-first search algorithms in Haskell. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, USA, 1995.

[85] J.H. Kingston. *Algorithms and Data Structures: Design, Correctness, Analysis.* International Computer Science Series. Addison-Wesley, 2nd edition, 1997.

[86] G. Kirchhoff. Ueber die Auflösung der Gleichungen, auf welche man bei der Untersuchung der linearen Vertheilung galvanischer Ströme geführt wird. *Annalen der Physik und Chemie*, 72:497–508, 1847.

[87] S.C. Kleene. *Introduction to Metamathematics.* Bibliotheca Mathematica, volume 1. North-Holland, 1952.

[88] B. Knaster. Un théorème sur les fonctions d'ensembles. *Annales de la Société Polonaise de Mathématique*, 6:133–134, 1928.

[89] F.E.J. Kruseman Aretz. Maximal strong components: An exercise in program presentation. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra*, Texts and Monographs in Computer Science, pages 251–261. Springer-Verlag, 1990.

[90] J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7, 1956.

[91] J. Launchbury. Graph algorithms with a functional flavour. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Lecture Notes in Computer Science 925, pages 308–331. Springer-Verlag, 1995.

[92] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[93] R.C. Lyndon. The representation of relational algebras. *Annals of Mathematics*, 51:707–729, 1950.

[94] R.D. Maddux. The origin of relation algebras in the development and axiomatization of the calculus of relations. *Studia Logica*, 50(3-4):421–455, 1991.

[95] R.D. Maddux. Relation-algebraic semantics. *Theoretical Computer Science*, 160:1–85, 1996.

[96] K. Madhukar, D. Pavan Kumar, C. Pandu Ragan, and R. Sundar. Systematic design of an algorithm for biconnected components. *Science of Computer Programming*, 25:63–77, 1995.

[97] G.R. Malcolm. *Algebraic Data Types and Program Transformation.* PhD thesis, Rijksuniversiteit Groningen, 1990.

[98] G.R. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2-3):255–279, 1990.

[99] Mathematics of Program Construction Group. Fixed-point calculus. *Information Processing Letters*, 53(3):131–136, 1995.

[100] L. Meertens. Algorithmics – Towards programming as a mathematical activity. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*. Springer-Verlag, 1987.

[101] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 523, pages 124–144. Springer-Verlag, 1991.

[102] E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Lecture Notes in Computer Science 925, pages 228–266. Springer-Verlag, 1995. Available from `http://www.cs.ruu.nl/~erik/`.

[103] A. Mili, J. Desharnais, and F. Mili. *Computer Program Construction*. Oxford University Press, 1994.

[104] B. Möller. Relations as a program development language. In B. Möller, editor, *Constructing Programs from Specifications*. North-Holland, 1991.

[105] B. Möller. Derivation of graph and pointer algorithms. In B. Möller, H. Partsch, and S. Schumann, editors, *Formal Program Development*, Lecture Notes in Computer Science 755, pages 123–160. Springer-Verlag, 1992.

[106] B. Möller and M. Russling. Shorter paths to graph algorithms. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science 669, pages 250–268. Springer-Verlag, 1992.

[107] B. Möller and M. Russling. Shorter paths to graph algorithms. *Science of Computer Programming*, 22(1-2):157–180, 1994. Earlier version in [106].

[108] O. de Moor. *Categories, Relations and Dynamic Programming*. DPhil thesis, Oxford University Computing Laboratory, Programming Research Group, 1992. Available as Technical Monograph PRG-98.

[109] O. de Moor. Categories, relations and dynamic programming. *Mathematical Structures in Computing Science*, 4:33–69, 1994.

[110] O. de Moor. An exercise in polytypic program derivation: repmin, 1995. Available from `http://www.comlab.ox.ac.uk/oucl/publications/books/algebra/`.

[111] O. de Moor. A generic program for sequential decision processes. In M. Hermenegildo and D.S. Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, Lecture Notes in Computer Science 982, pages 1–23. Springer-Verlag, 1995.

[112] A. de Morgan. On the syllogism: IV, and on the logic of relations. *Transactions of the Cambridge Philosophical Society*, 10:331–358, 1864. Reprinted in [113].

[113] A. de Morgan. *On the Syllogism, and Other Logical Writings*. Yale University Press, 1966.

[114] C.C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, 1988. Reprinted in [118].

[115] C.C. Morgan. *Programming from Specifications*. International Series in Computer Science. Prentice Hall, 2nd edition, 1994.

[116] C.C. Morgan and P.H.B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990. Reprinted in [118].

[117] C.C. Morgan and K.A. Robinson. Specification statements and refinement. *IBM Journal of Research and Development*, 31(5):546–555, 1987. Reprinted in [118].

[118] C.C. Morgan and T.N. Vickers, editors. *On the Refinement Calculus*. FACIT Series in Computer Science. Springer-Verlag, 1994.

[119] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, 1987.

[120] D.A. Naumann. Beyond Fun: Order and membership in polytypic imperative programming. In J. Jeuring, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 1422, pages 286–314. Springer-Verlag, 1998.

[121] M. Ortega and O. Meza. *Grafos y Algoritmos*. Equinoccio, Ediciones de la Universidad Simón Bolívar, 1993.

[122] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

[123] C.S. Peirce. Description of a notation for the logic of relatives, resulting from an amplification of the conceptions of Boole's calculus of logic. *Memoirs of the American Academy of Sciences*, 9:317–378, 1870. Reprinted in [124].

[124] C.S. Peirce. *Collected Papers*. Harvard University Press, 1933.

[125] B.C. Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing Series. MIT Press, 1991.

[126] R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36, 1957.

[127] J.N. Ravelo. A class of graph algorithms. Qualifying dissertation for transfer to DPhil status, Oxford University Computing Laboratory, 1996. Available from `http://www.comlab.ox.ac.uk/oucl/publications/books/algebra/`.

[128] J.N. Ravelo. Calculating with relations for graph algorithmics. Presented at RelMiCS'97 –3rd International Seminar on the Use of Relational Methods in Computer Science–, Hammamet, Tunisia, 6–10 January 1997. Available from `http://www.comlab.ox.ac.uk/oucl/people/jesus.ravelo.html`.

[129] J.N. Ravelo. Two graph algorithms derived. To appear in *Acta Informatica*, 1999. Earlier version in [128].

[130] F.J. Rietman. A note on extensionality. In J. van Leeuwen, editor, *Proceedings Computer Science in the Netherlands 91*, pages 468–483, 1991.

[131] M. Russling. An algebraic treatment of graph and sorting algorithms. In *Proceedings of the 14th International SCCC Conference*, Chile, 1994.

[132] M. Russling. A general scheme for breadth-first graph traversal. In B. Möller, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 947, pages 380–398. Springer-Verlag, 1995.

[133] M. Russling. Deriving a class of layer-oriented graph algorithms. *Science of Computer Programming*, 26:117–132, 1996. Earlier version in [132].

[134] M. Russling. *Deriving General Schemes for Classes of Graph Algorithms.* PhD thesis, Universität Augsburg, 1996. Published in the series "Augsburger Mathematisch-Naturwissenschaftliche Schriften", Wißner.

[135] G. Schmidt and T. Ströhlein. Relation algebras: Concept of points and representability. *Discrete Mathematics*, 54:83–92, 1985.

[136] G. Schmidt and T. Ströhlein. *Relations and Graphs: Discrete Mathematics for Computer Scientists.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1993.

[137] E. Schröder. *Vorlesungen über die Algebra der Logik (Exacte Logik)*, volume 3: "Algebra und Logik der Relative". Teubner, Leipzig, 1895.

[138] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 7(1), 1981.

[139] M. Sharir. Some observations concerning formal differentiation of set theoretic expressions. *ACM Transactions on Programming Languages and Systems*, 4(2):196–225, 1982.

[140] J.M. Spivey. A categorical approach to the theory of lists. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 375, pages 399–408. Springer-Verlag, 1989.

[141] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[142] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89, 1941.

[143] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[144] S. Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, 1996.

[145] K. Thulasiraman and M.N.S. Swamy. *Graphs: Theory and Algorithms.* John Wiley and Sons, 1992.

[146] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[147] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

Appendix A
Two Proofs for Chapter 4

This appendix presents the proofs of two statements that were left unproved in Chapter 4. These statements are, first, the key Thinning the Closure rule and, second, tbe extra invariant of the general algorithmic solution to the representatives problem. Each proof is preseuted in a section of this appendix.

A.1 The Thinning the Closure Rule

This section presents the proof of the Thinning the Closure rule (4.18). For the sake of convenience, we repeat and re-label the rule here:

$$\left.\begin{array}{lll} g(B \cup D,\, C - Q \cdot D) & \trianglelefteq & g(B,C) - Q \cdot D \\ & & \text{provided } C \cdot D^{\circ} \subseteq R\ , \\ & & \quad S \cdot D - Q \cdot B \subseteq C\ . \end{array}\right\} \quad \text{(A.1)}$$

Some Lemmas To prove (A.1), we will make use of some properties of function g, which we now present as lemmas.

First, we state the monotonicity properties of g. Take the definition of g (4.11), and note that both union and composition are monotonic while subtraction is antimonotonic on its second argument. Combining this with monotonicity of the least fixed-point operator μ (2.56), we obtain that g is antimonotonic on its first argument and monotonic on its second.

The second lemma reads as follows:

$$\left.\begin{array}{lll} g(B,C) & = & g(B,C) - Q \cdot D \quad \text{provided } C \cap Q \cdot D = \emptyset\ , \\ & & \qquad\qquad\qquad\quad D \subseteq B\ . \end{array}\right\} \quad \text{(A.2)}$$

To prove it, we choose to start manipulating the right-hand side –since it is more complex that the left-hand side and it thus offers more opportunities for "simplification"– and argue as follows:

$$
\begin{array}{ll}
 & g\,(B,C) - Q \cdot D \\
= & \quad \{\ \text{definition of } g\ (4.11)\text{, fixed-point computation } (2.54)\ \} \\
 & (C \ \cup\ (S \cdot g\,(B,C) \ -\ Q \cdot B)) \ -\ Q \cdot D \\
= & \quad \{\ \text{distribution of subtraction over union } (2.12)\ \} \\
 & (C - Q \cdot D) \ \cup\ (S \cdot g\,(B,C) \ -\ Q \cdot B \ -\ Q \cdot D) \\
= & \quad \left\{ \begin{array}{l} \text{subtraction/union } (2.13) \text{ and} \\ \text{distribution of composition over union } (2.14) \end{array} \right\} \\
 & (C - Q \cdot D) \ \cup\ (S \cdot g\,(B,C) \ -\ Q \cdot (B \cup D)) \\
= & \quad \left\{ \begin{array}{l} \text{by properties of the lattice structure:} \\ \quad \text{first proviso is equivalent to } C - Q \cdot D = C\ , \\ \quad \text{second proviso is equivalent to } B \cup D = B \end{array} \right\} \\
 & C \ \cup\ (S \cdot g\,(B,C) \ -\ Q \cdot B) \\
= & \quad \{\ \text{definition of } g\ (4.11)\text{, fixed-point computation } (2.54)\ \} \\
 & g\,(B,C)\ .
\end{array}
$$

Our third and final lemma corresponds to the following two inequalities:

$$g(B,C) \ \supseteq\ C\ , \tag{A.3}$$

$$g(B,C) \ \supseteq\ S \cdot g\,(B,C) \ -\ Q \cdot B\ . \tag{A.4}$$

Both of them are shown to hold by applying fixed-point computation (2.54) to the left-hand side via the definition of g (4.11), and then reducing the union thus obtained to just one of its operands.

The Main Proof Let us now deal with the big task: the proof of (A.1). By definition of the thinning relation $\trianglelefteq$ (4.7), we need to prove:

$$g\,(B \cup D\,,\, C - Q \cdot D) \ \subseteq\ g\,(B,C) - Q \cdot D\ , \tag{A.5}$$

$$g\,(B,C) - Q \cdot D \ \subseteq\ (Q \cap R) \cdot g\,(B \cup D\,,\, C - Q \cdot D)\ . \tag{A.6}$$

The first demonstrandum (A.5) is easily shown in only two steps:

$$
\begin{array}{ll}
 & g\,(B \cup D\,,\, C - Q \cdot D) \\
= & \quad \left\{ \begin{array}{l} \text{(A.2) with } B,C,D := B \cup D\,,\, C - Q \cdot D\,,\, D\,; \\ \text{the provisos hold trivially} \end{array} \right\} \\
 & g\,(B \cup D\,,\, C - Q \cdot D) \ -\ Q \cdot D \\
\subseteq & \quad \{\ \text{monotonicity properties of } g\ \}
\end{array}
$$

The second demonstrandum (A.6) will take much longer. To prove it, immediate intuition suggests that we move $Q \cdot D$ to the right-hand side by universal property of subtraction (2.7), and that we then apply fixed-point induction (2.55) on $g(B,C)$. Unfortunately, by doing so we would arrive at a false statement. In order to achieve a safe application of fixed-point induction, we will pull out a "rabbit" based on the following property of subtraction:

$$V - U \;=\; V - (V' \cap U) \qquad \text{provided} \quad V \subseteq V' \;. \tag{A.7}$$

Using this property to transform the left-hand side of (A.6), with a suitably chosen V', will allow us to proceed as intuition hinted initially. We argue thus:

$$\begin{array}{ll}
 & (A.6) \\
\equiv & \quad \{ \text{ introduce } W1, W2 := g(B,C), g(B \cup D, C - Q \cdot D) \; \} \\
 & W1 - Q \cdot D \;\subseteq\; (Q \cap R) \cdot W2 \\
\equiv & \quad \left\{ \begin{array}{l} \text{–apply "rabbit" on the left-hand side–} \\ \text{(A.7) with } V, V', U := W1 \,,\, (Q \cap R) \cdot W1 \,,\, Q \cdot D \,; \\ \text{the proviso holds on account of reflexivity of } Q \cap R \end{array} \right\} \\
 & W1 \;-\; ((Q \cap R) \cdot W1 \;\cap\; Q \cdot D) \;\subseteq\; (Q \cap R) \cdot W2 \\
\equiv & \quad \{ \text{ universal property of subtraction (2.7) } \} \\
 & W1 \;\subseteq\; (Q \cap R) \cdot W2 \;\cup\; ((Q \cap R) \cdot W1 \;\cap\; Q \cdot D) \\
\equiv & \quad \left\{ \begin{array}{l} \text{introduce} \\ \quad V1, V2 := (Q \cap R) \cdot W1 \;\cap\; Q \cdot D \,,\, (Q \cap R) \cdot W2 \,; \\ \text{commutativity of union} \end{array} \right\} \\
 & W1 \;\subseteq\; V1 \cup V2 \\
\Leftarrow & \quad \left\{ \begin{array}{l} \text{definitions of } W1 \text{ and } g \text{ (4.11)} \,, \\ \text{fixed-point induction (2.55)} \end{array} \right\} \\
 & C \;\cup\; (S \cdot (V1 \cup V2) - Q \cdot B) \;\subseteq\; V1 \cup V2 \\
\equiv & \quad \left\{ \begin{array}{l} \text{distribution of composition over union (2.14)} \\ \text{and of subtraction over union (2.12)} \end{array} \right\} \\
 & C \;\cup\; (S \cdot V1 - Q \cdot B) \;\cup\; (S \cdot V2 - Q \cdot B) \;\subseteq\; V1 \cup V2 \\
\equiv & \quad \{ \text{ universal property of union (2.3) } \} \\
 & C \;\subseteq\; V1 \cup V2 \qquad\qquad (A.8) \\
 & \quad \wedge \quad S \cdot V1 - Q \cdot B \;\subseteq\; V1 \cup V2 \qquad\qquad (A.9) \\
 & \quad \wedge \quad S \cdot V2 - Q \cdot B \;\subseteq\; V1 \cup V2 \;. \qquad\qquad (A.10)
\end{array}$$

Hence, it suffices to show these three conjuncts to complete the proof of the

Thinning the Closure rule.

For (A.8), we argue as follows, starting with the right-hand side:

$$
\begin{array}{ll}
 & V1 \cup V2 \\
\supseteq & \quad \{ \text{ definitions of } V1 \text{ and } V2 \text{, reflexivity of } Q \cap R \ \} \\
 & (W1 \cap Q \cdot D) \cup W2 \\
\supseteq & \quad \{ \text{ apply property (A.3) of } g \text{ both to } W1 \text{ and } W2 \ \} \\
 & (C \cap Q \cdot D) \cup (C - Q \cdot D) \\
= & \quad \{ \text{ complementation } \} \\
 & C \ .
\end{array}
$$

We now proceed to show (A.9). Note that the provisos of the Thinning the Closure rule (A.1) have not been used yet. They are only needed in the –long!– calculation that proves (A.9), which now follows:

$$
\begin{array}{ll}
 & S \cdot V1 - Q \cdot B \\
= & \quad \{ \text{ definition of } V1 \ \} \\
 & S \cdot ((Q \cap R) \cdot W1 \ \cap \ Q \cdot D) - Q \cdot B \\
\subseteq & \quad \left\{ \begin{array}{l} \text{Dedekind's rule (2.20) with} \\ \quad R, S, T := Q \text{ , } D \text{ , } (Q \cap R) \cdot W1 \end{array} \right\} \\
 & S \cdot ((Q \cap R) \cdot W1 \cdot D^{\circ} \cap Q) \cdot D - Q \cdot B \\
\subseteq & \quad \{ \text{ definition of } W1 \text{ , property (4.14) of } g \ \} \\
 & S \cdot ((Q \cap R) \cdot S^{*} \cdot C \cdot D^{\circ} \cap Q) \cdot D - Q \cdot B \\
\subseteq & \quad \{ \text{ first proviso! } \} \\
 & S \cdot ((Q \cap R) \cdot S^{*} \cdot R \cap Q) \cdot D - Q \cdot B \\
\subseteq & \quad \left\{ \begin{array}{l} \text{since } R \text{ is a preorder, requirement (4.16) on } R \text{ and } S \\ \text{entails } S^{*} \subseteq R \text{ by universal property of closure (2.62)} \end{array} \right\} \\
 & S \cdot ((Q \cap R) \cdot R \cdot R \cap Q) \cdot D - Q \cdot B \\
\subseteq & \quad \left\{ \begin{array}{l} \text{by intersection and transitivity of } R \\ \text{we have that } (Q \cap R) \cdot R \cdot R \subseteq R \end{array} \right\} \\
 & S \cdot (R \cap Q) \cdot D - Q \cdot B \\
\subseteq & \quad \{ \text{ requirement (4.17) on } Q \text{ , } R \text{ and } S \ \} \\
 & (Q \cap R) \cdot S \cdot D - Q \cdot B \\
\subseteq & \quad \left\{ \begin{array}{l} \text{by intersection and transitivity of } Q \text{ we} \\ \text{have that } (Q \cap R) \cdot Q \subseteq Q \text{, subtraction} \\ \text{antimonotonic on its second argument} \end{array} \right\} \\
 & (Q \cap R) \cdot S \cdot D - (Q \cap R) \cdot Q \cdot B \\
\subseteq & \quad \{ \text{ property of subtraction: } W \cdot V - W \cdot U \subseteq W \cdot (V - U) \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & (Q \cap R) \cdot (S \cdot D - Q \cdot B) \\
\subseteq & \quad \{ \text{ second proviso! } \} \\
 & (Q \cap R) \cdot C \\
\subseteq & \quad \left\{ \begin{array}{l} \text{(A.8) proved above,} \\ \text{distribution of composition over union (2.14)} \end{array} \right\} \\
 & (Q \cap R) \cdot V1 \ \cup \ (Q \cap R) \cdot V2 \\
= & \quad \{ \text{ definitions of } V1 \text{ and } V2 \ \} \\
 & (Q \cap R) \cdot ((Q \cap R) \cdot W1 \ \cap \ Q \cdot D) \ \cup \ (Q \cap R) \cdot (Q \cap R) \cdot W2 \\
\subseteq & \quad \left\{ \begin{array}{l} \text{distribution of composition over intersection (2.15),} \\ \text{transitivity of } Q \cap R \end{array} \right\} \\
 & ((Q \cap R) \cdot W1 \ \cap \ (Q \cap R) \cdot Q \cdot D) \ \cup \ (Q \cap R) \cdot W2 \\
\subseteq & \quad \left\{ \begin{array}{l} \text{by intersection and transitivity of } Q \\ \text{we have that } (Q \cap R) \cdot Q \subseteq Q \end{array} \right\} \\
 & ((Q \cap R) \cdot W1 \ \cap \ Q \cdot D) \ \cup \ (Q \cap R) \cdot W2 \\
= & \quad \{ \text{ definitions of } V1 \text{ and } V2 \ \} \\
 & V1 \cup V2 \ .
\end{array}
$$

We finalise the whole proof by showing (A.10):

$$
\begin{array}{ll}
 & S \cdot V2 - Q \cdot B \\
= & \quad \{ \text{ definition of } V2 \ \} \\
 & S \cdot (Q \cap R) \cdot W2 \ - \ Q \cdot B \\
\subseteq & \quad \{ \text{ requirement (4.17) on } Q, \ R \text{ and } S \ \} \\
 & (Q \cap R) \cdot S \cdot W2 \ - \ Q \cdot B \\
\subseteq & \quad \left\{ \begin{array}{l} \text{by intersection and transitivity of } Q \text{ we} \\ \text{have that } (Q \cap R) \cdot Q \subseteq Q \text{, subtraction} \\ \text{antimonotonic on its second argument} \end{array} \right\} \\
 & (Q \cap R) \cdot S \cdot W2 \ - \ (Q \cap R) \cdot Q \cdot B \\
\subseteq & \quad \{ \text{ property of subtraction: } W \cdot V - W \cdot U \subseteq W \cdot (V - U) \ \} \\
 & (Q \cap R) \cdot (S \cdot W2 - Q \cdot B) \\
= & \quad \left\{ \begin{array}{l} \text{complementation: } W \ = \ (W \cap V) \cup (W - V) \\ \qquad\qquad \text{with } W, V := S \cdot W2 - Q \cdot B \text{ , } Q \cdot D; \\ \text{distribution of composition over union (2.14)} \\ \text{–this helps to reach the goal } V1 \cup V2 \text{ in two halves–} \end{array} \right\} \\
 & (Q \cap R) \cdot ((S \cdot W2 - Q \cdot B) \cap Q \cdot D) \\
 & \quad \cup \ (Q \cap R) \cdot (S \cdot W2 - Q \cdot B - Q \cdot D) \\
= & \quad \{ \text{ introduce names } U1 \text{ and } U2 \text{ for the two halves } \} \\
 & U1 \cup U2 \ .
\end{array}
$$

Hence, it suffices to show that $U1 \subseteq V1$ and $U2 \subseteq V2$ hold. The key to proving both inclusions is property (A.4) of g. The first half is proved as follows:

$$
\begin{array}{ll}
 & U1 \\
\subseteq & \left\{ \begin{array}{l} \text{definition of } U1\,, \\ \text{monotonicity properties of } g \text{ entail } W2 \subseteq W1 \\ \text{–transforming } W2 \text{ into } W1 \text{ will take us to } V1 - \end{array} \right\} \\
 & (Q \cap R) \cdot ((S \cdot W1 - Q \cdot B) \cap Q \cdot D) \\
\subseteq & \quad \{ \text{ definition of } W1\,, \text{(A.4)} \} \\
 & (Q \cap R) \cdot (W1 \cap Q \cdot D) \\
\subseteq & \quad \{ \text{ distribution of composition over intersection (2.15)} \} \\
 & (Q \cap R) \cdot W1 \cap (Q \cap R) \cdot Q \cdot D \\
\subseteq & \left\{ \begin{array}{l} \text{by intersection and transitivity of } Q \text{ we have} \\ \text{that } (Q \cap R) \cdot Q \subseteq Q\,, \text{ definition of } V1 \end{array} \right\} \\
 & V1 \ .
\end{array}
$$

For the second half we argue:

$$
\begin{array}{ll}
 & U2 \\
= & \left\{ \begin{array}{l} \text{definition of } U2\,, \text{ subtraction/union (2.13) and} \\ \text{distribution of composition over union (2.14)} \end{array} \right\} \\
 & (Q \cap R) \cdot (S \cdot W2 \ - Q \cdot (B \cup D)) \\
\subseteq & \quad \{ \text{ definition of } W2\,, \text{(A.4)} \} \\
 & (Q \cap R) \cdot W2 \\
= & \quad \{ \text{ definition of } V2 \ \} \\
 & V2 \ .
\end{array}
$$

Done!

A.2 The Extra Invariant

When deriving the general algorithmic solution in Figure 4.20, an invariant with two conjuncts was proposed and used to guide the construction of the iteration. However, towards the end of the development, in page 75, a third extra invariant was made use of:

$$Inv3 \quad := \quad B \cdot C^{\circ} \cap Q = \emptyset \ .$$

We now show that $Inv3$ is indeed an invariant of the developed iteration.

Initial Establishment The initialisation statement $B, C := \emptyset, A$ establishes *Inv3* due to $\emptyset$ being a zero of both composition and intersection.

Maintenance Regarding the iteration body, since its first statement affects only variable D, it suffices to show that *Inv3* is maintained by the assignment:

$$B, C := B \cup D,\ (C \cup (S \cdot D - Q \cdot B)) - Q \cdot D \ . \tag{A.11}$$

Before analysing the effect of this assignment on *Inv3*, we take two preliminary steps.

First, we present a different phrasing of *Inv3* which will be more convenient to manipulate:

$$C \cdot B^\circ \cap Q = \emptyset \ . \tag{A.12}$$

This is equivalent to *Inv3* by properties of converse and symmetry of Q. Second, we transform the expression assigned to variable C in (A.11) as follows:

$$\begin{array}{ll} & (C \cup (S \cdot D - Q \cdot B)) - Q \cdot D \\ = & \quad \{ \text{ distribution of subtraction over union (2.12) } \} \\ & (C - Q \cdot D) \cup (S \cdot D - Q \cdot B - Q \cdot D) \\ = & \quad \left\{ \begin{array}{l} \text{subtraction/union (2.13) and} \\ \text{distribution of composition over union (2.14)} \end{array} \right\} \\ & (C - Q \cdot D) \cup (S \cdot D - Q \cdot (B \cup D)) \\ = & \quad \{ \text{ introduce names } \mathit{W1} \text{ and } \mathit{W2} \ \} \\ & \mathit{W1} \cup \mathit{W2} \ . \end{array}$$

Finally, we prove that assignment (A.11) maintains *Inv3* by showing it maintains (A.12). Assume (A.12) holds and then argue thus:

$$\begin{array}{ll} & \text{Left-hand side of (A.12) } [\, B, C := B \cup D, \mathit{W1} \cup \mathit{W2} \,] \\ = & \quad \{ \text{ substitution } \} \\ & (\mathit{W1} \cup \mathit{W2}) \cdot (B \cup D)^\circ \cap Q \\ = & \quad \left\{ \begin{array}{l} \text{distribution of composition, converse} \\ \text{and intersection over union} \end{array} \right\} \\ & (\mathit{W1} \cdot B^\circ \cap Q) \ \cup \ (\mathit{W1} \cdot D^\circ \cap Q) \ \cup \ (\mathit{W2} \cdot (B \cup D)^\circ \cap Q) \\ = & \quad \left\{ \begin{array}{l} \text{first operand is empty due to the following:} \\ \text{by subtraction we have } \mathit{W1} \subseteq C \text{, (A.12)} \end{array} \right\} \\ & (\mathit{W1} \cdot D^\circ \cap Q) \ \cup \ (\mathit{W2} \cdot (B \cup D)^\circ \cap Q) \end{array}$$

$$
\begin{array}{ll}
\subseteq & \left\{ \begin{array}{l} \text{by subtraction we have} \\ \quad W1 \subseteq \overline{Q \cdot D} \ \text{ and } \ W2 \subseteq \overline{Q \cdot (B \cup D)} \end{array} \right\} \\
& (\overline{Q \cdot D} \cdot D^{\circ} \cap Q) \ \cup \ (\overline{Q \cdot (B \cup D)} \cdot (B \cup D)^{\circ} \cap Q) \\
\subseteq & \quad \{ \text{ Dedekind's rule (2.20) twice } \} \\
& (\overline{Q \cdot D} \ \cap \ Q \cdot D) \cdot D^{\circ} \\
& \quad \cup \ (\overline{Q \cdot (B \cup D)} \ \cap \ Q \cdot (B \cup D)) \cdot (B \cup D)^{\circ} \\
= & \quad \{ \text{ complementation } \} \\
& \emptyset \ .
\end{array}
$$

Done!