The Kleene Algebra of Nested Pointer Structures: Theory and Applications

Dissertation
zur Erlangung des Doktorgrades der Naturwissenschaften
der Fakultät für Angewandte Informatik
der Universität Augsburg

vorgelegt von

Dipl.-Inf. Thorsten Ehm

Augsburg

Oktober 2003

1. Gutachter: Prof. Dr. Bernhard Möller
2. Gutachter: Prof. Dr. Rudolf Berghammer

Tag der mündlichen Prüfung: 16. Dezember 2003

Zusammenfassung

Softwaregesteuerte Systeme finden mehr und mehr Eingang in unser tägliches Leben. Damit steigt auch entscheidend die Wahrscheinlichkeit, auf Grund von schlampig programmiertem Code mit Systemabstürzen, Ausfällen und fehlerhaftem Verhalten konfrontiert zu werden. Während dies bei Produkten aus der Unterhaltungselektronik nur ärgerlich sein mag, kann es bei Kontrollsystemen für den Verkehr und für Kernkraftwerke oder bei medizinischen Geräten lebensgefährlich sein. Anwendungen aus diesen Gebieten verlangen nach einem formalen Software-Entwicklungsprozess zur Gewährleistung der Korrektheit.

Obwohl es einige Methoden zur Erlangung dieses Zieles gibt, haben sich die Verifikation und die Entwicklung korrekter Zeigeralgorithmen einer allgemeinen, formalen Behandlung weitgehend widersetzt.

In dieser Arbeit werden diese Unzulänglichkeiten in zweierlei Hinsicht behandelt. Zuerst wird ein abstrakter Kalkül zur Behandlung von markierten Graphen und Zeigerstrukturen vorgestellt. Dieses System basiert auf Kleene-Algebra, welche trotz ihres einfachen Aufbaus erfolgreich auf eine Vielzahl unterschiedlicher Problemstellungen angewendet werden konnte. Die Einfachheit und Prägnanz wird direkt von der hier definierten Zeiger-Kleene-Algebra geerbt. Diese ermöglicht eine kompakte Darstellung, ohne den Zugriff auf interne Strukturen zu verhindern. Auf einer höheren Ebene werden Operatoren zur Beschreibung von Erreichbarkeit, Speicherreservierung, Selektion und Projektion eingeführt. Damit werden Eigenschaften zur lokalen Einschränkung von Abänderungen auf bestimmte Teile des Speichers bewiesen.

In einem zweiten Teil werden Anwendungen der Zeigeralgebra in der Softwareentwicklung von Algorithmen vorgestellt. Die Algebra wird als formale Grundlage für ein Transformationssystem zur Herleitung korrekter Zeigeralgorithmen aus funktionalen Spezifikationen benutzt. Um den vollständigen Bereich von der Spezifikation bis zur Implementierung abzudecken wurde diese Methode um ein allgemeines Transformationsschema zur Erzeugung effizienter imperativer Algorithmen erweitert. In einer weiteren Anwendung wird gezeigt, daß die Zeigeralgebra auch als algebraisches Modell für ein auf dem Hoare-Kalkül basierendes Verifikationssystem für Algorithmen auf verzeigerten Datenstrukturen dienen kann.

Abstract

Software controlled systems more and more become established in our daily life. Thus, the probability to be confronted with system crashes, breakdowns or erroneous behaviour due to slovenly programmed code is increased considerably. While this may only be annoying for electronic entertainment products it could be dangerous to life in traffic and nuclear power plant control systems or medical tools. Applications from all these areas require a formal software development process to assure correctness.

Although there are several methods to achieve this goal in general, verification and development of correct pointer algorithms, which are most susceptible to errors, have to a large extent defied a general formal treatment.

In this thesis this insufficiency is dealt with in two ways. First, an abstract calculus for the treatment of labeled graphs and pointer structures is presented. The framework is based on Kleene algebra, which despite its simple structure has been successfully applied to a variety of different problems. Simplicity and succinctness is inherited directly by the pointer Kleene algebra defined here. It enables a compact representation without preventing access to the internal structure. We introduce higher-level operators to describe reachability constraints, allocation, selection and projection. Localization properties that allow restricting the effects of modifications to particular parts of the memory are proved.

A second part presents applications of pointer Kleene algebra to the software development process. The algebra is used as formal basis for a transformation system to derive correct pointer algorithms from functional specifications. To cover the whole scope from specification to implementation this method is extended by a general transformation scheme to create efficient imperative algorithms. As a further application it is shown that pointer Kleene algebra can also serve as an algebraic model behind a Hoare-style verification system for algorithms on linked data structures.

Contents

Chapter 1

Introduction

This thesis investigates the formal treatment of pointer structures and pointer algorithms. We show how to calculate properties of pointer-linked data structures and how to derive pointer manipulating algorithms from formal specifications. This is a step towards correct implementations of programs using dynamically allocated memory blocks. In practice, such programs cause serious problems in software and are resistant even to contemporary testing and debugging methods. The thesis gives a comprehensive view - theoretical concepts, specification, transformations - of a formal development process for pointer algorithms. We show that Kleene algebra is an appropriate mathematical foundation in the form of an algebraic framework. The main focus is the development of a suitable theory. Based on previous work by B. Möller the presented algebra is applied as the basis for a technique using functional specifications for algorithms on inductively defined data types. Since functional specifications are built on recursive concepts, the method yields recursive algorithms. In general, these waste a lot of memory on the stack for data of pending method calls. Therefore, in a final step we show transformation and optimization concepts to get efficient imperative algorithms.

1.1 Correct Software Development

In the history of software development intuitive programming has been replaced step by step by structured methodical approaches. In the early days of computing such improvements were the introduction of mnemonic codes for binary machine languages, symbolic assembly languages, macros for repeated tasks, high-level programming languages and so on. Nevertheless, it was early realized that these techniques do not suffice to create huge correct systems. Software development has remained more a craft than a science. This phenomenon was called the *software crisis* and caused that during the last decades a lot of effort was spent to solve problems arising from the immense size of contemporary software systems. Some of them, like usage of *design patterns* [GHJV94] and *refactoring* [Fow99], are only of informational character. Based on many years of experience they help to avoid well-known errors and serve as proposals how to implement common programming or restructuring tasks. Other approaches like the *unified modeling language* (UML) [RJB98] are intended as abstractions to allow talking about the design and managing the high complexity of software systems. These methods support the requirements engineering process to come from an imprecise informal description to a formal problem specification. Recently, *model driven architectures* (MDA) [KWBW03] have been introduced to make the particular models independent from a specific target language. All these approaches are considered to solve problems in the large. But the real difficulties arise in the refinement step from an abstract model to the concrete implementation. Even with code generating tools the programmer has to provide code pieces for the automatically created frame. These are the substantial parts that have to fulfill all the conditions and requirements the specification demands. In this step the programmer is left quite alone to produce high-quality software.

The quality of software deals with several notions like correctness, reliability, robustness, efficiency, portability, adaptability, maintainability, modularity and reusability [IEE83]. The main concern of this thesis is correctness, since incorrect software, even if it shows all the other properties, is useless. Intuitively, correctness is the extent to which a product meets the intended purpose. So the notion *correctness* is not restricted to software but can also be applied to custom products. The terminology to describe the particular method to determine correctness of a product slightly differs between quality management and software engineering. *Validation* of a product traditionally means to assure that the manufacturing process fulfills certain criteria so that it is most likely to yield high-quality products whereas *verification* describes the testing of the produced goods. This mostly is only applied to a small sample, since testing of real-world products often can only be achieved destructively, as for example in car crash-tests or examination of adhesion of two glued pieces. Failing tests then lead to investigations and improvements of the product design or the production process. By this quality management correctness continuously is improved by such process audits. It is sort of curious that not absence but the presence of errors helps to increase correctness. Since tests can only give a clue to correctness, in software engineering this weaker concept is called *validation*. *Verification* of software is a little bit different. It describes the formal process of proving correctness of an algorithm with mathematical methods. This assures that the implementation satisfies the properties fixed in the formal specification. Nevertheless, there remain the problems that the formal specification may not meet the intended behaviour, an inadequate model is used, or the hardware on which the program is executed is not correct either.

Verification can be avoided if the algorithm is correct by construction. Such an approach is called *transformational program construction* which was quite popular in the eighties [BBB$^+$85, BEH$^+$87] but went a little bit out of fashion in the meantime. There starting from the specification an implementation is derived by semantics-preserving manipulations. The derivation process is divided up into a number of small but manageable transformation steps. Complex and repeatedly arising tasks can be encapsulated and reused by general transformation rules. The design decisions are reflected in the derivation process by the choice of particular transformation rules. Applications of both methods, transformation and verification, are shown in Chapter 5.

Due to additional financial expenditure formal software development is rarely used in contemporary software engineering in industry. If someone pretends to use formal methods this means in the best case that there is a semi-formal specification and a test suite. It is accepted opinion that for often reused code in a standard programming framework or in safety critical areas it is sometimes mandatory to develop correct software by construction and not by testing. Since in the meantime a huge amount of time and resources is spent for testing and debugging of implementation errors, it could be advantageous to replace these parts by a formal development process even without big additional expenditure. But it is also clear that such a process will never be applied to "throw-away" software.

1.2 Pointer Algorithms

Pointers are the key to efficiency of many imperative algorithms. But experience shows that programs manipulating pointer data structures are prominent candidates for errors. Everybody working with a computer has had problems originating from dereferencing nil pointers and links referring to protected parts or non-allocated cells in memory. A reason for the omnipresence of pointer based mistakes is the potentially high complexity of dynamically created data. Even medium-sized applications form complicated net structures on the heap. Manipulation of these structures by allocation, deallocation or pointer modifications and the corresponding consequences add an additional dynamic factor. The symptoms of such errors are often observable only after a long system operation time, under high load or in a different environment and so refrain from being detected by standard test scenarios.

One of the most dangerous methods to work with dynamically allocated heap data is explicit pointer arithmetic as known from C [KR88]. It is evident, that the intention of the language designers was to increase performance with these constructs at machine code level. But as a consequence, the programmer needs exact knowledge of data representation on the heap as well as detailed assumptions about system internals as for example the width of processor registers. This is not a very abstract programming discipline and in the development of correct software such optimizations should be left to the (hopefully verified) compiler. Additionally, the arbitrary access to almost any memory location is not only hard to manage from a correctness point of view but also a security problem. In the sequel we will start from the assumption that pointer arithmetic is not an acceptable choice for implementing correct pointer algorithms. Nevertheless, there remain a lot of other potential sources of difficulties.

One problem in explicit storage management is the deallocation of cells. Complicated execution patterns of code often make it difficult to correctly determine the lifetime of objects. Too optimistic estimations then result in too early deallocation of memory. If a cell in memory is deallocated while another reference still points to this part of the store, there is no possibility to know in advance what will happen to the freed cell. As the affected cell can be reused by the memory manager to store other data, the dynamic behaviour is unpredictable. It is quite likely to get such effects called *dangling links* in systems with manually performed disposal of seemingly no longer needed cells. A comparable problem that arises by manually controlling the memory management of a system are *space leaks*. They show up if the last reference to a heap structure is destroyed before the structure itself is deallocated. Then there remains allocated but inaccessible data that cannot be freed as there is no handle to access the affected part of the store. If this happens once in a while the size of usable memory may decrease significantly. At first sight this seems not to be a problem concerning the correctness of a program. Nevertheless, a system may get completely out of step due to unexpectedly running out of memory. In the best case the system stops with an error message. In the worst case it resumes execution and unrecognizedly produces wrong output. A similar problem may arise if a program interchangeably allocates and frees small and large memory blocks. Although one would expect that the total amount of memory used is constant, the memory manager cannot fit large blocks into freed spaces that are too small in size. The *fragmentation* of the store is too high to reuse such parts and the memory manager has to allocate fresh cells. Thus, theoretically there is enough memory for allocating new data but in practice one will not find a large enough continuous part. The reason for fragmentation often is based on wasteful dealing with memory since programmers often are afraid of reusing memory cells due to complicated side-conditions and prefer to allocate new ones.

A solution to all these problems referring to deallocation would be to free no cells at all. But even with contemporary memory sizes, a program managing large data structures will rapidly reach the limit of the store. This memory overuse is called *hogging*. In contrast to memory leaks there may be remaining pointers to this part of the store that are not freed due to erroneous programs. Indeed not only large data fills up the memory. Most of the allocated cells' lifetime is only short-termed and so a lot of small-sized garbage arises. As mentioned before, the reuse of such cells often is refrained from, since this is complicated task and tends to be erroneous.

A remedy for these problems would be the automatic determination of the moment from which on an object is not used any longer. This can be pre-calculated by the compiler or decided during runtime. Unfortunately, static analysis techniques to perform this task never reached more than a theoretical status. The better automatic approach is to leave deallocation during runtime completely to the memory manager. The system's so called *garbage collector* decides which objects on the heap are still alive and frees the non-used ones. Most contemporary functional and object-oriented programming environments use garbage collectors to liberate the programmer from memory management issues.

Garbage collection will solve manual deallocation problems, but apart from this, most problems with pointer linked data structures arise from *sharing*. This multiple use of common parts in the storage mostly is used to save memory or to have alternative access possibilities. In the presence of sharing, changes of a pointer can damage invariants of arbitrary pointer variables referring to the same data on the heap. Therefore, sharing is also the main issue that makes reasoning about properties of pointer structures that complex and difficult. As we will see later, several operations can be simplified if the absence of sharing between two parts of the memory can be shown.

The main problem that makes syntactical reasoning about heavily inter-related objects so more difficult than reasoning about simple data types is pointer *aliasing*. This means that distinct expressions denote the same l-value and therefore the same memory cell in the store. As a consequence, changing one of them may alter an at first sight completely unrelated variable or object. Although in theoretical considerations this is often hidden by concurrent assignments, it is the rule and mostly unavoidable in pointer algorithms. Otherwise handles for memory data will get lost by simple assignments.

The best argument that it can be quite difficult for a programmer to understand all the implications of dynamically allocated data is the existence of a large number of debugging and profiling tools. These will not solve the implementation problems but help to find errors by testing. There are two sorts of tools. The first are visualization instruments like *Inuse*, a plug-in to *Insure* [Par02] that graphically displays and animates memory allocations. Such tools only help to find very wasteful use of memory. Others like *Valgrind* [Sew02] or *Electric Fence* [Per99] are dynamically linked to the programs and simulate the instructions concerned with memory allocation/deallocation or even emulate the whole CPU. They observe the allocated storage and discover errors like memory leaks, use of uninitialized memory and the under- or overrun of allocated buffers. These are reported together with the appearance in the code.

Despite all these pointer-specific difficulties there remain the same problems as for algorithms that only use simple data types. So, often the provided implementation does not or only partly match the specification or data structure invariants are broken. To avoid these problems and to be able to manage the pointer-specific difficulties a formal algebraic treatment of pointer structures is needed. This enables us to prove transformation and verification methods as described in Section 1.1 which yield pointer algorithms that are correct with respect to its specification.

1.3 Kleene Algebra

Iteration plays an essential rôle in almost all areas of computer science. There are a lot of different mathematical structures to formally treat iteration. Most of them are based on the Kleene star operator $*$, which algebraically expresses the properties of finite iteration and is one of the basic building blocks of Kleene algebra.

The theory of Kleene algebra goes back to a technical paper by S.C. Kleene in 1951 [Kle51] which later was published in a shortened and revised form in [SM56]. Influenced by the biological question to what kind of events an organism can respond, Kleene developed the theory of regular events. His studies are based on the input/output behaviour of McCulloch-Pitts nerve nets which he generalized to a model of finite automata. Kleene proved the equivalence of such machines and regular events. The operations of the class of regular events are motivated by constructions with neuron activation tables. These can be overlaid, composed with one another and iterated. So Kleene defined the regular set of tables as the least class that includes the unit tables and is closed under sum, product and iteration. In contrast to later approaches he defined iteration as a more general binary operation $E * F$, the iterate of E on F. Kleene gave some equalities for sets of tables but posed the axiomatization of regular events as an open problem.

In contrast to other algebraic areas even nowadays there is no agreed-on definition of regular algebras. One of the reasons is that axiomatization of the star operator is quite hard. In 1964 Redko [Red64] was the first who proved, that a finite equational axiomatization does not exist. Nevertheless, a lot of different axiomatizations where proposed. Salomaa [Sal66] was the first who gave two complete axiomatizations for the family of regular sets over a finite alphabet. His rules are sound when interpreted over the algebra of regular events but have some subtle problems in other contexts. Further approaches used schemes to represent an infinite number of equations [BE93a, Kro91]. The first who treated the algebra of regular events extensively was J.H. Conway [Con71]. He examined the different axiomatizations of Kleene algebra and the connection to finite machines under a more theoretical point of view than Kleene did. Conway gave a full mathematical account of Moore's identification theory [Moo56] which is concerned with the extraction of information about the internal structure of a sequential machine. Similar to Kleene's research, Moore supposed a black-box approach and considered the response on selected input sequences as the only means to get more knowledge about the internal states. In his book Conway proposed five different notions of Kleene algebra and compared their expressiveness. More generally, Bloom and Ésik [BÉ93b] were concerned with equational properties of fixed point solutions. They defined *iteration theories* which completely axiomatize valid fixed point identities in various structures. Kozen [Koz90a] gave a succinct and more transparent axiomatization than Conway did. He defined Kleene algebra to be an idempotent semiring with star using a finite set of universally quantified equations and two Horn-clauses over equations. Kozen axiomatized $*$ as the least fixed-point of a linear equation. The universal Horn theory of Kozen's axiomatization generates the equational theory of regular algebras. In $*$-continuous Kleene algebras [Koz81] which are equal to Conway's N-algebras an infinitary summation operator $\sum$ is used to define star as the supremum of the set of all powers. This definition implies the Horn-formula axioms of Kozen's formalization. Also closed semirings [Koz90b] use such a supremum operator to define star. They are similar to Conway's S-algebras but only require suprema of countable sets. Apart from these approaches there are a lot of other application-specific or slightly modified axiomatizations.

Apart from its simpleness regular structures have proved to be a useful tool in various contexts. Hence, also appearance of Kleene algebra is widespread. Strongly related to the automata theoretical roots is the theory of formal languages [KS86].

Following Chomsky's hierarchy, type-three grammars are built from the regular operations and exactly form the class of languages that are accepted by finite automata. The equational consequences of the Kleene algebra axioms as given by Kozen are exactly the regular identitities [Con71, Koz90a]. Therefore the family of regular languages over an alphabet A forms the free Kleene algebra on free generators A.

Moreover all standard models of relation algebra [Tar41, Ng84] are examples of Kleene algebras. The star there coincides with transitive reflexive closure. Thus, Kleene algebra has also its appearance in all relationally treated areas.

A significant rôle is played by Kleene algebra in the context of reasoning about computer programs. There have been several proposals for methods to verify and specify hard- and software, most notably [Flo67, Hoa69, Dij76]. All of them are based on reasoning about states or predicates that specify sets of states and transitions between them. A basic and important unifying concept to describe such systems are labeled transition systems (LTS). They present a picture of all possible states of a system together with all possible state transitions. The vertices of an LTS represent system states whereas edges are labeled by programs. Such a program maps the state represented by the source vertex to the target's state. Program logics, the formal system to reason about LTSs and hence about computational processes, are particular modal logics. In 1977 Pnueli [Pnu77] applied temporal logic (TL) to express properties of programs for verification tasks. Different semantics called linear- and branching-time and extensions to LTL, CTL, CTL^* were proposed in the meantime. TL is an endogenous approach, since it deals with internals of one specific program. A more abstract treatment that subsumes temporal logic is dynamic logic (DL). The first of these was propositional dynamic logic (PDL) [FL79] which later was extended to second order PDL and propositional μ-calculus. Dynamic algebra (DA) [Pra90b, Pra91, Koz79], the algebraic counterpart to PDL, is like PDL itself a two-sorted system. One sort corresponds to propositions which are used to characterize states. PDL imposes a Boolean structure B on propositions. The second sort K defines program behaviour and therefore shows a regular structure allowing union, composition and non-deterministic iteration. These two sorts are connected by a projection $\diamond : K \times B \to B$ called the *enables operator* by Pratt [Pra91]. Application of $\diamond(a,p)$ which often is written $\langle a \rangle p$ is intended to denote the predicate which has to hold so that application of a brings about a state satisfying p. In DA there is no corresponding operation that relates propositions to programs. Such an operation maps a proposition to a test program that returns the input state exactly if the proposition holds on this state. These tests preserve the current state or do not succeed otherwise. An approach to capture this algebraically are Kleene algebras with tests (KAT) [Koz97]. Such an, again two-sorted, algebra consists of a KA together with a Boolean algebra. As the Boolean sort is embedded into the set of Kleene elements, the test operator vanishes. Both approaches are lopsided, since reasoning in one of the two sorts is indirect and there either is no enables or no test operator. The extension of KA with a domain operator [DMS03] eliminates these defficiencies. The propositions also are embedded and we can express modalities as well as projections. Most theorems hold also in the more abstract structure of Kleene modules [EMS03], but for many practical application they suffer from similar deficiencies as DA and KAT.

Another improvement of DAs are action algebras as proposed by Pratt [Pra90a]. They form the algebraic counterpart to action logic. An advantage of action algebras is, that in contrast to regular algebras, they are finitely based due to the existence of residuals. The crucial axiom to define the star operator equationally is the pure induction rule $(a\backslash a)^* \leq (a\backslash a)$. Kozen remarks that, unlike Kleene algebras, action algebras are not closed under the formation of matrices. This is a property often used in automata theory and the design and implementation of algorithms. As an alternative Kozen proposed an extension of action algebras by a meet operator resulting in action lattices [Koz94]. They inherit all the merits of action algebra and are closed under the formation of matrices.

Similar to dynamic algebras, there are Peirce algebras which use relation algebra and sets instead of Kleene algebra and propositions. They show both directions, a set-forming operation on relations and a relation-forming operation on sets.

For several applications the possibility to describe infinite systems is an important task. Cohen [Coh00] therefore extended the axiomatics of Kleene algebra to omega algebra. He introduced an additional ω-operator that is able to describe infinite system behaviour. Omega exponentiation is defined by a generalized greatest fixed point law. Omega algebra axiomatizes the equational theory of omega-regular languages. This theory is useful for reasoning about concurrent programs, progress and termination. Nevertheless, Cohen has problems with modelling program termination and therefore is only able to reason about partial correctness. In contrast, von Wright [vW02] uses a slightly different approach to transfer the advantages of Kleene algebra into a framework of total correctness. By dropping the axiom $a \cdot 0 = 0$, which prevents a proper treatment of nontermination, he is able to model conjunctive predicate transformer semantics in a refinement framework.

A more general appearance is made by Kleene algebra in the computation calculus developed by R. Dijkstra [Dij98]. This subsumes for example wp-calculus, linear-time temporal logic and CTL^* in one unifying abstract theory. The basis of his algebra is formed by a complete Boolean algebra together with a composition operator and is called *semi-regular (Boolean) algebra* by R. Backhouse. Dijkstra requires some more properties to model infinite iterations similar to the ω-algebra constructs by Cohen.

There are also a number of non-standard examples for Kleene algebras. One of the best-known are the so-called $(min, +)$ algebras [AHU75]; they are also called tropical semirings [Pin98] and used for design and analysis of algorithms. A main application of $(min, +)$ algebras is for example the calculation of shortest paths in an edge-labeled graph. Not that prominent are the, in some sense symmetric, $(max, +)$ algebras used for all kinds of optimization problems. These arise in control and game theory, performance evaluations of discrete event systems [Plu90] and operations research. Special Kleene algebras called quantales [Yet90] are even used for modelling measurements in quantum physics.

1.4 Overview

The contribution of this thesis consists in two main parts. First, we investigate under which conditions Kleene algebra can be used as a formal basis to reason about pointer structures. Based on these observations we introduce pointer Kleene algebra that allows an algebraic treatment of labeled graphs. The simplicity and succinctness of reasoning in Kleene algebra directly is passed on to the presented calculus. In a second part, we show applications of pointer Kleene algebra to the formal software development process of pointer algorithms. In this, we extend a method to derive correct pointer programs from a functional specification to yield efficient imperative algorithms.
The thesis is structured as follows:
Chapter 2 gives a short introduction to the basics needed from areas this thesis is not directly concerned with. The concepts and theories presented are not treated formally but are used as models or supporting framework. The chapter introduces notions from graph theory and the connection between graphs and relations. In a short overview about fuzzy theory it is shown that this connection can be lifted to labeled graphs and fuzzy relations. The formalization of pointer structures in Kleene algebra presented later is heavily influenced by these observations. A last

section gives a short introduction to functional programming constructs. In Chapter 5 these are used in form of a restricted functional language to specify algorithms on inductively defined data types.
Chapter 3 presents the first main contribution of the thesis. After defining the basic notions of Kleene algebra and its operations we investigate the conditions needed to port concepts from fuzzy relation algebra to Kleene algebra. Although it is possible to lift several concepts treated with relation algebra to the more abstract level of Kleene algebra with tests, these are too weak to serve as a model for fuzzy relations. Properties of cut-operators in relational fuzzy theory heavily depend on a bijective correspondence between scalars and ideals. It is shown that the needed connection can be established in Kleene algebra by adding the concept of subordination. The introduced extensions support an algebraic treatment of labeled graphs in Kleene algebra. In contrast to former approaches that modelled labeled graphs by a family of unlabeled graphs, the presented calculus is more compact but nevertheless allows looking into the elements by projecting to particular subgraphs.
Chapter 4 defines the structure of pointer Kleene algebra based on the investigations made in Chapter 3. We define a set of higher-level operations to characterize reachability properties of pointer structures. This enables us to localize effects of pointer modifications to particular regions of the store which makes it possible to simplify pointer expressions considerably. Then further concepts based on reachability like sharing of common parts are defined. Some candidate expressions to characterize acyclicity are presented and compared.
Chapter 5 is concerned with applications of this theory to the concrete development process of correct pointer algorithms. We present a method for transformational derivation of pointer algorithms introduced by B. Möller. There, starting from a functional specification an implementation using pointer-linked data structures is derived. It is shown that pointer Kleene algebra can be used as a formal basis for the transformation rules in this framework. Further, we extend the calculus by a general rule to get efficient imperative algorithms from the derived recursive variants. This completes the presented method to a generally applicable framework for inductively-defined data types. A further section shows how pointer Kleene algebra can serve as foundation for deriving an extension of a Hoare-style calculus for pointer manipulation programs.
In **Chapter 6** we first present related work which is concerned with an algebraic treatment of graph algorithms, derivation and verification of pointer algorithms and comparable approaches. Then we discuss the progress achieved by the approach presented in this thesis and point out starting-points for future investigations and research.
In the **Appendix** some definitions, derivations, and non-essential proofs are given in more detail.

1.5 Acknowledgements

Many people have supported me during this work. My first thank is due to my advisor Prof. Dr. Bernhard Möller, Universität Augsburg for encouraging me to this research, valuable comments, and discussions. It was always a pleasant working environment in his group.

I owe thanks to Prof. Dr. Rudolf Berghammer, Universität Kiel, for reviewing this thesis. I also want to thank Prof. Dr. Werner Kießling, Universität Augsburg for his kind support.

I am grateful to my colleagues Dr. Georg Struth, Universität Augsburg, and Dr. Michael Winter, University of the Federal Armed Forces Munich for valuable hints and discussions.

Special thanks are due to my family - especially my parents - without their kind support I would have never been able to do these studies and this thesis would never have been written.

Last but not least I would like to thank Conny for her continual encouragement and creating the necessary diversion.

Chapter 2

Basics

This chapter is intended to provide the theoretical foundation for the subsequent parts of the thesis. We define the notions used and briefly explain facts from peripheral areas. More precisely, we show the connections between graphs and relations in Section 2.1. Further, we point out that this connection can be lifted to labeled graphs and fuzzy relations. We will show that labeled graphs can be used as a model for record-based pointer structures which is one of the key observations for the axiomatization presented in Chapter 4. Since properties of algebraic operations often are hard to understand we will give concrete models based on graphs and matrices to visualize the intentions behind the definitions. Another section gives a brief introduction to concepts of functional programming that are used in Chapter 5 to specify algorithms on inductively defined data structures.

2.1 Graphs and Relations

In principle, all problems in programming can be solved using simple unstructured data types. To make programming simpler and easier and to be able to reason in problem specific domains, more complex data structures are introduced. The most general of these are dynamic pointer-linked data types. Prominent and mainly used candidates are linked lists and trees. They are allocated during runtime and form a complex net of interconnected elements on the heap. Due to this dynamic behaviour they are quite resistant against static analysis methods and often are the reason for erroneous programs.

A unifying abstract model of these structures are graphs [SS93, Jun94]. In classical graph theory a *directed graph* $G = (V, E)$ over a node set N consists of a set of vertices $V \subseteq N$ and a relation $E \subseteq N \times N$ representing the edges. If we want to emphasize the embedding of a graph into a concrete memory we interchangeably will use the notion *cell* for the nodes of the graph. Starting and end points of edges are described by domain $\mathit{dom}(E)$ and codomain $\mathit{cod}(E)$ of the edge relation. Since for the further work graphs with pending edges do not make much sense, we additionally demand $\mathit{dom}(E) \subseteq V$ and $\mathit{cod}(E) \subseteq V$. The union of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ over the same node set N is the graph $G_\cup = (V_1 \cup V_2, E_1 \cup E_2)$ with nodes and edges from both graphs joined together. Intersection is defined similarly to be the graph $G_\cap = (V_1 \cap V_2, E_1 \cap E_2)$ with only the nodes and edges remaining which are present in both graphs. A graph $G' = (V', E')$ is called a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. $G = (V, E)$ is called a *labeled graph* if there exist two functions $m_V : V \to M_V$ and $m_E : E \to M_E$ with suitable label sets M_V and M_E that associate labels to the vertices and edges. In the sequel we will mainly use edge-labeled graphs. To get a representation for labeled graphs with

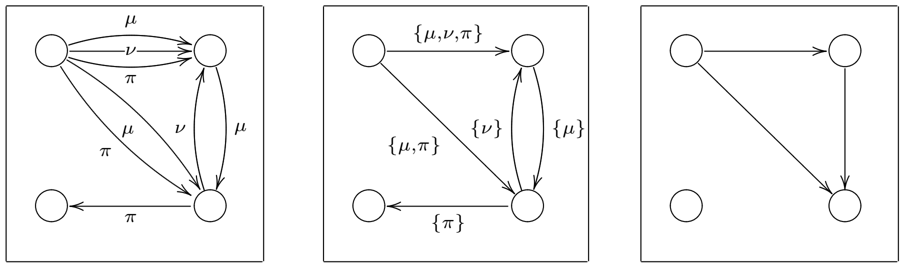

Figure 2.1: Isomorphic representation of labeled graphs and projection to the μ-subgraph

several differently labeled edges between two nodes one has to make more effort. Either it is possible to consider a family of graphs each representing edges with a particular label or one has to refrain from the relational treatment. We will take a different approach and join several distinctly labeled edges between two nodes to one set-labeled edge. Thus, we extend the labeling function to yield sets. If the empty set is interpreted as non-link this model is isomorphic to the representation by a family of graphs. The relation between these two formalizations is depicted by the two graphs on the left side in Figure 2.1. We will call the set-labeled variant *contracted* representation. Edges marked with the set of all possible labels are called *completely labeled*. The same notion is used for graphs if all existing connections are completely labeled. For abbreviation reasons we call a subgraph consisting of the same set of nodes and only edges with labels from set α an *α-subgraph*. We will formally denote the projection of graph G to its α-subgraph by $P_\alpha(G)$. An arbitrary edge-labeled graph can be split into the set of all its α-subgraphs. Obviously, this dismantling can be inverted since G can be reconstructed as the union of all its α-projections $G = \bigcup_{\alpha \subseteq M_E} P_\alpha(G)$.

A graph is said to be *finite* if the set of vertices V is finite. A *path* in a graph is a sequence of nodes where successive nodes are connected by edges. The *length* of a path is the number of connecting edges. A directed graph is called *connected* if for all pairs of nodes $m, n \in V$ there exists a path either from m to n or in the opposite direction. It is called *completely connected* if for each pair of nodes m, n a direct link from m to n exists. The unique completely labeled completely connected graph simply is called the *complete graph* over a particular node set. If the first and last node of a path coincide the path is called a *cycle*. Cycles of length one, that are edges with coinciding source and target node n, are called *loops* on n. A graph is called *cyclic* if it contains at least one cycle and *acyclic* if there is no cycle at all. Acyclic graphs with an injective edge relation are called *forests*. In a forest, the nodes without incoming links are the *roots*. Symmetrically, nodes that are only endpoints of links are called *leaves*. A *tree* is a forest with exactly one root.

As we can see from the definition of edges, every binary relation can be identified with an unlabeled graph and vice versa. Thus, the structure $REL(A) = (2^{A\times A}, \cup, \circ, \emptyset, \triangle)$ of relations over a set A can be used as a mathematical framework for algebraic calculations with graphs. $REL(A)$ is the power set of the Cartesian product $A \times A$ together with the operations of set union and relational composition defined by:

$$(x, z) \in R \circ S \overset{\text{def}}{\Leftrightarrow} \exists y.\ (x, y) \in R \wedge (y, z) \in S$$

The empty set $\emptyset$ denotes the empty relation and $\triangle$ denotes the identity relation $\triangle = \{(x, x) \mid x \in A\}$. Relational composition $\circ$ puts together two graphs G_1 and G_2 over the same set of nodes. The resulting graph connects nodes m and n exactly if there exists a node o such that there is an edge from m to o in G_1 and from o to n in G_2. We always have the two possibilities of looking at such a given set of pairs, either as binary relation or as a mapping. For the further treatment we often will use the second viewpoint of graphs mapping nodes to their successors. This is similar to the behaviour of the *enables operator* in dynamic algebra or the *Peirce product* in Peirce algebra. $REL(A)$ can be extended to a Kleene algebra by defining R^* as reflexive, transitive closure $R^* = \bigcup_{i\geq 0} R^i$ where $R^0 = \triangle$ and $R^{i+1} = R \circ R^i$. Thus, we can also use Kleene algebra to abstractly model graphs. A representation of $REL(A)$ is the structure of matrices over the Boolean semiring $(\{0,1\}, +, \cdot, 0, 1)$ with $+$ and $\cdot$ playing the rôle of disjunction and conjunction. These are A-indexed matrices with elements in $\{0,1\}$ representing associations. We will refer to this structure by the notion *standard relational model.*

Pointer based data structures are built from records consisting of several distinct fields. Fields of simple data types hold the information of a node whereas pointer data fields form the connections with the remaining records of the data structure. In this thesis we are only interested in the complex pointer structure evolving from these links and abstract our view by leaving the value fields behind. More precisely, we assume that there exist mappings from addresses to simple data types for each field name and identify value fields with their addresses. The names of the fields are used as unique selectors to access the record's components. Such a net of linked records can be represented by a labeled graph by taking records as nodes and selectors as edges labeled with the corresponding field names as shown in Figure 2.2. Object-oriented programming languages put together such data description and

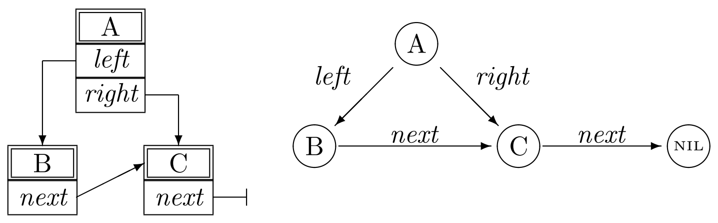

Figure 2.2: Isomorphic representation of record based pointer structure and edge-labeled graphs

methods operating on that data in classes. Objects as instances of classes contain the concrete data and can be viewed as enriched records. Therefore the data part of objects can be handled by the same framework as record based pointer structures.

Some problems require the consideration of a distinguished set of nodes. Take for example the question of reachability in a graph. This task needs a set of initial nodes from which the calculation should start. In the sequel such a tuple (m, a) of an entry point m together with a graph a is called a *pointer structure*. If appropriate we will freely extend the notion of pointer structures to sequences of entries paired with a graph representing the link structure.

2.2 Fuzzy Theory

In contrast to mathematics that is based on exact concepts and perfect notions, imprecise structures prevail in real life. The formal treatment of such unsharpness and inexactness goes back to Zadeh [Zad65]. In 1965 he introduced fuzzy set theory as a generalization of abstract set theory. Even earlier the invention of many-valued logic [ŁT30, Łuk70] made it possible to reason about uncertain or incomplete information. Nevertheless, it was Zadeh's work that influenced researchers from logics, relation algebra, measure and information theory and other areas to formally fix the treatment of such imprecise concepts. Subsequently, fuzzy set theory successfully was applied to a number of real-world applications. This not only includes engineering disciplines and computer science but also natural, life and social sciences as well as decision making in management and medicine. Nowadays, fuzzy tools even made their entrance into consumer products.

In conventional set theory, the sets considered are defined as collections of objects having some specific property. For example

$$X = \{x \mid x \text{ is a street}\}$$

Considering the subclass of long streets it is not clear how long a street has to be to be a member of this subclass. Obviously, this subclass of objects does not have a well-defined criterion of membership. It is not necessary for an object to belong or not to belong to the class. In contrast to classic sets the transition between full membership and no membership is gradual. This is the concept of fuzzy sets. Traditionally, membership in a set A over some universe $\mathcal{U}$ is determined by a characteristic function $\mu_A : \mathcal{U} \mapsto \{0, 1\}$. This function codes the truth and falsity of the predicate "$x \in A$" such that $\mu_A(x)$ returns 1 exactly if the element x is a member of A and 0 if the argument x does not belong to A. Such classic sets with a discrete membership function only returning 0 or 1 are called *crisp*. Zadeh generalized the characteristic function to return membership grades where larger values denote higher degrees of set membership. The grades of membership reflect a preorder of the objects in the universe. Although the most commonly used set of membership grades is the unit interval $[0, 1]$ which represents the percentage of belonging, any arbitrary partially ordered set can be used. It is even possible to assume a more sophisticated structure on the set of grades. In 1967 Goguen [Gog67] introduced the notion of *L-fuzzy sets* where the values of the membership function are supposed to form a lattice L. By assuming this lattice L to be Boolean one is able to abstractly describe the behaviour of set valued membership functions.

The concepts of fuzzy set theory can be ported to logic as well. Logic based on exact knowledge has a long tradition in mathematics and philosophy. Several calculi were proposed to reason formally in these logics. In propositional logics for example the basic building blocks are statements like "it is warm" and "I am sweating" which are composed by logical connectors. These propositions themselves can also describe relatively vague concepts as for example it is not mentioned explicitly what temperature or temperature interval is meant with "warm". Back in the 1920s the polish logician Łukasiewicz [ŁT30, Łuk70] developed many-valued logic to reason about such vague notions that are not simply characterizable by binary truth functions. Comparable to Zadeh, he also generalized the set of truth degrees to the real unit interval $[0, 1]$ and used minimum and maximum as truth functions to substitute conjunction and disjunction. In such a *fuzzy logic* it is possible to model values like *very true* or *fairly false* in an arbitrary gradation. This is the basis for approximate reasoning based on imprecise information as typical in natural languages. The main application of fuzzy logic are all kind of controls based on rules defined over linguistic variables. In contrast to Łukasiewicz's approach, fuzzy logic considers a whole class of different truth functions for conjunction, disjunction, implication and negation. These are defined by so called t-norms and t-conorms. So fuzzy logic can be seen as a further generalization of many-valued logic.

Since binary relations are isomorphic to sets of ordered pairs, the same generalizations introduced for classical sets can be applied to relations. Crisp relations represent the presence or absence of connections between elements of two sets. If we introduce degrees of associations we get *fuzzy relations*. This means that a fuzzy relation is identified with a fuzzy set whose support is a subset of a suitable crisp Cartesian product. There two elements are related only to a certain degree. Of course, this can be extended to n-ary fuzzy relations but will play no rôle in this thesis. The introduction of graded membership for these tuples implies degrees of associations. Fuzzy relations again can be generalized to *L-fuzzy relations* where the relation grades come from a lattice L.

Similarly, the standard model for binary relations can be extended to a matrix model $(\mathcal{P}(L)^{A\times A}, \cup, \cdot, 0, 1)$ for fuzzy relations on a finite set which we will call the *fuzzy model*. These are A-indexed matrices with subsets of L as entries. Let U, V be such matrices, then the operations and constants are defined point-wise by:

$$\begin{aligned}
(U \cup V)(x,y) &= U(x,y) \cup V(x,y) \\
(U \cdot V)(x,y) &= \bigcup_{z\in A} \{U(x,z) \cap V(z,y)\} \\
0(x,y) &= \emptyset \\
1(x,y) &= \begin{cases} L & , x = y \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

Above we argued that every crisp relation is isomorphic to a directed graph. Similarly, fuzzy relations can be visualized by labeled directed graphs where the edges are marked by the relation grades. For L-fuzzy relations, which are representations of contracted labeled graphs, the degrees of association are subsets of L. To see this connection we take the example of an atomic lattice L. Then an edge labeled with set $K \subseteq L$ can be split into several edges marked with the atoms of K. Therefore L-fuzzy relations based on an atomic distributive lattice L can be used as a formal model for labeled directed graphs.

Although there exists only a blurred notion of membership, there is a natural demand to exhibit elements that typically belong to a fuzzy set. These can be defined by introducing a threshold α that all such essential elements should exceed. The operation which maps fuzzy sets to crisp sets of elements that have a particular membership grade is called *α-cut*. The α-cut of a fuzzy set A is the level set $A_\alpha = \{x \in A \mid \mu_A(x) \geq \alpha\}$ of all elements that have membership grade at least α. The value of α serves as discrimination level to select the essential elements from the set A. Carried over to fuzzy relations, an α-cut is the relation that contains all the associations that are connected at least with a degree α. Interpreted in the graph model, the restriction $\alpha \cdot A_\alpha$ of the α-cut corresponds to an algebraic version of the projection P_α to the α-subgraph.

Similar to the representation of graphs by all its α-subgraphs it is possible to decompose fuzzy sets into their level sets through the *resolution* identity

$$A = \sum_\alpha \alpha \cdot A_\alpha$$

This is the union of all crisp sets A_α comprising the α-cuts of the fuzzy set A scaled by α. Each value α represents the minimal membership degree of elements in A_α. As before this identity similarly holds for fuzzy relations and in particular for L-fuzzy relations.

2.3 Functional Programming

Frequently, one of the most direct ways to write down a formal specification is to use a purely functional programming language. Even if there is a more intuitive non-functional formalization, it is often well-understood how to transform such a problem calculationally and derive a functional program [BdM96, Par90]. This section gives a short overview of the important aspects of functional programming used in this thesis. More detailed introductions can be found in the standard textbooks [BW89, Thi94].

The basic building blocks of functional languages are functions. They are first-class citizens, which means that there can be variables of function type and functions themselves can return function values. Functions are characterized by their domain, range and a mapping. To get more sophisticated functions they can be composed in several ways and applied to expressions. A major advantage of pure functional languages is that repeated application of a function to the same argument always returns the same result. Thus, the value of a function depends only on its parameters and produces no side-effects. As a consequence, two expressions that evaluate to the same value freely can be exchanged one by another. This is called *referential transparency* and simplifies proofs and reasoning about functional programs enormously. As most of the non-trivial programs are defined recursively, the main proof tool used is induction and the proof steps are determined by the structure of the functions. Such proofs are far easier than finding an invariant for a `while`-loop and considering all side-effects in an equivalent imperative program.

The type of a function only depends on the types of expressions it is built from. This *strong typing* makes it possible to automatically type-check a program. A type inference algorithm calculates the resulting type from the subexpressions used. Static type-checking and executability of functional programs makes them particularly well-suited for rapid prototyping. This can be used for an immediate validation with respect to the requirements and exposes ambiguities and inconsistencies in the specification.

In this thesis we use a notation following *Haskell* [Bir98] style to write down functional programs. To stay simple we will not cover sophisticated aspects of modern functional languages like type and constructor classes or monads. In Haskell simple types for numbers (`Int`) or characters (`Char`) are built into the language. Additional algebraic data types can be defined by enumeration or composition using the `data` declaration. So the type `Bool` for example is an enumeration of the nullary constructors `True` and `False`:

```
data Bool = True | False
```

Data types can also be parameterized with other types which leads to polymorphic types. A value of type

```
data Maybe a = Just a | Nothing
```

just encapsulates a value of type `a` or no value at all. The possibility to define recursive data types gives this concept its real power. The data types we mainly are working with are lists and binary trees.

```
data List a = Nil | Cons a (List a)
data Tree a = Empty | Fork (Tree a , a , Tree a)
```

As lists play a crucial part in most functional languages a special syntax for `Nil` and `Cons` is provided. The empty list `Nil` is written `[]` and `Cons a as` is written `a:as`. In addition, `[a]` can be used for the single element list `a:[]`. Although it is not available in Haskell, we will use a similar abbreviation for binary trees in this thesis. The empty tree will be denoted by $\langle\rangle$ and `Fork (l,a,r)` is written as $\langle l, a, r\rangle$.

In Haskell the formal arguments of a function are given as patterns that are matched against the actual parameters. The evaluation order is sequential in the program text. So the first matching pattern in the source code is applied. Therefore most of the function definitions are case distinctions via this pattern scheme. The arguments are treated in a *curried* fashion which means that if there are less parameters than needed a function that expects the remaining arguments is returned.

To simplify complexity of notation and abbreviate we will use two implemented special patterns. The *wildcard* `_` matches all values. There is no way to reference the value of this parameter inside the function definition and therefore this is used to indicate that the value is not needed, as for example in

```
const a _ = a
```

So `(const 3)` is a function that expects one argument that is ignored and always returns 3. The *as pattern* `@` is an alternative access possibility to structured arguments like tuples. The pattern `p@(x,y)` matches a pair whose first and second components are called `x` and `y`. The complete pair can also be referred to directly by `p`.

Chapter 3

Kleene Algebra and Extensions

In this chapter we introduce the notion of *Kleene algebra* (KA) and present general operations. The definitions and presented properties of the introduced operators are strongly oriented towards calculations and applications in the graph theoretic model. As in relation algebra we will see Kleene algebra elements as representations of the link structure of graphs. Enhancements to extend KA also to an abstract theory of labeled graphs are presented. This leads to the definition of *pointer Kleene algebra* considered in more detail in the next chapter.

The aim of the observations presented in this chapter was to remove a significant drawback of a former approach by B. Möller [Möl97a]. There labeled graphs are described by a family of relations each representing a particularly labeled subgraph as described in Section 2.1. Thus, each calculation results in case distinctions with respect to the set of labels. This makes proofs longer and more complicated than they could be. To simplify reasoning about labeled graphs we are heading for a compact algebraic representation of labeled graphs with possibilities to access particular substructures and to model selective changes.

We assume that each element of the Kleene algebra represents a particular graph and require that the set of labels shows a Boolean structure. This approach is slightly different from that of labeled transition systems used in verification and specification of computer systems. There the Boolean sort represents the nodes whereas the edges are labeled with elements forming a regular structure. Thus an LTS is represented by the whole Kleene algebra (for examples see [Pra91]) whereas in our framework each element models a labeled graph.

3.1 Kleene Algebra

The algebraic structure of a Kleene algebra axiomatizes the three components of regular events: sequential composition, nondeterministic choice and iteration, represented by composition $(\cdot)$, join $(+)$, and star $(\_^*)$. As formal basis in this thesis we consider Kleene algebra to be an idempotent semiring with star as introduced by Kozen [Koz97].

Definition 1 (Kleene algebra). *A* Kleene algebra $(\mathcal{K}, +, \cdot, 0, 1, ^*)$ *is an idempotent semiring with star, i.e.:*

$(\mathcal{K}, +, 0)$ is an idempotent commutative monoid:

$$a + (b + c) = (a + b) + c \tag{3.1}$$
$$a + b = b + a \tag{3.2}$$
$$a + 0 = a \tag{3.3}$$
$$a + a = a \tag{3.4}$$

$(\mathcal{K}, \cdot, 1)$ is a monoid:

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \tag{3.5}$$
$$1 \cdot a = a \tag{3.6}$$
$$a \cdot 1 = a \tag{3.7}$$

Composition distributes over $+$:

$$a \cdot (b + c) = a \cdot b + a \cdot c \tag{3.8}$$
$$(a + b) \cdot c = a \cdot c + b \cdot c \tag{3.9}$$

0 is a two-sided annihilator for $\cdot$:

$$0 \cdot a = 0 \tag{3.10}$$
$$a \cdot 0 = 0 \tag{3.11}$$

The $*$ operator satisfies:

$$1 + a \cdot a^* = a^* \tag{3.12}$$
$$1 + a^* \cdot a = a^* \tag{3.13}$$
$$b + a \cdot c \leq c \rightarrow a^* \cdot b \leq c \tag{3.14}$$
$$b + c \cdot a \leq c \rightarrow b \cdot a^* \leq c \tag{3.15}$$

Note, that we can do without either (3.12) or (3.13), as one of them is derivable from the other axioms (see [Hol98]). Alternatively, one can define both laws only by inequations. We will call a KA *Boolean* if the set $\mathcal{K}$ together with the respective operations forms a Boolean algebra. As known from lattice theory, idempotence, commutativity, and associativity of join induce a natural order relation over KA elements by:

$$a \leq b \overset{\text{def}}{\Leftrightarrow} a + b = b$$

A useful proof tool to show equality using inequations are the rules of indirect equality:

$$a = b \Leftrightarrow (\forall c.\ c \leq a \Leftrightarrow c \leq b) \Leftrightarrow (\forall c.\ a \leq c \Leftrightarrow b \leq c)$$

Abbreviating iterations that perform at least one step we additionally introduce:

$$a^+ \overset{\text{def}}{=} a \cdot a^*$$

Both operations $\_^*$ and $\_^+$ are monotonic and the well-known laws of regular languages hold:

Lemma 2.

1. $0^* = 1^* = 1$
2. $0^+ = 0$ *and* $1^+ = 1$
3. $1 \leq a^*$
4. $a \leq a^*$ *and* $a \leq a^+$
5. $(a + b)^* = a^* \cdot (b \cdot a^*)^*$
6. $a \cdot (b \cdot a)^* = (a \cdot b)^* \cdot a$
7. $a^* \cdot a^* = a^*$ *and* $a^+ \cdot a^+ = a \cdot a^+$
8. $a^+ \cdot a^* = a^+ = a^* \cdot a^+$
9. $(a^*)^+ = a^* = (a^+)^*$
10. $(a^*)^* = a^*$ *and* $(a^+)^+ = a^+$

The proofs are trivial or can be found in [Koz90a].

3.2 Predicates

To model also sets of nodes in a single sorted theory like Kleene algebra we have to find a representation for them as graphs. Then they can be embedded into the algebra and treated uniformly with the graph model. The trick is to use graphs with edges that only point from nodes to themselves. A node is then interpreted as an element of the modeled set if and only if there exists a loop on this node. The maximal graph in which each node has such a self-reference is formalized by the identity 1 which represents the universal set of all nodes. Since each subgraph of this graph also consists of loops only, a set of nodes can be represented algebraically by elements that are smaller than the identity. Influenced by KA applications to program optimization and verification, where such subidentities are used to represent program states, we will call them *predicates.*

Definition 3 (predicate). *A* predicate *of a Kleene algebra is an element s with $s \leq 1$.*

In the sequel we will use s, t to name predicates and denote the set of all predicates by $\mathcal{P} = \{s : s \leq 1\}$. To be able to calculate with predicates as sets of nodes, abstract properties of sets are needed. Therefore we assume that the predicates form a Boolean lattice $(\mathcal{P}, +, \cdot, \neg, 0, 1)$ with $\neg$ denoting the complement in $\mathcal{P}$. This is a restricted variant of Kleene algebra with tests (KAT) as defined by Kozen [Koz97]. Kozen considers our approach less desirable than using a second sort to represent tests for two reasons. First, he criticizes that not every KA can be extended to a KA with tests if the set of all predicates has to form a Boolean lattice. And second, he complains that considering all predicates as tests may contradict interpretations in his program semantics model. Both arguments do not hold for our purpose. To model sets of nodes we are not interested in extensibility of an arbitrary Kleene algebra into one satisfying our model. What we need is a representation showing a set-like behaviour of predicates which is achieved by the Boolean structure. Since the identity is a model for the universal set of all nodes and each smaller element represents a subset, this implies the choice of the complete structure lying under the identity. Obviously, iterated composition of predicates yields the predicate itself, since composition coincides with meet. This is reflected by:

Lemma 4. *For predicate s the following iteration laws hold:*

$$s^* = 1 \quad \textit{and} \quad s^+ = s$$

Proof. From $1^* = 1$ and monotonicity it follows that: $s^* \leq 1^* = 1 \leq s^*$ and $s^+ = s \cdot s^* = s \cdot 1 = s$ $\square$

Similar to Kozen's KAT, we define Kleene algebra with predicates as an enhancement of KA.

Definition 5 (KA with predicates). *A* Kleene algebra with predicates (KAP) *is a KA in which the set $\mathcal{P}$ of predicates forms a Boolean lattice $(\mathcal{P}, +, \cdot, \neg, 0, 1)$ with $\neg$ denoting the complement in $\mathcal{P}$.*

If constants and operations are clear from the context we simply say that $\mathcal{P}$ forms a Boolean lattice. In the sequel we will assume to work in KAPs and use the notion Kleene algebra interchangeably. Composition of a predicate s and graph a in the graph model can be derived directly from the definition of composition of two graphs in Section 2.1. Since s just consists of loops, composition restricts a to the set of nodes represented by s. Composition from the left results in a domain restricted subgraph and preserves only the links starting from nodes in s. Symmetrically, all links with targets not in s are removed by composition from the right. Calculations with restrictions often result in separate reasoning about a set of nodes satisfying a particular condition and the remaining nodes that do not. We will refer to such a partition of element a into the restriction $s \cdot a$ and the complementary restriction $\neg s \cdot a$ by the notion *case distinction.*

In the contracted labeled graph model, membership in the represented set indicated by a loop is generalized. Each loop is marked with a set of labels which can be interpreted as membership degree of the respective node. Thus, in the labeled graph model predicates represent fuzzy sets of nodes. To get more information in the context of this enhanced interpretation we need additional algebraic operations.

3.3 Residuals

A helpful tool to gain knowledge about the internal structure of elements but nevertheless staying in an abstract framework is residuation. Residuals go back to de Morgan [dM64] who introduced them for relation algebra and called the defining rule "Theorem K". The term residual itself was coined by Ward and Dilworth [WD39] and studied in lattices in more detail by Birkhoff [Bir67]. Later Conway used the term factors [Con71] for the same concept. Note that this notion of factors has nothing to do with factors in graph theory. Residuals also show up in a more application-specific way as *weakest pre-/post-specification* [HJ87] and form the basis of *division allegories* [FS90]. Residuals characterize largest solutions of certain linear equations and mostly are defined by Galois connections.

Definition 6 (residuals).

$$b \leq a\backslash c \overset{\text{def}}{\Leftrightarrow} a \cdot b \leq c \overset{\text{def}}{\Leftrightarrow} a \leq c/b$$

We will call a Kleene algebra where all residuals exist a *residuated KA*. As a direct consequence from existing residuals 0 is a two-sided annihilator for composition and the distributivity laws 3.8 and 3.9 hold. In a residuated KA we get a top element for free, which is $\top = 0\backslash 0$. Note, that this is not the only representation of the greatest element. In fact all terms $0\backslash a$ are equal to top. The structure of a residuated Kleene algebra is equivalent to that of action algebras introduced by Pratt [Pra90a] as an axiomatization for action logic. Action algebras have the advantage that they are finitely, equationally definable and so form a variety. Nevertheless, they are quite restrictive, since the existence of residuals is a rather strong demand. In our case it is possible to get enough information about the internal structure by restricting residuals to predicates. To motivate this we temporarily change our view to residuals of binary relations.

In the matrix model $S\backslash R$ relates two elements x and y if column x in S is covered by column y in R. R/S symmetrically works on rows. If S now is a predicate and so only consists of unit vectors we have two cases. Either the column in S is completely empty and therefore is covered by every column in R . Or the column consists of exactly one entry and residuation indicates if this entry is also present in the respective column in R. So $S\backslash R$ has completely filled rows where S is undefined and equals R otherwise. We can show this model theoretic observation generally in residuated Kleene algebras:

Lemma 7. *In a residuated KA, residuals with respect to predicate s can be expressed explicitly as*

$$s\backslash a = a + \neg s \cdot \top \qquad\qquad a/s = a + \top \cdot \neg s \tag{3.16}$$

Proof. We only show the first equality. Assume a residuated KA, then

$$\begin{aligned}(\leq) &: s\backslash a = s \cdot (s\backslash a) + \neg s \cdot (s\backslash a) \leq a + \neg s \cdot \top \\ (\geq) &: s \cdot (a + \neg s \cdot \top) = s \cdot a \leq a \Leftrightarrow a + \neg s \cdot \top \leq s\backslash a\end{aligned}$$

□

So we just restrict ourselves to Kleene algebra with a greatest element and axiomatize \ and / by the equations (3.16).

Definition 8 (KA with top). *A Kleene algebra with top $(\mathcal{K}, +, \cdot, 0, 1, \_^*, \top)$ is a KA $(\mathcal{K}, +, \cdot, 0, 1, \_^*)$ enhanced with an element $\top$ defined by: $\forall a \in \mathcal{K}.\ a \leq \top$.*

By definition it follows immediately that residuals are anti-monotone in the predicate argument and monotone in the other. Also the standard laws for residuals hold restricted to predicates:

Lemma 9.

1. $0\backslash a = \top$
2. $1\backslash a = a$
3. $s\backslash\top = \top$
4. $1 \leq s\backslash s$
5. $s \cdot (s\backslash a) = s \cdot a$
6. $s\backslash(s \cdot a) = s\backslash a$
7. $(s \cdot t)\backslash a = t\backslash(s\backslash a)$
8. $(s\backslash a)/t = s\backslash(a/t)$

The proofs follow immediately from Axioms (3.16). We can even show that in KAs with top the Galois connections of Definition 6 holds restricted to predicates.

Lemma 10.

$$s \cdot a \leq b \Leftrightarrow a \leq s\backslash b \quad \textit{and} \quad a \cdot s \leq b \Leftrightarrow a \leq b/s$$

Proof. Assume $s{\cdot}a \leq b$, then $a = s{\cdot}a{+}\neg s{\cdot}a \leq b{+}\neg s{\cdot}\top = s\backslash b$. Now assume $a \leq s\backslash b$, then $s \cdot a \leq s \cdot (s\backslash b) \overset{9.5}{=} s \cdot b \leq b$. The second proposition follows symmetrically. □

3.4 Kleene Algebra with Domain

The representation of sets of nodes by predicates enables us also to give abstract characterizations of domain and codomain of a graph. The domain represents the set of nodes links start from and symmetrically codomain results in all target nodes. As seen in Section 2.1 these operations coincide for directed graphs with the respective notions on the edge relation. In the sequel we will focus on domain, as the laws for codomain hold symmetrically. Abstractly, the set of nodes links start from can be expressed as the least predicate that does not restrict the graph by composition. We will use an equational axiomatization for domain based on the one given in [DMS03].

Definition 11 (domain). *The domain operation $\ulcorner\_$ is axiomatized by:*

$$a \leq \ulcorner a \cdot a \tag{3.17}$$

$$\ulcorner(s \cdot a) \leq s \tag{3.18}$$

To speak in the notions of [DMS03] we more precisely just demand $\ulcorner\_$ to be a predomain operator. Nevertheless, for practical applications that require equality propositions one cannot do without a rule called locality. As we will see this follows from one of the later added extensions of the algebra. Definition 11 implies distributivity over joins and therefore monotonicity, just as domain is strict, i.e. $\ulcorner a = 0 \Leftrightarrow a = 0$. Other properties which are easy to see are:

Lemma 12.

1. $\ulcorner a \leq 1$
2. $\ulcorner s = s$
   In particular: $\ulcorner 1 = 1$
3. $\ulcorner a \cdot a = a$
4. $\ulcorner(a \cdot \top) = \ulcorner a$
   In particular: $\ulcorner\top = 1$

With these rules we can show that indeed:

Lemma 13. *$\ulcorner a$ is the least solution of the equation $s \cdot a = a$*

Proof. $\ulcorner a$ is a solution by Lemma 12.3. Now assume that $t \leq \ulcorner a$ is also a solution, then: $\ulcorner a = \ulcorner(t \cdot a) \leq t$ and therefore $t = \ulcorner a$ □

By a straightforward calculation we can also show that for domain the Galois connection

$$\ulcorner a \leq s \Leftrightarrow a \leq s \cdot \top$$

holds if there is a greatest element in the algebra. The extension of KAP by domain and codomain operators leads us to:

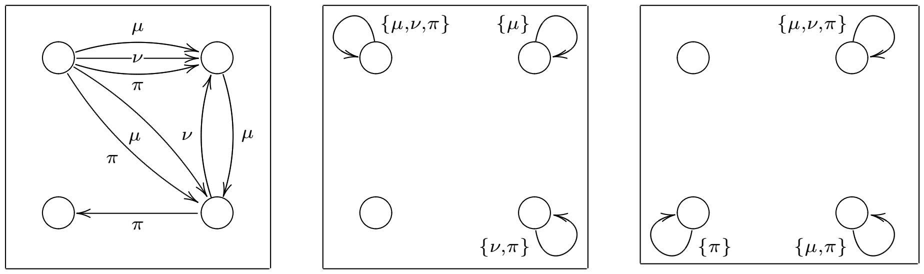

Figure 3.1: Application of domain and codomain

Definition 14 (KA with domain). *A* Kleene algebra with domain *(KAD) is a KAP with two operators* $\ulcorner\_$ *and* $\_\urcorner$ *for domain and codomain satisfying the laws of Definition 11 and the symmetrical versions respectively.*

For labeled graphs the domain operation not only yields a representation of the set of nodes links start from. Additionally it sums up the set of symbols these links are labeled with and assigns them to the respective loops (see Figure 3.1). This is the representation of a fuzzy set of nodes where the membership degree of each node corresponds to the set of labels of its outgoing links.

3.5 Scalars and Ideals

Based on the observation that the α-cut operation can be used to algebraically get a particularly labeled subgraph, this section is concerned with the foundations of such a projection. The first we need is a representation for the discrimination level α which in our case corresponds to a set of edge labels. In Section 3.2 it was shown that by representing sets of nodes as graphs, they can be treated in a single-sorted graph theory. The same idea now is used to model sets of labels. The representation of cut-level α will be a labeled graph with an α-labeled loop on each node. This corresponds to a universal fuzzy set containing each node with membership grade α. Algebraically this is fixed by the notion of a scalar.

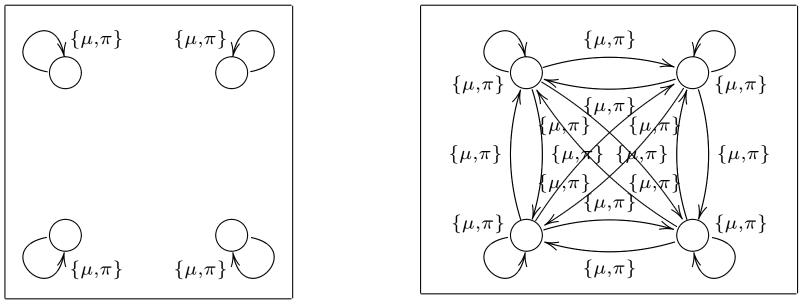

Figure 3.2: Representation of scalars and ideals

Definition 15 (scalar). *A* scalar *is an element* $\alpha \in \mathcal{P}$ *that commutes with the top element, i.e.* $\alpha \cdot \top = \top \cdot \alpha$.

We observe that in the fuzzy model for predicate s and greatest element $\top$ we have:

$$(s \cdot \top)(x, y) = s(x, x)$$
$$(\top \cdot s)(x, y) = s(y, y)$$

By equality of these two terms we get the intended interpretation for scalar α:

$$\forall x, y.\ \alpha(x,x) = \alpha(y,y)$$

Scalars are closed under the KA operations and as they are predicates also under complement:

Lemma 16. *The set $\mathcal{S} = \{\alpha \le 1 \mid \alpha \cdot \top = \top \cdot \alpha\}$ of scalars forms a Boolean sublattice of $\mathcal{P}$ and $(\mathcal{S}, +, \cdot, 0, 1, \_^*)$ is a Boolean KA.*

Proof. Closedness under $+$, $\cdot$ and $\_^*$ is trivial or follows from Lemma 4. The remaining closedness under $\neg$ is proven by case distinction:

$$\neg\alpha \cdot \top = \neg\alpha \cdot \top \cdot \alpha + \neg\alpha \cdot \top \cdot \neg\alpha = \neg\alpha \cdot \top \cdot \neg\alpha \le \top \cdot \neg\alpha$$

The other direction is shown symmetrically. □

In the sequel we will use Greek letters α, β, γ for scalars. Scalars not only commute with top but also show some other nice commutativity properties:

Lemma 17. *Let $\alpha \in \mathcal{S}$ be a scalar and $a \in \mathcal{K}$, then*

1. $\alpha \backslash a = a / \alpha$
2. $\alpha \cdot a = a \cdot \alpha$

Proof. 1. $\alpha \backslash a = a + \neg\alpha \cdot \top = a + \top \cdot \neg\alpha = a/\alpha$
2. By indirect equality and Lemma 10: $\alpha \cdot a \le b \Leftrightarrow a \le \alpha \backslash b \Leftrightarrow a \le b/\alpha \Leftrightarrow a \cdot \alpha \le b$ □

As seen in Section 2.1 an edge-labeled graph can be split up into the set of its α-subgraphs. Projection of the subgraph using only edges of labels from set α can be achieved by intersection with a completely connected α-subgraph. Such a completely connected graph with only identically labeled arrows algebraically corresponds to an ideal [JT51, JT52]. An ideal in a KA with top is an element that is invariant under composition with top.

Definition 18 (ideal). *A* right ideal *is an element $j \in \mathcal{K}$ that satisfies $j = j \cdot \top$. Symmetrically we define the notion of* left ideals. *An* ideal *then is an element that is a left and a right ideal, i.e. $\top \cdot j \cdot \top = j$.*

In the sequel we will denote ideals by j and k. The set $\mathcal{J} = \{j \mid \top \cdot j \cdot \top = j\}$ of ideals is closed under the KA operations $+$ and $\cdot$ but not under $\_^*$.

Lemma 19. *For iteration of ideals one has: $j^* = 1 + j$ and $j^+ = j$.*

Proof. We only prove the second claim from which the first follows immediately: $j^+ = j \cdot j^* \le j \cdot \top = j \le j^+$ □

Since for the set of ideals the top element $\top$ coincides with the identity, this set forms an algebra of subidentities:

Corollary 20. *The ideal elements of a KA with top together with the restricted operations form a semiring $(\mathcal{J}, +, \cdot, 0, \top)$ with top.*

It is evident that every non-trivial KA with top has at least the two ideals 0 and $\top$. For simple algebras these are the only ones.

Definition 21 (simple). *An algebra is called* simple, *if the Tarski rule $a \ne 0 \Rightarrow \top \cdot a \cdot \top = \top$ holds for all a.*

In [JT52] Jónsson and Tarski first showed equivalence of these two characterizations of simplicity for non-trivial relation algebras. The standard relation models all are simple and therefore do not help to give us a better intuition of ideals. Thus, we change our view to the fuzzy relational model. There an ideal corresponds to a matrix with only identical sets as entries. This is the representation of a contracted completely connected graph with all links labeled identically. Evidently, intersection of such an element with an arbitrary graph representation results in the according α-subgraph.

By idempotence of $\top$ we can see that for every element $a \in \mathcal{K}$ the composition $\top \cdot a \cdot \top$ is an ideal. By this operation each ideal is mapped onto itself and so every ideal can be written in this form. Also for every scalar α function $i_{\mathcal{S}\mathcal{J}}(\alpha) = \top \cdot \alpha \cdot \top$ is a mapping from scalars to ideals. By commutativity this can be simplified to

$$i_{\mathcal{S}\mathcal{J}}(\alpha) = \alpha \cdot \top$$

and we can show injectivity of this mapping:

Lemma 22. *For a scalar α the element $j = i_{\mathcal{S}\mathcal{J}}(\alpha)$ is an ideal and $i_{\mathcal{S}\mathcal{J}} : \mathcal{S} \to \mathcal{J}$ is injective.*

Proof. $\top \cdot j \cdot \top = \top \cdot (\alpha \cdot \top) \cdot \top = \alpha \cdot \top = j$, so j is an ideal. Let now $\alpha, \beta \in \mathcal{S}$ and $i_{\mathcal{S}\mathcal{J}}(\alpha) = i_{\mathcal{S}\mathcal{J}}(\beta)$, then $\alpha = \ulcorner(\alpha \cdot \top) = \ulcorner(\beta \cdot \top) = \beta$. $\square$

It is well-known that a category of L-relations in the sense of Goguen forms a Dedekind category [KF01]. The term Dedekind category was introduced by Olivier and Serrato [OS95] and is called locally complete division allegory by Freyd and Scedrov [FS90]. Kawahara and Furusawa [KF01] have proved the bijective correspondence between scalars and ideals in such categories. Based on this bijection Winter [Win01] has shown how to axiomatize cut operations to formalize crispness. It is now shown how these concepts can be ported to Kleene algebra.

Kawahara and Furusawa used the two functions

$$i_{\mathcal{S}\mathcal{J}}(\alpha) = \alpha \cdot \top \text{ and } i_{\mathcal{J}\mathcal{S}}(j) = j \sqcap 1$$

as bijective mappings between the set of scalars $\mathcal{S}$ and the set of ideals $\mathcal{J}$. Since there is no meet operation in KAs, we show how to achieve the mapping $i_{\mathcal{J}\mathcal{S}}$ with Kleene algebraic means. For the proof of injectivity of $i_{\mathcal{J}\mathcal{S}}$ in [KF01] the modular laws

$$\begin{aligned} Q \cdot R \sqcap S &\leq Q \cdot (R \sqcap Q^{\smile} \cdot S) \\ Q \cdot R \sqcap S &\leq (Q \sqcap S \cdot R^{\smile}) \cdot R \end{aligned}$$

find their place. To avoid unnecessary parentheses we assume that composition binds more tightly than meet. Although the modular laws make use of the converse operation $^{\smile}$, only a weaker version of them is needed. The top element, which remains unchanged under $^{\smile}$, is used as conversed element. To be able to replay the proof we will change our focus temporarily to standard Kleene algebras (SKA) which are similar to S-algebras as defined in [Con71]. This is a more restrictive structure based on a complete lattice. So there is a meet operation. For an axiomatization see Section A.1 in the appendix. We will now extend SKAs by the modular laws:

Definition 23 (weakly modular SKA). *We say a SKA is* weakly modular *if the modular laws hold for $\top$:*

$$\begin{aligned} \top \cdot a \sqcap b &\leq \top \cdot (a \sqcap \top \cdot b) \\ a \cdot \top \sqcap b &\leq (a \sqcap b \cdot \top) \cdot \top \end{aligned}$$

In a WMKA we can replay the proof of injectivity for $i_{\mathcal{JS}}$:

Proof.
$i_{\mathcal{JS}}(j) \cdot \top = (j \sqcap 1) \cdot \top \leq j \cdot \top = j = j \sqcap 1 \cdot \top \leq (j \cdot \top \sqcap 1) \cdot \top = (j \sqcap 1) \cdot \top = i_{\mathcal{JS}}(j) \cdot \top$, thus: $i_{\mathcal{JS}}(j) \cdot \top = j$ which immediately shows injectivity of $i_{\mathcal{JS}}$. Symmetrically $j = \top \cdot i_{\mathcal{JS}}(j)$, so that $i_{\mathcal{JS}}(j)$ indeed is a scalar. □

In a next step we show how to eliminate the need for a meet operation by giving conditions that are able to replace the restricted modular laws. A first observation shows that in the case of WMKAs there is a closed formula for domain:

Lemma 24. *Assume a WMKA, then*

$$\ulcorner a = a \cdot \top \sqcap 1$$

Proof. By the modular laws $\ulcorner a = \ulcorner(a \sqcap 1 \cdot \top) \leq \ulcorner((a \cdot \top \sqcap 1) \cdot \top) = a \cdot \top \sqcap 1$. On the other hand $a \cdot \top \sqcap 1 = \ulcorner a \cdot a \cdot \top \sqcap 1 = \ulcorner a \cdot a \cdot \top \sqcap \ulcorner a \leq \ulcorner a$, since in SKA composition with predicates distributes arbitrarily over meet. □

As a consequence the operation $i_{\mathcal{JS}}$ on ideals can be simplified to $i_{\mathcal{JS}}(j) = j \sqcap 1 = j \cdot \top \sqcap 1 = \ulcorner j$. We can give alternative conditions without using meet that $i_{\mathcal{JS}}$ and domain coincide:

Lemma 25. *The following conditions are equivalent in SKAs with domain:*

1. $\ulcorner a \leq a \cdot \top$
2. $\ulcorner a \cdot \top = a \cdot \top$
3. $\ulcorner a = a \cdot \top \sqcap 1$

Proof.

1. ⇒ 2.: $\ulcorner a \cdot \top \leq a \cdot \top \cdot \top = a \cdot \top = \ulcorner a \cdot a \cdot \top \leq \ulcorner a \cdot \top$
2. ⇒ 3.: $\ulcorner a = \ulcorner a \sqcap 1 \leq \ulcorner a \cdot \top \sqcap 1 = a \cdot \top \sqcap 1 = \ulcorner a \cdot a \cdot \top \sqcap 1 = \ulcorner a \cdot a \cdot \top \sqcap \ulcorner a \leq \ulcorner a$
3. ⇒ 1.: $\ulcorner a = a \cdot \top \sqcap 1 \leq a \cdot \top$ □

The reverse implication from 2. to 1. also holds in structures without a meet operation, so that they are equivalent even in KAPs with top and domain. Symmetrically, the same proposition holds for codomain.

Lemma 26. *The following formulas are equivalent in SKAs with codomain.*

1. $a^\urcorner \leq \top \cdot a$
2. $\top \cdot a^\urcorner = \top \cdot a$
3. $a^\urcorner = \top \cdot a \sqcap 1$

Motivated by the form of Equations 25.1 and 26.1 we will call these properties *subordination of domain* and *codomain* respectively.

Example 27. The algebra of regular languages $LAN = (\mathcal{P}(A^*), \cup, \cdot, \bullet, \emptyset, \epsilon)$ only has the two predicates $\emptyset$ and ϵ. Therefore all non-empty languages have domain ϵ. Nevertheless, LAN does not show subordination, as in general ϵ is not contained in the composition of an arbitrary language and A^* which corresponds to $\top$. □

Alternatively we can give more symmetric conditions equivalent to Lemmas 25/26:

$$\ulcorner a \cdot b \leq \ulcorner b \cdot a \cdot \top$$
$$a \cdot b^\urcorner \leq \top \cdot b \cdot a^\urcorner$$

Nevertheless, we will keep the characterizations from Lemmas 25/26 due to their simplicity. By adding subordination of domain and codomain we can show more sophisticated properties of ideals and some that are needed for later derivation steps. We only present the respective laws using domain, as the codomain variants follow symmetrically.

Lemma 28. *With subordination of domain composition of ideals is idempotent and commutative.*

Proof. $j \cdot j = j \cdot \top \cdot j = \ulcorner j \cdot \top \cdot j = \ulcorner j \cdot j = j$ shows idempotence and commutativity follows from $j \cdot k = j \cdot k \cdot j \cdot k \leq \top \cdot k \cdot j \cdot \top = k \cdot j$. The other direction is shown symmetrically. □

Remember that we introduced subordination to establish a bijection between scalars and ideals. We now can show some properties of the chosen mapping $\ulcorner\_$ on ideals:

Lemma 29. *Assume again subordination, then for $j \in \mathcal{J}$*

1. $\ulcorner j \leq j$
2. $j = \ulcorner j \cdot \top$
3. $\ulcorner j = j\urcorner$
4. $\ulcorner j \cdot \top = \top \cdot \ulcorner j$

Proof. 1. $\ulcorner j \leq j \cdot \top = j$
2. $\ulcorner j \cdot \top = j \cdot \top = j$
3. $\ulcorner j = (\ulcorner j)\urcorner \overset{1.}{\leq} j\urcorner$ and symmetrically $j\urcorner \leq \ulcorner j$
4. $\ulcorner j \cdot \top \overset{2.}{=} j = \top \cdot j\urcorner \overset{3.}{=} \top \cdot \ulcorner j$ □

The first law shows that the naming of subordination is justified. Indeed the domain of an ideal, which equals its corresponding scalar, is below the ideal itself. The third equation shows that it does not matter if one uses domain or codomain to map ideals to scalars. This mimics the fact that one is also free to choose composition with top either from the left or right to map a scalar to its corresponding ideal. With subordination we are in the position to choose $i_{\mathcal{JS}}(j) = \ulcorner j = j\urcorner$. Finally, we are able to show that this mapping is injective:

Lemma 30. *Assume subordination of domain, then $i_{\mathcal{JS}}$ is injective on ideals.*

Proof. Assume $i_{\mathcal{JS}}(j) = i_{\mathcal{JS}}(k)$, i.e. $\ulcorner j = \ulcorner k$, then $j \overset{29.2}{=} \ulcorner j \cdot \top = \ulcorner k \cdot \top \overset{29.2}{=} k$ □

As one can see by Lemma 29.4 function $i_{\mathcal{JS}}$ really maps into the set of scalars, viz commutes with the top element. Indeed the two functions $i_{\mathcal{JS}}$ and $i_{\mathcal{SJ}}$ are inverse:

Lemma 31. $i_{\mathcal{JS}}(i_{\mathcal{SJ}}(\alpha)) = i_{\mathcal{JS}}(\alpha \cdot \top) = \ulcorner(\alpha \cdot \top) = \alpha$
$i_{\mathcal{SJ}}(i_{\mathcal{JS}}(j)) = i_{\mathcal{SJ}}(\ulcorner j) = \ulcorner j \cdot \top = j$

By the now established bijection between scalars and ideals it is immediately clear that the ideals also form a Boolean lattice with composition as meet operation. The construction $a\backslash 0$, which in the presence of residuals often is used as pseudo complement, now coincides for ideals with the real Boolean complement. Since we only have residuals with respect to predicates, we first show:

Lemma 32. *In a residuated KA with subordination for $j \in \mathcal{J}$ the pseudo complements $j\backslash 0$ and $\ulcorner j\backslash 0$ coincide.*

Proof. $j\backslash 0 \leq \ulcorner j\backslash 0$ follows from anti-monotonicity in the first argument. The other inequality is shown by

$$j \cdot (\ulcorner j\backslash 0) = j \cdot (0 + \neg\ulcorner j \cdot \top) = j \cdot j\urcorner \cdot \neg\ulcorner j \cdot \top = j \cdot \ulcorner j \cdot \neg\ulcorner j \cdot \top \leq 0 \Leftrightarrow \ulcorner j\backslash 0 \leq j\backslash 0$$

□

So we are able to give a closed formula of the complement operation on ideals and conclude:

Lemma 33. *The set $\mathcal{J}$ of ideals forms a Boolean lattice $(\mathcal{J},+,\cdot,\,,0,\top)$ with $\overline{j} = \ulcorner j\backslash 0 = \neg\ulcorner j \cdot \top$ the complement of j.*

Proof. We just show the statement about the complement, since all the other parts were shown previously. It follows that $j + \overline{j} = j + \neg\ulcorner j \cdot \top \overset{29.2}{=} \ulcorner j \cdot \top + \neg\ulcorner j \cdot \top = \top$ and $j \cdot \overline{j} = 0$ was shown in the proof of Lemma 32. □

Summarizing, we have the following relations between scalars and ideals (here with the use of domain and composition on the right):

$$\begin{tikzcd}
\mathcal{J} \arrow[r, "\ulcorner\_\backslash 0"] \arrow[d, bend right, "\ulcorner\_"'] & \mathcal{J} \arrow[d, bend right, "\ulcorner\_"'] \\
\mathcal{S} \arrow[r, "\neg"'] \arrow[u, bend right, "\_\cdot\top"'] & \mathcal{S} \arrow[u, bend right, "\_\cdot\top"']
\end{tikzcd}$$

An important rule that neither follows from the axiomatization of Kleene algebra nor holds in SKAs is *locality*. In [Möl99b] this rule is called (left)-local composition and describes the fact that the domain of the composition of two elements does not depend on the composed element itself but only on its domain.

Definition 34 (locality). *A Kleene algebra shows left-locality if*

$$\ulcorner b = \ulcorner c \Rightarrow \ulcorner(a \cdot b) = \ulcorner(a \cdot c)$$

Right-locality is defined symmetrically. This definition is equivalent to

$$\ulcorner(a \cdot b) = \ulcorner(a \cdot \ulcorner b)$$

which is the form in which locality most often is used. Left-locality also implies immediately

$$\ulcorner(\ulcorner a \cdot b) = \ulcorner a \cdot \ulcorner b$$

We can show that Kleene algebras with subordination of domain show left-locality.

Lemma 35. *Assume subordination of domain, then $\ulcorner(a \cdot \ulcorner b) = \ulcorner(a \cdot b)$*

Proof. $\ulcorner(a \cdot \ulcorner b) \overset{25.1}{\leq} \ulcorner(a \cdot b \cdot \top) \overset{25.2}{=} \ulcorner(\ulcorner(a \cdot b) \cdot \top) = \ulcorner(a \cdot b)$ and the opposite direction holds in all Kleene algebras with domain. □

Conversely, right-locality follows from subordination of codomain. Thus, subordination makes $\ulcorner\_$ a real domain and $\_\urcorner$ a real codomain operator in the sense of [DMS03].

3.6 Updates and Images

This section will introduce two indispensable operations on graphs. One serves to change the representations of the link structure selectively with respect to the target nodes and a second one is used to get access to the mapping behaviour induced by the links between nodes. Selective updating of links in the structure is the most essential operation all algorithms working on pointer structures are based on. We will model this by an operation to overwrite a graph representation with another. This process is selective with respect to the source nodes of links, which are represented by the domain of the overwriting element.

Definition 36 (update). *Element b overwrites a by*

$$b \mid a \overset{\text{def}}{=} b + \neg{}^{\ulcorner}b \cdot a$$

The updated element is preserved exactly where the update is not defined. Hence, a link is in the graph $b \mid a$ if it is in b or the link is in a and there is no link with the same source node in b. This operator will be our tool to represent changes in a pointer structure. The structure $(\mathcal{K}, |, 0)$ is an idempotent monoid and we can show:

Lemma 37. *1.* $\top \mid a = \top$
2. $b \leq b \mid a$
3. $b = {}^{\ulcorner}b \cdot (b \mid a)$
4. ${}^{\ulcorner}(b \mid a) = {}^{\ulcorner}b + {}^{\ulcorner}a$
5. $c \mid (a + b) = c \mid a + c \mid b$

The proofs are straightforward calculations using the definition and case distinction.

Similar to the enables operator in DAs and the Peirce product in Peirce algebras we introduce an operation that mimics an action of a Kleene element on a predicate. Intuitively the representation of a graph operates on a set of nodes by yielding all successors of these nodes. This is a sort of image under the mapping represented by the Kleene element. We adapt the notation of Peirce products and define:

Definition 38 (image). *The image of s under a is defined by:*

$$s : a \overset{\text{def}}{=} (s \cdot a)^{\urcorner}$$

Hence, the direct successors are calculated by restricting graph a to the subgraph with edges that start at nodes represented by s and then taking all the link targets. In [EMS03] a more abstract axiomatization of the image operator in the context of Kleene modules is given. There it is shown that the image operator abstractly characterizes the relational image operation. It is also shown that for practical applications the abstract setting of Kleene modules is not of useful expressiveness as there are too few properties connecting the two sorts of the module. The embedding of the Boolean sort by identifying it with the set of predicates in KAD avoids these problems. In the sequel we assume that $\cdot$ binds more tightly than $:$ to avoid unnecessary parentheses if possible. Since the image operator is defined as the composition of two monotone functions, that both distribute through joins, these properties are directly inherited:

Corollary 39. Monotonicity: $a \leq b \Rightarrow s : a \leq s : b$
$s \leq t \Rightarrow s : a \leq t : a$
Distributivity: $s : (a + b) = s : a + s : b$
$(s + t) : a = s : a + t : a$

Locality of codomain is also inherited by the image operator. This results in a rule that supports successive calculation of an image under a composed element, which is one of the two defining laws for operations on groups:

Lemma 40. *The image operator shows locality:*

$$(s : a) : b = s : (a \cdot b)$$

Proof. $(s : a) : b = ((s \cdot a)^{\urcorner} \cdot b)^{\urcorner} = (s \cdot (a \cdot b))^{\urcorner} = s : (a \cdot b)$ □

Further properties of the image operator can be deduced from this law and the definition:

Lemma 41.

1. $s : t = s \cdot t$
Immediately: $s : 1 = s$ *and* $s : 0 = 0$
2. $0 : a = 0$
3. $1 : a = a^{\urcorner}$
4. $s : a^* = s + (s : a) : a^*$
5. $s : a = 0 \Leftrightarrow s \cdot a = 0$
6. $s : a^* = 0 \Leftrightarrow s = 0$
7. $s : (a \cdot t) = (s : a) \cdot t$
8. $s : (t \cdot a) = (s \cdot t) : a$

The induction rules for the star operator (Axioms (3.14) and (3.15)) can also be lifted directly to images. So we get an image induction rule to prove properties not only for direct but arbitrary successors:

Lemma 42. *A generalized induction principle for the image operator is*

$$s : a + t : b \le t \Rightarrow s : (a \cdot b^*) \le t$$

Proof. By the Galois connection for codomain and star induction we get:

$$\begin{aligned}
& s : (a \cdot b^*) \le t \\
\Leftrightarrow\ & s \cdot a \cdot b^* \le \top \cdot t \\
\Leftarrow\ & s \cdot a + \top \cdot t \cdot b \le \top \cdot t \\
\Leftrightarrow\ & s : a + (\top \cdot t \cdot b)^{\urcorner} \le t \\
\Leftrightarrow\ & s : a + t : b \le t
\end{aligned}$$

□

This law is an instance of the powerful μ-fusion rule known from fixed point calculus (see [Bac02]), that is used to reason about inequations in which star is not the principal function on the lower side.

A special rôle is played by the image of a predicate under the top element. This operation can be used to equationally express the least scalar a predicate is included in:

Lemma 43. *The image of a predicate s under $\top$ is the least scalar α with $s \le \alpha$.*

Proof. $s : \top = (s \cdot \top)^{\urcorner} = ((\top \cdot s)^{\urcorner} \cdot \top)^{\urcorner} = (\top \cdot s \cdot \top)^{\urcorner}$ which by Lemma 31 is a scalar and from the assumption $\beta \ge s$ we get $s : \top \le \beta : \top = (\beta \cdot \top)^{\urcorner} = (\top \cdot \beta)^{\urcorner} = \beta$, which shows that $s : \top$ is the least scalar comprising s. □

This can be seen as a sort of cylindrification [HMT71] with respect to the scalar dimension.

Since we are heading for an abstract framework to handle pointer structures, there may be cases in which the expressive power of the system will not be strong enough. There may be non-standard models for which particular equality propositions cannot be shown but in most of these cases the equivalence of mapping behaviour suffices for practical interests. Therefore we define:

Definition 44 (observational equivalence). *We say the Kleene elements a and b are observationally equivalent, if*

$$a \equiv b \overset{\text{def}}{\Leftrightarrow} \forall s \le 1.\ s : a = s : b$$

This is strongly related to separability in dynamic algebras. There an algebra is called separable if every element is uniquely determined by its mapping behaviour:

$$\forall a, b \in \mathcal{K}.\ a \equiv b \Rightarrow a = b \qquad \text{(separable)}$$

So for separable algebras observational equivalence and equality of two elements coincide. The definition of observational equivalence even can be tweaked to restricting the scope of s to the joined domain of both elements:

Lemma 45. $a \equiv b \Leftrightarrow \forall s \leq \ulcorner(a+b).\ s : a = s : b$

Proof. ($\Leftarrow$) is trivial.
($\Rightarrow$) Let $s \leq 1$ be an arbitrary predicate, then: $s : a = (s \cdot \ulcorner a) : a = (s \cdot \ulcorner a) : b = (s \cdot \ulcorner a \cdot \ulcorner b) : b = (s \cdot \ulcorner b \cdot \ulcorner a) : a = (s \cdot \ulcorner b) : a = (s \cdot \ulcorner b) : b = s : b$ □

3.7 Determinacy and Atomicity

The abstract properties that are modeled by the Kleene elements are based on the mapping behaviour of a pointer linked data structure. At the moment we have no access to the internal structure of such a representation. Each information about an element can only be derived from observable effects such as the image under a set of nodes. Nevertheless, we sometimes need to model a single link or want to be sure that each node has at most one successor. Such statements about the structure often can be expressed by quantification over the set of all elements. In [DM01] it is shown that determinacy of an element abstractly can be defined in Kleene algebra by such a formula.

Definition 46 (determinacy). *An element $a \in \mathcal{K}$ is called a map (deterministic) if*

$$map(a) \overset{\text{def}}{\Leftrightarrow} \forall b \leq a.\ b = \ulcorner b \cdot a$$

Since *every* restriction of a to $\ulcorner b$ has to yield b, there is no chance that some node is mapped to two distinct successors. Obviously, if in a graph each node has at most one successor this holds also for each subgraph. Thus, the set of deterministic elements in a Kleene algebra is downward closed:

Lemma 47. $map(a) \Rightarrow \forall b \leq a.\ map(b)$

Proof. Let $c \leq b \leq a$ and $map(a)$, then $\ulcorner c \cdot b = \ulcorner c \cdot \ulcorner b \cdot a = \ulcorner c \cdot a = c$ □

For deterministic stores we can show an important annihilation rule. Updating with links that are already in the store does not yield any changes:

Lemma 48. *Assume $map(a)$, then store a shows* annihilation*, i.e.:*

$$b \leq a \Rightarrow b \mid a = a$$

Proof. $b \mid a = b + \neg\ulcorner b \cdot a = \ulcorner b \cdot a + \neg\ulcorner b \cdot a = a$ □

Another important concept to get a really applicable framework is atomicity. In a sense, atomic elements represent smallest non-empty elements. As we are only able to talk about sets of nodes, atomicity is the key to describe a single node or a single link in a graph. Nevertheless, we try to stay on an abstract level and will see that most of the laws can be proven without atomicity.

Definition 49 (atom). *An element $0 \neq a \in \mathcal{K}$ is called an atom if*

$$at(a) \overset{\text{def}}{\Leftrightarrow} \forall b \leq a.\ b = 0 \vee b = a$$

Obviously, atomic elements are deterministic. These two concepts are in a sense related but not equal:

Lemma 50. *1.* $at(a) \Rightarrow map(a)$
2. $at(a) \Rightarrow at(\ulcorner a) \wedge at(a\urcorner)$
3. $at(s) \wedge map(a) \Rightarrow at(s \cdot a)$

Proof. 1. Assume $at(a)$, then $\forall b \leq a.\ b = 0 \vee b = a$. From $b = 0$ we get $\ulcorner b \cdot a = 0 = b$ and $b = a$ implies $\ulcorner b \cdot a = \ulcorner b \cdot b = b$ which shows the claim.

2. Let $s \leq \ulcorner a$, then $s \cdot a \leq a \overset{at(a)}{\Rightarrow} s \cdot a = 0 \vee s \cdot a = a$. The first disjunct simplifies by $s \leq \ulcorner a$ to $s = s \cdot \ulcorner a = \ulcorner(s \cdot a) = \ulcorner 0 = 0$ whereas the second one by $s \cdot \ulcorner a = \ulcorner(s \cdot a) = \ulcorner a$ and $s \leq \ulcorner a$ implies $s = \ulcorner a$ which shows the claim for domain. Atomicity of codomain is shown symmetrically.
3. Let $b \leq s \cdot a$ and therefore also $b \leq a$, hence $\ulcorner b \leq \ulcorner(s \cdot a) \leq s$ and by $at(s)$: $\ulcorner b = 0 \vee \ulcorner b = s \Leftrightarrow b = 0 \vee \ulcorner b = s \overset{map(a)}{\Rightarrow} b = 0 \vee b = \ulcorner b \cdot a = s \cdot a$ □

With these rules we can show that the image of an atom under a deterministic mapping again is an atom:

Corollary 51. $at(s) \wedge map(a) \Rightarrow at(s : a)$

More important than reasoning about atomic elements is atomicity of scalars. This makes it possible to denote single labels of graphs. We introduce the notion of scalar-atomicity that is not atomicity with respect to $\mathcal{K}$ but atomicity in the lattice of scalars.

Definition 52 (scalar-atomic). *A scalar $0 \neq \alpha \in \mathcal{S}$ is called scalar-atomic if*

$$sat(\alpha) \overset{\text{def}}{\Leftrightarrow} \forall \beta \in \mathcal{S}.\ \beta \leq \alpha \Rightarrow \beta = 0 \vee \beta = \alpha$$

This concept gives us a handle to access and denote equally labeled subgraphs as shown in the following sections.

3.8 Cut Operations

So far, our treatment of pointer structures in Kleene algebra is built on a record-based view of labeled graphs. Each node represents a record and components are modeled by labeled links to the nodes representing the field contents. We have presented operations to restrict graph representations to particular nodes, change them selectively and get information about successor relations. But currently, we are not able to select particularly labeled substructures. To solve this task we consider L-fuzzy relations as an abstract theoretical model for labeled graphs. In Section 2.2 we have seen that an α-cut on fuzzy relations can be used to select the α-subgraph. Our concern now is how to abstractly model an α-cut in Kleene algebra. For this we define two cut operators $^\uparrow$ and $^\downarrow$ similar to [Win01] that send an element to the least crisp element it is included in and to the greatest element it includes respectively. The effect of these operations carried over to labeled graphs is depicted in Figure 3.3. So for example assume we have three graphs (each an element of a Kleene algebra) fitted together in one. To distinguish the edges that come from different graphs each is labeled with a unique identifier, say μ, ν, π. In the original graph on the left side a completely labeled connection exists from node A to node B, as they are connected by all sorts of links. Application of $^\uparrow$ results in the graph in the middle, where nodes that were previously linked at all are completely labeled. Application of $^\downarrow$ yields the graph on the right side in which there remain only the previously completely labeled parts. As $^\uparrow$ and $^\downarrow$ produce related least and greatest elements we can use a Galois connection to define them.

Definition 53 (up and down). *The cut operators $^\uparrow$ and $^\downarrow$ are axiomatized as follows:*

1. *$(^\uparrow, ^\downarrow)$ form a Galois connection, i.e.* $a^\uparrow \leq b \Leftrightarrow a \leq b^\downarrow$.
2. $(a \cdot b^\downarrow)^\uparrow = a^\uparrow \cdot b^\downarrow$ *and* $(a^\downarrow \cdot b)^\uparrow = a^\downarrow \cdot b^\uparrow$.

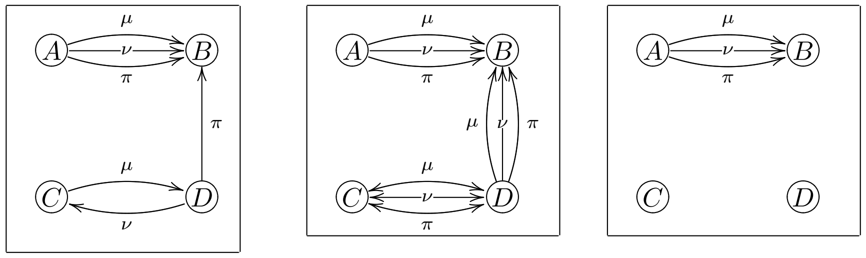

Figure 3.3: Example graph and application of $^{\uparrow}$ and $^{\downarrow}$

3. *α scalar and $\alpha \neq 0$, then $\alpha^{\uparrow} = 1$.*

The second law defines the completion of a graph that is built from paths with one part arising from a completely labeled graph. Then the same graph is yielded by first completing the other part and then performing the composition. The third law fixes the concrete semantics since by the first two laws $^{\uparrow}$ and $^{\downarrow}$ could also coincide with the identical mapping. Monotonicity and the cancellation laws follow directly from the Galois connection and therefore are given without proof. The interested reader may have a look at [Aar92] for general properties of Galois connections.

Corollary 54. *1. $^{\uparrow}$ and $^{\downarrow}$ are monotone*
2. $a \leq a^{\uparrow\downarrow}$ and $a^{\downarrow\uparrow} \leq a$
3. $a^{\uparrow} = a^{\uparrow\downarrow\uparrow}$ and $a^{\downarrow} = a^{\downarrow\uparrow\downarrow}$

By these observations and the further parts of the definition we can show the following properties of the cut operators $^{\uparrow}$ and $^{\downarrow}$:

Lemma 55.

1. $1^{\uparrow} = 1$
2. $a^{\downarrow\uparrow} = a^{\downarrow}$
3. $a^{\uparrow\downarrow} = a^{\uparrow}$
4. $a^{\uparrow\uparrow} = a^{\uparrow}$ *and* $a^{\downarrow\downarrow} = a^{\downarrow}$
5. $a \leq a^{\uparrow}$ *and* $a^{\downarrow} \leq a$
6. $a^{\uparrow} = a \Leftrightarrow a^{\downarrow} = a$
7. $0^{\uparrow} = 0$ *and* $\top^{\uparrow} = \top$
8. *(a)* $a^{\uparrow} = 0 \Leftrightarrow a = 0$
   (b) $a^{\downarrow} = \top \Leftrightarrow a = \top$
   (c) $s^{\downarrow} = 1 \Leftrightarrow s = 1$
9. $(a \cdot b^{\uparrow})^{\uparrow} = a^{\uparrow} \cdot b^{\uparrow} = (a^{\uparrow} \cdot b)^{\uparrow}$

Proof. 1. Assume $1 \neq 0$, then apply Definition 53.3. Otherwise for all a holds $a = 1 \cdot a = 0 \cdot a = 0$ and so $1^{\uparrow} = 0 = 1$.
2. $a^{\downarrow\uparrow} = (1 \cdot a^{\downarrow})^{\uparrow} = 1^{\uparrow} \cdot a^{\downarrow} = 1 \cdot a^{\downarrow} = a^{\downarrow}$
3. $a^{\uparrow} = a^{\uparrow\downarrow\uparrow} = a^{\uparrow\downarrow}$
4. $a^{\uparrow} = a^{\uparrow\downarrow\uparrow} = a^{\uparrow\uparrow}$
5. $a \leq a^{\uparrow\downarrow} = a^{\uparrow}$ and $a^{\downarrow} = a^{\downarrow\uparrow} \leq a$
6. By Galois connection and Lemma 55.5
7. $0^{\downarrow} \leq 0$, thus $0^{\downarrow} = 0$ and by Lemma 55.6 follows the proposition. The second is immediate from Lemma 55.5 as $\top \leq \top^{\uparrow}$.
8. We only show the first claim, as the others follow similarly. $a^{\uparrow} = 0 \Leftrightarrow a^{\uparrow} \leq 0 \Leftrightarrow a \leq 0^{\downarrow} \Leftrightarrow a \leq 0 \Leftrightarrow a = 0$
9. $(a \cdot b^{\uparrow})^{\uparrow} \overset{55.3}{=} (a \cdot b^{\uparrow\downarrow})^{\uparrow} \overset{53.2}{=} a^{\uparrow} \cdot b^{\uparrow\downarrow} \overset{55.3}{=} a^{\uparrow} \cdot b^{\uparrow}$. The second property is shown symmetrically.

□

As we can see by Corollaries 54.1, 55.4 and Lemma 55.5, up is a closure and down an interior operator. Some of the laws for up (e.g. Lemma 55.9) remind us of axioms used in a cylindric algebra [HMT71]. Indeed one can see the up operator as a sort of cylindrification or completion with respect to the labeling.

Focusing on the interaction between the cut operators and ideals we get some more insight into their behaviour. As we established by subordination of domain and codomain a bijection between scalars and ideals, the special case for scalars in Definition 53.3 is inherited by ideals and we can show a sort of specialized Tarski rule.

Lemma 56. *1. If $j \neq 0$ is an ideal, then $j^{\uparrow} = \top$.*
2. $a \neq 0 \Rightarrow \top \cdot a^{\uparrow} \cdot \top = \top$.

Proof. 1. Assume $\alpha = i_{\mathcal{JS}}(j)$ to be the scalar corresponding to j. Thus, j can be represented by $i_{\mathcal{SJ}}(\alpha) = \alpha \cdot \top$. By strictness of $i_{\mathcal{JS}}$ the proposition $j \neq 0$ implies $\alpha \neq 0$ and therefore $j^{\uparrow} = (\alpha \cdot \top)^{\uparrow} = \alpha^{\uparrow} \cdot \top = \top$.
2. From $a \neq 0$ it follows that $\top \cdot a \cdot \top$ is a non-zero ideal. Thus, by $\top = \top^{\uparrow}$ and Lemma 55.9 we get: $\top \cdot a^{\uparrow} \cdot \top = (\top \cdot a \cdot \top)^{\uparrow} \overset{1.}{=} \top$ □

This perfectly meets our intuition, since non-zero ideals are models of equally labeled completely connected graphs. Application of $^{\uparrow}$ clearly should yield the complete graph represented by $\top$. But we can also turn the tables. As the set of ideals forms a Boolean lattice the symmetric law holds for all ideals not equal to top and for application of $^{\downarrow}$. Using again the bijective correspondence we get as an consequence the symmetric version of Law 53.3 for scalars and application of $^{\downarrow}$.

Lemma 57. *For ideal j and scalar α the following implications hold:*

$$j \neq \top \Rightarrow j^{\downarrow} = 0$$
$$\alpha \neq 1 \Rightarrow \alpha^{\downarrow} = 0$$

Proof. Assume $j^{\downarrow} \neq 0$. By Lemmas 55.5, 55.2, and 56.2 this implies $j = \top \cdot j \cdot \top \geq \top \cdot j^{\downarrow} \cdot \top = \top \cdot j^{\downarrow\uparrow} \cdot \top = \top$. Inversion of this implication shows the first claim. Now assume $\alpha \neq 1$, then $i_{\mathcal{SJ}}(\alpha) = \alpha \cdot \top$ is an ideal with $\alpha \cdot \top \neq \top$ and application of the first law yields: $\alpha^{\downarrow} \leq (\alpha \cdot \top)^{\downarrow} = 0$ □

In summary we have shown that cutting a scalar either by $^{\uparrow}$ or $^{\downarrow}$ results only in 0 or 1 whereas application on ideals yields either 0 or $\top$. So the set of scalars as well as the set of ideals are closed under the cut operations:

Corollary 58. *Let $\alpha \in \mathcal{S}$ and $j \in \mathcal{J}$, then $\alpha^{\uparrow}, \alpha^{\downarrow} \in \mathcal{S}$ and $j^{\uparrow}, j^{\downarrow} \in \mathcal{J}$.*

The down operator can now be used to abstractly model an α-cut operation. We use a scalar α as unique handle for the desired set of edge labels. Since under $^{\downarrow}$ only completely labeled links survive we first have to prepare the graph. This is achieved by enlarging each link labeled at least with marks from set α to a completely labeled edge. By $\alpha \backslash a = a + \neg\alpha \cdot \top$ we can see that the residual exactly performs this task by completing the graph with all labels from the complement of α. So after the operation two nodes are connected by a completely labeled link if and only if they were connected at least by all labels represented by α. Application of $^{\downarrow}$ then removes all remaining not completely connected links. We will extend this α-cut operation slightly by a restriction to α to get a projection function yielding all the α-links of the original graph:

$$P_{\alpha}(a) = \alpha \cdot (\alpha \backslash a)^{\downarrow}$$

In contrast to the original α-cut this projection behaves a little bit more pleasantly. The trouble in the calculation of an α-cut arises in connection with the empty set. All cuts with other discrimination levels respect the original link structure and only remove links that do not exceed level α. But the cut with discrimination level 0 adds new links that previously were not present as by $(0 \backslash a)^{\downarrow} = \top^{\downarrow} = \top$ it returns the complete graph. P_{α} is idempotent and monotone, since $^{\downarrow}$ is and residuals

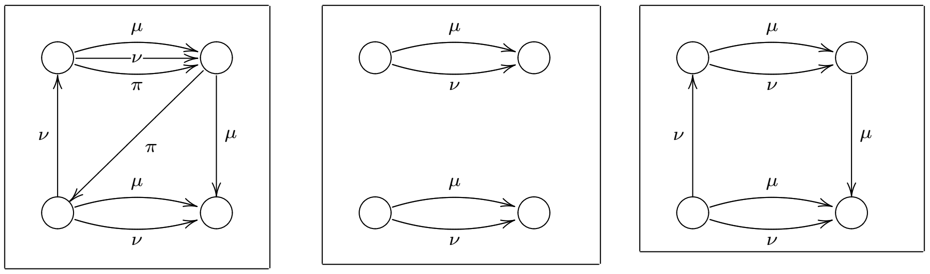

Figure 3.4: Effect of $P_{\{\mu,\nu\}}$ and restriction to $\{\mu, \nu\}$

are monotone in the non-predicate argument. We can show that the projection is smaller than just restricting the graph to the α-level since restriction also yields links labeled only by a subset of α (see Figure 3.4). We will see later that projection and restriction coincide for atomic α.

Lemma 59. *1.* $P_\alpha(a) \leq \alpha \cdot a$

2. $P_\alpha(P_\alpha(a)) = P_\alpha(a)$

3. $P_\alpha(a + \neg\alpha \cdot b) = P_\alpha(a)$

4. $\alpha \cdot {}^\ulcorner a = 0 \Rightarrow P_\alpha(a) = 0$

Proof. 1. $P_\alpha(a) = \alpha \cdot (\alpha \backslash a)^\downarrow \overset{55.5}{\leq} \alpha \cdot (\alpha \backslash a) \overset{9.5}{=} \alpha \cdot a$

2. $P_\alpha(P_\alpha(a)) \leq \alpha \cdot P_\alpha(a) = P_\alpha(a) = \alpha \cdot (\alpha\backslash a)^\downarrow = \alpha \cdot (\alpha\backslash a)^{\downarrow\downarrow}$
$\leq \alpha \cdot ((\alpha\backslash a)^\downarrow + \neg\alpha \cdot \top)^\downarrow = \alpha \cdot (\alpha\backslash(\alpha\backslash a)^\downarrow)^\downarrow = \alpha \cdot (\alpha\backslash(\alpha \cdot (\alpha\backslash a)^\downarrow))^\downarrow$
$= P_\alpha(P_\alpha(a))$

3. $P_\alpha(a + \neg\alpha \cdot b) = \alpha \cdot (\alpha\backslash(a + \neg\alpha \cdot b))^\downarrow = \alpha \cdot (a + \neg\alpha \cdot b + \neg\alpha \cdot \top)^\downarrow$
$= \alpha \cdot (a + \neg\alpha \cdot \top)^\downarrow = \alpha \cdot (\alpha\backslash a)^\downarrow = P_\alpha(a)$

4. $P_\alpha(a) \overset{1.}{\leq} \alpha \cdot a = 0$

□

Obviously, if there is no α-labeled link in the graph the α-subgraph is empty, which is shown algebraically by Lemma 59.4. To be able to model the decomposition of graphs into its α-subgraphs we further need the resolution identity from fuzzy theory as explained in Section 2.2. Since the other inequality holds trivially, we define:

Definition 60 (resolution). *The cut operations show* resolution *if*

$$a \leq \sum_{\alpha \in \mathcal{S}} P_\alpha(a)$$

Resolution of the cut operators is an important tool to perform practically usable calculations in the algebra. As an immediate consequence we are able to represent a completely labeled element by the sum of all its real α-cuts:

Corollary 61. $a^\uparrow = \sum_{0 \neq \alpha \in \mathcal{S}} (\alpha\backslash a)^\downarrow$

Here the previously described anomaly of discrimination level 0 can be observed. By Figure 3.4 we can see that the projection function $P_\alpha(a)$ can be considerably simplified to a restriction if α is scalar-atomic. At the same time, this simplification is a defining feature of scalar-atomicity if the cut operators show resolution.

Lemma 62. $sat(\alpha) \Leftrightarrow \forall a \in \mathcal{K}.\ P_\alpha(a) = \alpha \cdot a$

Proof. ($\Rightarrow$) Immediately from $sat(\alpha)$ and $\alpha \cdot \beta \leq \alpha$ we get $\alpha \cdot \beta = 0 \vee \alpha \cdot \beta = \alpha$. The second disjunct is equivalent to $\beta \geq \alpha$, so

$$\alpha \cdot a = \alpha \cdot \sum_{\beta \in \mathcal{S}} P_\beta(a) = \sum_{\beta \in \mathcal{S}} \alpha \cdot \beta \cdot (\beta \backslash a)^{\downarrow} = \sum_{\beta \geq \alpha} \alpha \cdot (\beta \backslash a)^{\downarrow} \leq \sum_{\beta \geq \alpha} P_\alpha(a) = P_\alpha(a)$$

The other inequality follows from Lemma 59.1.
($\Leftarrow$) Assume there is scalar β in between 0 and α, i.e. $0 < \beta < \alpha$, then $\beta = \alpha \cdot \beta = \alpha \cdot (\alpha \backslash \beta)^{\downarrow} = \alpha \cdot (\beta + \neg\alpha \cdot \top)^{\downarrow} \leq \alpha \cdot ((\beta + \neg\alpha) \cdot \top)^{\downarrow}$. By the assumption we have $\beta + \neg\alpha < \alpha + \neg\alpha = 1$ and thus $(\beta + \neg\alpha) \cdot \top \neq \top$. Now Lemma 57 implies $\beta = 0$ which is a contradiction. □

By this equivalence projections with respect to atomic scalars behave more friendly than general projections:

Corollary 63. *Assume $sat(\alpha)$ and $sat(\beta)$, then*

1. $P_\alpha(a) + P_\alpha(b) = P_\alpha(a + b)$
2. $P_\alpha(a) \cdot P_\alpha(b) = P_\alpha(a \cdot b) = a \cdot P_\alpha(b)$
3. $P_\alpha(s : a) = P_\alpha(s) : P_\alpha(a)$
4. $\alpha \backslash P_\alpha(a) = \alpha \backslash a$
5. $P_\alpha(a) + P_\beta(a) = P_{(\alpha+\beta)}(a)$

This shows closedness under join and composition. Thus, the set

$$\mathcal{K}_\alpha = \{P_\alpha(a) \mid a \in \mathcal{K}\}$$

of all elements of an atomic scalar α forms an idempotent semiring $(\mathcal{K}_\alpha, +, \cdot, 0, \alpha)$ with top. The ideal $j = P_\alpha(\top) = \alpha \cdot \top$ that corresponds to α forms the top element, which is shown by Lemma 59.1. The set $\mathcal{K}_\alpha$ is not closed under the star operation, since $1 \leq P_\alpha(a)^*$ which is not representable in the general form $P_\alpha(a)$ for arbitrary α. But we can define a new operation $\_^\star$ that completes the idempotent semiring to a Kleene algebra with top by:

$$P_\alpha(a)^\star \overset{\text{def}}{=} P_\alpha(a^*) = \alpha \cdot a^*$$

The operation satisfies the expansion and induction laws that have to hold for a star operator in this structure, which are:

$$P_\alpha(a)^\star = P_\alpha(1) + P_\alpha(a) \cdot P_\alpha(a)^\star$$
$$P_\alpha(b) + P_\alpha(a) \cdot P_\alpha(c) \leq P_\alpha(c) \Rightarrow P_\alpha(a)^\star \cdot P_\alpha(b) \leq P_\alpha(c)$$

and the symmetric versions.

Proof. We calculate:

$$\begin{aligned}
\text{a)}\quad & P_\alpha(1) + P_\alpha(a) \cdot P_\alpha(a)^\star = P_\alpha(1) + P_\alpha(a) \cdot P_\alpha(a^*) = P_\alpha(1) + P_\alpha(a \cdot a^*) \\
& \qquad = P_\alpha(1 + a \cdot a^*) = P_\alpha(a^*) = P_\alpha(a)^\star \\
\text{b)}\quad & P_\alpha(b) + P_\alpha(a) \cdot P_\alpha(c) \leq P_\alpha(c) \Leftrightarrow P_\alpha(b) + a \cdot P_\alpha(c) \leq P_\alpha(c) \\
& \qquad \Rightarrow a^* \cdot P_\alpha(b) \leq P_\alpha(c) \\
& \qquad \Leftrightarrow P_\alpha(a^*) \cdot P_\alpha(b) \leq P_\alpha(c) \\
& \qquad \Leftrightarrow P_\alpha(a)^\star \cdot P_\alpha(b) \leq P_\alpha(c)
\end{aligned}$$

□

Thus, each set $\mathcal{K}_\alpha$ together with the respective operations forms a Kleene algebra $(\mathcal{K}_\alpha, +, \cdot, 0, \alpha, \_^\star, \alpha \cdot \top)$ which is embedded in $\mathcal{K}$. They form algebraic counterparts of the set of α-subgraphs.

For scalar-atomic α we can also strengthen the idempotence law from Lemma 59.2 to projections completed by $^{\uparrow}$:

Lemma 64. *Assume sat(α), then $P_\alpha(P_\alpha(a)^\uparrow) = P_\alpha(a)$*

Proof. We have: $P_\alpha(a) = \alpha \cdot (\alpha\backslash a)^\downarrow \leq (\alpha\backslash a)^\downarrow \Leftrightarrow P_\alpha(a)^\uparrow \leq \alpha\backslash a$

$$\begin{aligned} &\Leftrightarrow P_\alpha(a)^\uparrow \leq \alpha\backslash P_\alpha(a) \\ &\Leftrightarrow \alpha \cdot P_\alpha(a)^\uparrow \leq P_\alpha(a) \\ &\Leftrightarrow P_\alpha(P_\alpha(a)^\uparrow) \leq P_\alpha(a) \end{aligned}$$

The other direction follows from idempotence of P_α with Lemma 55.5. □

3.9 Crispness

Using the cut operations from the last section we are able to characterize crisp elements in a Kleene algebra. These represent graphs only consisting of completely labeled links. Obviously, a crisp element coincides with the least crisp element it is included in. So we can define crisp elements as invariant under $^\uparrow$.

Definition 65 (crispness). *An element $a \in \mathcal{K}$ is called* crisp, *if $a^\uparrow = a$.*

Certainly, by Lemma 55.6 $a^\downarrow = a$ is an equivalent characterization. We denote the set of crisp elements by $\mathcal{C} = \{a \mid a^\uparrow = a\}$. In addition to crispness of particular elements we will also introduce a notion of crispness with respect to the whole algebra. A Kleene algebra will be called crisp if *every* element $a \in \mathcal{K}$ is crisp, which is equivalent to the condition $\mathcal{K} = \mathcal{C}$. Such crisp algebras can be defined by the structure of their scalars. A first observation shows that most of the crisp elements lie outside the set of scalars. In fact, there are exactly two crisp scalars:

Lemma 66. *The only crisp scalars are* 0 *and* 1*.*

Proof. By Lemmas 55.1 and 55.7 the elements 0 and 1 are crisp. Now suppose that $0 \neq \alpha \in \mathcal{S}$ and α crisp. Then $\alpha = \alpha^\uparrow = 1$ by Definition 53.3. □

So in every crisp KA the structure of scalars is minimal and consists exactly of the two elements 0 and 1 (disregarding the trivial algebra where $0 = 1$). But the other direction also holds if the cut operators show resolution:

Lemma 67. *A Kleene algebra is crisp if and only if* 0 *and* 1 *are the only scalars.*

Proof. First assume that 0 and 1 are the only scalars, then: $a = \sum_\alpha P_\alpha(a) = 0 \cdot (0\backslash a)^\downarrow + 1 \cdot (1\backslash a)^\downarrow = a^\downarrow$. So for every element $a = a^\downarrow = a^\uparrow$ holds. Now assume a crisp Kleene algebra. Then all elements are crisp and therefore also the scalars. By Lemma 66 they can only be 0 and 1. □

Example 68. Since α, which coincides with the identity, is scalar-atomic, the structure of scalars in the algebra $(\mathcal{K}_\alpha, +, \cdot, 0, \alpha, \_^\star, \alpha \cdot \top)$ of α-subgraphs is minimal. Therefore the algebra is crisp. This means that there is only one particular labeling of edges, which perfectly meets our intuition of α-subgraphs. □

By the bijective connection between scalars and ideals we are also able to describe a crisp algebra by the structure of its ideals. But by Definition 21 Kleene algebras with 0 and $\top$ as only ideals are exactly the simple ones. So the notions of simple and crisp Kleene algebras coincide. In all crisp algebras $^\uparrow$ and $^\downarrow$ coincide with the identity function, since the differentiating Law 53.3 gets ineffective.

By definition of crispness it is easy to see that the crisp elements of an arbitrary Kleene algebra are closed under join and composition. As the set of crisp elements also involves the constants 0, 1 and $\top$ and crisp predicates are closed under negation we observe:

Lemma 69. *The set of crisp elements $\mathcal{C} \subseteq \mathcal{K}$ together with the restricted Kleene operations forms a KAP $(\mathcal{C}, +, \cdot, 0, 1, \_^*, \top)$ with top.*

Proof. Crispness of 0,1 and $\top$ was shown in Lemma 55 and the closure properties of join and composition follow from the Galois connection and Lemma 55.9. Assume $s^\uparrow = s$, then closedness of predicates under complement is shown by:

$$\begin{aligned} s + (\neg s)^\uparrow &= s^\uparrow + (\neg s)^\uparrow = (s + \neg s)^\uparrow = 1^\uparrow = 1 \\ s \cdot (\neg s)^\uparrow &= s^\uparrow \cdot (\neg s)^\uparrow = (s^\uparrow \cdot \neg s)^\uparrow = (s \cdot \neg s)^\uparrow = 0^\uparrow = 0 \end{aligned}$$

It remains to show crispness of $(a^\uparrow)^*$. To avoid parentheses we abbreviate $(a^\uparrow)^*$ by $a^{\uparrow^*}$. One inequality is trivial by Lemma 55.5 whereas the other follows by the induction principle from:

$$\begin{aligned} &(1 + a^\uparrow \cdot (a^{\uparrow^*})^\downarrow)^\uparrow = 1 + a^\uparrow \cdot (a^{\uparrow^*})^\downarrow \leq 1 + a^\uparrow \cdot a^{\uparrow^*} = a^{\uparrow^*} \\ &\Leftrightarrow 1 + a^\uparrow \cdot (a^{\uparrow^*})^\downarrow \leq (a^{\uparrow^*})^\downarrow \\ &\Rightarrow a^{\uparrow^*} \leq (a^{\uparrow^*})^\downarrow \\ &\Leftrightarrow (a^{\uparrow^*})^\uparrow \leq a^{\uparrow^*} \end{aligned}$$

$\square$

Another interesting point is the interaction between domain and codomain and the cut operators. Again we will focus on domain, as the laws for codomain follow symmetrically. We can show that application of $\ulcorner\_$ and $^\uparrow$ can be commuted whereas for $\ulcorner\_$ and $^\downarrow$ in general only an inequality can be shown. The reason for this is that the domain operator in a sense sums up the labels of all outgoing links. Thus, after this operation there may arise new completely labeled loops, which are not removed by the down operator. For predicates this summing up disappears and so equality follows. The up operator just completes arbitrary existing links which does not make any difference if performed before or after the application of domain.

Lemma 70. *1.* $\ulcorner(a^\uparrow) = (\ulcorner a)^\uparrow$
2. $\ulcorner(a^\uparrow)$ *and* $\ulcorner(a^\downarrow)$ *are crisp, i.e.* $(\ulcorner(a^\uparrow))^\uparrow = \ulcorner(a^\uparrow)$ *and* $(\ulcorner(a^\downarrow))^\uparrow = \ulcorner(a^\downarrow)$
3. $\ulcorner(a^\downarrow) \leq (\ulcorner a)^\downarrow$ *but* $\ulcorner(s^\downarrow) = (\ulcorner s)^\downarrow$

Proof.

1. $\ulcorner(a^\uparrow) = \ulcorner(a^\uparrow \cdot \top) = \ulcorner((a \cdot \top)^\uparrow) \overset{25}{=} \ulcorner((\ulcorner a \cdot \top)^\uparrow) = \ulcorner((\ulcorner a)^\uparrow \cdot \top) = (\ulcorner a)^\uparrow$
2. $(\ulcorner(a^\uparrow))^\uparrow \overset{1.}{=} \ulcorner(a^{\uparrow\uparrow}) \overset{55.4}{=} \ulcorner(a^\uparrow)$ and $(\ulcorner(a^\downarrow))^\uparrow \overset{1.}{=} \ulcorner(a^{\downarrow\uparrow}) \overset{55.2}{=} \ulcorner(a^\downarrow)$
3. $\ulcorner(a^\downarrow) \overset{2.}{=} (\ulcorner(a^\downarrow))^\downarrow \leq (\ulcorner a)^\downarrow$ and $(\ulcorner s)^\downarrow = s^\downarrow = \ulcorner(s^\downarrow) \cdot s^\downarrow \leq \ulcorner(s^\downarrow)$ $\square$

This implies immediately that the composition laws for crisp elements from Definition 53.2 and Lemma 55.9 can be lifted to images.

Corollary 71. *1.* $(s : a^\uparrow)^\uparrow = s^\uparrow : a^\uparrow = (s^\uparrow : a)^\uparrow$
2. $(s : a^\downarrow)^\uparrow = s^\uparrow : a^\downarrow$ *and* $(s^\downarrow : a)^\uparrow = s^\downarrow : a^\uparrow$

For the complement of crisp predicates we are only able to show inequalities.

Lemma 72. *1.* $\neg(s^\uparrow) \leq (\neg s)^\uparrow$
2. $(\neg s)^\downarrow \leq \neg(s^\downarrow)$

Proof. We only show the first proposition. The second is proven symmetrically. $s \leq s^\uparrow \Leftrightarrow \neg(s^\uparrow) \leq \neg s \Rightarrow (\neg(s^\uparrow))^\uparrow \leq (\neg s)^\uparrow$ and by $\neg(s^\uparrow) \leq (\neg(s^\uparrow))^\uparrow$ the proposition follows. $\square$

The other directions do not hold. Assume for example a scalar $\alpha \neq 0, 1$. Then $\neg(\alpha^\uparrow) = \neg 1 = 0$ and $(\neg\alpha)^\uparrow = 1$, which shows the claim for the first expression. A counterexample for the second one follows symmetrically from Lemma 57.

Chapter 4

Pointer Kleene Algebra

Based on the extensions to model labeled graphs in Kleene algebra presented in the last chapter, this part of the thesis is concerned with higher level operations on pointer structures. Following [Möl97a] we define a pointer algebra that is able to express reachability conditions, allocation and modification of pointer structures. Localization properties are shown that allow reasoning about the effects of changes, perform simplifications and verify correctness of data structure implementations.

4.1 Operations and Notations

To fix the formal basis for the subsequent observations we introduce the structure of a pointer Kleene algebra:

Definition 73 (pointer Kleene algebra). *A* pointer Kleene algebra *(PKA) is a KAD with top that shows subordination of domain and codomain and has two cut operators* $^{\uparrow}$ *and* $^{\downarrow}$ *as defined in Section 3.8 that show resolution.*

In the sequel we will assume our algebraic environment to be a pointer Kleene algebra.

From Section 3.2 we know that predicates can be used to represent sets of nodes in graphs. Each member of the set is identified with a loop on this node. For labeled graphs there may be several distinctly labeled loops on the same node. Hence, to get a unique representation of sets of nodes in arbitrary Kleene algebras we use completely labeled loops. These elements are modeled by crisp predicates and can play the part of addresses in pointer structures:

Definition 74. *A crisp element* $m \leq 1$ *is called an* address.

In the sequel we will use letters m, n, o to denote addresses. As addresses are crisp predicates, by Lemma 69 they form a Boolean sublattice of the predicates. This shows that they really behave like sets of nodes.

Corollary 75. *The set* $\mathcal{A} = \mathcal{P} \cap \mathcal{C}$ *of addresses forms a Boolean lattice.*

Similar to atomic scalars we mean atoms in this Boolean lattice when we refer to atomic addresses. The predicate *cat* will be used to characterize such *crisp atomic* elements. Obviously, this represents a singleton set of nodes. For local reasoning starting from an address we often have to advance exactly one step further along particular labeled edges. We will abbreviate the successor of address m under selector α in graph a by the functional notation $a_\alpha(m)$. As the result normally should again represent a set of nodes, we additionally define $\hat{a}_\alpha(m)$ that yields addresses.

Definition 76 (restricted image).

1. $a_\alpha(m) \overset{\text{def}}{=} m : P_\alpha(a)$
2. $\hat{a}_\alpha(m) \overset{\text{def}}{=} a_\alpha(m)^\uparrow$

Considering the anomaly of the α-cut we can simplify $\hat{a}_\alpha(m)$ to $m : (\alpha\backslash a)^\downarrow$ if $m \neq 0$. To change such successor relations selectively we define a ministore representing particular links between two addresses. This can be used to overwrite parts of a store with a new link structure. We assume that such a ministore relates all nodes of its domain m to all nodes of its codomain n by α-labeled links. Therefore we derive the definition of a ministore by projecting the α-subgraph of the complete graph represented by $\top$ and restricting it to m on its domain and to n on its codomain:

$$m \cdot P_\alpha(\top) \cdot n = m \cdot \alpha \cdot (\alpha\backslash\top)^\downarrow \cdot n = m \cdot \alpha \cdot \top^\downarrow \cdot n = m \cdot \alpha \cdot \top \cdot n$$

This construct certainly is equal to first restricting the graph and then projecting to the α-subgraph:

Corollary 77. $m \cdot P_\alpha(\top) \cdot n = P_\alpha(m \cdot \top \cdot n)$

Definition 78 (ministore). *Let $m, n \in \mathcal{K}$ be addresses and $\alpha \in \mathcal{S}$ a selector. Then we call the element $(m \xrightarrow{\alpha} n) \overset{\text{def}}{=} P_\alpha(m \cdot \top \cdot n)$ an* α-ministore *with source addresses m and target addresses n.*

If addresses m and n are atomic and $sat(\alpha)$ holds, the ministore $(m \xrightarrow{\alpha} n)$ models exactly a single α-labeled pointer link from address m to n. Obviously, the domain of an α-ministore equals the set of starting addresses restricted to α and the codomain for target addresses symmetrically. The image of starting address m under the ministore $(m \xrightarrow{\alpha} n)$ should yield target node n in the α-subgraph.

Lemma 79. *Let $\alpha \in \mathcal{S}$ a selector and $m, n \in \mathcal{K}$ be addresses*

1. $\ulcorner(m \xrightarrow{\alpha} n) = m \cdot \alpha$
2. $(m \xrightarrow{\alpha} n)\urcorner = \alpha \cdot n$
3. $m : (m \xrightarrow{\alpha} n) = \alpha \cdot n$
4. $\neg m \cdot ((m \xrightarrow{\alpha} n) \mid a) = \neg m \cdot a$
5. $\alpha \neq 0 \Rightarrow m : (\alpha \cdot ((m \xrightarrow{\alpha} n) \mid a))^\uparrow = n$

Proof.

1. $m \cdot \alpha = \ulcorner(m \cdot \alpha \cdot \top) \overset{55.2}{=} \ulcorner(m \cdot \alpha \cdot \top \cdot n \cdot \top) \overset{25}{=} \ulcorner(\ulcorner(m \cdot \alpha \cdot \top \cdot n) \cdot \top) = \ulcorner(m \cdot \alpha \cdot \top \cdot n)$
2. Symmetrically to 1.
3. $m : (m \xrightarrow{\alpha} n) = (m \cdot (m \cdot \alpha \cdot \top \cdot n))\urcorner = (m \cdot \alpha \cdot \top \cdot n)\urcorner = \alpha \cdot n$
4. $\neg m \cdot ((m \xrightarrow{\alpha} n) \mid a) = \neg m \cdot (m \xrightarrow{\alpha} n) + \neg m \cdot \neg(m \cdot \alpha) \cdot a = \neg m \cdot a$
5. $m : (\alpha \cdot ((m \xrightarrow{\alpha} n) \mid a))^\uparrow = m : ((m \xrightarrow{\alpha} n) + \alpha \cdot \neg m \cdot a)^\uparrow$
$$= (m : (m \xrightarrow{\alpha} n))^\uparrow = (\alpha \cdot n)^\uparrow = n$$

□

We can show that the change of a β-labeled link does not affect an α-subgraph if $\alpha \neq \beta$. On the other hand, a change of an α-labeled link is only reflected in the α-subgraph:

Lemma 80. *Assume $sat(\alpha)$ and $sat(\beta)$, then*

1. $P_\alpha((m \xrightarrow{\alpha} n) \mid a) = (m \xrightarrow{\alpha} n) + \neg m \cdot P_\alpha(a)$
2. $\alpha \cdot \beta = 0 \Rightarrow P_\alpha((m \xrightarrow{\beta} n) \mid a) = P_\alpha(a)$

Proof. 1.
$$\begin{aligned}
&P_\alpha((m \overset{\alpha}{\to} n) \mid a)\\
&= P_\alpha((m \overset{\alpha}{\to} n) + \neg(m \cdot \alpha) \cdot a)\\
&= \alpha \cdot (m \overset{\alpha}{\to} n) + \alpha \cdot \neg(m \cdot \alpha) \cdot a\\
&= (m \overset{\alpha}{\to} n) + \neg m \cdot \alpha \cdot a\\
&= (m \overset{\alpha}{\to} n) + \neg m \cdot P_\alpha(a)
\end{aligned}$$

2.
$$\begin{aligned}
&P_\alpha((m \overset{\beta}{\to} n) \mid a)\\
&= P_\alpha((m \overset{\beta}{\to} n) + \neg(m \cdot \beta) \cdot a)\\
&= \alpha \cdot (m \overset{\beta}{\to} n) + \alpha \cdot \neg(m \cdot \beta) \cdot a\\
&= \alpha \cdot \neg m \cdot a + \alpha \cdot \neg\beta \cdot a\\
&= P_\alpha(a)
\end{aligned}$$

□

To get an intuitive notation of pointer structures and operations we introduce some syntactic sugar. In the sequel we will assume pointer structures $p = (m, a)$ to consist of an address m (a crisp predicate) and a store a (an arbitrary Kleene element). Throughout this thesis we will denote such pointer structures by variables p, q, and r. For convenience we introduce the access functions

$$ptr(m, a) = m \qquad\qquad sto(m, a) = a$$

to project entry point and store of a pointer structure. Following traditional programming languages we use a dot to denote selection of a particular record field:

Definition 81 (selection). $(m, a).\alpha \overset{\text{def}}{=} (a_\alpha(m), a)$

Selection returns a pointer structure with the root node of field α as entry.

Similar to selection we will use a notation for changing the α-successor of the record stored at address m to point to n. This operation is based on the selective update operator presented in Section 3.6.

Definition 82 (update). $(m, a).\alpha := (n, b) \overset{\text{def}}{=} (m, (m \overset{\alpha}{\to} n) \mid b)$

For the interaction between update and selection we can show that modification of the α-subgraph in a pointer structure has no effect on the selection of the β successor.

Lemma 83. *Assume $sat(\alpha)$, $sat(\beta)$, $\alpha \cdot \beta = 0$, $p = (m, a)$, $q = (n, b)$, and $r = (m, b)$, then*
$$ptr((p.\alpha := q).\beta) = ptr(r.\beta)$$

Proof. Setting $c = (m \overset{\alpha}{\to} n) \mid b$ we get

$$ptr((p.\alpha := q).\beta) = ptr((c_\beta(m), c) = m : P_\beta((m \overset{\alpha}{\to} n) \mid b) \overset{80.2}{=} m : P_\beta(b) = ptr(r.\beta)$$

□

We are also in the position to show that overwriting of an α-successor with the original value lets the store untouched, e.g. $(p.\alpha := p.\alpha) = p$ with $p = (m, a)$. Nevertheless, by the more abstract model we are only able to show observational equivalence of the two terms.

Lemma 84. *Assume $sat(\alpha)$, then $(m \overset{\alpha}{\to} a_\alpha(m)) \mid a \equiv a$*

Proof. Let m, n be addresses and m crisp atomic, then

$$m \cdot n \leq m \Rightarrow m \cdot n = 0 \vee m \cdot n = m$$

So we handle two cases:

$m \cdot n = 0$**:** By assumption $n : (m \overset{\alpha}{\to} \hat{a}_\alpha(m)) = 0$ and $n : ((m \cdot \alpha) \cdot a) = 0$.

$$\begin{aligned} n : ((m \overset{\alpha}{\to} \hat{a}_\alpha(m)) \mid a) &= n : (m \overset{\alpha}{\to} \hat{a}_\alpha(m)) + n : (\neg(m \cdot \alpha) \cdot a) \\ &= 0 + n : (\neg(m \cdot \alpha) \cdot a) + n : ((m \cdot \alpha) \cdot a) \\ &= n : a \end{aligned}$$

$m \cdot n = m$**:** With a first auxiliary calculation

$$\begin{aligned} n : (m \overset{\alpha}{\to} \hat{a}_\alpha(m)) &= (m \overset{\alpha}{\to} \hat{a}_\alpha(m))^\urcorner = P_\alpha(\hat{a}_\alpha(m)) = P_\alpha(m : P_\alpha(a)^\uparrow) \\ &= P_\alpha(m) : P_\alpha(P_\alpha(a)^\uparrow) = P_\alpha(m) : P_\alpha(a) = P_\alpha(m \cdot a)^\urcorner \\ &= P_\alpha(n \cdot m \cdot a)^\urcorner = (n \cdot P_\alpha(m) \cdot a)^\urcorner = n : (P_\alpha(m) \cdot a) \end{aligned}$$

we can show: $n : ((m \overset{\alpha}{\to} \hat{a}_\alpha(m)) \mid a)$

$$\begin{aligned} &= n : (m \overset{\alpha}{\to} \hat{a}_\alpha(m)) + n : (\neg(m \cdot \alpha) \cdot a) \\ &= n : ((m \cdot \alpha) \cdot a) + n : (\neg(m \cdot \alpha) \cdot a) \\ &= n : a \end{aligned}$$

□

4.2 Reachability

The complexity of reasoning about pointer structures mainly is caused by complicated connections in the graph. Knowledge about associated nodes can simplify calculations and propositions considerably. In contrast to the image operator that returns the direct successors, we are now interested in all nodes reachable by any number of steps. This leads us to:

Definition 85 (reach). *The addresses reachable from node m in store a are*

$$reach(m, a) \overset{\text{def}}{=} m : (a^\uparrow)^*$$

By using the completed graph $a^\uparrow$ for the calculation we really get every somehow reachable node independent of the link labels. To avoid unnecessary parentheses in the sequel we abbreviate $(a^\uparrow)^*$ by $a^{\uparrow^*}$ like in the proof of Lemma 69. Monotonicity of *reach* in both arguments follows immediately. Distributivity over join in the first argument is directly inherited from the image operator. The same inheritance relation holds for image induction and we can formulate a reach induction law:

Corollary 86. $m + n : a^\uparrow \leq n \Rightarrow reach(m, a) \leq n$

As an immediate consequence of strictness of codomain and $1 \leq a^*$ we obtain that *reach* is strict in its first argument:

$$reach(m, a) = 0 \Leftrightarrow m = 0$$

By using the defining laws of star (Axioms 3.12 and 3.13) recursion equations for *reach* can be derived.

Lemma 87. *1.* $reach(m, a) = m + reach(m, a) : a^\uparrow$
2. $reach(m, a) = m + reach(m : a^\uparrow, a)$

Proof. 1. $reach(m,a) = m : a^{\uparrow^*} = m : (1 + a^{\uparrow^*} \cdot a^\uparrow) = m + m : (a^{\uparrow^*} \cdot a^\uparrow)$
$= m + (m : a^{\uparrow^*}) : a^\uparrow = m + reach(m,a) : a^\uparrow$

2. Symmetrically to 1.

□

The first rule defines $reach(m,a)$ as fixed point of the equation $x = m + x : a^\uparrow$ and the second immediately can be implemented as recursive calculation of reachable nodes in a labeled graph. The standard efficiency improvements to proceed only with the part of the store not yet considered follows from *reach* induction:

Lemma 88. $reach(m,a) = m + reach(m : a^\uparrow, \neg m \cdot a)$

Proof. By monotonicity and Lemma 87.2 $\geq$ is trivial. The claim follows from Corollary 86 and case distinction with:

$$\begin{aligned}
&m + (m + reach(m : a^\uparrow, \neg m \cdot a)) : a^\uparrow \\
&= m + m : a^\uparrow + reach(m : a^\uparrow, \neg m \cdot a) : (m \cdot a^\uparrow) + reach(m : a^\uparrow, \neg m \cdot a) : (\neg m \cdot a^\uparrow) \\
&\leq m + m : a^\uparrow + 1 : (m \cdot a^\uparrow) + reach(m : a^\uparrow, \neg m \cdot a) : (\neg m \cdot a^\uparrow) \\
&\overset{41.8}{=} m + m : a^\uparrow + reach(m : a^\uparrow, \neg m \cdot a) : (\neg m \cdot a^\uparrow) \\
&\overset{87.1}{=} m + reach(m : a^\uparrow, \neg m \cdot a)
\end{aligned}$$

□

The star decomposition rule from Lemma 2 to resolve the iteration of two joined elements directly can be lifted to *reach*:

Lemma 89.

$$reach(m, a + b) = reach(reach(m,a), b \cdot a^{\uparrow^*}) = reach(reach(m, a^{\uparrow^*} \cdot b), a)$$

Proof. We only show the first equality since the second follows symmetrically:

$$\begin{aligned}
reach(m, a + b) &= m : (a^\uparrow + b^\uparrow)^* = m : (a^{\uparrow^*} \cdot (b^\uparrow \cdot a^{\uparrow^*})^*) \\
&= (m : a^{\uparrow^*}) : (b \cdot a^{\uparrow^*})^{\uparrow^*} = reach(reach(m,a), b \cdot a^{\uparrow^*})
\end{aligned}$$

□

For several tasks it is also necessary to calculate the nodes reachable only by proper paths in the graph. This is the set of addresses reachable by at least one step. Thus, in contrast to *reach* starting addresses are not in this set by default. They only appear if they are reachable from any direct successor as for example in a cyclic structure. We will see later how this can be used to characterize acyclic graphs. The operation *sreach* is introduced by:

Definition 90 (sreach). *The set of addresses reachable from m by at least one step in store a is*

$$sreach(m,a) \overset{\text{def}}{=} m : (a^\uparrow)^+$$

Obviously, *reach* consists of *sreach* and the set of starting addresses and for the interaction of these two operators we can show:

Lemma 91. *1.* $sreach(m,a) = reach(m : a^\uparrow, a)$

2. $reach(m,a) = m + sreach(m,a)$
Immediately: $sreach(m,a) \leq reach(m,a)$

3. $reach(sreach(m,a), a) = sreach(m,a)$

4. $sreach(reach(m,a), a) = sreach(m,a)$

$$\begin{tikzcd}
\mathcal{A} \times \mathcal{K} \arrow[r, "reach_\alpha"] \arrow[d, "P_\alpha \times P_\alpha"'] & \mathcal{A} \arrow[d, "P_\alpha"] \\
\mathcal{A}_\alpha \times \mathcal{K}_\alpha \arrow[r, "reach"] & \mathcal{A}_\alpha
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathcal{A} \times \mathcal{K} \arrow[r, "sreach_\alpha"] \arrow[d, "P_\alpha \times P_\alpha"'] & \mathcal{A} \arrow[d, "P_\alpha"] \\
\mathcal{A}_\alpha \times \mathcal{K}_\alpha \arrow[r, "sreach"] & \mathcal{A}_\alpha
\end{tikzcd}$$

Figure 4.1: Reachability observations in subgraphs

Proof. 1. $reach(m : a^\uparrow, a) = (m : a^\uparrow) : a^{\uparrow^*} = m : a^{\uparrow^+} = sreach(m, a)$

2. Immediate from Lemma 87.1.

3. $$\begin{aligned} reach(sreach(m,a),a) &= reach(reach(m : a^\uparrow, a), a) = reach(m : a^\uparrow, a) \\ &= sreach(m,a) \end{aligned}$$

4. $$\begin{aligned} sreach(reach(m,a),a) &= (m : a^{\uparrow^*}) : a^{\uparrow^+} = m : (a^{\uparrow^*} \cdot a^{\uparrow^+}) = m : a^{\uparrow^+} \\ &= sreach(m,a) \end{aligned}$$

□

In contrast, we define restricted versions that only are concerned with reachability along particularly α-labeled links. If α is an atom, we can use the simplification for the projection function $P_\alpha(a)$ for links labeled at least by α.

Definition 92 **(α-reach).**

$$\begin{aligned} reach_\alpha(m,a) &\overset{\text{def}}{=} reach(m, P_\alpha(a)) \overset{sat(\alpha)}{=} reach(m, \alpha \cdot a) \\ sreach_\alpha(m,a) &\overset{\text{def}}{=} sreach(m, P_\alpha(a)) \overset{sat(\alpha)}{=} sreach(m, \alpha \cdot a) \end{aligned}$$

Certainly, these operations should be equivalent to reachability observations in the (crisp) algebra $(\mathcal{K}_\alpha, +, \cdot, 0, \alpha, \_^\star, \alpha \cdot \top)$ of α-subgraphs and the diagrams in Figure 4.1 should commute. Indeed we can show:

$$\begin{aligned} P_\alpha(reach_\alpha(m,a)) &= P_\alpha(m : P_\alpha(a)^{\uparrow^*}) \\ &= P_\alpha(m) : P_\alpha(P_\alpha(a)^{\uparrow^*}) \\ &= P_\alpha(m) : P_\alpha(P_\alpha(a)^\uparrow)^\star \\ &= P_\alpha(m) : P_\alpha(a)^\star \end{aligned}$$

Remember that the algebra of α-subgraphs is crisp. This is the reason that $^\uparrow$ and $^\downarrow$ have no effect and therefore the last term equals *reach* in this subalgebra. The proof for *sreach* works similar.

As *reach* and *sreach* should yield all addresses reachable along particular paths from a set of starting addresses in a given pointer structure, we have to assure that they really return sets of addresses. Remember that addresses are modeled by crisp predicates. Since *reach* and *sreach* are defined as images, it is immediate that they yield predicates. So we are left with showing crispness of the calculated result:

Lemma 93. *The operators reach and sreach return addresses, i.e.*

$$reach(m,a)^\uparrow = reach(m,a) \text{ and } sreach(m,a)^\uparrow = sreach(m,a)$$

Proof. By Lemma 91.1 it suffices to show the claim for *reach*. With additivity of $^\uparrow$, Corollary 71, Lemma 55.5 and Lemma 87 it follows that:

$$(m + reach(m,a)^\downarrow : a^\uparrow)^\uparrow \leq m + reach(m,a) : a^\uparrow = reach(m,a)$$

which by Galois connection and *reach* induction implies:

$$\begin{aligned} & m + reach(m,a)^{\downarrow} : a^{\uparrow} \leq reach(m,a)^{\downarrow} \\ & \Rightarrow reach(m,a) \leq reach(m,a)^{\downarrow} \\ & \Leftrightarrow reach(m,a)^{\uparrow} \leq reach(m,a) \end{aligned}$$

The other inequality is trivially true by 55.5. □

After advancing one step the entry address will not be reachable anymore if there is no way back from its successors as for example in a cyclic pointer structure. So obviously the set of reachable nodes in such a successor pointer structure cannot grow.

Lemma 94. $reach((m,a).\alpha) \leq reach(m,a)$

Proof. $reach((m,a).\alpha) \leq reach(m : a^{\uparrow}, a) = sreach(m,a) \leq reach(m,a)$ □

Similar to *reach* we introduce the operator *from* that calculates the subgraph built from all reachable nodes. The yielded structure contains all the links and nodes that are reachable from the given set of starting addresses. This is a sort of projection onto the *live part* of the store.

Definition 95 (from). *The part of store a which is reachable from m is:*

$$from(m,a) \overset{\text{def}}{=} reach(m,a) \cdot a$$

The original definition of *from* in [Möl97a] returns a pointer structure with the same entry as the argument. In contrast to the pointer algebra given there we abstract from this definition and just focus on the store. Similar to $reach_{\alpha}$ we define a restricted variant of *from* that only considers selectors coming from set α. This can be based on the previous definition of $reach_{\alpha}$:

Definition 96 (α-from). $from_{\alpha}(m,a) \overset{\text{def}}{=} reach_{\alpha}(m,a) \cdot a$

The returned subgraph consists of all α-reachable nodes together with all links between them which also entails the non-α-labeled ones.

We can show that an equality proposition about the live part of two stores is stronger than the same statement about the set of reachable addresses:

Lemma 97. $from(m,a) = from(m,b) \Rightarrow reach(m,a) = reach(m,b)$

Proof. The claim follows immediately from Lemma 87.1 as we know that *reach* can be expressed by *from*: $reach(m,a) = m + reach(m,a) : a^{\uparrow} = m + (from(m,a))^{\urcorner\uparrow}$ □

By investigation of the iterated reachability operators *reach* and *from* we can show that they form closure and interior operators respectively. Idempotence of *reach* is achieved by an application of locality of images which is entailed in the rules of interaction between *reach* and *sreach*. Additionally, we can show that:

Lemma 98. *reach is a closure operator, viz*

1. *Extensive:* $m \leq reach(m,a)$
2. *Idempotent:* $reach(reach(m,a),a) = reach(m,a)$
3. *Monotone:* $m \leq n \Rightarrow reach(m,a) \leq reach(n,a)$

Proof. 1. Follows immediately from 87.1.

2. $$\begin{aligned} reach(reach(m,a),a) &= reach(m,a) + sreach(reach(m,a),a) \\ &= reach(m,a) + sreach(m,a) \\ &= reach(m,a) \end{aligned}$$
3. By monotonicity of all involved operators. □

Idempotence of $from$ is a little bit more tricky. To be able to reason about the star of $reach$ we have to use the reach induction principle:

Lemma 99. *$from$ is an interior operator, viz*

1. *Reductive:* $from(m,a) \leq a$
2. *Idempotent:* $from(m, from(m,a)) = from(m,a)$
3. *Monotone:* $a \leq b \Rightarrow from(m,a) \leq from(m,b)$

Proof. 1. Trivial

2. Let $b = reach(m,a) \cdot a$, then $reach(m,b) \leq reach(m,a)$ is trivial and the opposite direction follows from $reach$ induction and case distinction by:

$$\begin{aligned} & m + reach(m,b) : a^{\uparrow} \\ &= m + (reach(m,b) \cdot reach(m,a)) : a^{\uparrow} + (reach(m,b) \cdot \neg reach(m,a)) : a^{\uparrow} \\ &\leq m + reach(m,b) : (reach(m,a) \cdot a^{\uparrow}) + (reach(m,a) \cdot \neg reach(m,a)) : a^{\uparrow} \\ &= m + reach(m,b) : b^{\uparrow} \\ &= reach(m,b) \end{aligned}$$

Hence: $$\begin{aligned} from(m,b) &= reach(m,b) \cdot b = reach(m,b) \cdot reach(m,a) \cdot a \\ &= reach(m,a) \cdot a = from(m,a) \end{aligned}$$

3. Follows immediately from monotonicity of $reach$. □

By this we can show that $from$ indeed is the projection to a subgraph and does not change the connections in the live part of the store. So for example the same addresses are reachable as the ones that were before.

Lemma 100. $reach(m, from(m,a)) = reach(m,a)$

Proof. By Lemma 97 the claim can be reduced to idempotency of $from$. □

4.3 Non-reachability

If we can specify the allocated addresses in a store, we are able to define a complementary operator to $reach$ that calculates all used but not reachable records in a pointer structure. This is an abstract description of garbage nodes in the store - cells that were in use but are no longer reachable from the roots. For this purpose we define $recs$ that returns all addresses of records where a pointer link starts from.

Definition 101 (allocated records). $recs(a) \overset{\text{def}}{=} (\ulcorner a)^{\uparrow}$

By Lemma 70 we know that $\ulcorner\_$ and $^{\uparrow}$ can be commuted. So the definition is equivalent to $recs(a) = \ulcorner(a^{\uparrow})$ which we will use if appropriate. Distributivity over joins is directly inherited from the operators involved. Further simplification in the calculation of $recs$ can be achieved by

Lemma 102.

1. $recs(\ulcorner a) = recs(a)$
2. $recs(\ulcorner b \cdot a) \leq recs(b)$
3. $recs(b \mid a) = recs(b) + recs(a)$
4. $recs(m \cdot a) = m \cdot recs(a)$
5. $\alpha \neq 0 \Rightarrow recs((m \xrightarrow{\alpha} n)) = m$
6. $\alpha \neq 0 \Rightarrow recs((m \xrightarrow{\alpha} n) \mid a)$
   $= m + \neg m \cdot recs(a)$

Proof. 1. $recs(\ulcorner a) = (\ulcorner(\ulcorner a))^{\uparrow} = (\ulcorner a)^{\uparrow} = recs(a)$

2. $recs(\ulcorner b \cdot a) \overset{1.}{=} recs(\ulcorner(\ulcorner b \cdot a)) = recs(\ulcorner b \cdot \ulcorner a) \leq recs(\ulcorner b) \overset{1.}{=} recs(b)$

3. $recs(b \mid a) = recs(b) + recs(\neg\ulcorner b \cdot a) \overset{2.}{=} recs(b) + recs(\ulcorner b \cdot a) + recs(\neg\ulcorner b \cdot a)$
   $= recs(b) + recs(a)$

4. $recs(m \cdot a) = (\ulcorner(m \cdot a))^{\uparrow} = (m \cdot \ulcorner a)^{\uparrow} = m \cdot (\ulcorner a)^{\uparrow} = m \cdot recs(a)$

5. $recs((m \xrightarrow{\alpha} n)) = (\ulcorner(m \xrightarrow{\alpha} n))^{\uparrow} = (m \cdot \alpha)^{\uparrow} = m$

6. $recs((m \xrightarrow{\alpha} n) \mid a) = recs((m \xrightarrow{\alpha} n)) + \neg m \cdot recs(a) \overset{5.}{=} m + \neg m \cdot recs(a)$ □

Symmetrically we define the set of all addresses links point to by:

Definition 103 (link targets). $targets(a) \overset{\text{def}}{=} (a\urcorner)^{\uparrow}$

For symmetry reasons all the laws that hold for *recs* hold correspondingly. As direct consequence from Lemma 102.4 follows that the allocated records in the live part of the store are all reachable nodes where links start from:

Corollary 104. $recs(from(m, a)) = reach(m, a) \cdot recs(a)$

Reachability gets a trivial task if we know that the starting nodes are only non-allocated records:

Lemma 105. *Assume* $m \cdot recs(a) = 0$*, then*

$$sreach(m, a) = 0 \qquad reach(m, a) = m \qquad from(m, a) = m \cdot a$$

Proof. a) $sreach(m, a) = reach(m : a^{\uparrow}, a) = reach(0, a) = 0$
b) $reach(m, a) = m + sreach(m, a) = m$
c) $from(m, a) = reach(m, a) \cdot a = m \cdot a$

□

With these laws we can further improve the calculation of *reach* defined in Section 4.2 to a more efficient version by considering only not yet visited addresses as new starting points. This rule directly can be lifted to *from*:

Lemma 106. *reach and from can be calculated efficiently by:*

1. $reach(m, a) = m + reach((m : a^{\uparrow}) \cdot \neg m, \neg m \cdot a)$
2. $from(m, a) = m \cdot a + from((m : a^{\uparrow}) \cdot \neg m, \neg m \cdot a)$

Proof. 1. $reach(m, a)$

$$\overset{88}{=} m + reach((m : a^{\uparrow}) \cdot m, \neg m \cdot a) + reach((m : a^{\uparrow}) \cdot \neg m, \neg m \cdot a)$$
$$\overset{105}{=} m + reach((m : a^{\uparrow}) \cdot \neg m, \neg m \cdot a)$$

2. $from(m, a) \overset{1.}{=} m \cdot a + reach((m : a^{\uparrow}) \cdot \neg m, \neg m \cdot a) \cdot a$

$$= m \cdot a + reach((m : a^{\uparrow}) \cdot \neg m, \neg m \cdot a) \cdot m \cdot a$$
$$+ reach((m : a^{\uparrow}) \cdot \neg m, \neg m \cdot a) \cdot \neg m \cdot a$$
$$= m \cdot a + reach((m : a^{\uparrow}) \cdot \neg m, \neg m \cdot a) \cdot \neg m \cdot a$$
$$= m \cdot a + from((m : a^{\uparrow}) \cdot \neg m, \neg m \cdot a)$$

□

These rules can immediately be implemented as efficient calculation of the reachable nodes and the live part in a labeled graph. To derive and prove the calculation law for $from$ based on a calculus of maps in [BMM91] several pages were needed. The solution presented there does not even directly handle different selectors. This again shows the succinctness of the Kleene-algebraic approach.

By use of $recs$ to describe all allocated records we are able to define the used but non-reachable records.

Definition 107 (noreach). $noreach(m, a) \overset{\text{def}}{=} recs(a) \cdot \neg reach(m, a)$

The operator $noreach$ returns the set of addresses from which links start but which are not reachable from the given root nodes. We observe that $noreach$ is anti-monotone in its first argument by monotonicity of $reach$ and anti-monotonicity of the complement. Similar to $reach_\alpha$ and $from_\alpha$ we can define $noreach_\alpha$ to express non-reachability via particular labels.

Definition 108 (α-noreach). $noreach_\alpha \overset{\text{def}}{=} recs(a) \cdot \neg reach_\alpha(m, a)$

Corollary 104 can be used to write $noreach$ in an alternative form based on $from$ instead of $reach$.

Lemma 109. $noreach(m, a) = recs(a) \cdot \neg recs(from(m, a))$

Proof.
$$\begin{aligned} recs(a) \cdot \neg recs(from(m, a)) &\overset{104}{=} recs(a) \cdot \neg(reach(m, a) \cdot recs(a)) \\ &= recs(a) \cdot \neg reach(m, a) + recs(a) \cdot \neg recs(a) \\ &= noreach(m, a) \end{aligned}$$
□

By anti-monotonicity of $noreach$ we can lift Lemma 94. Evidently, we potentially increase the number of non-reachable addresses if we start the calculation at an arbitrary successor node.

Lemma 110. $noreach(m, a) \leq noreach((m, a).\alpha)$

Proof. Immediate from Lemma 94 □

More sophisticatedly we can show that changing the α-successor of address m yields the same set of allocated but non-α-reachable records as in the structure of its new successor and the store restricted to addresses not in m [1].

Lemma 111. $sat(\alpha) \Rightarrow noreach_\alpha((m, a).\alpha := (n, a)) = noreach_\alpha(n, \neg m \cdot a)$

Proof.

$noreach_\alpha((m, a).\alpha := (n, a))$

$=$ {[definition of assignment]}

$noreach_\alpha(m, (m \xrightarrow{\alpha} n) \mid a)$

$=$ {[definition of $noreach_\alpha$ and $reach_\alpha$]}

$recs((m \xrightarrow{\alpha} n) \mid a) \cdot \neg reach(m, \alpha \cdot ((m \xrightarrow{\alpha} n) \mid a))$

$=$ {[Lemmas 102.3, 102.5 and 88]}

$(m + recs(a)) \cdot \neg(m + reach(m : (\alpha \cdot ((m \xrightarrow{\alpha} n) \mid a))^\uparrow, \neg m \cdot \alpha \cdot ((m \xrightarrow{\alpha} n) \mid a)))$

[1] Incidentally we noticed a copy error on the right hand side of this lemma in [Möl99a] as we tried to prove it in the form given there. The same lemma was noted correctly in the former articles [Möl97a] and [Möl97b]. Nevertheless, in all these papers the restriction to a single selector is not mentioned.

$=$ $\{\![$ Boolean algebra and definition of $|$ $]\!\}$

$\neg m \cdot recs(a) \cdot \neg reach(m : (\alpha \cdot ((m \overset{\alpha}{\to} n) \mid a))^{\uparrow}, \alpha \cdot \neg m \cdot a)$

$=$ $\{\![$ Lemmas 102.4 and 79.5 $]\!\}$

$recs(\neg m \cdot a) \cdot \neg reach(n, \alpha \cdot \neg m \cdot a)$

$=$ $\{\![$ definition of $reach_\alpha$ and $noreach_\alpha$ $]\!\}$

$noreach_\alpha(n, \neg m \cdot a)$ □

We should explain this pictorially in Figure 4.2. To abbreviate we will denote the

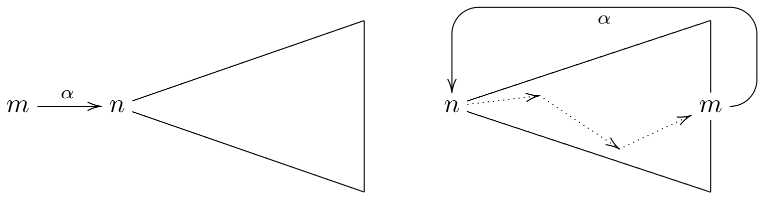

Figure 4.2: Explanation of Lemma 111

left-hand side of the equation by (lhs) and similarly the right-hand side by (rhs). The new link $(m \overset{\alpha}{\to} n)$ in (lhs) adds m to the set of allocated records of store a and the restriction $\neg m \cdot a$ in (rhs) removes m from this set. Thus, it suffices to consider address m and to show that it is neither in (lhs) nor in (rhs). We distinguish two cases. First assume m is not reachable from n (see left picture of Figure 4.2). Then the set of reachable nodes from m in (lhs) consists of m itself together with the nodes reachable from n. Thus, m is not included in (lhs) and the reachable parts of (lhs) and (rhs) exactly differ by m. Now assume m is reachable from n (see right picture of Figure 4.2). Then the reachable parts of (lhs) and (rhs) are equal and both contain m. Also in this case m is neither in (lhs) nor in (rhs).

To be able to express reachability conditions succinctly we additionally define reachability predicates. They should evaluate to true if a particular node is reachable from a pointer structure (m, a) and otherwise return false. In contrast to the point-wise approach in [Möl97a] we are only able to model sets of nodes by predicates. This abstraction is taken into account by distinguishing three cases:

Definition 112.

1. *Every node in n is reachable:* $(m, a) \vdash n \overset{\text{def}}{\Leftrightarrow} n \leq reach(m, a)$
2. *Some nodes in n are reachable:* $(m, a) \vdash n \overset{\text{def}}{\Leftrightarrow} 0 < reach(m, a) \cdot n < n$
3. *None of the nodes in n is reachable:* $(m, a) \nvdash n \overset{\text{def}}{\Leftrightarrow} reach(m, a) \cdot n = 0$

If n is an atomic predicate $\vdash$ can never be fulfilled. In this case we have the point-wise view. Each address element represents exactly one node and $\vdash$ and $\nvdash$ are complementary reachability predicates. Both are downward closed in their second argument. Transitivity of $\vdash$ follows immediately from its definition and idempotence of *reach*. Non-reachability by $\nvdash$ and *noreach* are strongly connected. The validity of predicate $(m, a) \nvdash n$ can be deduced from non-reachability. Nevertheless, they are not equivalent as could be presumed, since *noreach* respects allocated records only. But they coincide if n is in the set of allocated records. Thus we can give the equivalent characterization:

Lemma 113. $n \leq noreach(m, a) \Leftrightarrow n \leq recs(a) \wedge (m, a) \nvdash n$

Proof. The two conjuncts on the right side follow from:

$$n \le noreach(m,a) = recs(a) \cdot \neg reach(m,a) \le recs(a) \quad \text{and}$$
$$n \cdot reach(m,a) \le recs(a) \cdot \neg reach(m,a) \cdot reach(m,a) = recs(a) \cdot 0 = 0$$

whereas the opposite direction is shown by case distinction:

$$n = n \cdot reach(m,a) + n \cdot \neg reach(m,a) \le recs(a) \cdot \neg reach(m,a) = noreach(m,a)$$

□

4.4 Localization

Most of the expressions modifying pointer structures can be simplified if effects of changes can be shown to take place only in particular parts of the store. Such locality can be expressed by reachability and non-reachability conditions that have to hold. So if we can show that the records of store b are not reachable from a pointer structure (m,a), we do not have to expect side-effects on b of changing (m,a). First we show some simple consequences from reachability constraints:

Lemma 114. *Assume that* $(m,a) \nvdash recs(b)$ *which by definition is equivalent to* $reach(m,a) \cdot recs(b) = 0$*, then*

1. $reach(m,a) \cdot {\ulcorner b} = 0$
2. $reach(m,a) \cdot b = 0$
3. $reach(m,a) \cdot b^{\uparrow} = 0$

Proof. 1. $reach(m,a) \cdot {\ulcorner b} \le reach(m,a) \cdot {\ulcorner (b^{\uparrow})} = reach(m,a) \cdot recs(b) = 0$
2. $reach(m,a) \cdot b = reach(m,a) \cdot {\ulcorner b} \cdot b = 0$
3. $reach(m,a) \cdot b^{\uparrow} = reach(m,a) \cdot {\ulcorner (b^{\uparrow})} \cdot b^{\uparrow} = reach(m,a) \cdot recs(b) \cdot b^{\uparrow} = 0$ □

By strictness of codomain all these laws can be lifted from composition to image. Using these prerequisites, the assumption of $(m,a) \nvdash recs(b)$ gives us a lot of information for dealing with pointer structures. So for example in the calculation of reachable addresses we can completely ignore whole regions of the store and concentrate on important parts. This is particularly helpful for stores that are built from two parts joined together.

Lemma 115 (Localization I). *Assume* $(m,a) \nvdash recs(b)$*, then*

1. $reach(m, a+b) = reach(m,a)$
2. $reach(m, b \mid a) = reach(m,a)$

Proof. 1. $reach(m, a+b) \overset{89}{=} reach(reach(m,a), b \cdot a^{\uparrow *}) \overset{105}{=} reach(m,a)$
2. $(m,a) \nvdash recs(b)$ implies $(m, \neg{\ulcorner b} \cdot a) \nvdash recs(b)$ and $(m, \neg{\ulcorner b} \cdot a) \nvdash recs({\ulcorner b})$, thus:

$$reach(m, b \mid a) = reach(m, b + \neg{\ulcorner b} \cdot a) \overset{1.}{=} reach(m, \neg{\ulcorner b} \cdot a)$$
$$\overset{1.}{=} reach(m, {\ulcorner b} \cdot a + \neg{\ulcorner b} \cdot a) = reach(m,a)$$

□

As the definition of *from* is based on *reach* the previous lemma can be lifted to *from*. So under the respective conditions the calculation of the live part of a composed store can also be simplified:

Lemma 116 (Localization II). *Assume* $(m,a) \nvdash recs(b)$*, then*

1. $from(m, a+b) = from(m,a)$
2. $from(m, b \mid a) = from(m,a)$

Proof. 1. $$\begin{aligned} from(m, a+b) &= reach(m, a+b) \cdot (a+b) \\ &\overset{115.1}{=} reach(m,a) \cdot a + reach(m,a) \cdot b \\ &\overset{114.2}{=} reach(m,a) \cdot a \\ &= from(m,a) \end{aligned}$$

2. $$\begin{aligned} from(m, b \mid a) &= reach(m, b \mid a) \cdot (b \mid a) \\ &\overset{115.2}{=} reach(m,a) \cdot b + reach(m,a) \cdot (\neg\ulcorner b \cdot a) \\ &\overset{114.1}{=} 0 + reach(m,a) \cdot (\neg\ulcorner b \cdot a) + reach(m,a) \cdot (\ulcorner b \cdot a) \\ &= from(m,a) \end{aligned}$$

□

In particular, with pointer structures $p = (m,a), q = (n,b)$, and $r = (m,b)$ we can show some of the most sophisticated rules that are needed to derive algorithms on pointer structures with selective updates.

Lemma 117. *Assume $sat(\alpha)$ and $sat(\beta)$ then*

1. $q \not\vdash ptr(p) \Rightarrow from((p.\alpha := q).\alpha) = from(q)$
2. $\alpha \cdot \beta = 0 \wedge r.\beta \not\vdash ptr(p) \Rightarrow from((p.\alpha := q).\beta) = from(r.\beta)$

Proof. Let $c = (m \overset{\alpha}{\to} n) \mid b$, then

$$\begin{aligned} & from((p.\alpha := q).\alpha) = from(q) \\ \Leftrightarrow\ & from(((m,a).\alpha := (n,b)).\alpha) = from(q) \\ \Leftrightarrow\ & from((m,c).\alpha) = from(q) \\ \Leftrightarrow\ & from(c_\alpha(m), c) = from(q) \\ \Leftrightarrow\ & from(n,c) = from(q) \\ \overset{116.2}{\Leftarrow}\ & (n,b) \not\vdash m \end{aligned}$$

The second proposition is shown similarly. For the proof one needs to show that $c_\beta(m) = b_\beta(m)$ which follows from $\alpha \cdot \beta = 0$ and Lemma 80.2. □

The essence of the first equation immediately is accepted by everyone. But it is often forgotten that changing the α-successor of p may also influence q. This is the case exactly if p is in the structure reachable from q. So in fact it is a matter of localization which is expressed.

4.5 Meaningful Pointer Structures

We now want to show how correctness properties of pointer structures can be expressed with respect to the modeled data structures. To have an anchor for inductively defined data types we need a special address that serves as model for nil pointers. In contrast to [HJ99] who proposed to model it by a node with all links pointing to itself we more intuitively choose nil to be a special address that no link starts from. This better reflects the property that it can not be dereferenced.

Definition 118 (nil). *The special value $\diamond \in \mathcal{A}$ is an address that has no image under any store, i.e. $\diamond : a = 0$ for all stores a*

In the sequel we assume that all stores used fulfill this requirement and use it for proofs if necessary. We can also show that the definition intuitively is correct, as it implies that $\diamond$ is not in the set of allocated addresses:

Lemma 119. $recs(a) \cdot \diamond = 0$

Proof. $\diamond : a = 0 \Leftrightarrow \diamond \cdot a = 0 \Leftrightarrow \diamond \cdot {}^{\ulcorner}a = 0 \Rightarrow recs(a) \cdot \diamond = {}^{\ulcorner}(a^{\uparrow}) \cdot \diamond = ({}^{\ulcorner}a \cdot \diamond)^{\uparrow} = 0$ □

The definition implies that no proper addresses are reachable from $\diamond$:

Corollary 120. $sreach(\diamond, a) = 0 \qquad reach(\diamond, a) = \diamond \qquad from(\diamond, a) = 0$

We also can give the set of terminal nodes which are important for example in automata theory by calculating all reachable nodes that no further link starts from:

Definition 121 (final nodes). $final(m, a) \overset{\text{def}}{=} reach(m, a) \cdot \neg recs(a)$

The intuitive interpretation of final nodes - that they have no successors - immediately follows:

Corollary 122. $final(m, a) : a = final(m, a) : a^{\uparrow} = 0$

Obviously, from terminal nodes no further addresses are reachable and $final$ is an idempotent operator

Lemma 123. *1.* $sreach(final(m, a), a) = 0$
2. $reach(final(m, a), a) = final(m, a)$
3. $final(reach(m, a), a) = final(m, a)$
4. $final(m, from(m, a)) = final(m, a)$
5. $final(final(m, a), a) = final(m, a)$

Proof. 1. $sreach(final(m, a), a) \overset{91.1}{=} reach(final(m, a) : a^{\uparrow}, a) \overset{122}{=} reach(0, a) = 0$
2. $reach(final(m, a), a) = final(m, a) + sreach(final(m, a), a) \overset{1.}{=} final(m, a)$
3. $final(reach(m, a), a) = reach(reach(m, a), a) \cdot \neg recs(a)$
$$= reach(m, a) \cdot \neg recs(a) = final(m, a)$$
4. $final(m, from(m, a)) = reach(m, from(m, a)) \cdot \neg recs(from(m, a))$
$$= reach(m, a) \cdot \neg recs(a) = final(m, a)$$
5. $final(final(m, a), a) = reach(final(m, a), a) \cdot \neg recs(a)$
$$\overset{2.}{=} final(m, a) \cdot \neg recs(a) = final(m, a)$$
□

Since dynamic pointer implementations of non-recursive data types with a fixed size, like e.g. tuples, are senseless, such representations are mainly used for recursive data structures. Meaningful implementations of these types should be terminated by nil pointers. With the use of $final$ we can define a predicate that serves as a sort of invariant for operators on pointer structures. This expresses that nil should be the only final state in a pointer structure:

Definition 124 (meaningful). *The store a is a* meaningful *representation of inductively defined pointer data structures if for all available entries m of recursive data types the condition $final(m, a) \leq \diamond$ is satisfied.*

Additionally one can demand that the store is closed. This excludes dangling links by permitting only allocated records and nil as possible link targets.

Definition 125 (closedness). *We say store a is* closed *if*

$$targets(a) \leq recs(a) + \diamond$$

In a closed store there are no links that point to non-allocated records. Nevertheless, this does not assure that a link points to an address that represents an object of the desired type. This has to be checked by the type system and should not be the concern of this thesis.

With respect to the store we can also define the set of sources and sinks of the graph. These are the addresses where pointer-links only start from or where they just end. With this we can define the inner nodes that have entering and leaving edges.

Definition 126 (source, sink and inner nodes).

$$\begin{aligned}
src(a) &\overset{\text{def}}{=} recs(a) \cdot \neg targets(a) \\
snk(a) &\overset{\text{def}}{=} targets(a) \cdot \neg recs(a) \\
inner(a) &\overset{\text{def}}{=} recs(a) \cdot \neg src(a) = targets(a) \cdot \neg snk(a) = recs(a) \cdot targets(a)
\end{aligned}$$

4.6 Acyclicity and Sharing

A higher concept that is based on reachability is acyclicity of graphs and pointer structures. The standard way in relation algebra to define acyclicity is

Definition 127 (relational acyclicity (RA)).

$$acyclic_{RA}(a) \overset{\text{def}}{\Leftrightarrow} a^+ \sqcap 1 = 0$$

As there is no meet operation in EKA we have to find a different characterization. One possibility is to switch to observational equivalence. This means that the image of an arbitrary address under both sides has to be equal. So we work in the set of predicates where we have a meet (namely composition) at hand.

Definition 128 (observational acyclicity (OA)).

$$acyclic_{OA}(a) \overset{\text{def}}{\Leftrightarrow} \forall m.\ m \cdot (m : a^+) = 0$$

This definition seems quite natural, since it follows from a non-reachability proposition:

$$\forall m.\ (m : a^{\uparrow}, a) \not\vdash m \Rightarrow acyclic_{OA}(a)$$

But by choosing $m = 1$ we immediately get that $acyclic_{OA}(a)$ is equivalent to $a = 0$! As address elements model sets of nodes, a logical step would be to switch to atomic address elements representing only single nodes. This also solves the problem $m = 1$ for algebras with a non-trivial predicate structure.

Definition 129 (atomic observational acyclicity (AOA)).

$$acyclic_{AOA}(a) \overset{\text{def}}{\Leftrightarrow} \forall cat(m).\ m \cdot (m : a^+) = 0$$

An alternative characterization comes from graph theory. There one says that a graph is *progressively finite* [SS93] if all paths in the graph have finite length. So there are no infinite chains which means that the graph is Noetherian or well-founded.

Definition 130 (progressively finite (PF)).

$$acyclic_{PF}(a) \overset{\text{def}}{\Leftrightarrow} \forall m.\ m \leq m : a^+ \Rightarrow m = 0$$

For finite graphs it is well-known that progressive finiteness and acyclicity are equivalent. So we can also define progressive finiteness for atoms.

Definition 131 (atomic progressively finite (APF)).

$$acyclic_{APF}(a) \overset{\text{def}}{\Leftrightarrow} \forall cat(m).\ m \leq m : a^+ \Rightarrow m = 0$$

For these characterization candidates for acyclicity we can show the following relations:

$$\begin{tikzcd}
& RA & \\
OA \arrow[ur, Rightarrow, tail] \arrow[rr, Rightarrow, maps to] \arrow[d, Rightarrow, maps to] & & PF \arrow[ul, Rightarrow, tail] \arrow[d, Rightarrow] \arrow[dll, Rightarrow] \\
AOA \arrow[rr, Leftrightarrow] & & APF
\end{tikzcd}$$

Here an arrow with an open tail stands for an unknown connection between these two characterizations in the respective direction. A closed tail ($\mapsto$) means that the characterization on this side is strictly stronger than the one pointed to.

Proof. The unifying counterexample that proves that OA neither follows from RA, AOA nor PF is the graph a with two nodes and only a single connection between them:

$$\begin{tikzcd}
① \arrow[r] & ②
\end{tikzcd}$$

We choose $m = \{1, 2\}$, then $m : a^+ = \{2\}$ and $m \cdot (m : a^+) = \{2\}$. By this, OA does not hold, but PF holds for all m, AOA holds for all atomic m and RA trivially holds. A counter example for $RA \Rightarrow PF$ can be found in PAT, the algebra of paths. Assume $P = \{aa\}$ a path in PAT, then $P^+ = \{aa, aaa, \ldots\}$ and therefore $P^+ \sqcap 1 = 0$. But $0 \neq \{a\} \subseteq \{a\} : P^+ = \{a\}$. For finite graphs RA and PF are equivalent (see [SS93]).
The implications from OA and PF to the respective atomic versions are trivial. The other implications are shown as follows:

$OA \Rightarrow PF$: Assume OA and $m \leq m : a^+$, then $m = m \cdot m \leq m \cdot (m : a^+) = 0$
$AOA \Rightarrow APF$: Similar to $OA \Rightarrow PF$ with additional assumption $cat(m)$.
$PF \Rightarrow AOA$,$APF \Rightarrow AOA$: $cat(m) \Rightarrow m \cdot (m : a^+) = m \vee m \cdot (m : a^+) = 0$. The first term is equivalent to $m \leq m : a^+$ and by PF/APF we get $m = 0$.
$OA \Rightarrow RA$: OA holds for all addresses, so also for $m = 1$, then $0 = m \cdot (m : a^+) = 1 \cdot (1 : a^+) = (a^+)^\urcorner \Rightarrow a^+ = 0 \Rightarrow a^+ \sqcap 1 = 0$
$PF \Rightarrow RA$: $a^+ \sqcap 1 = 0$ follows from PF by:

$$a^+ \sqcap 1 = (a^+ \sqcap 1)^\urcorner = ((a^+ \sqcap 1) \cdot (a^+ \sqcap 1))^\urcorner \leq ((a^+ \sqcap 1) \cdot a^+)^\urcorner = (a^+ \sqcap 1) : a^+$$

In the sequel we will use characterization PF as definition of acyclicity as OA is too strong and RA is not expressible in Kleene algebra. Obviously, if a graph is free of cycles all its subgraphs also are. Thus, acyclicity is a downward closed predicate:

Lemma 132. *$acyclic(a) \Rightarrow \forall b \leq a.\ acyclic(b)$*

Proof. Assume $m \leq m : b^+$, then $m \leq m : b^+ \leq m : a^+ \Rightarrow m = 0$ □

Acyclicity of $a^\uparrow$ now can be expressed using *sreach*:

$$m \leq sreach(m, a) \Rightarrow m = 0$$

With the additional assumption of acyclicity we can show stronger localization properties of pointer algebra operations like in Section 4.3, since non-reachability conditions follow. So one can reason about reachability after having performed one step:

Lemma 133. *$acyclic(a^\uparrow) \wedge m \neq 0 \Rightarrow reach(m : a, a) < reach(m, a)$*

Proof. Obviously, $reach(m : a, a) \leq sreach(m, a) \leq reach(m, a)$ holds. So we assume $reach(m, a) \leq reach(m : a, a)$, which implies:

$$\begin{aligned}
&reach(m, a) \leq reach(m : a, a) \leq sreach(m, a) = sreach(reach(m, a), a)\\
&\Rightarrow reach(m, a) = 0\\
&\Leftrightarrow m = 0
\end{aligned}$$

This is a contradiction to $m \neq 0$ and thus shows the claim. □

Standard consequences from acyclicity can also be proven. Assume an element n is reachable from m in more than one step. It follows that m is not in the part of the store reachable from n if the store is acyclic. In contrast to the corresponding lemmas in [Möl97a] we always have to demand, that the involved address is not 0. This is a consequence of the set representation of addresses and ensures non-emptyness.

Lemma 134. $n \neq 0 \wedge n \leq sreach(m, a) \wedge acyclic(a^{\uparrow}) \Rightarrow \neg((n, a) \vdash m)$

Proof. Assume $(n, a) \vdash m$ which is equivalent to $m \leq reach(n, a)$, then

$$n \leq sreach(m, a) \leq sreach(reach(n, a), a) = sreach(n, a)$$

It follows by acyclicity that $n = 0$, which contradicts the precondition. □

If m is an atomic address the implication simplifies to $(n, a) \nvdash m$. By this observation specialized versions of the localization properties for single selective updates in Lemma 117 follow from acyclicity:

Lemma 135 (Localization III). *Assume $cat(m)$, $a_\beta(m) \neq 0$ and $acyclic(a^{\uparrow})$, then*

1. $from((p.\alpha := p.\beta).\alpha) = from(p.\beta)$
2. $\alpha \cdot \beta = 0 \Rightarrow from((p.\alpha := p.\gamma).\beta) = from(p.\beta)$

Proof. Using Lemma 117 for both claims $(a_\beta(m), a) \nvdash m$ needs to be shown which by $cat(m)$ and Lemma 134 follows from the preconditions and $a_\beta(m) \leq sreach(m, a)$. Hence, we are left to show: $a_\beta(m) \leq m : a^{\uparrow} \leq reach(m : a^{\uparrow}, a) = sreach(m, a)$ □

Using the reachability operator from Section 4.2 we are able to define a predicate that expresses the sharing of parts of two pointer structures. As $\diamond$ is used as terminator for all linked data structures it plays a special rôle. We say that two pointer structures do not share any parts if the intersection of their reachable addresses is at most $\diamond$.

Definition 136 (sharing). $\neg sharing(m, n, a) \overset{\text{def}}{\Leftrightarrow} reach(m, a) \cdot reach(n, a) \leq \diamond$

As an immediate consequence from Lemma 94 it follows that if two pointer structures have no nodes in common the successor structures also do not show sharing:

Lemma 137. $\neg sharing(m, n, a) \Rightarrow \neg sharing(a_\alpha(m), n, a)$

Proof. $reach(a_\alpha(m), a) \cdot reach(n, a) \leq reach(m, a) \cdot reach(n, a) \leq \diamond$ □

By calculations in our algebra we observed, that the following lemma from [Möl97a] in fact does not need acyclicity as a precondition.

Lemma 138. $n \leq sreach(m, a)$ *implies* $\forall o.\ \neg sharing(m, o, a) \Rightarrow \neg sharing(n, o, a)$

Proof. $reach(n, a) \leq reach(sreach(m, a), a) = sreach(m, a) \leq reach(m, a)$ and thus $reach(m, a) \cdot reach(o, a) \leq \diamond \Rightarrow reach(n, a) \cdot reach(o, a) \leq \diamond$ □

Chapter 5

Pointer Algorithms

The main contribution of this chapter presents the process of correct construction of pointer algorithms. As the object to be examined we use a library of list processing functions. Following Möller [Möl97a] these are specified in a functional programming style and transformed to algorithms working on pointer structures. The purpose of this chapter is to show the pointer Kleene algebra at work and to close the gap between recursive algorithms that arise from the development process and concrete imperative implementations.

5.1 A Formal Derivation Method

The method to derive correct pointer algorithms is based on previous work by B. Möller [Möl97a]. Pointer Kleene algebra forms the formal basis over which to express properties about pointer structures and prove correctness of transformation steps. The method is intended to use the simplicity of denoting and proving correctness of algorithms in functional programming languages. This abstract level is the starting point in a transformation process to formally derive concrete algorithms on pointer structures. We establish the connection between concrete and abstract level by partial abstraction functions as proposed by Hoare [Hoa72]. The implementation of an object is represented by a pointer structure (m, L) with a single, atomic entry m and store L. An abstraction function maps such a pointer structure into an abstract data type representation. The standard abstraction function for singly linked lists with selectors hd to reference the member values and tl to link the structure is for example:

$$\begin{aligned} list(p) = \mathsf{if}\, ptr(p) = \diamond\ &\mathsf{then}\, [] \\ &\mathsf{else}\, p.hd : list(p.tl) \end{aligned}$$

Specification of pointer implementations f_p of a given functional description f works by requiring that f_p mimics the abstract input/output behaviour of f on the pointer structure level. Given abstraction functions $F, F_1, \ldots, F_i$ the implementation of f_p is specified by the equation

$$f(F_1(p_1), \ldots, F_i(p_i)) = F(f_p(p))$$

Here p_j with $1 \leq j \leq i$ denotes the projection from a multi-entry pointer structure $(m_1, \ldots, m_i, L)$ which can be seen as representation of an i-tuple to the pointer structure (m_j, L) representing one particular data type. The whole specification process can be depicted as follows:

$$\begin{tikzcd}
\boxed{\begin{array}{c}\text{functional}\\ \text{data objects}\\ (O_1,\ldots,O_i)\end{array}} \arrow[rr, "\begin{array}{c}\text{functional}\\ \text{specification}\end{array}", "f"'] & & \boxed{\begin{array}{c}\text{functional}\\ \text{data object}\\ O\end{array}} \\
\boxed{\begin{array}{c}\text{pointer structure}\\ (m_1,\ldots,m_i,L)\end{array}} \arrow[u, "F_1,\ldots,F_i"'] \arrow[rr, "\text{pointer algorithm}", "f_p"'] & & \boxed{\begin{array}{c}\text{pointer structure}\\ (m,L')\end{array}} \arrow[u, "F"']
\end{tikzcd}$$

To derive a pointer implementation f_p from this specification one tries to transform the expression $f(F_1(p_1),\ldots,F_i(p_i))$ by equational reasoning into an expression $F(E)$ such that E does not contain an abstraction function anymore. Then we can define f_p by setting $f_p(p) = E$. The derivation can be seen as a step-by-step refinement to a deterministic function. Since an abstract specification entails several distinct implementations, this is a decision process towards the desired program. There may be several concrete pointer algorithms satisfying the equation. The required functionality is achieved by restricting the input/output behaviour by suitable predicates. A natural condition is for example that allocated but non-reachable records cannot be made reachable by f_p. This can be expressed by the following predicate:

Definition 139 (norea).

$$f_p \in \mathit{norea} \overset{\text{def}}{\Leftrightarrow} \forall (p,q) \in f_p.\ \mathit{noreach}(p) \leq \mathit{noreach}(q)$$

We should note that this does not prevent an algorithm from allocating new cells, since *norea* refers only to previously used records.

To avoid magically constructed abstract objects we assume abstraction functions to be *reasonable*. This prevents for example the use of random generators or similar non-deterministic concepts in the abstraction process. An abstraction function is called reasonable if equality of the reachable parts of two pointer structures implies equal abstractions. To abbreviate this equality we define the equivalence relation $\sim_F$ by:

$$p \sim_F q \overset{\text{def}}{\Leftrightarrow} F(p) = F(q)$$

Definition 140 (reasonable). *F is* reasonable *if*

$$\mathit{from}_{s(F)}(p) = \mathit{from}_{s(F)}(q) \Rightarrow p \sim_F q$$

Here $s(F)$ denotes the scalar that describes the set of field selectors used by the abstraction function F, e.g. *hd* and *tl* in *list*. Obviously, *list* is a reasonable abstration function. By definition of reasonability we are able to transfer Lemma 117 to reasonable abstraction functions:

Corollary 141. *Assume F to be a reasonable abstraction function and pointer structures $p = (m,a)$, $q = (n,b)$, and $r = (m,b)$, then*

1. $q \nvdash \mathit{ptr}(p) \Rightarrow (p.\alpha := q).\alpha \sim_F q$
2. $\alpha \cdot \beta = 0 \wedge r.\beta \nvdash \mathit{ptr}(p) \Rightarrow (p.\alpha := q).\beta \sim_F r.\beta$

Since normally we have no knowledge about reachability conditions of arbitrary pointer structures that arise in the transformation process, we have to derive them from higher concepts like acyclicity or absence of sharing:

Lemma 142. *Assume $a = \mathit{sto}(p) = \mathit{sto}(q)$ and $q' \in f_p(q)$, then*

$$\begin{aligned} f_p \in \mathit{norea} \wedge \mathit{ptr}(p) \cdot \diamond = 0 \wedge \mathit{ptr}(p) \leq \mathit{recs}(a) \wedge \neg \mathit{sharing}(\mathit{ptr}(p), \mathit{ptr}(q), a) \\ \Rightarrow q' \nvdash \mathit{ptr}(p) \end{aligned}$$

Proof.

$\neg sharing(ptr(p), ptr(q))$

$\Leftrightarrow$ $\{\!|$ definition of *sharing* $|\!\}$

$reach(p) \cdot reach(q) \leq \diamond$

$\Rightarrow$ $\{\!|$ $ptr(p) \leq reach(p)$, $ptr(p) \cdot \diamond = 0$ $|\!\}$

$ptr(p) \cdot reach(q) = 0$

$\Leftrightarrow$ $\{\!|$ definition of $\nvdash$ $|\!\}$

$q \nvdash ptr(p)$

$\Leftrightarrow$ $\{\!|$ $ptr(p) \leq recs(a)$ and Lemma 113 $|\!\}$

$ptr(p) \leq noreach(q)$

$\Rightarrow$ $\{\!|$ $f_p \in norea$ $|\!\}$

$ptr(p) \leq noreach(q')$

$\Rightarrow$ $\{\!|$ Lemma 113 $|\!\}$

$q' \nvdash ptr(p)$

With these prerequisites we are able to derive pointer algorithms from functional implementations. This will be shown at the example of list concatenation specified by:

```
cat []     ys = ys
cat (x:xs) ys = x : cat xs ys
```

We can now derive a pointer implementation of `cat` by reasoning about the two cases of its definition:

Case $m = \diamond$**:**

$cat\ list(p)\ list(q)$

$=$ $\{\!|$ definition of *list* $|\!\}$

$cat\ []\ list(q)$

$=$ $\{\!|$ definition of *cat* $|\!\}$

$list(q)$

Thus, we can choose $cat_p(\diamond, n, L) = (n, L)$. If the first list is not empty we calculate:

Case $m \neq \diamond$ **:**

$cat\ list(p)\ list(q)$

$=$ $\{\!|$ definition of *list* $|\!\}$

$cat\ (p.hd : list(p.tl))\ list(q)$

$=$ $\{\!|$ definition of *cat* $|\!\}$

$p.hd : cat\ list(p.tl)\ list(q)$

$=$ $\{\!|$ choose an arbitrary $q' \in cat_p(L_{tl}(m), n, L)$ $|\!\}$

$p.hd : list(q')$

$=$ $\{\!|$ set $r = p.tl := q'$, Lemma 83 $|\!\}$

$r.hd : list(q')$

$= \quad \{\![$ Corollary 141.1 and Lemma 142 $]\!\}$

$\quad r.hd : list(r.tl)$

$= \quad \{\![$ definition of $list$ $]\!\}$

$\quad list(r)$

By resubstitution of r and composing both cases we get the following pointer algorithm under the condition that both input lists do not share any parts and the first list argument does not show any cycles:

$$\boxed{\begin{array}{l} cat_p(m,n,L) = \textsf{if}\ m \neq \diamond\ \textsf{then}\ p.tl := cat_p(L_{tl}(m),n,L) \\ \qquad\qquad\qquad\qquad\qquad\qquad \textsf{else}\ q \end{array}}$$

By abbreviating

$$\begin{array}{llll} K(m,n,L) \overset{\text{def}}{=} (L_{tl}(m),n,L) & B(m,n,L) \overset{\text{def}}{=} m \neq \diamond & \phi_\alpha(u,v) \overset{\text{def}}{=} v.\alpha := u \\ H(m,n,L) \overset{\text{def}}{=} (n,L) & E(m,n,L) \overset{\text{def}}{=} (m,L) \end{array}$$

We get a standardized form of this algorithm that will be used for schematic transformations in later sections:

$$\boxed{\begin{array}{l} f(x) = \textsf{if}\ B(x)\ \ \textsf{then}\ \phi(f(K(x)),E(x)) \\ \qquad\qquad\qquad\quad \textsf{else}\ H(x) \end{array}}$$

Function ϕ is used to encapsulate the selective update operation. We assume, that no other function except ϕ modifies the store, i.e. for all appearing functions F we have:

$$sto(F(x)) = sto(x)$$

K is the function used to advance in the data structure. Similarly to `cat` we are able to derive pointer implementations for other list processing functions. Some of them that will serve as examples are presented in Appendix A.2. Further examples can be found in [Ehm03].

5.2 From Recursion to Iteration

For implementation issues the method presented in Section 5.1 has one major drawback. Since functional specification works recursively, the derivation process also yields recursive algorithms. But manipulating pointers is a highly imperative concept and recursive algorithms are quite inefficient in execution on a conventional computer architecture. To get programs with sufficient performance that immediately can be fed into a computer we have to make one more step from recursion to iteration. As target language we assume a subset of an imperative language consisting of loops, conditionals and concurrent assignments. Most contemporary compilers do not support concurrent assignments but it is well-known how to resolve them using auxiliary variables. Thus, such an extension does not enhance the expressiveness of a language but helps to simplify reasoning about complicated things anyway. Rather than considering each transformation task independently and therefore be faced with the same problems over and over again, we derive a universally applicable rule for transferring the recursive algorithms into an imperative world. This is achieved by considering the most general function pattern that can arise from the derivation process. The abstract transformation scheme solves the task comprehensively, since all reasonable results are comprised.

As we have seen, all the evolving functions are built from case distinctions with different behaviours. We assume that in each of these branches there is only one appearance of the function itself. This makes sense, since the structure potentially can be changed by one of these recursive calls. So we would face problems with aliasing, get side-effects and the evaluation order would play a significant rôle if there are multiple calls. The form of these branches can syntactically be classified into three groups. If the function symbol f itself does not occur in the branch we have a termination case, since there is no further recursive call. If f is the outermost function symbol we have a tail recursive call which is used to ignore the parameters of the current recursion level. If the actual parameters should not be dismissed we need a gluing function to compose the partial results. For all pointer structure processing algorithms this is the selective update operator ϕ. It is used to walk through the pointer structure as well as for changing links. In the first case ϕ overwrites the store with links that already are present and thus performs no changes at all. This seemingly complicated concept is a consequence of the bottom-up style by which functional recursive algorithms compile their results. So the third case is a linear recursive call where ϕ is the outermost function symbol and f appears as parameter inside.

We can show that all branches of the same form can be fused together into one. For several alternative termination cases this is an easy task. We just use sequential conjunction && which is defined by:

Definition 143 (sequential logical operators).

$$\begin{aligned} B \mathbin{\&\&} C &= \mathsf{if}\ B\ \mathsf{then}\ C\ \mathsf{else\ false} \\ B \mathbin{||} C &= \mathsf{if}\ B\ \mathsf{then\ true\ else}\ C \end{aligned}$$

Assume $H_0, \ldots, H_n$ are all termination cases and M only consists of case distinctions between tail and linear recursive calls. Then we immediately get:

$$\boxed{\begin{array}{l} f(x) = \mathsf{if}\ C_0(x)\ \ \mathsf{then\ if}\ C_1(x)\ \ \mathsf{then}\ \ldots \mathsf{if}\ C_n(x)\ \ \mathsf{then}\ M(x) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ H_n(x) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad \vdots \qquad\qquad\qquad \mathsf{else}\ H_1(x) \\ \qquad\qquad\qquad\quad \mathsf{else}\ H_0(x) \\ \hline \qquad\qquad\qquad\qquad\qquad\qquad \updownarrow \\ \hline f(x) = \mathsf{if}\ C_0(x) \mathbin{\&\&} C_1(x) \ldots \mathbin{\&\&} C_n(x)\ \ \mathsf{then}\ M(x) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ \mathsf{if}\ \neg C_0(x)\ \mathsf{then}\ H_0(x) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{elsif}\ \neg C_1(x)\ \mathsf{then}\ H_1(x) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{elsif}\ \neg C_{n-1}(x)\ \mathsf{then}\ H_{n-1}(x) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ H_n(x) \end{array}}$$

By abbreviating the condition of the if -statement and summarizing the else part into one function symbol we are left with exactly one terminating branch.

Similarly, multiple linear recursive branches with different arguments can be merged by using the conditional operator _ ? _ : _ known from contemporary programming languages as abbreviation. We just show the transformation for two branches:

$$\boxed{\begin{array}{l} f(x) = \mathsf{if}\ B(x)\ \ \mathsf{then\ if}\ C(x)\ \ \mathsf{then}\ \phi(f(K_0(x)), E_0(x)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \phi(f(K_1(x)), E_1(x)) \\ \qquad\qquad\qquad\quad \mathsf{else}\ H(x) \\ \hline \qquad\qquad\qquad\qquad\qquad\qquad \updownarrow \\ \hline f(x) = \mathsf{if}\ B(x)\ \ \mathsf{then}\ \phi(f(C(x)\ ?\ K_0(x) : K_1(x)), C(x)\ ?\ E_0(x) : E_1(x)) \\ \qquad\qquad\qquad\quad \mathsf{else}\ H(x) \end{array}}$$

Certainly, this transformation is valid only if $C(x)$ has no side-effects, as for the tests $C(x)$ is evaluated twice. But this is no problem, since all these functions come from the functional derivation process, they satisfy referential transparency. The same method can be applied to tail recursive branches which leads us to the following most general function pattern (MGFP) that can evolve from the derivation process:

$$\begin{array}{l} f(x) = \mathsf{if}\, B(x)\ \ \mathsf{then\, if}\, C(x)\ \ \mathsf{then}\, f(K(x)) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\, \phi(f(K(x)), E(x)) \\ \qquad\qquad\qquad \mathsf{else}\, H(x) \end{array}$$

Most general function pattern

By the previously described method all other patterns can be reduced to this one. The scheme comprises two special cases. If we set $C(x)$ to true we get a purely linear recursive function and by setting $C(x)$ to false we get a tail recursive algorithm. Such tail recursive functions directly can be transformed into a while-loop by the well-known standard transformation scheme [Par90]:

```
$f(x) = \mathsf{if}\, B(x)\ \ \mathsf{then}\, f(K(x))$
                   $\mathsf{else}\, H(x)$
──────────────────$\updownarrow$──────────────────
$f(x) = \mathsf{var}\, vx := x$
        $\mathsf{while}\, B(vx)\, \mathsf{do}\, vx := K(vx)$
        $\mathsf{return}\, H(vx)$
```

Thus, to get a general transformation for the MGFP we have to focus on the more difficult case of purely linear recursive algorithms. Then the term calculated by $f(x)$ is

$$\phi(\ldots\phi(\phi(H(K^{n0}(x)), E(K^{n0}(x))), E(K^{n0-1}(x))), \ldots, E(x))$$

where the innermost part results from the deepest recursion level. The standard way to transform such a linear recursive function into an imperative algorithm works in two steps. First linear recursion is transformed into tail recursion and then the previously presented scheme is applied.

Methods to get tail recursive versions of linear recursive functions heavily depend on properties of the function ϕ that combines the recursive and the non-recursive parts [BW82]. One of the best known procedures is changing the evaluation order of parentheses. The resulting expression then is calculated in a tail recursive way using a function ψ satisfying $\phi(\phi(r, s), t) = \phi(r, \psi(s, t))$. Obviously, if ϕ is associative one can choose $\psi = \phi$. Another method restructures the resulting expression by changing the order of operands. For that it is necessary that $\phi(\phi(r, s), t) = \phi(\phi(r, t), s)$ holds or more generally a function ψ is needed that satisfies $\phi(\psi(r, s), t) = \psi(\phi(r, t), s)$. In general, such functions ψ satisfying these conditions only exist in very rare cases. Alternatively, one can use function inversion to iteratively calculate parameter values from the arguments of the following recursion level backwards. This demands for invertability of function K that calculates the next parameter. But a direct consequence is the need for an additional preparation pass to get the argument of the deepest recursion level as starting point. An unacceptable method is the usage of tabulation. This requires the introduction of a global data structure e.g. a stack and simply imitates the execution behaviour of recursively defined functions with the same disadvantages. All of the presented methods demand that the involved functions are good-natured enough to satisfy the respective conditions. Unfortunately, in our applications ϕ coincides with the selective update operator or slightly altered variants ϕ_α of it which are not associative and also do not satisfy any of the other conditions given. Just as function K that is used to advance along a pointer link in general is not invertable.

Nevertheless, in 1970 Paterson and Hewitt (P & H) presented a transformation scheme [PH70] that makes it possible to transform any linear recursive algorithm into a tail recursive one. They applied the idea of function inversion to arbitrarily shaped functions by calculating the inverse of function K that is used to advance in the structure. By repeatedly calculating $K^i(x)$ from the starting value x to get the reverse image of value $K^{i+1}(x)$ under function K, they where in the position to make the step backwards also for non-injective functions. The function in focus is transformed into a system of three tail recursive functions. The evolving scheme is:

$$\boxed{\begin{array}{l}
f(x) = \textsf{if } B(x) \ \textsf{ then } \phi(f(K(x)), E(x)) \\
\qquad\qquad\qquad\quad \textsf{else } H(x) \\
\hline \updownarrow \quad [\text{ P \& H} \\
f(x) = G(n0, H(m0)) \ \textsf{where} \\
\qquad (m0, n0) = num(x, 0) \\
\qquad num(y, i) = \textsf{if } B(y) \ \textsf{ then } num(K(y), i+1) \\
\qquad\qquad\qquad\qquad\qquad\quad \textsf{else } (y, i) \\
\qquad it(y, i) = \textsf{if } i \neq 0 \ \textsf{ then } it(K(y), i-1) \\
\qquad\qquad\qquad\qquad\quad \textsf{else } y \\
\qquad G(i, z) = \textsf{if } i \neq 0 \ \textsf{ then } G(i-1, \phi(z, E(it(x, i-1)))) \\
\qquad\qquad\qquad\qquad\quad \textsf{else } z
\end{array}}$$

The actual calculation is done in function G whereas num and it are auxiliary tools to perform the inversion. Function num concurrently calculates the number of iterations $n0$ that have to be performed and the final value $K^{n0}(x)$ that satisfies condition B. For all the other $i < n0$ it holds that $B(K^i(x)) = false$. G so to say ascends from the deepest level of recursion starting from $K^{n0}(x)$ by successively calculating the inverse of K. This is achieved by iterating K on parameter x one time less often than has to be done for the current value. G uses function it to calculate the powers of K and we have $it(y, i) = K^i(y)$. Therefore we can abbreviate $\phi(z, E(it(x, i-1)))$ to $\phi(z, E(K^{i-1}(x)))$ and eliminate it from the scheme:

$$\boxed{\begin{array}{l}
f(x) = \textsf{if } B(x) \ \textsf{ then } \phi(f(K(x)), E(x)) \\
\qquad\qquad\qquad\quad \textsf{else } H(x) \\
\hline \updownarrow \quad [\text{ P \& H II} \\
f(x) = G(n0, H(m0)) \ \textsf{where} \\
\qquad (m0, n0) = num(x, 0) \\
\qquad num(y, i) = \textsf{if } B(y) \ \textsf{ then } num(K(y), i+1) \\
\qquad\qquad\qquad\qquad\qquad\quad \textsf{else } (y, i) \\
\qquad G(i, z) = \textsf{if } i \neq 0 \ \textsf{ then } G(i-1, \phi(z, E(K^{i-1}(x)))) \\
\qquad\qquad\qquad\qquad\quad \textsf{else } z
\end{array}}$$

Certainly, this is only a cosmetic change and gives us no efficiency improvement as the powers of K still have to be calculated. But in the sequel it is easier to use the powers of K than reasoning about function it. As one immediately notices, the P & H scheme yields very inefficient execution patterns. This is the reason why it normally is only of theoretical interest. Nevertheless, under particular conditions it is the starting point for further simplification.

5.3 Transformation of Linear Recursive Algorithms

Clearly, the bad runtime performance of the P & H scheme evolves from the repeated exhaustive calculation of the powers of K starting again and again from scratch. To improve efficiency of the transformation rule for the derived pointer algorithms we investigate under which conditions the evaluation order can be reversed. Then the function value and the powers of K can be calculated concurrently. This would imply the need of only one pass through the structure and improve efficiency considerably. As G successively applies the selective update operator ϕ that alters the pointer structure we have to investigate under which conditions a reordering of the applications of function ϕ can be performed. We first focus on the case of two successive updates.

Lemma 144. *Let m, n, o, p be atomic addresses, then:*

$$m \neq o \vee n = p \;\Rightarrow\; (m \overset{\alpha}{\to} n) \mid ((o \overset{\alpha}{\to} p) \mid L) = (o \overset{\alpha}{\to} p) \mid ((m \overset{\alpha}{\to} n) \mid L)$$

Proof. By definition of $\mid$ the left hand side of the equation can be rewritten as:

$$(m \overset{\alpha}{\to} n) + \neg(m \cdot \alpha) \cdot (o \overset{\alpha}{\to} p) + \neg(m \cdot \alpha) \cdot \neg(o \cdot \alpha) \cdot L$$

Whereas the right hand side is:

$$(o \overset{\alpha}{\to} p) + \neg(o \cdot \alpha) \cdot (m \overset{\alpha}{\to} n) + \neg(o \cdot \alpha) \cdot \neg(m \cdot \alpha) \cdot L$$

By commutativity of predicates the last terms are equal and we can ignore them. The precondition is equivalent to the statement $(m = o \wedge n = p) \vee m \neq o$. Therefore we split the rest of the proof into two cases:

$m = o \wedge n = p:$ The remainder of both sides simplifies to $(m \overset{\alpha}{\to} n)$

$m \neq o:$ As m and o are atomic it follows that $\neg m \cdot o = o$ and $\neg o \cdot m = m$. This makes the domain restrictions vanish and the terms are equal by commutativity of join. □

We use this lemma to inductively change the execution order of the updates performed by the Paterson/Hewitt scheme. For abbreviation reasons we will write s_i for $ptr(E(K^i(x)))$ and s_{n0} for $ptr(H(K^{n0}(x)))$.

As the last parameter in the evaluation process of G is $E(K^0(x))$ the P & H scheme always returns a pointer structure (s_0, X) with some store X. We first focus only on this calculated store. Assuming the update operator to be right-associative the performed changes of the store result in:

$$(s_0 \overset{\alpha}{\to} s_1) \mid (s_1 \overset{\alpha}{\to} s_2) \mid \ldots \mid (s_{n0-2} \overset{\alpha}{\to} s_{n0-1}) \mid (s_{n0-1} \overset{\alpha}{\to} s_{n0}) \mid sto(x) \qquad (*)$$

To get efficient single-pass algorithms on pointer structures our goal is the reversal of these updates to:

$$(s_{n0-1} \overset{\alpha}{\to} s_{n0}) \mid (s_{n0-2} \overset{\alpha}{\to} s_{n0-1}) \mid \ldots \mid (s_1 \overset{\alpha}{\to} s_2) \mid (s_0 \overset{\alpha}{\to} s_1) \mid sto(x) \qquad (**)$$

This term easily can be calculated by a tail recursive function counting the indices of s from 0 to $n0 - 1$. In fact, we introduce a somewhat more general function g that makes it possible to describe an arbitrary but coherent section of the update sequence.

$$\begin{aligned} g(j, i, L) = \mathsf{if}\ j \neq i\ \ &\mathsf{then}\ g(j+1, i, (s_j \overset{\alpha}{\to} s_{j+1}) \mid L) \\ &\mathsf{else}\ L \end{aligned}$$

We implicitly assume that $j \leq i$ holds, since otherwise g would not terminate. The first ministore applied as update to store L is $(s_j \overset{\alpha}{\to} s_{j+1})$ whereas $(s_{i-1} \overset{\alpha}{\to} s_i)$ is the last . Thus $g(0, n0, L)$ exactly yields expression (**). With this generalization the effect of pushing an update through the sequence can be expressed by lifting Lemma 144 from the binary case to a sequence of updates. Certainly, also the conditions have to be extended to all indices the binary change rule is applied to. As an abbreviation we define

$$PRE(i,j) \overset{\text{def}}{\Leftrightarrow} s_i \neq s_j \vee s_{i+1} = s_{j+1}$$

to be able to succinctly express the precondition and show a sort of commutativity rule for function g:

Lemma 145.

$$\bigwedge_{j \leq k \leq i} PRE(i,k) \Rightarrow g(j,i,(s_i \xrightarrow{\alpha} s_{i+1}) \mid L) = g(j,i+1,L)$$

Proof. $j = i$:

$$\begin{aligned} & g(i,i,(s_i \xrightarrow{\alpha} s_{i+1}) \mid L) \\ &= (s_i \xrightarrow{\alpha} s_{i+1}) \mid L \\ &= g(i+1,i+1,(s_i \xrightarrow{\alpha} s_{i+1}) \mid L) \\ &= \textsf{if } i \neq i+1 \ \textsf{ then } g(i+1,i+1,(s_i \xrightarrow{\alpha} s_{i+1}) \mid L) \\ &\qquad\qquad\quad \textsf{else } L \\ &= g(i,i+1,L) \end{aligned}$$

$j \to j-1$:

$$\begin{aligned} & g(j,i,(s_i \xrightarrow{\alpha} s_{i+1}) \mid L) \\ &= \textsf{if } j \neq i \ \textsf{ then } g(j+1,i,(s_j \xrightarrow{\alpha} s_{j+1}) \mid (s_i \xrightarrow{\alpha} s_{i+1}) \mid L) \\ &\qquad\qquad \textsf{else } L \\ &\overset{144}{=} g(j+1,i,(s_i \xrightarrow{\alpha} s_{i+1}) \mid (s_j \xrightarrow{\alpha} s_{j+1}) \mid L) \\ &\overset{I.H.}{=} g(j+1,i+1,(s_j \xrightarrow{\alpha} s_{j+1}) \mid L) \\ &= \textsf{if } j \neq i+1 \ \textsf{ then } g(j+1,i+1,(s_j \xrightarrow{\alpha} s_{j+1}) \mid L) \\ &\qquad\qquad\quad \textsf{else } L \\ &= g(j,i+1,L) \end{aligned}$$

□

This shows that under the given conditions terms (*) and (**) are equivalent. Hence, using this lemma we can calculate the result of the Patterson/Hewitt scheme by replacing G by g.

Lemma 146.

$$\forall i \leq n0 : \bigwedge_{0 \leq k \leq i} PRE(i,k) \Rightarrow G(i,(s_i,L)) = (s_0, g(0,i,L))$$

Proof. The proof again is an induction over i:

$i = 0$:

$$\begin{aligned} G(0,(s_0,L)) &= \textsf{if } 0 \neq 0 \ \textsf{ then } G(-1,\phi_\alpha((s_0,L),E(K^{-1}(x)))) \\ &\qquad\qquad\qquad \textsf{else } (s_0,L) \\ &= (s_0,L) \\ &= (s_0, \textsf{if } 0 \neq 0 \, \textsf{then } g(1,0,(s_0 \xrightarrow{\alpha} s_1) \mid L) \, \textsf{else } L) \\ &= (s_0, g(0,0,L)) \end{aligned}$$

$i \to i+1$:

$$\begin{aligned} & G(i+1,(s_{i+1},L)) \\ &= \textsf{if } i+1 \neq 0 \ \textsf{ then } G(i,\phi_\alpha((s_{i+1},L),E(K^i(x)))) \\ &\qquad\qquad\qquad \textsf{else } (s_{i+1},L) \\ &= G(i,E(K^i(x)).\alpha := (s_{i+1},L)) \\ &= G(i,(s_i,(s_i \xrightarrow{\alpha} s_{i+1}) \mid L)) \\ &\overset{I.H.}{=} (s_0, g(0,i,(s_i \xrightarrow{\alpha} s_{i+1}) \mid L)) \\ &\overset{145}{=} (s_0, g(0,i+1,L)) \end{aligned}$$

□

We take advantage of the fact that the evaluation order of updates is reversed to further simplify g. The quadratic cost factor can be eliminated by calculating the powers of K concurrently with the evaluation of g. This is achieved by introducing a new parameter z that sums up the applications of K. Additionally, we reverse the substitution of s_i. Here we handle the distinct case s_{n0} by introducing the choice function S. Since we now have access to the powers of K, function S is able to manage without referencing i or $n0$:

$$S(z) = \text{if } B(z) \text{ then } E(z) \text{ else } H(z)$$

Similarly, testing the termination condition can be performed by applying B directly to z. So we change g to an equivalent function g' defined by:

$$g'(j,i,z,L) = \text{if } B(z) \text{ then } g'(j+1,i,K(z),(ptr(E(z)) \xrightarrow{\alpha} ptr(S(K(z)))) \mid L) \text{ else } L$$

Obviously, parameters i and j are now neither used anymore in the function nor returned as part of the result. Hence, we can remove them securely and transform g' to g''.

$$g''(z,L) = \text{if } B(z) \text{ then } g''(K(z),(ptr(E(z)) \xrightarrow{\alpha} ptr(S(K(z)))) \mid L) \text{ else } L$$

As a direct consequence of these simplifications there is no longer any need for function num in the P & H scheme. So the only remaining term is G that is replaced by g''. As g'' is tail recursive we get an imperative version by simply using the standard transformation scheme presented previously. Some extra attention has to be drawn to the case $n0 = 0$. Then the entry address returned by f is s_0 which in this case is not $ptr(E(x))$ but $ptr(H(x))$. So here the auxiliary function S also has to be used. The evolving scheme is depicted in Figure 5.1. Additionally we propagate the if statement at the end over the while-loop. The derived transformation pattern now can be applied to linear recursive algorithms as for example mix_p.

Example 147. We use the derived recursive algorithm from Appendix A.2, replace the formal parameters, abbreviate $L_{tl}(m)$ by the more object oriented syntax $m.tl$, eliminate the ineffective update $(vm \xrightarrow{tl} vm.tl)$, and achieve:

```
$mix_p(m,n,L) =$
    var $(vm,vn,vL) := (m,n,L)$
    if $m \neq \diamond$ then
        while $vm \neq \diamond$ do
            if $vn \neq \diamond$ then $(vm,vn,vL) := (vn, vm.tl, (vm \xrightarrow{tl} vn) \mid vL)$
                            else $(vm,vn,vL) := (vn, vm.tl, vL)$
        return $(m, vL)$
    else return $(n, L)$
```

Condition $vn == \diamond$ inside the while-loop immediately implies by the assignments that $vm == \diamond$ at the next loop and therefore the while-loop terminates. Since the value of vn is not used afterwards and vL is not changed, we can eliminate these assignments from the else branch. By removing unused variables like vn and resolving the concurrent assignment the algorithm directly can be noted in C-syntax:

```
$f(x) = (ptr(S(x)), g''(x, sto(x)))$ where
    $g''(y) =$ if $B(y)$ then $g''(ptr(K(y)), (ptr(E(y)) \xrightarrow{\alpha} ptr(S(K(y)))) \mid sto(y))$
                          else $sto(y)$
——————————————— $\updownarrow$ ——————————————— [ tail to while
$f(x) =$ var $vx@(vy, vL) := x$
         while $B(vx)$ do
            $vx := (ptr(K(vx)), (ptr(E(vx)) \xrightarrow{\alpha} ptr(S(K(vx))) \mid vL)$
         return $(ptr(S(x)), vL)$
——————————————— $\updownarrow$ ——————————————— [ unfolding $S$
$f(x) =$ var $vx@(vy, vL) := x$
         while $B(vx)$ do
            if $B(K(vx))$
               then $vx := (ptr(K(vx)), (ptr(E(vx)) \xrightarrow{\alpha} ptr(E(K(vx))) \mid vL)$
               else $vx := (ptr(K(vx)), (ptr(E(vx)) \xrightarrow{\alpha} ptr(H(K(vx))) \mid vL)$
         if $B(x)$ then return $(ptr(E(x)), vL)$
                  else return $(ptr(H(x)), vL)$
——————————————— $\updownarrow$ ——————————————— [ if propagation
$f(x) =$ var $vx@(vy, vL) := x$
         if $B(x)$ then
            while $B(vx)$ do
               if $B(K(vx))$
                  then $vx := (ptr(K(vx)), (ptr(E(vx)) \xrightarrow{\alpha} ptr(E(K(vx))) \mid vL)$
                  else $vx := (ptr(K(vx)), (ptr(E(vx)) \xrightarrow{\alpha} ptr(H(K(vx))) \mid vL)$
            return $(ptr(E(x)), vL)$
         else return $H(x)$
```

Figure 5.1: Transformation scheme for linear recursive algorithms

```
list mix(list m, list n) {

  list vhm,vm = m;

  if (m) {
    while (vm)
      if (n) {
        vhm = vm;
        vm = n;
        n = vhm->tail;
        vhm->tail = vm;
      }
      else
        vm = n;
    return m;
  }
  else
    return n;
}
```

□

The applicability condition of the transformation scheme follows immediately from acyclicity of lists and $\neg sharing(m, n, L)$. Under the class of linear recursive pointer algorithms there is a particular subclass that can be treated in a more efficient way. The members of this class can be characterized as algorithms modifying exactly one link of the pointer structure as for example inserting an element into a sorted list or concatenation of two lists. They use function ϕ_α to pass through the pointer structure to find the position where the change has to be performed. A characterizing feature is that the selective update in the then branch has no effect. This follows from

$$\forall x, i \leq n0.\ ptr(E(K^i(x))) \xrightarrow{\alpha} ptr(E(K^{i+1}(x))) \leq L \qquad \text{(ann)}$$

by annihilation. Since function K does not change the pointer structure it follows:

$$(ptr(K(x)), vL) = (ptr(K(x)), sto(K(x))) = K(x)$$

which simplifies the then branch. For the first iteration of the while-loop $B(vx)$ holds since $vx = x$. If additionally the value of B is independent from the change of the store in the else part which is expressed by:

$$\forall x.\ B(x) = B(ptr(x), (ptr(E(K^{n0-1}(x))) \xrightarrow{\alpha} ptr(H(K^{n0}(x)))) \mid sto(x)) \qquad \text{(ind)}$$

the assignment to vx can be propagated and the resulting function pattern can be transformed to:

```
$f(x) =$ var $vx := x$
         if $B(x)$ then while $B(K(vx))$ do $vx := K(vx)$
                        return $(ptr(E(x)), (ptr(E(vx)) \xrightarrow{\alpha} ptr(H(K(vx)))) \mid L)$
                   else $H(x)$
```

Example 148. For the concatenation of two lists condition (ann) holds, since the left-hand side evaluates to $(L^i_{tl}(m) \xrightarrow{tl} L^{i+1}_{tl}(m))$ and (ind) is satisfied since $B(m, n, L) = m \neq \diamond$ only depends on the entries. Therefore the more efficient pattern can be applied to cat_p and we immediately get the C-program:

```
list cat(list m,list n) {
  list vm = m;
  if (m) {
    while (vm->tail) vm=vm->tail;
    vm->tail=n;
    return m;
  }
  else
    return n;
 }
```

□

The same transformation pattern was derived in [Ehm01] from scratch. Nevertheless, there only an informal description of the class of treated algorithms was given. By the two conditions (ann) and (ind) the members of this class are characterized formally.

5.4 Improving the Scheme

By the observations of the previous section we are able to transform linear recursive pointer algorithms into imperative variants. This section is concerned with extending the scheme to treat algorithms matching the most general function pattern.

The problem we currently are not able to deal with is the concurrent appearance of a linear and a tail recursive branch. Such patterns mainly appear in deletion algorithms. The linear recursive branch is used to traverse the data structure and the tail recursive one performs the deletion of elements. To reproduce this behaviour we introduce an auxiliary function that skips over all the elements to be dismissed. Then the tail recursive part can be embedded into a linear recursive one and the previously derived scheme can be applied. We define function δ that summarizes successive execution of a tail recursive branch by multiple application of advance function K:

Definition 149. *The function $\delta_{B,C,K}$ is defined by*

$$\delta_{B,C,K}(x) = \mathsf{if}\, B(x)\,\&\&\, C(x)\ \begin{array}{l}\mathsf{then}\, \delta(K(x))\\ \mathsf{else}\, x\end{array}$$

If B, C and K are obvious from the context we simply will write $\delta(x)$ for $\delta_{B,C,K}(x)$. In the sequel we abbreviate the application of δ prior to the application of function f by writing:

$$\overrightarrow{f} \stackrel{\text{def}}{=} f \circ \delta$$

We present some properties of δ that are needed to generalize the linear recursive scheme. Evidently, iterated application of δ has no further effect:

Lemma 150. *The function δ is idempotent, i.e. $\delta(\delta(x)) = \delta(x)$*

Proof. If $\delta(x)$ does not terminate the expressions on both sides of the equation do not succeed. So assume that the calculation terminates and let $y = \delta(x)$, then $\neg(B(y)\,\&\&\,C(y))$ holds and it follows that

$$\delta(\delta(x)) = \delta(y) = \mathsf{if}\, B(y)\,\&\&\, C(y)\,\mathsf{then}\,\delta(K(y))\,\mathsf{else}\, y = y = \delta(x)$$

□

It is also clear at first sight that if $B(y)$ holds for $y = \delta(x)$ then C cannot hold. Otherwise δ would not have terminated for y. By denoting sequential implication, where the left argument is evaluated first, by $\triangleright$, we have:

Lemma 151. $\overrightarrow{B}(x) \triangleright \neg\overrightarrow{C}(x)$

Proof. Let $y = \delta(x)$, then by termination of δ we get:

$$\begin{aligned}\neg(B(y)\,\&\&\,C(y)) &\Leftrightarrow \neg B(y)\,||\,\neg C(y)\\ &\Leftrightarrow B(y) \triangleright \neg C(y)\\ &\Leftrightarrow \overrightarrow{B}(x) \triangleright \neg\overrightarrow{C}(x)\end{aligned}$$

□

We can show that $\delta(x)$ has no effect if B does not hold for x or if $C(x)$ fails under the precondition that B terminates. We will use the predicate *def* to express definedness. This means that the calculation is solvable and for example will not run forever. For a formal definition see [Par90].

Lemma 152.
$$\begin{aligned}&\neg B(x) &&\Rightarrow f(x) = \overrightarrow{f}(x)\\ &\mathit{def}\,(B) \wedge \neg C(x) &&\Rightarrow f(x) = \overrightarrow{f}(x)\end{aligned}$$

Proof. Assume $\neg B(x)$, then:

$$\overrightarrow{f}(x) = f(\delta(x)) = f(\mathsf{if}\ B(x)\ \&\&\ C(x)\ \mathsf{then}\ \delta(K(x))\ \mathsf{else}\ x) = f(x)$$

The second proposition is shown the same way. □

With these prerequisites the most general function pattern can be transformed into a linear recursive form:

$$\begin{aligned}
& f(x) = \mathsf{if}\ B(x)\ \ \mathsf{then\ if}\ C(x)\ \ \mathsf{then}\ f(K(x)) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \phi(f(K(x)), E(x)) \\
& \qquad\qquad\qquad\quad \mathsf{else}\ H(x) \\
= \quad & \{\!\![\ \text{introducing } \delta\ ]\!\!\} \\
& f(x) = \mathsf{if}\ B(x)\ \ \mathsf{then\ if}\ C(x)\ \ \mathsf{then}\ f(\delta(x)) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \phi(f(K(x)), E(x)) \\
& \qquad\qquad\qquad\quad \mathsf{else}\ H(x) \\
= \quad & \{\!\![\ \text{unfold } f\ ]\!\!\} \\
& f(x) = \mathsf{if}\ B(x)\ \ \mathsf{then\ if}\ \ C(x)\ \mathsf{then} \\
& \qquad\qquad\qquad\qquad\quad \mathsf{if}\ B(\delta(x))\ \ \mathsf{then\ if}\ \ C(\delta(x)) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{then}\ f(\delta(\delta(x))) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \phi(f(K(\delta(x))), E(\delta(x))) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ H(\delta(x)) \\
& \qquad\qquad\qquad\qquad\quad \mathsf{else}\ \phi(f(K(x)), E(x)) \\
& \qquad\qquad\qquad\quad \mathsf{else}\ H(x) \\
= \quad & \{\!\![\ \text{Lemmas 150, 151 and 152 and Definition of } \overrightarrow{f}\ ]\!\!\} \\
& f(x) = \mathsf{if}\ B(x)\ \ \mathsf{then\ if}\ C(x)\ \ \mathsf{then\ if}\ \overrightarrow{B}(x)\ \ \mathsf{then}\ \phi(f(\overrightarrow{K}(x)), \overrightarrow{E}(x)) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \overrightarrow{H}(x) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \phi(f(\overrightarrow{K}(x)), \overrightarrow{E}(x)) \\
& \qquad\qquad\qquad\quad \mathsf{else}\ \overrightarrow{H}(x) \\
= \quad & \{\!\![\ \text{simplification of conditions } (*)\ ]\!\!\} \\
& f(x) = \mathsf{if}\ B(x)\ \&\&\ \overrightarrow{B}(x)\ \ \mathsf{then}\ \phi(f(\overrightarrow{K}(x)), \overrightarrow{E}(x)) \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \overrightarrow{H}(x)
\end{aligned}$$

The simplification $(*)$ works as follows. The two branches where f is called recursively can be described by $B\,\&\&(C\,\&\&\,\overrightarrow{B})$ and $B\,\&\&\,\neg C$. Since by Lemma 152 we have $B\,\&\&\,\neg C \Rightarrow \overrightarrow{B}$, the second formula equals $B\,\&\&(\neg C\,\&\&\,\overrightarrow{B})$ and we can summarize the two expressions and simplify them to $B\,\&\&\,\overrightarrow{B}$. So we can apply the transformation scheme for linear recursive functions derived in the previous section and get:

```
$f(x) =$ var $vx@(vy, vL) := x$
      if  $B(x)\ \&\&\ \overrightarrow{B}(x)$ then
         while $B(vx)\ \&\&\ \overrightarrow{B}(vx)$ do
            if  $B(\overrightarrow{K}(vx))\ \&\&\ \overrightarrow{B}(\overrightarrow{K}(vx))$
               then $vx := (ptr(\overrightarrow{K}(vx)), (ptr(\overrightarrow{E}(vx)) \xrightarrow{\alpha} ptr(\overrightarrow{E}(\overrightarrow{K}(vx)))) \mid vL)$
               else $vx := (ptr(\overrightarrow{K}(vx)), (ptr(\overrightarrow{E}(vx)) \xrightarrow{\alpha} ptr(\overrightarrow{H}(\overrightarrow{K}(vx)))) \mid vL)$
         return $(ptr(\overrightarrow{E}(x)), vL)$
      else return $\overrightarrow{H}(x)$
```

Further we can make some improvements concerning the efficiency. To prevent the repeated calculation of $\delta(vx)$ we introduce a new variable $v\delta$ to hold this value. A second new variable vh is used to store the initial value of $\delta(x)$ which is needed after the while-loop in the calculation of the entry of the returned pointer structure.

```
$f(x) =$ var $vx@(vy, vL) := x$
         var $v\delta, vh := \delta(x)$
         if $B(x)$ && $B(v\delta)$ then
           while $B(vx)$ && $B(v\delta)$ do
             $vx := v\delta$
             $v\delta := \delta(K(vx))$
             if $B(K(vx))$ && $B(v\delta)$
               then $vx := (ptr(K(vx)), (ptr(E(vx)) \overset{\alpha}{\to} ptr(E(v\delta))) \mid vL)$
               else $vx := (ptr(K(vx)), (ptr(E(vx)) \overset{\alpha}{\to} ptr(H(v\delta))) \mid vL)$
           return $(ptr(E(vh)), vL)$
         else return $H(v\delta)$
```

Finally, we can unfold the definition of δ and transform the tail recursive function into a while-loop. The resulting transformation scheme for the MGFP then looks like follows:

```
$f(x) =$ if $B(x)$ then if $C(x)$ then $f(K(x))$
                                else $\phi(f(K(x)), E(x))$
                    else $H(x)$
──────────────────────$\updownarrow$──────────────────────
$f(x) =$ var $vx@(vy, vL) := x$
         var $v\delta := x$
         while $B(v\delta)$ && $C(v\delta)$ do $v\delta := K(v\delta)$
         var $vh := v\delta$
         if $B(x)$ && $B(v\delta)$ then
           while $B(vx)$ && $B(v\delta)$ do
             $vx := v\delta$
             $v\delta := K(vx)$
             while $B(v\delta)$ && $C(v\delta)$ do $v\delta := K(v\delta)$
             if $B(K(vx))$ && $B(v\delta)$
               then $vx := (ptr(K(vx)), (ptr(E(vx)) \overset{\alpha}{\to} ptr(E(v\delta))) \mid vL)$
               else $vx := (ptr(K(vx)), (ptr(E(vx)) \overset{\alpha}{\to} ptr(H(v\delta))) \mid vL)$
           return $(ptr(E(vh)), vL)$
         else return $H(v\delta)$
```

Certainly, the conditions to apply the linear recursive transformation scheme have to hold for the respective instances of functions.

Example 153. The pattern immediately can be applied to $delete_p$. By removing the explicitly mentioned store and resolving concurrent assignments we get the program:

```
list delete(int v,list m) {
  list vh, vl, vm = m, va = vm;

  while (va && v==va->head) va = va->tail;
  vh = va;
  if (m && va) {
    while (vm && va) {
      vm = va;
      va = vm->tail;
```

```
          while (va && v==va->head) va = va->tail;
          if (vm->tail && va) {
            vl = vm;
            vm = vl->tail;
            vl->tail = va;
          }
          else {
            vl = vm;
            vm = vl->tail;
            vl->tail = NULL;
          }
        }
        return vh;
      }
      else
        return NULL;
```

□

At first sight the algorithm looks a little bit complicated compared to the ones given in present algorithm textbooks. But most of them use header cells for lists to avoid a number of cases concerning nil pointers and empty lists. We are sure that given to experienced programmers the exercise to code the deletion algorithm without using header cells on a sheet of paper, there would be a minimal number of correct solutions disregarding syntactic errors.

5.5 The Other Way 'Round: Verification

Another application area of pointer Kleene algebra can be found in verification of pointer structures. In contrast to a transformational approach where an executable implementation is derived step by step from a specification, verification works the other way round. Given a specification the software engineer invents an implementation and has to prove that the specification is fulfilled. The transformational approach has its advantages if there is a sufficient set of applicable transformation rules that encapsulates most of the work to be done. On the other hand, verification is a universal method but often requires a great deal of painstaking work. The creative programming process is reflected in the invention of loop invariants and assertions that have to hold at intermediate calculation steps. We will show how pointer Kleene algebra can be used as a formal foundation of Hoare-style rules to verify pointer algorithms. In a second step we use these rules to verify one of the algorithms yielded by the transformation method described before.

We take a verification formalism presented by Bornat [Bor00]. Based on previous work on correctness of assignments by Burstall [Bur72] and Morris [Mor81a] Bornat extends Hoare logic by object-component assignment rules. In contrast to Morris's, Bornat's method also is capable of whole-object assignments and multiple object component references. This is accomplished by considering a spatial separated heap representation. His approach is strongly related to the work presented in [Rey00, ORY01] concerned with the same problem. To fix the object-component assignment axioms Bornat uses an embedding of the heap into a two-dimensional array. He treats the heap as a pointer-indexed collection of objects each of which is a name-indexed collection of components. This heap model bijectively can be transferred to labeled graphs and therefore treated with pointer Kleene algebra. The first index step corresponds to the selection of an address and the second one is represented by following a labeled link. Therefore the double indexing of an object component reference $A.f$ in a heap H in Bornat's model is reflected by an image calculation in pointer algebra:

$$[\![A.f]\!]H \overset{\text{def}}{=} ([\![A]\!]H) : P_f(H)$$

We here have used Bornat's naming conventions and notions to support easy comparison of both formalizations. He uses $[\![E]\!]H$ to denote the semantics of expression E in heap H which in this case maps a pointer expression to an address. As we identify object names and addresses in our pointer algebra model, the interpretation of $[\![A]\!]H$ of object A simplifies to A itself. By assigning a value to $A.f$ the mapping of object A is changed at component f. This results in a selective change of the heap representation which directly can be expressed in pointer Kleene algebra by the update operator:

$$[\![F^{/A.f}_E]\!]H \overset{\text{def}}{=} [\![F]\!]H' \text{ with } H' = (A \xrightarrow{f} E) \mid H$$

Here F denotes an arbitrary object-component reference expression. As a consequence, we can formally validate the axioms for object component substitution. This reveals the advantages and succinctness of the pointer Kleene algebra approach. In contrast to the array treatment our calculation is half as long and half as complicated. For distinct component names f and g we obtain:

$$\begin{aligned}
[\![(B.g)^{/A.f}_E]\!]H &= [\![B.g]\!]H' \text{ with } H' = (A \xrightarrow{f} E) \mid H\\
&= ([\![B]\!]H') : P_g(H')\\
&\overset{80.2}{=} ([\![B]\!]H') : P_g(H)\\
&= ([\![B^{/A.f}_E]\!]H) : P_g(H)\\
&= [\![(B^{/A.f}_E).g]\!]H
\end{aligned}$$

With identical component names we calculate:

$$\begin{aligned}
[\![(B.f)^{/A.f}_E]\!]H &= [\![B.f]\!]H' \text{ with } H' = (A \xrightarrow{f} E) \mid H\\
&= ([\![B]\!]H') : P_f(H')\\
&\overset{80.1}{=} ([\![B]\!]H') : (A \xrightarrow{f} E) + ([\![B]\!]H') : \neg A \cdot P_f(H)\\
&= (A \cdot B) : (A \xrightarrow{f} E) + (\neg A \cdot B) : P_f(H)\\
&= \mathsf{if}\ A = B\ \mathsf{then}\ A : (A \xrightarrow{f} E)\ \mathsf{else}\ [\![B]\!]H' : P_f(H)\\
&= \mathsf{if}\ A = B\ \mathsf{then}\ E\ \mathsf{else}\ (B^{/A.f}_E) : P_f(H)\\
&= \mathsf{if}\ A = B\ \mathsf{then}\ E\ \mathsf{else}\ [\![(B^{/A.f}_E).f]\!]H
\end{aligned}$$

These calculations result in the following axioms for assignments of object components:

$$\frac{}{(B.g)^{/A.f}_E \triangleq (B^{/A.f}_E).g} \qquad \frac{}{(B.f)^{/A.f}_E \triangleq \mathsf{if}\ A = B^{/A.f}_E\ \mathsf{then}\ E\ \mathsf{else}\ (B^{/A.f}_E).f}$$

The complete set of Hoare-triple rules for a language with assignment to variables and object components, conditionals, while-loop and instruction sequence can be found in Appendix A.3. We will now use this method to show correctness of one of the algorithms derived with the transformational approach. This should show two things. First, how verification of pointer algorithms can be achieved and second, give an evidence that the transformation schemes derived are correct. Certainly, we can also apply the verification method directly to the transformation pattern. But this leads to a very abstract verification template with a bunch of proof obligations. Such a high abstraction also suffers from the lack of knowledge about the underlying pointer structure to perform simplifications, the concrete abstraction function and so on.

For the case study we will use the function mix_p derived in Appendix A.2 that in a C-like style similar to Bornat's notation is:

```
$\{PRE\}$
if $p \neq \diamond$ then $r := p$
                 $\{INV\}$
                 while $r \neq \diamond$ do
                    if $q \neq \diamond$ then $s := r; r := q; q := s.tl; s.tl := r$
                              else $r := q;$
                 $u := p$
           else $u := q$
$\{POST\}$
```

To have an access possibility the result is returned in a new variable u. The algorithm is enriched by assertions satisfied at the respective locations. To denote these conditions Bornat uses a sort of generalized abstraction function for lists $A \Rightarrow_f B$ starting at object A and following links in f components until a pointer equal to B is encountered:

$$A \Rightarrow_f B \overset{\text{def}}{=} \textsf{if } A = B \textsf{ then } [] \textsf{ else } [A] \mathbin{@} (A.f \Rightarrow_f B)$$

He uses the @ operator to concatenate two lists. The list abstraction $list(m, a)$ from the transformation method is equal to $(m \Rightarrow_{P_{tl}(a)} \diamond)$. He uses another predicate $list$ to express listness of such sequences. This can be seen as an abstract test for cycle-freeness. Additionally $\not\cap$ is used to denote disjointness of two sequences. With these tools the assertions can be expressed by:

$$\begin{aligned}
PRE &= \{list(p \Rightarrow_{tl} \diamond) \wedge list(q \Rightarrow_{tl} \diamond) \wedge (p \Rightarrow_{tl} \diamond) = S \wedge (q \Rightarrow_{tl} \diamond) = T\} \\
INV &= \{list(r \Rightarrow_{tl} \diamond) \wedge list(q \Rightarrow_{tl} \diamond) \wedge (r \Rightarrow_{tl} \diamond) \not\cap (q \Rightarrow_{tl} \diamond) \wedge \\
&\qquad (p \Rightarrow_{tl} r) \mathbin{@} mix (r \Rightarrow_{tl} \diamond)\ (q \Rightarrow_{tl} \diamond) = mix\ S\ T\} \\
POST &= \{(u \Rightarrow_{tl} \diamond) = mix\ S\ T\}
\end{aligned}$$

As termination measure for the while-loop we use the sum of length of lists represented by r and q:

$$t = length(r \Rightarrow_{tl} \diamond) + length(q \Rightarrow_{tl} \diamond)$$

The complete proof including all calculations is somewhat longer and can be found in Appendix A.3.

Chapter 6

Discussion

6.1 Related Work

Since this thesis considers the complete development process of pointer algorithms, a broad spectrum of areas is touched. Thus, we try to classify the related work into groups of related topics although such a categorization does not always match exactly.

We first give an overview of papers that present approaches to treat graphs or pointer structures formally. These are either specialized logics to reason about pointer linked structure or algebraic frameworks to perform calculations and transformations. Since graph theory has a long tradition in mathematics, there are a lot of publications in this area. Nevertheless, most of them just present more efficient calculation methods and are not concerned with correct program development. We here will focus on work that uses graphs from a software engineering point of view, and that is involved with derivations and transformations of pointer algorithms.

In [Möl93] a relational approach based on the algebra of regular languages and path algebra is used to derive graph and pointer algorithms. Möller sticks primarily to recursively defined equations as specifications. The derivation of an algorithm for in-situ chain concatenation can be seen as preliminary work for the transformation method presented in [Möl97a], which is used in this thesis. The passage from recursive to imperative programs in general is not considered and is only shown for the simple tail recursive case of chain reversal.

In contrast, in [Bvv94] imperative loops enriched by equational invariants are used immediately. The equational reasoning is based on a matrix algebra over a regular algebra. By this approach it is possible to have access to the elements inside a matrix but the calculus loses abstractness and turns towards point-wise reasoning. Nevertheless, the presented derivation is abstract, since a general implementation of path algorithm that comprises finding of shortest paths, reachability and bottleneck problems is derived. The concrete interpretation is achieved by instantiating the regular algebra by a suitable structure.

A comparison between the two approaches [Bvv94] and [Möl93] can be found in [Cle95]. Clenaghan identifies dynamic algebra as a common abstract algebraic framework that unites both methods. At this abstract level he derives Dijkstra's shortest path algorithm. This is achieved independently of a specific $(min, +)$ algebra by adding extra properties characterizing the used structure.

A quite general approach to the derivation of graph algorithms is presented in [Rus96]. The thesis derives abstract algorithm schemes for classes of graph algorithms. These entail layer-oriented graph traversal and problems based on the calculation of hamiltonian paths as for example topological sorting. The underlying calculus is based on an algebra of formal languages and paths and allows to express graph problems transparently and concise. The problems treated origin more from the previously mentioned mathematical view of graph theory and are not focused on pointer algorithms but the transformational approach and the abstract framework used is similar to the ones presented in this thesis.

Butler [But99] uses enriched trees to derive pointer algorithms from applicative specifications. He extends abstract trees by paths to access subtrees. Instead of working on trees themselves all changes are performed on these path descriptions. In contrast to [Möl97a] Butler does not have to carry along a representation for the whole store in the code itself. A data refinement method is presented to transform operations on enriched trees to pointer operations. Similar to the transformation rule presented in Chapter 5 Butler introduces a generic refinement rule for search algorithms to make the step to imperative algorithms. Although he shows how to derive an algorithm to delete an element from a sorted tree, we cannot see that his rule also covers the more general case of deleting all appearances from a sorted tree with potential multiple occurrences of elements. Such a scheme would be on the same level as the transformation rule for the MGFP.

In [BEZ88] several reachability algorithms for directed graphs are presented. The paper is mainly focused on application of the transformation framework CIP. Based on an abstract data type of graphs a predicate logical specification of the set of reachable vertices is transformed into reachability algorithms implemented by depth-first-search, breadth-first-search or more application-specific strategies. It is shown how well-known transformation tactics like iterative augmentation and formal differentiation can be used to get these concretizations of a derived general abstract algorithm pattern.

A second group of papers is concerned with verification of pointer algorithms as seen in Section 5.5. Most of these use an extension of weakest precondition or Hoare-like verification rules that are able to cope with pointer assignments and aliasing.

Although there is earlier work on this task [Bur72, LS79, Kow79] most of these approaches to verify pointer algorithms are based on previous work by Morris. In [Mor81a] he presents a new axiom for pointer assignments where assignments of references are broken down to conditions over scalar variables. With this, one is able to verify correctness of pointer algorithms and programs modifying linked data structures. Morris works with observations of aliasing which are sufficient for languages like Pascal, where no calculations with pointers are available. Nevertheless this method is not transferable to *C*-like languages with direct manipulations of pointers, since there it is syntactically not decidable if two references point to the same location or not. In the same volume [Mor81b] Morris applied this framework to prove correctness of the Schorr-Waite marking algorithm.

A completely functional treatment of pointer structures is presented in [Bir01]. Bird derives simple list processing functions and as main contribution also the Schorr-Waite marking algorithm. The theory is heavily influenced by [Möl97a] and also explicitly uses a variable for the store. Bird completely stays in a functional setting until he reaches a tail recursive variant. Although the derivation has some subtle problems the method itself constitutes a substantial contribution to a formal development process of pointer algorithms. Nevertheless, it is more driven by intuition than the method presented in Chapter 5.

In [Mas88] a LISP based method to derive programs that destructively manipulate their data is introduced. Mason presents a theory to determine equivalence between pure functional and imperative implementations. This is used to introduce transformation laws to get destructive imperative algorithms from functional specifications. The approach is strongly related to the method presented in Chapter 5. Although compared to [Bir01] it shows succinctness in the derivation of the

Schorr-Waite marking algorithm, there is a lack of abstraction and generality since the theory is tightly embedded into the LISP environment.

Bijlsma [Bij89] pretends that his calculus is superior to Morris' approach, since assertions are not limited to reachability conditions. He uses a function to denote the length of a sequence between two pointers following a particular field type. His case study about insertion into a list shows that this construct is only used to express reachability in a complicated way. This leads to an immense blow-up of complexity of calculations but the gain of the more general treatment is not used.

In [Nel83] verification of reachability invariants in linked data structures is achieved by introducing an axiomatization of a reachability predicate. A simple set union algorithm based on a sort of union/find structure is derived from an abstract specification and verified on several pages. In contrast to our approach Nelson has the possibility to talk about the number of steps that have to be performed to reach a particular node but his system is also restricted to such reachability statements. Similarly to Bijlsma, this leads to complex formulas flooded with quantifiers.

The main focus in [Rey00] is laid on restricting assertions about a heap structure to statements over independent disjoint parts. This extends former work by Burstall [Bur72] and Kowaltowski [Kow79]. They assume to have a sequence of assertions for distinct regions so that by an assignment to a single location only one of these is affected. Reynolds argues that this situation is not always achievable. He uses a mix of an imperative language together with inductively defined data types. Reynolds himself admits that his work is very preliminary and needs to be investigated more exactly.

Another approach to verification of pointer algorithms is presented in [Kub03]. Specification of pointer data structures there is achieved by using a temporal specification of dynamic algebra. This allows to talk about paths in the pointer structure. The model is more related to the trace model of pointers introduced in [HJ99]. Kubica presents a deduction system for an extended version of Hoare logic that allows temporal dynamic formulae as assertions. It is not completely clear how this approach exactly relates to the one presented here. Since we are able to model temporal operators like always and sometimes, it seems possible that most statements in some way can be expressed in our calculus, too.

Although we presented in Chapter 5 transformational program development from a functional specification as an application of pointer Kleene algebra, this approach has some limitations. The main problem is that graphs are not naturally inductively definable data types like lists and trees. So there are several approaches originating from the functional programming community to deal with graphs and linked data structures in functional programming languages. We will only dwell upon two selected representative approaches here.

Although it is unusual to think of graphs as an algebraic type with constructors, Erwig [Erw01] proposed an inductive graph definition. There graphs are constructed from the empty graph and extensions with new nodes together with edges from and to the nodes already present. He uses unique identifiers for the nodes to establish edge connections and tries to resolve the ambiguity of representations by a particular kind of pattern matching. This works well to write functional programs on graphs and get efficient implementations. Nevertheless, the approach cannot serve as solution to treat graphs with the method presented in Chapter 5 due to the complex pattern matching concept.

Also Klarlund and Schwartzbach [KS93] defined *graph types* as extensions of classical recursive data types. They use a spanning tree as underlying backbone enhanced with additional routing links. These links are denoted by routing expressions which describe relative addresses within the backbone. They show how to decide which routing fields of a data type have to be updated and that efficient imperative algorithms can be derived. Monadic second-order logic is used to define shape invariants on these data types. To implement imperative style concepts they use monads and get efficient execution patterns.

Finally, we present papers concerned with an abstract description of fuzzy concepts in a relational environment. These approaches heavily influenced the algebraic treatment presented in this thesis.

The algebraic treatment of fuzzy relations goes back to the thesis of Furusawa [Fur98]. Together with Kawahara [KF01] he introduced the concept of scalars and proposed several notions of crispness. Winter [Win01] has shown that it is impossible to characterize L-fuzzy relations in Dedekind categories for arbitrary distributive lattices L. He introduced Goguen categories which are Dedekind categories extended with cut-operators to specify crisp L-relations. However, Kawahara and Furusawa again tried to do without these extensions but assumed the existence of unit objects in the category and a linear ordering of L around the least element. Despite the evidence that L-fuzzy relations are able to model labeled graphs, to our knowledge this abstract framework has not been applied to graph theory.

6.2 Summary

We have seen how an extension of Kleene algebra can serve as an algebraic foundation for the treatment of labeled graphs. Simplicity and succinctness of Kleene algebra is inherited by the calculus and leads to short proofs at a high abstract level. We defined pointer Kleene algebra and more sophisticated operations to express properties of pointer structures or to characterize particular sets of nodes and substructures. In contrast to a relational approach considering each equally labeled subgraph distinctly we developed a compact representation of labeled graphs. On the other hand, the matrix theoretical treatment often found in the literature mostly can be replaced by the presented abstract framework. This prevents unclear and complex point-wise reasoning.

Since subordination holds in relation algebra, the usage of fuzzy relation algebra or equivalently Goguen categories with transitive closure would also be an appropriate choice as formal basis for labeled graphs. Nevertheless, calculating with a converse operator makes proofs intransparent and from an algorithmic point of view reversal of all pointers in memory is a highly inefficient task. We have shown how several properties can be defined in a structure without converse and meet. If the existence of a meet is required the right choice would be to use action lattices. Since they are extensions of action algebras, we get residuals for free.

As application we have shown how to use pointer Kleene algebra as formal basis to show the correctness of rules to transform pointer manipulating programs. In this course we extended a method to derive correct pointer algorithms from specifications in a functional programming style. We characterized a general syntactic form of function patterns that may arise and provided a transformation rule to get efficient imperative pointer manipulating programs. This particular method is only usable to derive algorithms on inductively defined data types which mostly are lists and trees. Nevertheless, the underlying calculus is not limited to this class of programs.

In sum, we have presented an algebraic framework to formalize labeled graphs which is simple but of strong expressive power. None of the previous approaches to achieve this task in the literature are as compact and concise as the treatment based on Kleene algebra introduced in this thesis. The algebraic foundation is the basis for reasoning about pointer structures at a high abstract level. Application to transformation and verification techniques is the key to get correct implementations of pointer algorithms.

6.3 Outlook and Future Research

There are some points where the two big areas covered in this thesis can be further developed.

Since one of the most often used systems in a programming environment are the compiler and the executing machine, these are natural candidates to be verified. An important rôle there plays the garbage collector. It would be an interesting case study to port the derivation of a copying garbage collector in [BMM91] to the framework presented here. In this thesis we first concentrated on a general algebraic treatment of pointer structures and did not let us guide from such a specific application. Since the derivation in the paper is based on a calculus of partial maps and the representation of chains, the transfer seems to be no problem. Nevertheless, to get the same concrete algorithm on a linearly ordered memory one has to add additional properties or has to leave the abstract framework.

The method to transform functional specifications into correct implementations of pointer algorithms has been proved suitable to derive algorithms on inductively defined data structures. Since general graphs are not naturally describable inductively, the transformation schemes demand for more sophisticated rules that consider such cyclic data types. We will investigate the approaches of [Erw01] and [KS93] in more detail and try to extend the derivation method also to a high level treatment of graphs without refraining from the advantages of applicative specification.

Due to additional expenditure with respect to time and money formal methods will never be applied if not necessary or demanded. To increase efficiency and make complexity in dealing with formulas and proof obligations manageable a high degree of automation and tool support is needed. Some preliminary work has been done by implementing different axiomatizations into the automatic proof system KIV [RSSB98]. To further automation of proofs we plan to develop decision procedures for at least fragments of pointer Kleene algebra, since decidability is not solved.

Another step towards an efficient treatment of pointer algorithms was performed by implementing a prototype of a transformation system [Vog03]. This can be used to rapidly derive transformation rules similar to the one presented in Chapter 5. Further research may confirm our supposition, that the derivation of pointer algorithms on inductively defined data types follows a simple unfold/fold heuristic.

The crucial extension of Kleene algebra to be able to perform the step to pointer Kleene algebra is the concept of subordination. From a theoretical point of view it is important to study the relation to extensions from other areas. This would give more insight into the expressiveness of Kleene algebras with subordination. So we know for example that subordination strictly implies the concept of local linearity as defined by R. Dijkstra [Dij98] in his computation calculus. Thus, it would be interesting to see how much of this calculus can be transferred to Kleene algebra with subordination.

Appendix A

The appendix shows definitions, derivations and the complete verification of mix_p. The definition of standard Kleene algebra is used in the derivation of subordination whereas the derivation of list processing functions shows the transformation method at work and yields the motivating examples. Section A.3 presents a detailed proof of correctness of the derived algorithm mix_p.

A.1 Standard Kleene Algebra

In [Con71] Conway gives five different notions of Kleene algebra. As reason for this multiplicity he mentioned that the formal laws of regular operations are not easily codified. So for example already in 1964 Redko [Red64] proved that star is not finitely axiomatizable. The most restricted algebra Conway gave he called *S-algebra* (standard Kleene algebra). We will give here a slightly different axiomatization that better meets our requirements.

Definition 154 (SKA). *A standard Kleene algebra is a sixtuple* $(\mathcal{K}, \leq, \top, \cdot, 0, 1)$ *satisfying the following properties:*

1. $(\mathcal{K}, \leq)$ *is a complete lattice with least element* 0 *and greatest element* $\top$*.*
2. $(\mathcal{K}, \cdot, 1)$ *is a monoid.*
3. *The operation* $\cdot$ *is universally disjunctive (i.e. distributes through arbitrary suprema) in both arguments.*

We only summarize the important laws that hold in SKAs due to the existence of a meet operator.

Lemma 155. *Consider a SKA and* $s, t \in \mathcal{P}$*.*

1. $s \cdot t = s \sqcap t$
2. $s \cdot (a \sqcap b) = s \cdot a \sqcap s \cdot b$
3. $(s \sqcap t) \cdot a = s \cdot a \sqcap t \cdot a$
4. $s \cdot a \sqcap \neg s \cdot b = 0$
5. $a \sqcap s \cdot b = s \cdot a \sqcap s \cdot b$
   In particular: $a \sqcap s \cdot \top = s \cdot a$

A.2 Selected Derivations

The most important and wide-spread dynamically allocated pointer linked data types are lists and trees. Simultaneously lists are the simplest non-trivial of such structures as there is only one successor record. Trees in so far are generalizations of lists as they consist of two descendants. We present functional definitions of some list-processing functions that are examined in Chapter 5. For some of them which are used as examples for the application of the MGFP transformation pattern we also provide the derivation of pointer manipulating variants. Further examples can be found in [Ehm03].

The constructor to build sorted lists is a function that inserts an element before the first element that is greater:

```
insert a []     = [a]
insert a (x:xs) = if a <= x then a:(x:xs)
                            else x : insert a xs
```

To remove the same element from the list we define

```
del a []     = []
del a (x:xs) = if a==x then xs
                       else x : del a xs
```

Obviously, this only holds under the promise that `del` is applied only to sorted lists. Function `del` only removes the first occurrence of `a` in the list. As we did not assume repetition free sorted lists, the generalization `delete` removes all occurrences of element `a`. Actually, we define `delete` even to be applicable to arbitrary lists, since we do not stop if a greater element is encountered.

```
delete a []     = []
delete a (x:xs) = if a==x then delete a xs
                          else x : delete a xs
```

From a pointer implementation point of view the two functions `insert` and `del` change exactly one link in the pointer structure imlementation of the considered lists. `delete` performs as many modifications as elements `a` are in the list. From this observation the algorithm that changes all links is the function `mix` that shuffles the elements of two lists element by element. If one of the lists is longer than the other the remainder is concatenated to the end of the result. This can be specified by:

```
mix []     ys = ys
mix (x:xs) ys = x : mix ys xs
```

To get pointer implementations of `delete` and `mix` we use the method presented in Section 5.1. The derivation for the case $m = \diamond$ works similarly to the derivation of `cat` in this section and we choose

$$delete_p\ a\ (\diamond, L) = (\diamond, L)$$

For the other case we get:

Case $m \neq \diamond$:

$delete\ a\ list(p)$

$=$ $\{\![$ unfold definitions of $list$ and $delete$ $]\!\}$

$\mathsf{if}\ a == p.hd\ \mathsf{then}\ delete\ a\ list(p.tl)$
$\qquad\qquad\quad \mathsf{else}\ p.hd : delete\ a\ list(p.tl)$

$=$ $\{\![$ fold with spec. of $delete_p$; choose an arbitrary $q \in delete_p\ a\ p.tl$ $]\!\}$

$\mathsf{if}\ a == p.hd\ \mathsf{then}\ list(q)$
$\qquad\qquad\quad \mathsf{else}\ p.hd : list(q)$

$=$ $\{\![$ set $r = p.tl := q$, Lemma 83 $]\!\}$

$\mathsf{if}\ a == p.hd\ \mathsf{then}\ list(q)$
$\qquad\qquad\quad \mathsf{else}\ r.hd : list(q)$

$=$ $\{\![$ Corollary 141.1 and Lemma 142 $]\!\}$

$$\begin{array}{l}\textsf{if}\ a == p.hd\ \ \textsf{then}\ list(q)\\ \qquad\qquad\quad\ \textsf{else}\ r.hd : list(r.tl)\end{array}$$

$=\quad \{\![$ fold with definition of $list$ $]\!\}$

$\textsf{if}\ a == p.hd\ \textsf{then}\ list(q)\ \textsf{else}\ list(r)$

$=\quad \{\![$ if propagation $]\!\}$

$list(\textsf{if}\ a == p.hd\ \textsf{then}\ q\ \textsf{else}\ r)$

By resubstituting q and r we get the pointer algorithm:

$$\boxed{\begin{array}{l}delete_p\ a\ p = \textsf{if}\ \ m \neq \diamond\ \ \textsf{then if}\ \ a \neq L_{hd}(m)\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textsf{then}\ p.tl := delete_p\ a\ p.tl\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textsf{else}\ delete_p\ a\ p.tl\\ \qquad\qquad\qquad\qquad\quad \textsf{else}\ (\diamond, L)\end{array}}$$

In a similar way we derive for `mix` in the case $m = \diamond$:

$$mix_p\ (\diamond, n, L) = (n, L)$$

and calculate:

Case $m \neq \diamond$:

$mix\ list(p)\ list(q)$

$=\quad \{\![$ unfold definitions of $list$ and mix $]\!\}$

$p.hd : mix\ list(q)\ list(p.tl)$

$=\quad \{\![$ fold with spec. of mix_p; choose an arbitrary $r \in mix_p(n, L_{tl}(m), L)$ $]\!\}$

$p.hd : list(r)$

$=\quad \{\![$ set $s = p.tl := r$, Lemma 83 $]\!\}$

$s.hd : list(r)$

$=\quad \{\![$ Corollary 141.1 and Lemma 142 $]\!\}$

$s.hd : list(s.tl)$

$=\quad \{\![$ fold with definition of $list$ $]\!\}$

$list(s)$

Again resubstitution yields the algorithm:

$$\boxed{\begin{array}{l}mix_p(m, n, L) = \textsf{if}\ \ m \neq \diamond\ \ \textsf{then}\ p.tl := mix_p(n, L_{tl}(m), L)\\ \qquad\qquad\qquad\qquad\qquad\quad \textsf{else}\ (n, L)\end{array}}$$

A.3 Verification of mix_p

This section shows the verification of the imperative version of mix_p from Section 5.5. We use the Hoare-style rules depicted in Figure A.1. For the assignment of the result to u in both branches of the if statement we first calculate:

then part:

$\{POST\}$

$\quad u := p$

$\{(p \Rightarrow_{tl} \diamond) = mix\ S\ T\}$

else part:

$\{POST\}$

$\quad u := q$

$\{(q \Rightarrow_{tl} \diamond) = mix\ S\ T\}$

$$\frac{Q \Rightarrow R^x_E}{\{Q\}x := E\{R\}} \qquad \frac{Q \Rightarrow R^f_{f\oplus A\to E}}{\{Q\}A.f := E\{R\}} \qquad \frac{\{Q\}S\{R'\} \quad R' \Rightarrow R}{\{Q\}S\{R\}}$$

$$\frac{Q \Rightarrow P \quad \{P \wedge B\}S\{P\} \quad P \wedge \neg B \Rightarrow R \quad P \wedge B \Rightarrow t > 0 \quad \{P \wedge B \wedge t = vt\}S\{t < vt\}}{\{Q\}\mathsf{while}\, B\, \mathsf{do}\, S\{R\}}$$

$$\frac{\{Q \wedge B\}S_{then}\{R\} \quad \{Q \wedge \neg B\}S_{else}\{R\}}{\{Q\}\mathsf{if}\, B\, \mathsf{then}\, S_{then}\, \mathsf{else}\, S_{else}\{R\}} \qquad \frac{\{Q\}S1\{Q'\} \quad \{Q'\}S2\{R\}}{\{Q\}S1; S2\{R\}}$$

Figure A.1: Hoare-triple rules

The rest of the else branch is simply proven. From PRE and $p = \diamond$ follows $S = [] \wedge T = (q \Rightarrow_{tl} \diamond)$ which implies by definition of mix:

$$mix\ S\ T = mix\ []\ T = T = (q \Rightarrow_{tl} \diamond)$$

Thus, we are left with showing correctness of the rest of the then part which mainly consists of the verification of the while-loop. We will use Q as abbreviation for the condition that has to hold after the loop. First, we show the following implications that can be derived from INV and $r \neq \diamond$:

Lemma 156. *Assume INV and $r \neq \diamond$, then*

1. $(r.tl \Rightarrow_{tl\oplus r\to q} \diamond) = (r.tl \Rightarrow_{tl} \diamond)$
2. $(q \Rightarrow_{tl\oplus r\to q} \diamond) = (q \Rightarrow_{tl} \diamond)$

Proof. 1. $list(r \Rightarrow_{tl} \diamond) \wedge r \neq \diamond$
$\Rightarrow list(r.tl \Rightarrow_{tl} \diamond) \wedge [r] \not\cap (r.tl \Rightarrow_{tl} \diamond)$
$\Rightarrow (r.tl \Rightarrow_{tl\oplus r\to q} \diamond) = (r.tl \Rightarrow_{tl} \diamond)$

2. $list(q \Rightarrow_{tl} \diamond) \wedge (r \Rightarrow_{tl} \diamond) \not\cap (q \Rightarrow_{tl} \diamond)$
$\Rightarrow list(q \Rightarrow_{tl} \diamond) \wedge [r] \not\cap (q \Rightarrow_{tl} \diamond)$
$\Rightarrow (q \Rightarrow_{tl\oplus r\to q} \diamond) = (q \Rightarrow_{tl} \diamond)$

To show that invariant INV holds after the initialization we calculate:

$\{INV\}$

$r := p$

$$\left\{\begin{array}{l} list(p \Rightarrow_{tl} \diamond) \wedge list(q \Rightarrow_{tl} \diamond) \wedge (p \Rightarrow_{tl} \diamond) \not\cap (q \Rightarrow_{tl} \diamond) \wedge \\ (p \Rightarrow_{tl} p) @ mix(p \Rightarrow_{tl} \diamond)\ (q \Rightarrow_{tl} \diamond) = mix\ S\ T \end{array}\right\}$$

This follows immediately from PRE by a straightforward calculation. For the loop body we show that INV indeed is an invariant:

$$\left\{\begin{array}{l} list(r \Rightarrow_{tl} \diamond) \wedge list(q \Rightarrow_{tl} \diamond) \wedge (r \Rightarrow_{tl} \diamond) \not\cap (q \Rightarrow_{tl} \diamond) \\ \wedge (p \Rightarrow_{tl} r) @ mix(r \Rightarrow_{tl} \diamond)\ (q \Rightarrow_{tl} \diamond) = mix\ S\ T \end{array}\right\}$$

$s.tl := r$

$$\left\{\begin{array}{l} list(r \Rightarrow_{tl\oplus s\to r} \diamond) \wedge list(q \Rightarrow_{tl\oplus s\to r} \diamond) \wedge (r \Rightarrow_{tl\oplus s\to r} \diamond) \not\cap (q \Rightarrow_{tl\oplus s\to r} \diamond) \\ \wedge (p \Rightarrow_{tl\oplus s\to r} r) @ mix(r \Rightarrow_{tl\oplus s\to r} \diamond)\ (q \Rightarrow_{tl\oplus s\to r} \diamond) = mix\ S\ T \end{array}\right\}$$

$q := s.tl$

$$\left\{\begin{array}{l} list(r \Rightarrow_{tl\oplus s\to r} \diamond) \wedge list(s.tl \Rightarrow_{tl\oplus s\to r} \diamond) \wedge (r \Rightarrow_{tl\oplus s\to r} \diamond) \not\cap (s.tl \Rightarrow_{tl\oplus s\to r} \diamond) \\ \wedge (p \Rightarrow_{tl\oplus s\to r} r) @ mix(r \Rightarrow_{tl\oplus s\to r} \diamond)\ (s.tl \Rightarrow_{tl\oplus s\to r} \diamond) = mix\ S\ T \end{array}\right\}$$

$r := q$

$$\left\{\begin{array}{l} list(q \Rightarrow_{tl\oplus s\to q} \diamond) \wedge list(s.tl \Rightarrow_{tl\oplus s\to q} \diamond) \wedge (q \Rightarrow_{tl\oplus s\to q} \diamond) \not\cap (s.tl \Rightarrow_{tl\oplus s\to q} \diamond) \\ \wedge (p \Rightarrow_{tl\oplus s\to q} q) @ mix(q \Rightarrow_{tl\oplus s\to q} \diamond)\ (s.tl \Rightarrow_{tl\oplus s\to q} \diamond) = mix\ S\ T \end{array}\right\}$$

$s := r$

$$\left\{\begin{array}{l} list(q \Rightarrow_{tl\oplus r\to q} \diamond) \wedge list(r.tl \Rightarrow_{tl\oplus r\to q} \diamond) \wedge (q \Rightarrow_{tl\oplus r\to q} \diamond) \not\cap (r.tl \Rightarrow_{tl\oplus r\to q} \diamond) \\ \wedge (p \Rightarrow_{tl\oplus r\to q} q) \mathbin{@} mix(q \Rightarrow_{tl\oplus r\to q} \diamond)\ (r.tl \Rightarrow_{tl\oplus r\to q} \diamond) = mix\ S\ T \end{array}\right\}$$

This follows immediately from INV and $r \neq \diamond$ with Lemmas 156.1 and 156.2 and by the straightforward calculation:

$$\begin{aligned} &(p \Rightarrow_{tl} r) \mathbin{@} mix(r \Rightarrow_{tl} \diamond)\ (q \Rightarrow_{tl} \diamond) \\ &= (p \Rightarrow_{tl} r) \mathbin{@} [r] \mathbin{@} mix(q \Rightarrow_{tl} \diamond)\ (r.tl \Rightarrow_{tl} \diamond) \\ &= (p \Rightarrow_{tl} r.tl) \mathbin{@} mix(q \Rightarrow_{tl\oplus r\to q} \diamond)\ (r.tl \Rightarrow_{tl\oplus r\to q} \diamond) \\ &= (p \Rightarrow_{tl\oplus r\to q} q) \mathbin{@} mix(q \Rightarrow_{tl\oplus r\to q} \diamond)\ (r.tl \Rightarrow_{tl\oplus r\to q} \diamond) \end{aligned}$$

After the while-loop Q has to hold. Since INV and $r = \diamond$ imply $q = \diamond$, this can be shown by:

$$\begin{aligned} mix\ S\ T &= (p \Rightarrow_{tl} r) \mathbin{@} mix(r \Rightarrow_{tl} \diamond)\ (q \Rightarrow_{tl} \diamond) \\ &= (p \Rightarrow_{tl} \diamond) \mathbin{@} mix(\diamond \Rightarrow_{tl} \diamond)\ (\diamond \Rightarrow_{tl} \diamond) \\ &= (p \Rightarrow_{tl} \diamond) \mathbin{@} mix\ []\ [] \\ &= (p \Rightarrow_{tl} \diamond) \mathbin{@}\ [] \\ &= (p \Rightarrow_{tl} \diamond) \end{aligned}$$

For measure t we have to show that it is positive after the initialization. From $r \neq \diamond$ we get:

$$\begin{aligned} t &= length(r \Rightarrow_{tl} \diamond) + length(q \Rightarrow_{tl} \diamond) \\ &\geq length(r \Rightarrow_{tl} \diamond) \\ &= length([r] \mathbin{@} (r.tl \Rightarrow_{tl} \diamond)) \\ &= 1 + length(r.tl \Rightarrow_{tl} \diamond) \\ &> 0 \end{aligned}$$

To show termination of the loop we have to prove that the loop body reduces measure t. We just show the calculation for the then branch, since the else branch works similar:

$\{length(r \Rightarrow_{tl} \diamond) + length(q \Rightarrow_{tl} \diamond) < vt\}$

$s.tl := r$

$\{length(r \Rightarrow_{tl\oplus s\to r} \diamond) + length(q \Rightarrow_{tl\oplus s\to r} \diamond) < vt\}$

$q := s.tl$

$\{length(r \Rightarrow_{tl\oplus s\to r} \diamond) + length(s.tl \Rightarrow_{tl\oplus s\to r} \diamond) < vt\}$

$r := q$

$\{length(q \Rightarrow_{tl\oplus s\to q} \diamond) + length(s.tl \Rightarrow_{tl\oplus s\to q} \diamond) < vt\}$

$s := r$

$\{length(q \Rightarrow_{tl\oplus r\to q} \diamond) + length(r.tl \Rightarrow_{tl\oplus r\to q} \diamond) < vt\}$

By Lemmas 156.1 and 156.2 under the assumption of INV and $r \neq \diamond$ this is equivalent to:

$$length(q \Rightarrow_{tl} \diamond) + length(r.tl \Rightarrow_{tl} \diamond) < vt$$

which follows immediately from INV by:

$$\begin{aligned}
vt &= length(r \Rightarrow_{tl} \diamond) + length(q \Rightarrow_{tl} \diamond) \\
&= length([r] \,@(r.tl \Rightarrow_{tl} \diamond)) + length(q \Rightarrow_{tl} \diamond) \\
&= 1 + length(r.tl \Rightarrow_{tl} \diamond) + length(q \Rightarrow_{tl} \diamond) \\
&> length(r.tl \Rightarrow_{tl} \diamond) + length(q \Rightarrow_{tl} \diamond)
\end{aligned}$$

Bibliography

[Aar92] C.J. Aarts. Galois connections presented calculationally. Afstudeer verslag (Graduating Dissertation), Department of Computing Science, Eindhoven University of Technology, July 1992.

[AHU75] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1975.

[Bac02] R. Backhouse. Galois connections and fixed point calculus. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures*, volume 2297 of *Lecture Notes in Computer Science*, pages 89–148. Springer-Verlag, 2002.

[BBB$^+$85] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[BdM96] R.S. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall International, 1996.

[BE93a] S.L. Bloom and Z. Ésik. Equational axioms for regular sets. *Mathematical structures in computer science*, 3:1–24, 1993.

[BÉ93b] S.L. Bloom and Z. Ésik. *Iteration Theories: The Equational Logic of Iterative Processes*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1993.

[BEH$^+$87] F.L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[BEZ88] R. Berghammer, H. Ehler, and H. Zierer. Development of several reachability algorithms for directed graphs. In H. Göttler and H.J. Schneider, editors, *Graph-Theoretic Concepts in Computer Science, International Workshop, WG '87, Kloster Banz/Staffelstein, Germany, June 29 - July 1, 1987*, volume 314 of *Lecture Notes in Computer Science*, pages 206–218. Springer-Verlag, 1988.

[Bij89] A. Bijlsma. Calculating with pointers. *Science of Computer Programming*, 12(3):191–206, September 1989.

[Bir67] G. Birkhoff. Lattice theory. *A.M.S. Colloquium Publications*, 25, 1967.

[Bir98] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall International, 1998. 2nd edition.

[Bir01] R.S. Bird. Functional pearl: Unfolding pointer algorithms. *Journal of Functional Programming*, 11(3):347–358, May 2001.

[BMM91] U. Berger, W. Meixner, and B. Möller. Calculating a garbage collector. In M. Broy and M. Wirsing, editors, *Methods of programming*, volume

544 of *Lecture Notes in Computer Science*, pages 137–192. Springer-Verlag, 1991.

[Bor00] R. Bornat. Proving pointer programs in Hoare logic. In R. Backhouse and J.N. Oliveira, editors, *Mathematics of Program Construction, 5th International Conference, MPC 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer-Verlag, 2000.

[Bur72] R.M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland, 1972.

[But99] M. Butler. Calculational derivation of pointer algorithms from tree operations. *Science of Computer Programming*, 33(3):221–260, March 1999.

[Bvv94] 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, April 1994.

[BW82] F.L. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer-Verlag, 1982.

[BW89] R. Bird and Ph. Wadler. *Introduction to Functional Programming.* Prentice Hall International, 1989.

[Cle95] K. Clenaghan. Calculational graph algorithmics: reconciling two approaches with dynamic algebra. Technical report CS-R9518, CWI - Centrum voor Wiskunde en Informatica, March 1995.

[Coh00] E. Cohen. Separation and reduction. In R. Backhouse and J.N. Oliveira, editors, *Proceedings of Mathematics of Program Construction, 5th International Conference, MPC 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 45–59. Springer-Verlag, 2000.

[Con71] J.H. Conway. *Regular Algebra and Finite Machines.* Chapman & Hall, London, 1971.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall International, 1976.

[Dij98] R. Dijkstra. Computation calculus — bridging a formalization gap. In J. Jeuring, editor, *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings*, volume 1422 of *Lecture Notes in Computer Science*, pages 151–174, 1998.

[dM64] A. de Morgan. On the syllogism, no. iv, and on the logic of relations. *Transactions of the Cambridge Philosophical Society*, 10:331–358, 1864.

[DM01] J. Desharnais and B. Möller. Characterizing determinacy in Kleene algebras. In J. Desharnais, M. Frappier, A. Jaoua, and W. MacCaull, editors, *Relational Methods in Computer Science. Int. Seminar on Relational Methods in Computer Science, Jan 9–14, 2000 in Québec*, volume 139 of *Information Sciences — An International Journal*, pages 153–273, 2001.

[DMS03] J. Desharnais, B. Möller, and G. Struth. Kleene algebra with a domain operator. Technical report 2003-7, Institut für Informatik, Universität Augsburg, 2003.

[Ehm01] T. Ehm. Transformational Construction of Correct Pointer Algorithms. In D. Bjørner, M. Broy, and A.V. Zamulin, editors, *Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*, pages 116–130. Springer-Verlag, July 2001.

[Ehm03] T. Ehm. Case studies for the derivation of pointer algorithms. Technical report 2003-9, Institut für Informatik, Universität Augsburg, 2003.

[EMS03] T. Ehm, B. Möller, and G. Struth. Kleene modules. Submitted to RelMiCS, 2003.

[Erw01] M. Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

[FL79] J.M. Fischer and R.F. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Science*, 18(2):194–211, 1979.

[Flo67] R.W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, pages 19–32, 1967.

[Fow99] M. Fowler. *Refactoring. Imporving the design of existing code.* Addison-Wesley, 1999.

[FS90] P.J. Freyd and A. Scedrov. *Categories, Allegories*, volume 39 of *North-Holland Mathematical Library.* North-Holland, Amsterdam, 1990.

[Fur98] H. Furusawa. *Algebraic Formalizations of Fuzzy Relations and their Representation Theorems.* PhD thesis, Kyushu University, 1998.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[Gog67] J.A. Goguen. L-fuzzy sets. *Journal of Mathmatical Analysis and Applications*, 18:145–157, 1967.

[HJ87] C.A.R. Hoare and H. Jifeng. Weakest prespecification. *Information Processing Letters*, 24, 1987.

[HJ99] C.A.R. Hoare and H. Jifeng. A trace model for pointers and objects. In R. Guerraoui, editor, *ECCOP'99 - Object-Oriented Programming, 13th European Conference, Lisbon, Portugal, June 14-18, 1999, Proceedings*, volume 1628 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 1999.

[HMT71] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras I*, volume 64 of *Studies in logic and the foundations of mathematics.* North-Holland, 1971.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[Hoa72] C.A.R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[Hol98] M. Hollenberg. Equational axioms of test algebra. In M. Nielsen and W. Thomas, editors, *Computer Science Logic, 11th International Workshop, CSL '97, Annual Conference of the EACSL, Aarhus, Denmark, August 23-29, 1997, Selected Papers*, volume 1414 of *Lecture Notes in Computer Science*, pages 295–310, 1998.

[IEE83] *IEEE Standard Glossary of software engineering terminology*, 1983.

[JT51] B. Jónsson and A. Tarski. Boolean algebras with operators, Part I. *American Journal of Mathematics*, 73:891–939, 1951.

[JT52] B. Jónsson and A. Tarski. Boolean algebras with operators, Part II. *American Journal of Mathematics*, 74:127–167, 1952.

[Jun94] D. Jungnickel. *Graphs, Networks and Algorithms*, volume 5 of *Algorithms and Computation in Mathematics.* Springer-Verlag, 1994.

[KF01] Y. Kawahara and H. Furusawa. Crispness in Dedekind categories. *Bulletin of Informatics and Cybernetics*, 33(1–2):1–18, 2001.

[Kle51] S.C. Kleene. Representation of events in nerve nets and finite automata. Technical report, The Rand Corporation, 1951.

[Kow79] T. Kowaltowski. Data structures and correctness of programs. *Journal of the ACM (JACM)*, 26(2):283–301, 1979.

[Koz79] D. Kozen. A representation theorem for $*$-free PDL. Technical Report RC7864, IBM, 1979.

[Koz81] D. Kozen. On induction vs. $*$-continuity. In D. Kozen, editor, *Proceedings of Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 167–176. Springer-Verlag, 1981.

[Koz90a] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. Technical report TR90-1123, Cornell University, Computer Science Department, May 1990.

[Koz90b] D. Kozen. On Kleene algebras and closed semirings. In B. Rovan, editor, *Proceedings of Mathematical Foundations of Computer Science*, volume 452 of *Lecture Notes in Computer Science*, pages 26–47. Springer-Verlag, 1990.

[Koz94] D. Kozen. On action algebras. In J. van Eijck and A. Visser, editors, *Logic and Information Flow*, pages 78–88. MIT Press, 1994.

[Koz97] D. Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[KR88] B. Kernighan and D. Ritchi. *The C programming language.* Prentice Hall International, 1988.

[Kro91] D. Krob. A complete system of b-rational identities. *Journal of Theoretical Computer Science*, 89(2):207–343, October 1991.

[KS86] W. Kuich and A. Salomaa. *Semiring, Automata, and Languages.* Springer-Verlag, 1986.

[KS93] N. Klarlund and M. Schwartzbach. Graph types. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 196–205, Charleston, South Carolina, 1993.

[Kub03] M. Kubica. A temporal approach to specification and verification of pointer data-structures. In Mauro Pezzè, editor, *Fundamental Approaches to Software Engineering, 6th International Conference, FASE 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003*, volume 2621 of *Lecture Notes in Computer Science*, pages 231–245. Springer-Verlag, 2003.

[KWBW03] A. Kleppe, J. Warmer, W. Bast, and A. Watson. *MDA Explained: The Model Driven Architecture - Practice and Promise.* Addison-Wesley, 2003.

[LS79] D. Luckham and N. Suzuki. Verication of array, record, and pointer operations in pascal. *ACM Transactions on Programming Languages and Systems*, 1(2):226–244, October 1979.

[ŁT30] J. Łukasiewicz and A. Tarski. Untersuchungen über den Aussagenkalkül. *Comptes Rendus Séances Société des Sciences et Lettres Varsovie, cl. III*, 23:30–50, 1930.

[Łuk70] J. Łukasiewicz. Selected works, 1970.

[Mas88] I.A. Mason. Verification of programs that destructively manipulate data. *Science of Computer Programming*, 10(2):177–210, April 1988.

[Möl93] B. Möller. Derivation of graph and pointer algorithms. In B. Möller, H.A. Partsch, and S.A. Schuman, editors, *Formal program development*, volume 755 of *Lecture Notes in Computer Science*, pages 123–160. Springer-Verlag, 1993.

[Möl97a] B. Möller. Calculating with pointer structures. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 24–48. Proc. IFIP TC2/WG2.1 Working Conference, Le Bischenberg, Feb. 1997, Chapman & Hall, 1997.

[Möl97b] B. Möller. Linked Lists Calculated. Technical report 1997-7, Institut für Informatik, Universität Augsburg, 1997.

[Möl99a] B. Möller. Calculating with acyclic and cyclic lists. In A. Jaoua and G. Schmidt, editors, *Relational Methods in Computer Science. Int. Seminar on Relational Methods in Computer Science, Jan 6–10, 1997 in Hammamet*, volume 119 of *Information Sciences — An International Journal*, pages 135–154, 1999.

[Möl99b] B. Möller. Typed Kleene Algebras. Technical report 1999-8, Institut für Informatik, Universität Augsburg, 1999.

[Moo56] E.F. Moore. Gedanken-experiments on sequential machines. *Automata Studies*, pages 129–153, 1956.

[Mor81a] J.M. Morris. A general axiom of assignment. In *Theoretical Foundations of Programming Methodology*, volume 91 of *NATO Advanced Study Institutes Series C Mathematical and Physical Sciences*, pages 25–34. Dordrecht, Reidel, 1981.

[Mor81b] J.M. Morris. A proof of the Schorr-Waite algorithm. In *Theoretical Foundations of Programming Methodology*, volume 91 of *NATO Advanced Study Institutes Series C Mathematical and Physical Sciences*, pages 35–51. Dordrecht, Reidel, 1981.

[Nel83] G. Nelson. Verifying reachability invariants of linked structures. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 38–47, Austin, Texas, 1983.

[Ng84] K.C. Ng. *Relation Algebras with Transitive Closure.* PhD thesis, University of California, Berkley, 1984.

[ORY01] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic: 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2001.

[OS95] J.P. Olivier and D. Serrato. Squares and rectangles in relational categories - three cases: semilattice, distributive lattice and non-unitary. *Fuzzy sets and systems*, 72:167–178, 1995.

[Par90] H.A. Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development.* Monographs in Computer Science. Springer-Verlag, Berlin, 1990.

[Par02] Parasoft. Insure++: An automatic runtime error detection tool, October 2002. White Paper.

[Per99] B. Perens. Electric fence, 1999. available from: `http://perens.com/FreeSoftware/`.

[PH70] M.S. Paterson and C.E. Hewitt. Comparative schematology. In *Rec. Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 119–128, Woods Hole, MA, December 1970.

[Pin98] J.E. Pin. Tropical semirings. *Idempotency*, pages 50–69, 1998.

[Plu90] Max Plus. Linear systems in $(max, +)$ algebra. In *Proceedings of the 29th Conference on Decision and Control*, Honolulu, December 1990.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of 19th IEEE Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[Pra90a] V. Pratt. Action logic and pure induction. In J. van Benthem and J. Eijck, editors, *Proceedings of JELIA-90, European Workshop on Logics in AI*, Amsterdam, September 1990.

[Pra90b] V. Pratt. Dynamic Algebras as a well-behaved fragment of Relation Algebras. In C.H. Bergman, R.D. Maddux, and D.L. Pigozzi, editors, *Algebraic Logic and Universal Algebra in Computer Science*, volume 425 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[Pra91] V. Pratt. Dynamic algebras: Examples, constructions, applications. *Studia Logica*, 50:571–605, 1991.

[Red64] V.N. Redko. On defining relations for the algebra of regular events. *Ukrain. Math. Z.*, 16:120–126, 1964. In russian.

[Rey00] J.C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Mill-*

ennial Perspectives in Computer Science, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.

[RJB98] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[RSSB98] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P.H. Schmidt, editors, *Automated Deduction: A Basis for Applications. Volume II, Systems and Implementation Techniques*. Kluwer Academic Publishers, Dordrecht, Reidel, 1998.

[Rus96] M. Russling. *Deriving General Schemes for Classes of Graph Algorithms*. PhD thesis, Universität Augsburg, 1996. Augsburger mathematisch-naturwissenschaftliche Schriften, 13, Wißner.

[Sal66] A. Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM*, 13(1):158–169, January 1966.

[Sew02] J. Seward. The design and implementation of valgrind, 2002. available from: `http://developer.kde.org/\char126\relaxsewardj`.

[SM56] C.E. Shannon and J. McCarthy. *Automata Studies*. Princeton University Press, Princeton, 1956.

[SS93] G. Schmidt and T. Ströhlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists*. EATCS-Monographs on Theoretical Computer Science. Springer-Verlag, 1993.

[Tar41] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):65–106, 1941.

[Thi94] P. Thiemannn. *Grundlagen der funktionalen Programmierung*. B.G. Teubner, Stuttgart, 1994.

[Vog03] W. Vogl. Fallstudie zur entwicklung eines programmtransformationssystems unter verwendung von xml und xpath. Master's thesis, Universität Augsburg, 2003. in german.

[vW02] J. von Wright. From Kleene algebra to refinement algebra. In B. Möller and E. Boiten, editors, *Mathematics of Program Construction, 6th International Conference, MPC 2002*, volume 2386 of *Lecture Notes in Computer Science*, pages 233–262. Springer-Verlag, 2002.

[WD39] M. Ward and R.P. Dilworth. Residuated lattices. *Transactions of the American Mathematical Society*, 45:335–354, 1939.

[Win01] M. Winter. Relational constructions in Goguen categories. In H. de Swart, editor, *6th International Seminar on Relational Methods in Computer Science (RelMiCS)*, pages 222–236, 2001.

[Yet90] D. Yetter. Quantales and (noncommutative) linear logic. *Journal of Symbolic Logic*, 55(1):41–64, 1990.

[Zad65] L.A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.