A Language-Theoretic Approach to Algorithms

by

Deepak Goyal

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy
Department of Computer Science
Graduate School of Arts and Science
New York University
January 2000

Dissertation Advisor

To my parents, and To Bob

Acknowledgments

This thesis was completed in very sad circumstances. Bob Paige (my thesis advisor) finally lost his battle with cancer on October 5, 1999, only ten days before my final thesis defense. Bob had been bravely fighting his cancer for over three and a half years. Despite his poor health, he continued to help and advise me up to the very end. Ever since the fall of 1994, when I started working with Bob, he was a constant source of support and encouragement. Without his guidance, this thesis would not have been possible. I would like to express my deepest gratitude to Bob, who was not just a wonderful advisor and teacher, but also a very dear friend.

I am also very grateful to Prof. Alan Siegel on whose recommendation I started working with Bob, and Nieba Paige for taking such good care of Bob, and for her personal concern about my progress.

I would like to thank Annie Liu and G. Ramalingam for their careful reading of earlier drafts of this dissertation and for their valuable comments and suggestions.

I would also like to thank Prof. Zvi Kedem for helping me get a visa for entry into the United States more than five years ago, without which I would not have been able to attend NYU, and Anina Karmen for the numerous letters she has written for me as the Graduate Program Coordinator of the Computer Science Department.

I would like to thank my friends and fellow students Gediminas Adomavicius, Hseuming Chen, Raoul Sam Daruwala, Archisman Rudra, David Tanzer, and Zhe Yang for all their help and for being such good friends.

Most of all, I would like to thank my parents, and my sister Anu, for always having faith in me and for always being there for me.

Abstract

An effective algorithm design language should be 1) wide-spectrum in nature, i.e. capable of expressing both abstract specifications and low-level implementations, and 2) "computationally transparent", i.e. facilitate accurate estimation of time and space requirements. The conflict between these requirements is exemplified by SETL which is wide-spectrum, but lacks computational transparency because of its reliance on hash-based data structures. The first part of this thesis develops an effective algorithm design language, and the second demonstrates its usefulness for algorithm explanation and discovery.

In the first part three successively more abstract set-theoretic languages are developed and shown to be computationally transparent. These languages can collectively express both abstract specifications and low-level implementations. We formally define a data structure selection method for these languages using a novel type system. Computational transparency is obtained for the lowest-level language through the type system, and for the higher-level languages by transformation into the next lower level. We show the effectiveness of this method by using it to improve a difficult database query optimization algorithm from expected to worst-case linear time. In addition, a simpler explanation and a shorter proof of correctness are obtained.

In the second part we show how our data structure selection method can be made an effective third component of a transformational program design methodology whose first two components are finite differencing and dominated convergence. Finite differencing replaces costly repeated computations by cheaper incremental counterparts, and dominated convergence provides a generalized iteration scheme for computing fixed-points. This methodology has led us to a simpler explanation of a complex linear-time model-checking algorithm for the alternation-free modal mu-calculus, and to the discovery of an $O(N^3)$ time algorithm for computing intra-procedural may-alias information that improves over an existing $O(N^5)$ time algorithm.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The task of *programming* has been variously described as an *art* (Knuth, The Art of Computer Programming [57]), a *discipline* (Dijkstra, A Discipline of Programming [37]), and a *craft* (Reynolds, The Craft of Programming [79]). The prevailing belief among the computing community is that program or algorithm design is *inspired*, and that the most significant steps in an algorithm derivation are unexplained *eureka* steps. There lacks an adequate understanding of the algorithm design process, which has had a detrimental effect on how algorithms are explained. This problem has been eloquently described by Dijkstra in his book *A Discipline of Programming* ([37]), where he says

> ... on the one hand I knew that programs could have a compelling and deep logical beauty, on the other hand I was forced to admit that most programs are presented in a way fit for mechanical execution but, even if of any beauty at all, totally unfit for human appreciation. A second reason for dissatisfaction was that algorithms are often published in the form of finished products, while the majority of the considerations that had played their role during the design process and should have justified the eventual shape of the finished program were often hardly mentioned.

We believe that one of the main reasons behind this problem is the lack of a notation or a language for describing the algorithm design process. The algorithms community has long regarded the use of notation as a burden on the algorithm designer. This aversion to the use of notation is apparent from the way most algorithm texts are presented. The algorithms are presented in some low-level pseudo-code which facilitates a relative straightforward translation into an implementation, but is not helpful for understanding the algorithm. Quite often, the algorithm is *explained* by mechanically going over the steps of the algorithm on a carefully selected input instance that is designed to illuminate the more subtle aspects of the algorithm. However, we believe that the use of suitable notation and language can greatly benefit the algorithm design process and can help improve the understanding, and exposition of algorithms. The goal of this thesis is to present evidence in support of our belief, which we do in the following way.

1. We contribute to the development of a systematic, widely-applicable, transformational program design methodology, which relies on the use of an expressive program specification language that is capable of expressing both abstract problem specifications, and their efficient low-level implementations.

2. We demonstrate the feasibility of our methodology by showing that it can be used for getting simplified explanations for complex existing algorithms, and also for discovering new improved algorithms.

A program transformation is a meaning-preserving mapping defined on a programming language. Transformational programming is a program development methodology in which an implementation I is obtained from a specification S by applying a sequence of transformations $T_1, T_2, \ldots, T_k$. The correctness of the implementation I follows from the correctness of the initial specification S and the correctness of each of the transformations. Thus, transformational programming is closely related to Dijkstra's notion of step-wise refinement, in which the algorithm and its proof are developed hand-in-hand. However, Dijkstra was skeptical about the possibility of automating the transformational programming methodology through the use of a program transformation system (Why Naive Program Transformation Systems are Unlikely to Work? [36]). By a *naive* program transformation system, Dijkstra meant a system in which the mathematical concern of correctness, and the engineering concern of efficiency, would be two well-separated stages. The separation of the two primary concerns of programmer, viz. correctness and efficiency, was desirable since it allowed the two concerns to be addressed independently of each other. However, it was also true that for the majority of problems, the best known (most efficient) algorithms were based on mathematical theorems (for example, Euclid's GCD algorithm rests on the fact that $gcd(x, y) = gcd(x - y, y)$), and therefore for such problems, the concerns of correctness and efficiency could not be easily separated.

However, Dijkstra did not discount the possibility of the development of non-naive program transformation systems in which the efficiency considerations would provide a valuable and vital guiding principle for the derivation of correct programs. In our methodology, the selection of transformations is guided by complexity considerations, and this is what distinguishes our work from much of the existing work in the area of program transformations [11, 94, 74, 98, 81].

1.1 Background and Related Work

This thesis has been greatly inspired and influenced by the work of Bob Paige, who, in the late 1970's, set upon the goal of developing a transformational program development methodology for improving the productivity of designing and maintaining correct software. Paige was driven by a strong belief that a small number of rules, whose selection when guided by complexity considerations, could help in the scientific design of a large number of algorithms.

Paige believed that the most important component of a viable transformational program design methodology would be a wide-spectrum language that could express programs at all levels of abstraction, starting from problem specifications down to machine-level implementations. The use of SETL [84, 87], a programming language based closely on the mathematical dictions of finite set theory [102], as the starting point of Paige's research was to some extent, accidental to his being a student at NYU with Jack Schwartz (the creator of SETL) as his advisor. In retrospect, the choice of SETL turned out to be a blessing because of its wide-spectrum and mathematical nature. SETL proved to be highly effective for illustrating the power and broad applicability of finite differencing in the design and analysis of a large number of algorithms. In the classroom, SETL provided students with a simple, powerful and executable mathematical notation, and its small but powerful repertoire of abstract operations proved to be sufficient for modeling a variety of computer science concepts without a need for language extension.

The first transformation developed by Paige was finite differencing, which was a generalization of strength reduction [3], and Earley's Iterator Inversion [39]. The goal was to speedup programs by replacing costly repeated computations by their more efficient incremental counterparts. The finite differencing transformation was different from the other existing transformations in two respects. Firstly, finite differencing could lead to asymptotically large algorithmic improvements to time complexity of algorithms, while the other existing transformations could at best improve the performance of algorithms by constant factors. Secondly, finite differencing was more formalizable and systematically applicable than other existing transformations. Paige's thesis [70] showed that the finite differencing rules could be easily implemented in a program transformation system that would be capable of transforming many abstract, but inefficient program specifications into asymptotically more efficient implementations. The first examples of non-trivial algorithms being derived by finite differencing were presented in [70, 68]. Micha Sharir also did some related work on algorithm derivation by transformations [94] using a more algebraic reformulation [95] of Paige's finite differencing methodology.

In the early 1980's, the area of *Program Analysis* gained a lot of importance, especially due to the presence of very high-level languages like SETL, because it was believed that program analysis could help compile programs in these very high-level languages into efficient implementations. Many program analysis problems (e.g. live variable analysis, reaching definitions, constant propagation etc.) could be specified as least or greatest fixed point computations on monotonic functions. The problem of efficiently computing least/greatest fixed points of monotonic functions led Paige to the discovery of the second transformation, i.e. a crude form of *dominated convergence* [17, 13], a generalized iteration schema for computing fixed points.

Paige used this crude form of dominated convergence together with finite differencing to derive many increasingly difficult algorithms including Hecht's workset algorithms [48] for solving global program analysis problems. However, it was not until his collaboration with Jiazhen Cai that a comprehensive investigation of the dominated convergence transformation was first carried out and presented in [17, 13]. Paige and Cai defined a functional language SQ^+, comprising of the functional subset of SETL augmented with least and greatest fixed points. This language was shown to be useful for specifying numerous programming language and compiler analysis problems, and it was shown how these specifications could be transformed into efficient implementations by using dominated convergence and finite differencing. For example, in [12], an SQ^+ specification of Reif and Lewis's constant propagation algorithm [77] was shown to be transformable into an implementation guaranteed to run in time linear in the size of the program dataflow relation on a uniform cost sequential RAM [1, 62]. (Later on, we show how our methodology can be used to transform an SQ^+ specification of the constant propagation problem into an algorithm that runs in worst-case linear time even on a pointer machine. To the best of our knowledge, this is first linear-time pointer machine algorithm for the constant propagation problem.)

It was realized that the transformational methodology would be incomplete without a final transformation, that would help in the selection of data structures for implementing the set-theoretic associative access operations (such as set membership testing) efficiently. Thus, a three-stage transformational program design methodology was envisioned which comprised of a) dominated convergence, b) finite differencing, and c) data structure selection. First, the dominated convergence and finite differencing transformations would be used to transform an abstract set-theoretic specification into a low-level set-theoretic program, and next, the data structure selection transformation would be used to transform this program into an efficient machine-level implementation.

The problem of data structure selection for efficiently implementing set-theoretic operations had been the subject of much research in the SETL project and had led to the development of many ingenious ideas such as the idea of *basings* developed by Schwartz et al. [82, 34]. The original SETL implementation used hashing to implement all associative access operations, and the idea of basings was intended to reduce the use of hashing. Basings was a way of aggregating data (related by storage or retrieval operations) around finite sets called *bases* that are used like a *bulletin board.* For example, if the elements of two sets A and B were stored in the same *base*, then the operation $A \cap B$ could be performed by traversing and marking the elements of A, and subsequently traversing the elements of B and retrieving the marked elements.

However, SETL's data structure selection method had the following shortcomings. Firstly, the idea of basings was presented as an ad-hoc low-level compiler optimization that was never intended to eliminate hashing entirely, but only to *reduce* the use of hashing. Secondly, this optimization was never amenable to formal complexity analysis, i.e. it was not possible to predict the improvement in the time complexity of the optimized code. Thirdly, this optimization could only be applied to low-level SETL programs. The higher-level SETL programs (those containing more abstract operations such as set union, set comprehension etc.) had to be first transformed into low-level code before this optimization could be applied. However, there was no way to guide the transformation from high-level SETL to low-level SETL in a way so as to ensure that the generated low-level code would gain more benefit from the basings-based optimization.

It was realized that a more desirable solution to the problem of data structure selection would be one that would be applicable to both high-level and low-level languages, be amenable to formal complexity analysis, and that would eliminate hashing entirely, i.e. in which the selected data structures could always guarantee associative access in worst-case $O(1)$ time without the use of hashing. Such data structures which eliminated hashing entirely were first presented in [67, 65]. This led to the beginnings of the development of a methodology for data structure selection based on real-time simulation of set-theoretic operations on a uniform cost sequential RAM. The first non-trivial demonstration of this three-stage methodology was presented in [69] in which an SQ^+ specification of the Single Function Coarsest Partition Problem was transformed using dominated convergence, finite differencing, and real-time simulation, into a new linear-time algorithm.

The greatest impact of this three-step methodology was perceived from its ability to lead to the discovery of new algorithms. A new linear-time pointer machine algorithm for Horn Clause Propositional Satisfiability was presented in [67], improving the previously known linear-time algorithm by Dowling and Gallier [38] that relied heavily on array accesses. Dominated convergence and finite differencing were used in [9] to derive a low-level executable SETL prototype from an SQ$^+$ specification of the Ready Simulation problem. Informal descriptions of data structures were given that improved the algorithm by five orders of magnitude over the previous solution in Blooms thesis. The three-stage methodology was again effectively used to derive a new improved solution to the classical problem of DFA minimization [55].

However, the data structure selection transformation developed thus far, had some major shortcomings. The main hurdles in the effective use of this transformation were as follows.

1. There lacked a formal theory behind the data structure selection transformation. As a result, there was no systematic way of applying this transformation. For each of the problems mentioned above, the transformation was applied on an ad-hoc case-by-case basis.

2. There lacked a suitable notation to describe this transformation. Consequently, the transformation was both difficult to explain, and to prove correct.

3. In each of the above problems, the dominated convergence and finite differencing transformations were applied to abstract functional specifications that were simple to understand. However, the data structure selection transformation had to be applied to the result of the application of these transformations, i.e. to a low-level imperative program. This added to the difficulty of understanding the transformation.

4. Data structure selection could be performed only after the first two transformations had been applied. Consequently, the first two transformations could not be guided by concerns that influenced data structure selection. This was undesirable because if there were more than one ways of maintaining an expression E differentially[1], then ideally one would like the data structure selection transformation to guide the selection of that method of maintaining E differentially which would work most effectively in conjunction with the data structure selection transformation.

Moreover, it was only recently, that a suitable read method from initially creating the desired data structures in linear time (linear in the size of the input string) was developed ([73]).

The goal of this thesis is to extend the above line of research by address the above-mentioned problems with the data structure selection transformation, and demonstrating how the three-step transformational program design methodology can be used for both algorithm explanation and discovery.

1.2 Contribution Of This Thesis

The specific contributions of this thesis may be categorized as follows.

1. Formal development of a data structure selection transformation for a wide-spectrum set-theoretic language containing associative access operations such as set membership testing. A novel contribution of this thesis is the use of a type system (a variant of the Curry/Hindley type discipline for the λ-calculus [30, 52]) for data structure selection. By associating types with data structures, we show that both well-typed abstract functional problem specifications and well-typed low-level imperative programs in our set-theoretic language can be transformed into pointer machine ([57, pages 462-463], [105, 83, 106, 7]) implementations in which each associative access operation is guaranteed to execute in worst-case $O(1)$ time.

2. Demonstration of the effectiveness of the data structure selection methodology for *scaled-up* applications. In this thesis, we present an application of data structure selection to an complex database query optimization algorithm. Willard defined a subset of the relational calculus (called RCS) ([111, 112, 113]), and showed that queries in RCS could be implemented in expected linear time. Willard's original algorithm was extremely difficult, and involved over 80 pages of proofs. Another testament to the difficulty of his algorithm is the fact that it was submitted for publication to JCSS (Journal of

[1]Recall that he finite differencing transformation is concerned with the differential/incremental maintenance of expressions that are expensive to compute repeatedly from scratch

Computing Systems and Sciences) in 1983, but it took thirteen years of refereeing before it was finally accepted for publication in 1996. In this thesis, we show how our type system can be used to transform RCS queries into pointer machine implementations guaranteed to run in worst-case linear time. Thus, Willard's time bound for query-processing is improved from expected linear time to worst-case linear time without degradation of space complexity. Our approach not only improves upon Willard's algorithm, but also simplifies it by yielding shorter, and more precise constructive proofs that lead to an implementation design. This application is the first successful example of scale-up of our methodology for *algorithm explanation and discovery.*

3. Demonstration of how the data structure selection transformation may be used in conjunction with finite differencing and dominated convergence for algorithm explanation and discovery.

   Algorithm Explanation: We use the three-stage program transformation methodology to obtain a much simplified explanation of a linear time algorithm ([5]) for computing the least fixed point of a system of equations on a transition system.

   Algorithm Discovery: We use the three-stage program transformational methodology to discover a new worst-case $O(N^3)$ time algorithm for computing *intra-procedural may-alias information* for a program (where N is the size of the program being analyzed). Our hashing-free $O(N^3)$ time algorithm is a vast improvement over the previously best known algorithm ([51]) that runs in $O(N^5)$ time under the assumption that hashing unit-space data takes $O(1)$ time.

In the next section we provide a brief overview of our approach.

1.3 Overview of Our Approach

The most important component of our transformational program design methodology is an algorithm specification language that is both

wide-spectrum: i.e. capable of expressing both abstract functional problem specifications and their concrete efficient implementations, and

computationally transparent [16]: i.e. amenable to algorithmic analysis, or in other words, a language that facilitates accurate estimation of time and space requirements.

For example, the language SETL is wide-spectrum but not computationally transparent, while the language C is computationally transparent but not wide-spectrum. These properties address the two most important concerns of the programmer, viz. correctness and efficiency. A wide-spectrum language provides a uniform notation for expressing both problem specifications and implementations, and therefore, is useful for proving that the semantics of the implementation are faithful to the specification. Similarly, computational transparency is vital for addressing the issue of efficiency because unless the language is computationally transparent, it would be impossible to compare two programs in terms of efficiency. In order to get a better understanding of the terms *wide-spectrum* and *computationally transparent*, let us see why the language SETL is wide-spectrum but not computationally transparent.

Consider the problem of *Graph Reachability*, in which given a directed graph $G = (V, E)$ (having a set of vertices V and a set of edges E), and a set of vertices W, the problem is to compute the set of vertices reachable along paths of length 0 or more, from the vertices in W. An abstract high-level specification of the graph reachability problem may be written as

$$\text{the least } S \supseteq W \text{ such that } \mathit{Neighbors}[S] \subseteq S. \tag{1.1}$$

Specification 1.1 can be expressed as a valid SETL program by just applying a few syntactic modifications. A more efficient, low-level implementation of graph reachability can also be expressed in SETL as shown below.

```
$S := \{\};$
$WorkSet := W;$
**while** $WorkSet \neq \{\}$**loop**
    $x$ *from* $WorkSet;$
    $S$ **with** $:= x;$                                  (1.2)
    **for** $y$ **in** $Neighbors(x) \mid y \notin S$
        $WorkSet$ **with** $:= x;$
    **endfor**
**endwhile**
```

An implementation of SETL in which the associative access operations such as set membership test could be performed in $O(1)$ time, would generate a linear time implementation of graph reachability from Specification 1.2. Thus, we see that SETL can be used to express both an abstract problem specification (Specification 1.1) and its low-level implementation (Specification 1.2).

Before we see why SETL (and its later implementation SETL2 [99]) is not computationally transparent, let us take a look at the precise definition of computational transparency, originally given by Cai and Paige [16]. Given a prototyping language PL, and an implementation language IL into which programs in PL are transformed, the language PL is said to computationally transparent relative to IL if it has the following properties.

Algorithmic Analyzability: The performance (time complexity) of PL programs is as easily predictable as those of IL programs.

Algorithmic Expressiveness: For every PL computable function f, there is a PL program whose performance matches the best IL program computing f.

Algorithmic Writability: Given an IL program I that implements a PL computable function f, it is straightforward to find a PL program that computes f with its performance matching that of I.

For the purpose of this thesis, the main criterion for computational transparency will be Algorithmic Analyzability. The other two criteria Algorithmic Expressiveness and Algorithmic Writability, though desirable, will not be considered necessary.

The two main reasons for the lack of computational transparency in SETL are 1) the use of hashing to implement associative access operations such as membership testing in a set, and 2) the hidden copy problem that arises because of SETL's copy-value semantics. The following examples illustrate why these two problems lead to the lack of computational transparency in SETL.

Hash-based implementation of associative access operations: Let us consider the following simple fragment of SETL code that reads in two sets S and T and removes the elements in $S \cap T$ from set S.

```
read($S$, $T$);
for $x$ in $S$ loop
    if $x$ in $T$ then
        $S$ less:= $x$;                     (1.3)
    end if;
end loop;
```

The SETL operator "less" performs element deletion from a set. In order to analyze the time complexity of this code fragment, we need some information about how SETL is implemented. SETL uses hash-tables [2] to implement sets so that membership testing in a set involves a hash-table lookup. Moreover, the elements of a set in a hash-table are linked together into a doubly linked list, so that operations such as iteration over the set (e.g. for x in S loop), and element deletion (S less:= x) can also be performed efficiently.

Let us now try to determine the time complexity of the Program 1.3. If sets S and T are sets of integers, one might suppose that a good implementation of hash-tables would, on average, perform a set membership test (x in T) in constant time. Unfortunately, Program 1.3 does not say anything about the type of the elements in sets S and T. For example, the elements of sets S and T may themselves be sets of integers, or worse, sets of sets of integers. In such a case, even the best hash-table based implementations would not perform set membership testing operations in $O(1)$ time even on the average. In fact, it is not even clear how to hash an arbitrary set in constant time. Although it may be reasonable to assume that a hash-table based implementation of sets containing integers or other unit-sized objects may allow constant time membership testing on average, the same assumption for testing membership in arbitrarily nested sets and maps is certainly questionable. Thus, the hash-based implementation of SETL2 makes it impossible to accurately predict the time complexity of *associative access* operations such as set membership testing and map access.

Hidden-Copy Problem: The second reason for the lack of computational transparency in SETL and SETL2 is the *Hidden Copy Problem* that arises because of their *copy-value semantics.* For example, consider the following fragment of code

$$\begin{array}{l} S := T; \\ S \text{ with:= } x; \end{array} \qquad (1.4)$$

The SETL operator "with" performs element insertion into a set. As per the SETL semantics, the effect of the execution of Code Fragment 1.4 must be the creation of a copy of set T into which the element x is inserted, leaving the original copy of set T intact. For the sake of efficiency, the copy-creation in SETL is done in a *lazy* manner. The assignment $S := T$ is implemented by a pointer copy. In addition, a reference count is maintained for all sets and maps, and all update operations (such as S with:= x) are implemented by performing the update in place if the reference count of the set or map being updated is 1, and creating and updating a new copy otherwise. The lazy strategy improves over the *eager* strategy of always making a copy on assignment. Unfortunately, the lazy strategy depends on the dynamic reference counts of objects at run-time. As a result it is impossible to statically predict which update operations will result in the creation of a copy at run-time. Thus, it is not possible to accurately ascertain the time complexity of simple update operations such as "S with:= x".

The above discussion suggests that it would be desirable to have a SETL-like wide-spectrum algorithm specification language as long as it could be made computationally transparent. The first part of this thesis deals with the development of such a language. Instead of selecting one wide-spectrum language, we use three set-theoretic languages which provide increasingly higher levels of abstraction to the programmer. Low SETL, the lowest level language, is a statically typed, executable variant of SETL2 [99], including primitive set operations such as membership testing, set element addition and deletion, arbitrary choice, for-loops through a set, map application, indexed map assignment, and so forth. At the next higher level of abstraction, we have High SETL, which is a statically typed, imperative, executable superset of Low SETL augmented with such abstract operations as set comprehension, quantification, and a variety of operations on binary relations. Finally, the highest level specification language is SQ^+ ([13]), a non-executable, statically typed, functional subset of High SETL augmented with least and greatest fixed points.

The problem of efficient implementation of associative access operations, and of eliminating the hidden copy problem for these languages is dealt with in the following way.

Efficient Implementation of Associative Access: This problem is solved by a novel use of a *type system.* We present a static type system for Low SETL, and show how to make the statically typed subset of Low SETL computationally transparent with respect to a pointer machine. We do this by proving that all well-typed Low SETL programs can be transformed into pointer machine implementations (that are free of hashing) in which the time complexity of execution of each associative access operation is guaranteed to be $O(1)$ time in the worst case. Each Low SETL type is associated with a carefully selected data structure that is designed to facilitate efficient ($O(1)$ time) implementation of a certain set of operations. If x is an object (set or map) having type τ in a program P, and if D is the data structure associated with type τ, then the well-typedness of program P guarantees that the set of operations that could be performed on object x during the execution of program P are a subset of the operations that can be performed efficiently using data structure D. Thus, the well-typedness of a Low SETL program guarantees an $O(1)$ time implementation for all associative access operations in the program.

However, it is important to note that the type system for Low SETL is somewhat *non-standard*. In standard type systems, it is normally assumed that given a program and a type assignment (i.e. a map from variables to types), it is possible to automatically verify whether the program is well-typed or not. In some cases (such as ML [63]), the compiler even automatically derives the types to whatever extent possible. In the case of Low SETL however, this is not true and it is not always possible for a compiler to automatically perform type verification. However, this does not a pose a serious problem because we think of Low SETL not as a language for directly writing programs in, but as a target language for the translation of programs written in High SETL and SQ^+. The type systems for High SETL and SQ^+ do not suffer from the same problem as the Low SETL type system (i.e. type verification can be automatically performed). Moreover, we define a translation from High SETL and SQ^+ to Low SETL, and prove that the Low SETL programs generated from well-typed High SETL and SQ^+ programs, are always well-typed. Thus, it is unnecessary to test the well-typedness of a Low SETL program that is obtained as the result of transformation from a well-typed High SETL or SQ^+ program.

Hidden Copy Problem: The hidden copy problem, which is the second main reason for the lack of computational transparency in SETL2, is a more difficult problem to contend with. It has been the major source of inefficiency in two generations of SETL compilers, and has been known to cause asymptotically-large slow-downs in SETL and SETL2 implementations. Schwartz [88, 85, 86, 90, 89] developed an interesting but complicated value flow analysis for SETL, which was based on statically detecting variables that were guaranteed to be unshared at run-time, thereby allowing in-place updates on these unshared variables. All attempts ([41]) to implement Schwartz's analysis were abandoned after Sharir showed that such an analysis would either be intractable, or too approximate to be useful [93]. The lazy copy strategy employed by the SETL2 compiler based on dynamic reference counting also proved to be unsatisfactory. A pragmatic solution to the hidden copy problem was proposed by Goyal and Paige [44], in which a combination of dynamic reference counts, a static liveness analysis, and a static must-alias analysis provides an effective heuristic for eliminating hidden copies. A very local implementation of this approach for SETL2 was shown to speed up a computation intensive SETL2 application APTS by a factor of 10. Although this solution was able to improve the performance of SETL in practice, it cannot be used to statically guarantee the elimination of copies.

In this thesis we side-step the problem of hidden copies for Low SETL by imposing the requirement that all legal Low SETL programs have the same behavior under copy-value and reference semantics (analogous to the notion of an Ada-83 program being *erroneous* when it has a different meaning under call by reference, and call by value-result). This makes Low SETL computationally transparent because the fact that the program has identical behavior under both copy-value and reference semantics implies that it is unnecessary to create any copies during the execution of the program and that all updates can be performed in place. The caveat to our approach is that this requirement makes Low SETL unsuitable for manual programming of large applications because the burden of proving that the program behavior is identical under both copy-value and reference semantics lies on the programmer. However, once again we do not see this as a serious problem because we consider SETL as the target language for transformation of programs written at in High SETL and SQ^+. It turns out that the translation of a functional subset of High SETL (i.e. High SETL expressions) and SQ^+ is guaranteed to produce Low SETL code which has identical behavior under copy-value and reference semantics. We obtained substantial credibility for this approach in [16, 43, 42] where complex SQ^+ and High SETL specifications were translated to Low SETL programs that were guaranteed not to be erroneous[2] a priori.

[2]in the sense of having identical behavior under both copy-value and reference semantics

1.4 Testing the Viability of Our Approach

As outlined in the previous section, our approach to making Low SETL, High SETL, and SQ$^+$ *algorithmically analyzable* is based on the use of a type system, wherein the well-typedness of a program ensures that it can be implemented in a way such that each associative access operation executes in $O(1)$ time without the use of hashing. A legitimate question, then arises about the *algorithmic expressiveness* of these languages. The question of mere expressiveness is answered by the fact that all three languages are Turing Complete. However, the question of algorithmic expressiveness asks about whether for any computable function f, it is a possible to write a program in these languages that matches the algorithmic complexity of the best known pointer machine implementation of function f.

We will not attempt to address the question of algorithmic expressiveness formally in this thesis, as the question is interesting and complex enough to be the subject of another dissertation. Instead, we will partially address the issue by considering examples of algorithmic problems drawn from various fields of computer science. In this thesis, we show that our languages are capable of expressing the best known algorithms for textbook problems such as Graph Reachability, Cycle Testing etc. and also for more involved problems from the subject of program analysis, such as Constant Propagation, and May-alias Analysis. Perhaps the most convincing argument about the expressiveness of Low SETL and High SETL comes from the worst-case linear-time algorithm for Willard's Relational Calculus Subset (RCS). As mentioned earlier, Willard's original algorithm is extremely difficult and makes extensive use of hashing in virtually all the data structures used in the algorithm. The fact that our approach simplified the presentation of the algorithm, and eliminated hashing in entirety, is an avid testimonial to the algorithmic expressiveness of these languages.

1.5 Outline of the Thesis

The rest if the thesis is organized as follows.

Chapter 2 defines the language Low SETL. An operational semantics is defined for Low SETL and that can be easily simulated on a pointer machine. A static type system is defined for Low SETL and it is shown that every well-typed Low SETL program is guaranteed to have a pointer machine implementation in which all associative access operations can be executed in $O(1)$ time.

Chapter 3 defines the language High SETL. A static type system is defined for High SETL and an operational semantics is defined for the well-typed subset of High SETL in terms of Low SETL implementations. It is shown that the specified translation of well-typed High SETL programs is always guaranteed to lead to well-typed Low SETL implementations. Moreover, it is shown how the worst-case time complexity of execution of the functional subset of High SETL may be computed in a systematic manner at a very high level of abstraction, without resorting to bean counting arguments.

Chapter 4 takes a comprehensive look at our first application, i.e. an algorithmic problem related to database query optimization. We show how queries in the expected linear-time subset of Willard's RCS can be transformed into High SETL programs that are guaranteed to run in worst-case linear time without the use of hashing. This application provides evidence of the expressiveness of the Low SETL and High SETL type systems, and of the usefulness of the data structure selection transformation for *algorithm explanation and discovery.*

Chapter 5 defines the language SQ$^+$. Once again a static type system is defined and sufficient conditions are stated under which well-typed SQ$^+$ programs are shown to be transformable into well-typed High SETL implementations.

Chapter 6 defines a subset L_{IO} of SQ$^+$ consisting of programs that are guaranteed to have implementations that run in time linear in the sum of the sizes of the inputs and the outputs, in the worst case. This chapter demonstrates an application of the three-stage transformational program design methodology by using dominated convergence, finite differencing and data structure selection together to get asymptotically optimal linear-time algorithms for a large and interesting class of problems. We show

how text book problems such as Graph Reachability and Cycle Testing, and more intricate program analysis problems such as Constant Propagation can be shown to belong to L_{IO} and thus, shown to have efficient linear-time implementations.

Chapter 7 takes a look at our second main application, i.e. an algorithmic problem of computing the least fixed point of a system of equations. By showing that this problem belongs to L_{IO}, we get an algorithm that matches the best known algorithm for the problem. More importantly, by comparing our algorithm with some of the previous algorithms (that are highly unintuitive and difficult to understand), we demonstrate the usefulness of our methodology for *algorithm explanation*.

Chapter 8 takes a look at our final application, i.e. the problem of computing a intra-procedural may-alias analysis. We show how our language-theoretic approach leads to the discovery of a new $O(N^3)$ time (where N is the size of the program being analyzed) algorithm which vastly improves that previously known best algorithm having a time complexity of $O(N^5)$.

Chapter 9 discusses some limitations of our work and possibilities for future work.

Chapter 2

Low SETL

In this chapter we define Low SETL, a statically typed, executable variant of SETL2, including primitive set operations such as membership testing, set element addition and deletion, arbitrary choice, for-loops through a set, map application, indexed map assignment, and so forth. Our goal is to make Low SETL *computationally transparent* with respect to a pointer machine ([57, pages462-463], [105, 83, 106, 7]). In order to do this, we make use of a novel type system in which types are associated with data structures, and the well-typedness of a Low SETL program guarantees that each associative access operation in the program can be implemented efficiently (in $O(1)$ time) without the use of hashing.

In Section 2.1 we define the syntax, the set of valid types, and the set of values associated with each type, for Low SETL. In Section 2.2 we give an informal description of the data structures associated with the types. In Section 2.3 we define a dynamic operational semantics for Low SETL, and in Section 2.4 we describe how Low SETL operations may be simulated on a pointer machine. In Section 2.5 we define a static semantics for Low SETL, and in Section 2.6 we define a notion of *consistency* between the static and the dynamic semantics. Finally, in Section 2.7 we present the main result of this chapter, i.e. *every well-typed Low SETL program may be transformed into a pointer machine implementation in which each associative access operation executes in* $O(1)$ *time.*

2.1 Definition of Low SETL

The syntax for Low SETL is given in Figure 2.1. Low SETL is a strongly typed imperative language having finite *sets* and *maps* as built-in datatypes, along with some primitive set theoretic operations such as set membership test, set element addition, etc.

If v is a set[1] then expression $\ni v$ evaluates to an arbitrary element of v if v is non-empty, and a special value om (undefined) otherwise. A Low SETL map may be thought of as a set of pairs of values. A map v_1 is an *smap* i.e. a single-valued map if for all pairs $[x_1, y_1], [x_2, y_2] \in v_1$, $(x_1 = x_2) \implies (y_1 = y_2)$. For the sake of brevity, we only consider single-valued maps here. The generalization to multi-valued maps (which we call *mmaps*) is straightforward. If v_1 is a single-valued map, then expression $v_1(v_2)$ evaluates to value v_3 if $[v_2, v_3] \in v_1$, and the undefined value om otherwise. The boolean expression $v_1 \in v_2$ returns *true* if v_1 is an element of set v_2, and *false* otherwise. The boolean expression $\mathit{IsEmptySet}(v)$ (respectively $\mathit{IsEmptyMap}(v)$) returns *true* if set (respectively map) v is empty, and *false* otherwise. If expression e evaluates to the value y, then the indexed map assignment $v_1(v_2) := e$ adds the pair $[v_2, y]$ to map v_1, and removes any existing pair which has v_1 as its first component. The operators *with* and *less* are the set element addition and deletion operators respectively. The command v_1 *from* v_2 extracts an arbitrary element from set v_2 and assigns its value to v_1. The command $\mathit{InitSet}(v)$ (respectively $\mathit{InitMap}(v)$) initializes set (respectively map) v to the empty set (respectively empty map).

Low SETL also includes arithmetic expressions of the form n (an integer literal), expressions $e_1 + e_2$,

[1]Actually v is a set-valued variable. For now we will use the variable name to represent both the variable and its value.

$v, v_1, \ldots,$: Variable Names
$e, e_1, \ldots$: Expressions
$be, be_1, \ldots$: Boolean Expressions
$c, c_1, \ldots$: Commands
$P, P_1, \ldots$: Command Sequences
$op : with | less$

Expressions

$$e ::= v \mid \ni v \mid v_1(v_2) \mid \text{om}$$

Boolean Expressions

$$\begin{aligned} be \quad ::= \quad & v_1 \in v_2 \mid v_1 == v_2 \mid \mathit{IsEmptySet}(v) \mid \mathit{IsEmptyMap}(v) \mid \\ & \neg be \mid be_1 \wedge be_2 \mid be_1 \vee be_2 \end{aligned}$$

Command Sequences

$$P ::= c \mid c; P$$

Commands

$$\begin{aligned} c \quad ::= \quad & v := e \mid v_1(v_2) := e \mid v\ op := e \mid v_1\ \mathit{from}\ v_2 \mid \mathit{InitSet}(v) \mid \mathit{InitMap}(v) \mid \\ & \text{if } be \text{ then } P_1 \text{ else } P_2 \mid \text{while } be \text{ loop } P \text{ endloop} \mid \\ & \text{for } v_1 \in v_2 \text{ loop } P \text{ endloop} \mid \\ & \text{for } v_1 \in \mathit{domain}(v_2) \text{ loop } P \text{ endloop} \end{aligned}$$

Figure 2.1: Syntax for Low SETL

$e_1 - e_2$, $e_1 \times e_2$, e_1/e_2 etc. where e_1 and e_2 are themselves integer-valued expressions. Similarly, Low SETL also contains the boolean literals *true* and *false*. We have omitted these from the grammar in Figure 2.1 and the operational semantics given later for the sake of brevity. The semantics for evaluating arithmetic and boolean expressions are straightforward, and may be found in any semantics book like [114]. We have separated boolean expressions from other expressions (arithmetic, and set or map-valued expressions) just for the sake of convenience.

A Low SETL program is a sequence of one or more commands separated by semi-colons. A command c may either be a simple assignment, an indexed map assignment, a set update operation, an initialization of a set or map, a conditional statement, a for loop or while loop.

We disallow modifications to variables v_1 and v_2 inside for loops of the form "for $v_1 \in v_2$ loop P endloop", or "for $v_1 \in \mathit{domain}(v_2)$ loop P endloop". In other words, the value of variable v_2 must be the same before and after execution of the for loop, and the value of variable v_1 should remain fixed for the duration of each iteration. Since every valid Low SETL program must have the same behavior under copy-value and reference semantics, we consider the programs that contain modifications to the values of variables v_1 or v_2 as a result of aliasing, to be erroneous. For ease of presentation, we shall assume that all loop-iteration variables (i.e. variables like v_1 in "for $v_1 \in v_2$ loop ... endloop") are distinct from each other, and only used inside their corresponding loops.

In this thesis, we restrict our attention to batch-input programs, i.e. programs that read their entire input once at the beginning of the program. We assume that each Low SETL program has a single read statement of the form $read(v_1, v_2, \ldots)$ at the beginning of the program, and that there are no more reads in the rest of the program. The function of this read statement is to read-in the input and set up the appropriate data structures corresponding to each type. The details of how the task of reading may be accomplished can be found in several published papers ([66, 73]), and are omitted from this thesis. Paige and Yang ([73]) showed that the time complexity of the read routine required for Low SETL is linear in the size of the input. Thus, the cost of reading does not add any asymptotic factors to the cost of algorithms that must at least read in their entire input, and can be safely ignored.

Low SETL is a statically typed language. The type system for Low SETL will be presented in Section 2.5. The set of valid types *Type* ranged over by variable τ is given by

$$\begin{array}{lcl} \tau & ::= & \mathit{strong\_set}(b) \mid \mathit{strong\_smap}(b, \sigma) \mid \sigma \mid \mathit{bool} \\ \sigma & ::= & \mathit{int} \mid b \mid \mathit{set}(\sigma) \mid \mathit{smap}(\sigma_1, \sigma_2) \end{array} \tag{2.1}$$

where the type variable b corresponds to special types called base types. The types ranged over by σ include the types *int*, base types, and arbitrarily nested sets and single-valued maps[2] of types ranged over by σ. The set of valid types *Type* in Low SETL includes the type *bool*, the types ranged over by σ, and special types of sets and maps called strongly based sets (denoted by $\mathit{strong\_set}(b)$) and strongly based maps (denoted by $\mathit{strong\_smap}(b, \sigma)$) respectively[3]. For the sake of brevity, we have not added other simple atomic types such as *bool*, *real*, *char*, etc. in the production for σ. The type *int* should be considered representative of all such atomic types. The type *bool* in the production for τ allows us to differentiate between boolean expressions and other non-boolean expressions. Inclusion of type *bool* in the production for σ would not cause much change except for allowing additional types such as $\mathit{set}(\mathit{bool})$, $\mathit{smap}(\mathit{bool}, \mathit{bool})$ etc.

The elements of a strongly based set of type $\mathit{strong\_set}(b)$ must be of base type b. Similarly, the domain of a strongly based map of type $\mathit{strong\_smap}(b, \sigma)$ must contain elements of base type b, and its range must contain elements of type σ. The types $\mathit{strong\_set}(b)$ and $\mathit{strong\_smap}(b, \sigma)$ are associated with special data structures that allow associative access operations to be performed in $O(1)$ time. Grammar 2.1 disallows types of the form

$$\mathit{set}(\mathit{strong\_set}(b)), \mathit{strong\_smap}(\mathit{strong\_set}(b), \mathit{strong\_set}(b')), \text{etc. },$$

for reasons that will soon become clear.

Each type τ is associated with a set of values and a specific implementation. For example, the type *int* is associated with the set of all integers and any conventional implementation of integers. Similarly, the type $\mathit{set}(\mathit{int})$ is associated with the set of all finite sets of integers, and an implementation in the form of a doubly linked list of *distinct* integers.

Each base type b appearing in the program is uniquely associated with a subtype constraint of the form $b < \sigma$. In conventional type systems, the subtype constraint $b < \sigma$ would imply that the set of values associated with type b is a subset of the set of values associated with type σ. However, in our type system, the set of values associated with types b and σ are the same, but the implementations associated with types b and σ are different. The subtype constraint $b < \sigma$ implies that the implementation associated with type b is more constrained (in a sense that will become clear later) than the implementation associated with type σ, and that an implementation corresponding to base type b can be coerced into an implementation corresponding to type σ in $O(1)$ time. In other words, one can efficiently go from a more constrained implementation (for type b) to a less constrained implementation (for type σ). The reverse, i.e. going from a less constrained implementation for type σ to a more constrained implementation for type b, cannot be done efficiently, and is therefore disallowed. In conventional type systems, the subtype constraint $b < \sigma$ implies that a value of type b may be used wherever a value of type σ is expected, but not vice-versa. Analogously, in our type system, the fact that an implementation for type b can be coerced into an implementation for type σ in $O(1)$ time implies that an implementation for type b may be used wherever an implementation for type σ is expected, but not vice versa.

Each Low SETL program is associated with a set of subtype constraints C containing exactly one subtype constraint of the form $b < \sigma$ for each base type b appearing in the program (i.e. appearing in the type of some variable in the program). Given a set of subtype constraints C, we define the subtype constraint graph G_C to be a graph that has exactly one vertex for each base type b appearing in C, and an edge from vertex b_1 to b_2 iff C contains a subtype constraint of the form $b_1 < \sigma_1$ where b_2 appears in σ_1. A set of subtype constraints C for a program is said to be admissible iff the subtype graph G_C is acyclic, and C does not contain any subtype constraint of the form $b_1 < b_2$ where b_1 and b_2 are both base types. The acyclicity

[2]the "s" in "smap" means single-valued map

[3]The notation used for strongly based sets and maps in this thesis is different from the notation used in previous work such as [12, 67, 65, 73, 43]. Previously the notation used to denote a strongly based sets and smaps was $\mathit{set}(b\text{-}\mathit{strong})$ and $\mathit{smap}(b\text{-}\mathit{strong}, \sigma)$ respectively

constraint ensures that the values corresponding to each base type b are identical to the values corresponding to some unique type σ that does not contain any base types. For example, consider the subtype constraint set

$$C = \{b_1 < set(b_2),\ b_2 < int\}.$$

Then the values corresponding to types b_1 and b_2 are the values corresponding to types $set(int)$ and int. It is convenient to disallow constraints of the form $b_1 < b_2$ for technical reasons that make later proofs easier.

For a given set of subtype constraints C, each type τ is associated with a set of values through a binary relation $T \subseteq Type \times Values$ defined as follows:

$$\begin{array}{rcl}
T[bool] & = & \{true, false\} \\
T[int] & = & \{0, -1, 1, -2, 2, \ldots\} \\
T[b] & = & T[\sigma] \text{ where } b < \sigma \in C \\
T[set(\sigma)] & = & \{s : s \subseteq T[\sigma] \mid |s| < \infty\} \\
T[smap(\sigma, \sigma')] & = & \{s : s \subseteq T[\sigma] \times T[\sigma'] \mid |s| < \infty \ \wedge \ s \text{ is single-valued}\} \\
T[strong\_set(b)] & = & T[set(b)] \\
T[strong\_smap(b, \sigma)] & = & T[smap(b, \sigma)]
\end{array}$$

As mentioned previously, a subtype constraint $b < \sigma$ implies that the implementation associated with type b is more constrained than the implementation associated with type σ. The key difference in the implementations associated with types b and σ is that the former is specially designed to allow a test for equality of two values in $O(1)$ time. For example, if a base type b satisfies the subtype constraint $b < set(int)$, then although the set of values associated with type b and type $set(int)$ are the same, the implementation of values of type b allows two values to be compared for equality in $O(1)$ time, whereas the implementation of values of type $set(int)$ does not allow comparison for equality in $O(1)$ time. The idea behind the implementation of base types is simple. We simply preprocess the input (at read time) to identify all distinct values of base type b and assign unique value numbers to these values. We do not allow values of a base type to be created dynamically at run-time[4], and all comparisons of values of a base type are performed by comparing value numbers.

The idea of base types comes from the ingenious idea of basings developed by Schwartz et al [82, 34] during the SETL project. Basings is a way of aggregating data (related by storage or retrieval operations) around finite sets called *bases* that are used like a *bulletin board*. For example, if the elements of two sets A and B were stored in the same *base*, then the operation $A \cap B$ could be performed by traversing and marking the elements of A, and subsequently traversing the elements of B and retrieving the marked elements. More details about the implementation of base types and how they are used to implement associative access operations efficiently are given later.

In Section 2.2 we informally describe the implementations (i.e. data structures) associated with different types, and the operations that may be efficiently performed using these data structures. These ideas are formalized in Sections 2.3 and 2.5 in which a dynamic operational semantics and a type system for Low SETL are defined. The informal description in Section 2.2 should prove helpful in understanding the intuition behind the formal (and rather terse) development in the later sections.

2.2 Informal Description of Data Structures

2.2.1 Base Types

We start by explaining how values of a base type b are implemented. Each base type b satisfying the subtype constraint $b < \sigma$ is implemented as a finite set R_b of records such that each record corresponds to a distinct value of type b read from the input. Note that the set R_b contains records corresponding to only those values

[4]This restriction can be partially relaxed. We can allow creation of a value of base type b under the restriction that the newly created value be distinct from all existing values of type b.

of type b that are read from the input, and is hence finite. A value of base type b is implemented as a pointer to a record in set R_b. Moreover, each record in set R_b contains a key field containing a pointer to a different implementation (corresponding to type σ) of the same value (except for the case when σ is *int*, in which case the key field contains the integer directly rather than a pointer to the integer). This implementation of elements of base types satisfies the following two properties.

1. Given an implementation corresponding to a base type b satisfying the subtype constrain $b < \sigma$, we can get an implementation corresponding to type σ in $O(1)$ time. This is obvious from the fact that the key field of each record in R_b contains a pointer to the implementation corresponding to type σ.

2. Two values of type b are comparable for equality in $O(1)$ time. This is possible because the initial construction of set R_b at input read time, guarantees that each distinct record in R_b corresponds to a distinct value of type b. Thus, the comparison of two values of type b can be implemented simply as a pointer comparison. Note that the task of preprocessing the input to create the set R_b can be done in time linear in the input size [66, 73], and therefore does not add to the asymptotic time complexity of the algorithm[5].

2.2.2 Sets and Maps

A value of type $set(\sigma)$, which by definition is a finite set of values of type σ, is implemented as a doubly linked list of pointers to implementations of values of type σ. Again, the only exception to this is when type σ is *int*, in which case we implement $set(int)$ as a doubly linked list of integers rather than pointers to integers. For every set, the *first* and *last* pointers point to the first and last elements of the doubly-linked list.

Let us look at a few illustrative examples. Figures 2.2, 2.3, and 2.4 show the data structures created when two sets S and T are read as input. In each case, the two sets S and T are $\{\{1\},\{1,2\}\}$, and $\{\{1\},\{1,3\}\}$ respectively. The different data structures correspond to the different types of sets S and T. In Figure 2.2 the types of sets S and T are

$$\begin{array}{l} S : set(set(int)) \\ T : set(set(int). \end{array} \tag{2.2}$$

In Figure 2.3 the types of sets S and T are

$$\begin{array}{l} b < set(int) \\ S : set(b) \\ T : set(b). \end{array} \tag{2.3}$$

Finally, in Figure 2.4 the types of sets S and T are

$$\begin{array}{l} b_1 < int \\ b_2 < set(b_1) \\ S : set(b_2) \\ T : set(b_2). \end{array} \tag{2.4}$$

Sets such as S and T in Figure 2.2, whose elements are not of a base type, are called *unbased* sets. In Figure 2.2 an equality test between two arbitrary elements selected from sets S and T respectively would involve comparing two arbitrary sets of integers for equality, and in general, cannot be performed in $O(1)$ time. However, in Figure 2.3 the elements of sets S and T are of a base type b, and two arbitrary elements selected from sets S and T respectively can be compared for equality in $O(1)$ time. Sets such as S and T in Figure 2.3, whose elements are of a base type, are called *weakly based* sets. By definition, sets are unordered and therefore any ordering of elements in a linked list implementation of sets is permissible. For example, the ordering of elements in set S is different in Figures 2.2 and 2.3. In Figure 2.4, in addition to the elements

[5]assuming that the algorithm must at least read its entire input

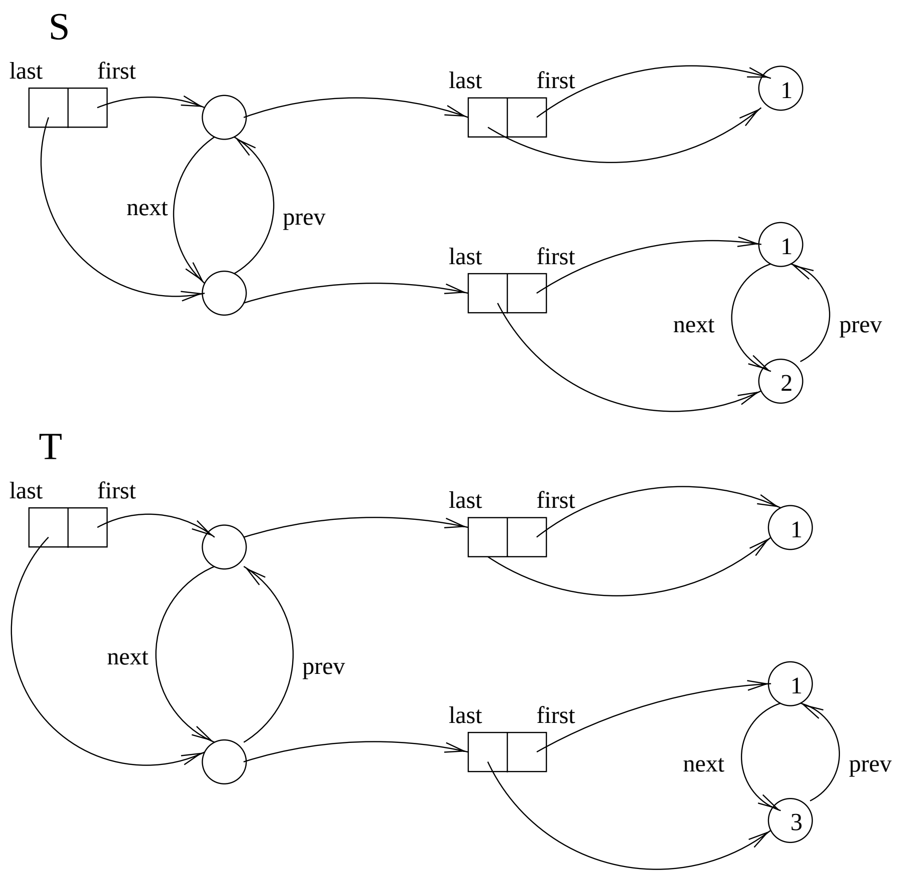


Figure 2.2: Data Structure for $S : set(set(int))$ and $T : set(set(int))$

of sets S and T being based, even the elements of the elements of sets S and T are based. This illustrates that based elements can be nested arbitrarily deeply in sets and maps.

A single-valued map f of type $smap(\sigma_1, \sigma_2)$ can be thought of as a set of (ordered) pairs of values of type σ_1 and σ_2. It is implemented by implementing the domain of the map as a set, and having a map *image* that associates every element x in the domain with the corresponding element y in the range such that the pair $[x, y]$ belongs to map f. A multi-valued map f of type $mmap(\sigma_1, \sigma_2)$ (which is omitted here for the sake of brevity) would be implemented similarly. Its domain would be implemented as a set, and the map *image* would associate each element x in the domain with a non-empty set of elements $\{y_1, y_2, \dots\}$ (called the image of x under map f), such that each pair $[x, y_i]$ belongs to f. Thus, the implementation of a multi-valued map of type $mmap(\sigma_1, \sigma_2)$ is similar to the implementation of a single-valued map of type $smap(\sigma_1, set(\sigma_2))$[6]. If the elements in the domain of a map are not of a base type, then the map is said to be unbased. If the elements in the domain of a map are of a base type, and its domain is implemented like a weakly based set, then the map is said to be weakly based.

2.2.3 Strongly based Sets and Maps

Before we see how the types *strong_set*(b) and *strong_smap*(b, σ_1) are implemented, let us first see why these types are needed. In Figure 2.3 sets S and T are weakly based and therefore two arbitrary elements selected from S and T respectively can be compared for equality in $O(1)$ time. However, the data structure in

[6]There is however one subtle difference. In the case of a single-valued map f of type $smap(\sigma_1, set(\sigma_2))$, it is possible for the pair $[x, \{\}]$ to belong to map f. However, in the case of a multi-valued map g of type $mmap(\sigma_1, \sigma_2)$, we say that an element x belongs to the domain of g only if the image of x under g is non-empty.

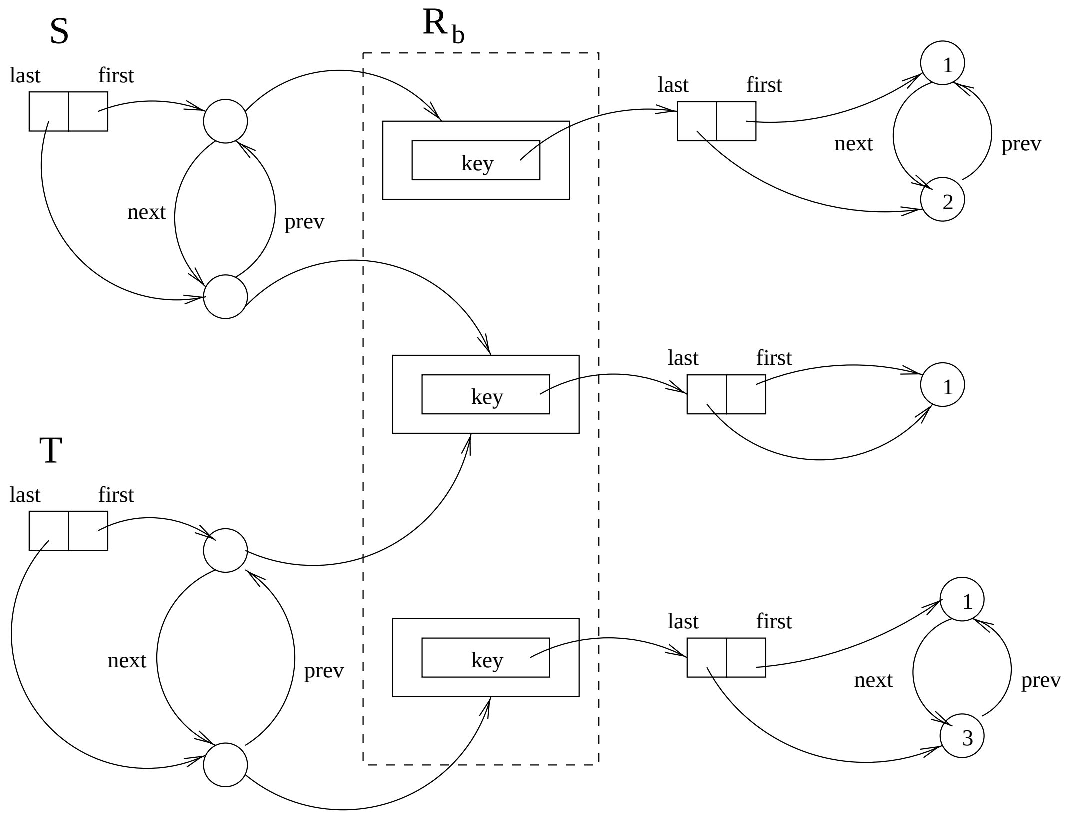


Figure 2.3: Data Structure for $b < set(int), S : set(b)$, and $T : set(b)$

Figure 2.3 does not allow us to test whether an arbitrary element of set S belongs to set T in $O(1)$ time. Values of type $strong\_set(b)$ or $strong\_smap(b, \sigma_1)$ are implemented using data structures that allow such membership tests and other associative access operations in $O(1)$ time.

Suppose we want to test whether an arbitrary element of base type b belongs to set T in $O(1)$ time. A simple way of doing this is to add a boolean field to every record in set R_b, which is true if and only if the value corresponding to this record belongs to set T. Thus, the membership test can be performed in $O(1)$ time by inspecting this boolean field. Now suppose that in addition to membership testing, we also want to be able to delete an arbitrary element x from set T in $O(1)$ time. The deletion of element x from set T requires 1) setting the boolean field in the record for x to false, and 2) locating and deleting the pointer to record x in the doubly linked list implementation of set T. The former can be done in $O(1)$ time, but the latter would take time proportional to the cardinality of set T. This problem is solved by incorporating the *prev* and *next* pointers of the doubly linked list implementation of record T within the records of R_b (see Figure 2.5). Thus, if element x of base type b is a member of set T, then the base record corresponding to x contains *true* in its boolean field, and pointers to other elements of set T in the *prev* and *next* fields. If, however, element x does not belong to set T, then the boolean field is *false*, and the *prev* and *next* fields are empty (or *null*). Such a data structure, shown in Figure 2.5, allows operations such as membership testing, and element addition and deletion in set T in $O(1)$ time. It also allows iteration over the elements of set T in time proportional to the cardinality of T. Such a set T is said to be strongly based with type $strong\_set(b)$. Thus, in Figure 2.5, set S is weakly based and set T is strongly based.

The implementation of a strongly-based single-valued map is done by implementing its domain as a strongly based set. In the case of a map f, the map access operation $f(x)$ also involves an associative access. So, for a strongly based map, every record in R_b contains four additional fields: a boolean field (for membership in the domain of map f), *prev* and *next* fields (for a doubly linked list implementation of the domain of f), and finally an *image* field storing a pointer to the image of this element. Thus, a strongly

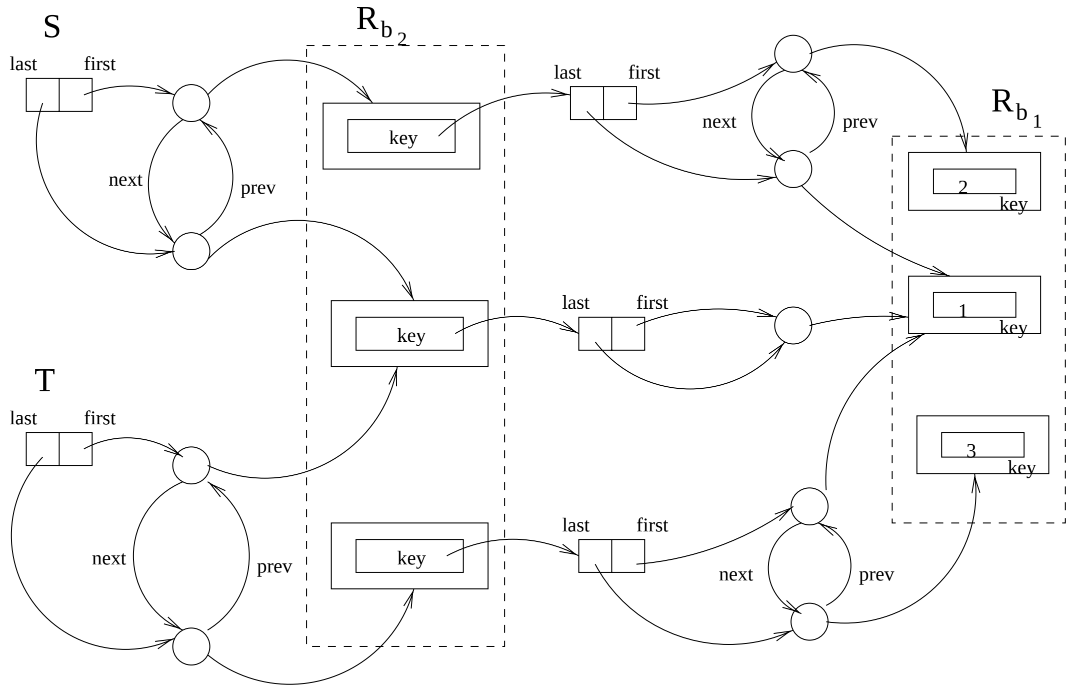


Figure 2.4: Data Structure for $b_1 < int, b_2 < set(b_1), S : set(b_2)$, and $T : set(b_2)$

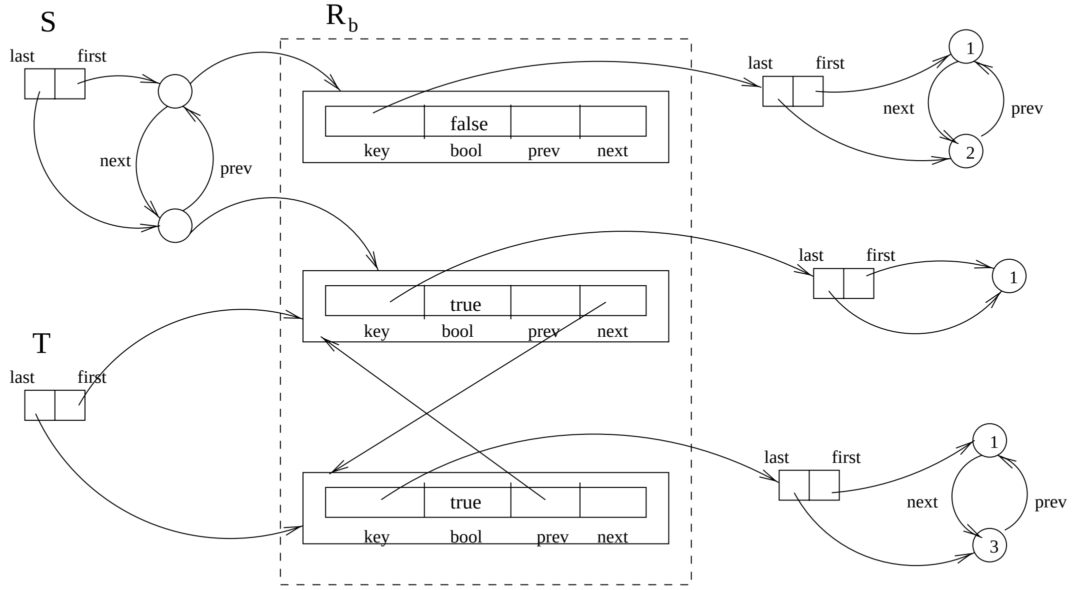


Figure 2.5: Data structures for $b < set(int)$, $S : set(b)$, $T : strong\_set(b)$. Set S is weakly based and set T is strongly based.

based map facilitates an $O(1)$ time implementation for map access operations such as $f(x)$.

Suppose we want to test the membership of an element x of base type b in both sets S and T. The obvious solution is to make both sets S and T strongly based, i.e. to have every record in R_b contain separate boolean, *prev*, and *next* fields for both sets S and T. In general, every base record in R_b contains a separate set of fields for every set or map based strongly on base type b. Such an implementation is possible because the number of sets and maps based strongly on any base type b, is a constant for each program. This is because

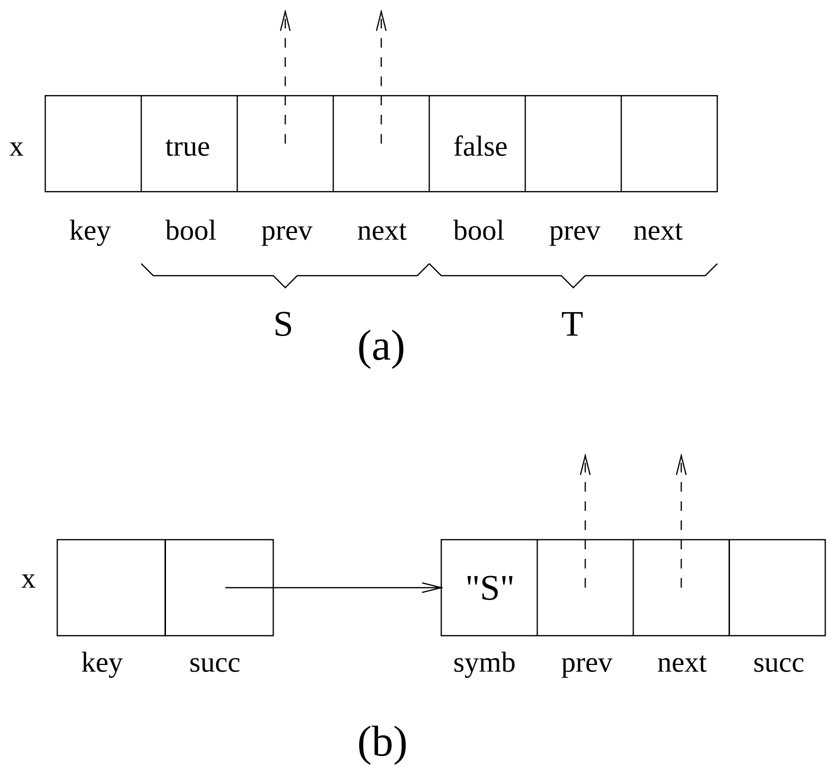


Figure 2.6: x is an element of base type b. Both sets S and T are strongly based.(a) Implementation of a base record on a RAM. (b) Implementation of a base record on a Pointer Machine.

our type system does not allow the nesting of strongly-based sets and maps inside other sets and maps. For example, the types $set(strong\_set(b))$ and $set(strong\_smap(b, \sigma_1))$ are not valid in our type system. Therefore, the total number of strongly-based sets and maps in a Low SETL program are bounded by the number of variables in a program. Notice, however, that if types such as $set(strong\_set(b))$ were allowed, then it would be possible to have arbitrarily many strongly-based sets in the program, since a set may have arbitrarily many elements.

There is one remaining problem with the data structure for base records described above. Consider a strongly based set T of type $strong\_set(b)$, which is one of the many strongly based sets and maps on base type b. In order to perform the membership test $x \in T$, we need to locate the boolean field corresponding to set T in the base record implementation of element x. On a RAM [1, 62] this can be done easily by using indices for each strongly based set and map. A pointer machine, however, does not allow address arithmetic, and therefore the base records have to be implemented a little differently. A base record for element x is implemented as a record containing a key field and pointer to a singly linked list of nodes which contains exactly one node for every strongly based set containing x, and every strongly based map containing x in its domain. Each node in this singly-linked list has a *symb* field (each *symb* being uniquely associated with some strongly based set or map), fields for *prev* and *next* pointers, and a field for the *image* pointer in the case of strongly based maps.

In Figure 2.6 an example shows the two implementations of base records. We have an element x of base type b and two strongly based sets S and T, each of type $strong\_set(b)$. In this example we see that element x is a member of set S but not of set T. Therefore, the pointer machine implementation of the base record contains only a node for S in its singly linked list. The presence of the node for S signifies that element x is a member of set S, and the absence of a node for T indicates that element x is not a member of set T. The pointer machine implementation of base records incurs a greater cost in terms of the time complexity of associative access operations. For example, in order to test the membership of an element x in some strongly based set T, we need to traverse the singly-linked list until we either find a node for set T, or we reach the end of the list. However, as mentioned before, the total number of strongly based sets and maps in a Low SETL program is a constant independent of the input to the program. Furthermore, each record's linked list can contain at most one node per strongly based set or map. Thus, the time to search in this list is $O(1)$ time.

2.2.4 A Special Note on Self-Access

As explained in Section 2.2.3, the map-access operation $f(x)$ can be performed in $O(1)$ time if map f is strongly based, and element x is of the suitable base type. If however, map f is not a strongly based map, then the operation $f(x)$ requires a traversal over the domain of map f to locate an element that equals x. Moreover, if element x is not of a base type, then the equality tests between element x and the elements in the domain of map f would be expensive operations. Thus, if map f is weakly based or unbased, then the map-access operation $f(x)$ cannot, in general, be performed efficiently. There is, however, an important exception. If the expression $f(x)$ appears inside a for-loop of the form "for $x \in domain(f)$ loop ... endloop", then the map-access operation $f(x)$ can be performed efficiently by simply keeping track of the position of element x in the domain of map f. We use the term *self-access* to describe this kind of map-access on a weakly based or unbased map, which can always be performed in $O(1)$ time.

Now that we have an informal understanding of Low SETL data structures, we go on to define an operational semantics for Low SETL in Section 2.3. Next, in Section 2.4, we will describe how this semantics can be implemented on a pointer machine.

2.3 Dynamic Operational Semantics

The operational semantics for Low SETL is a combination of *big-step* and *small-step semantics.* Small-step semantics, also known as *Structural operational semantics* was introduced by Gordon Plotkin in [75]. Structural operational semantics is used to describe how the individual steps of the computation take place. Big-step semantics, also called *Natural semantics* (or Kahn's style natural semantics) is derived from the structural operational semantics, and the basic ideas were first presented in [26] for a functional language. Natural semantics is used to describe how the overall results of executions are obtained. We assume that the reader is familiar with both small-step and big-step semantics, and do not include further explanation of these concepts. For readers who are not familiar with these concepts, an introductory treatment may be found in numerous books on semantics such as [49, 64, 97].

We use a big-step semantics to define evaluation of expressions and boolean expressions, both of which evaluate to a particular value in a given state[7] of the computation but do not modify the state. A small-step semantics is used to define the evaluation of commands, which act as state transformers, i.e. lead from one state of the computation to another. Such a combination of big-step and small-step semantics is sometimes called *transition* semantics ([46])

We assume that V_{loc} is an infinite set of locations including a special location l_{om} that implements the Low SETL expression om (which corresponds to the value *undefined*). Each type τ is associated with a set V_τ, which is the set of implementations of all values of type τ. V_τ is defined recursively as

$$\begin{array}{rcl} V_{bool} & = & \{\mathit{true}, \mathit{false}\} \\ V_{int} & = & \{\mathrm{int}(x) : x = 0, 1, -1, 2, -2, \ldots\} \\ V_b & = & \{\mathrm{base}(l) : l \in V_{loc}\} \\ V_{set(\sigma)} & = & \{\mathrm{set}(l) : l \in V_{loc}\} \\ V_{smap(\sigma_1,\sigma_2)} & = & \{\mathrm{smap}(l) : l \in V_{loc}\} \\ V_{strong\_set(b)} & = & \{\mathrm{strongset}(l) : l \in V_{loc}\} \\ V_{strong\_smap(b,\sigma)} & = & \{\mathrm{strongsmap}(l) : l \in V_{loc}\}. \end{array} \tag{2.5}$$

In Definition 2.5 the symbols int, base, set, smap, strongset, strongsmap are labels for locations. Each implementation in V_τ corresponds to a value of type τ, which can be computed using the function *extract* (defined

[7]a state may be thought of as a map from variables to values

later in this section). Note that function *extract* is a many-one function, i.e. many different implementations may represent the same value, but each implementation cannot represent more than one value.

The precise definition of how Low SETL values are implemented requires the following additional semantic objects.

1. The environment ρ is a partial map from variables to a pair comprising of a location l, and a type τ (denoted by $l : \tau$), i.e.
$$\rho : \mathit{Vars} \mapsto V_{loc} \times \mathit{Type}.$$
2. The store γ is a partial map from locations (of type τ) to a value in set V_τ, or the value *undefined*, i.e.
$$\gamma : V_{loc} \mapsto V_\tau \cup \{\mathit{undefined}\}.$$
3. The maps *first*, *last*, *prev*, *next*, *succ*, *baseval*, and *image* are all partial maps from locations to locations, i.e.
$$\mathit{first}, \mathit{last}, \mathit{prev}, \mathit{next}, \mathit{succ}, \mathit{baseval}, \mathit{image} : V_{loc} \mapsto V_{loc}.$$
4. Finally, the map *symb* is a partial map from locations to identifiers, i.e.
$$\mathit{symb} : V_{loc} \mapsto \text{Identifier}.$$

Value $\gamma(l_{om})$ is defined to be the value *undefined*. Our goal is to define an operational semantics that closely corresponds to the informal descriptions of data structures given in Section 2.2. The maps *first*, *last*, *prev*, and *next* correspond to the first, last, prev and next pointers in a doubly linked list. For example, consider a value v of type $\mathit{set}(\mathit{int})$ that is implemented as a labelled location set(l). Then the locations $\mathit{first}(l)$ and $\mathit{last}(l)$ correspond to the first and the last integers in the doubly linked list implementation of v. The other integers in the linked list can be accessed by either starting from location $\mathit{first}(l)$ and using map *next* (in a way similar to using the next pointer to traverse a linked list), or by starting at location $\mathit{last}(l)$ and using map *prev*.

Figure 2.7 shows the implementation of a set S of type $\mathit{set}(\mathit{set}(\mathit{int}))$ with the value $\{\{1\}, \{1, 2\}\}$, a map M of type $\mathit{smap}(\mathit{int}, \mathit{int})$ with the value $\{[1, 2], [2, 3]\}$, and a variable x of base type b (satisfying the subtype constraint $b < \mathit{int}$) corresponding to the integer value 1. In Figure 2.7 an arrow labelled ρ from S to $l_1 : \mathit{set}(\mathit{set}(\mathit{int}))$ should be read as $\rho(S) = l_1 : \mathit{set}(\mathit{set}(\mathit{int}))$. Similarly, an arrow labelled γ from l_1 to set(l_2) should be read as $\gamma(l_1) = \text{set}(l_2)$. The absence of an arrow labelled *prev* coming out of location l_3 should be taken to imply that $\mathit{prev}(l_3) = \perp$. The environment ρ maps variable S to location l_1 which corresponds to the value $\{\{1\}, \{1, 2\}\}$. Locations l_3 and l_4 correspond to the values $\{1\}$ and $\{1, 2\}$ respectively, and Locations l_7, l_8, and l_9 correspond to the values 1, 1, and 2 respectively. Also, as seen in Figure 2.7, given a variable x of base type b satisfying the subtype constraint $b < \sigma$, the corresponding value of type σ can be obtained through the map *baseval*. The maps *succ*, and *symb* are used in the implementation of strongly based sets and maps in a manner similar to what is shown in Figure 2.6(b). In Figure 2.7 we see that x is an element of a strongly based set T of type $\mathit{strong\_set}(b)$. Moreover, $\mathit{succ}(l_{19}) = \perp$ (since there is no arrow labelled *succ* coming out of location l_{19}) implies that T is the only strongly based set containing element x, and that no strongly based map contains element x in its domain.

Before we give a precise definition of how a value v of type τ (i.e. $v \in T[\tau]$) may be extracted from an implementation $v_i \in V_\tau$, we define a few auxiliary functions that are used in the definition of function *extract* (Definition 2.3.1), and in the definition of the dynamic operational semantics of Low SETL.

- The function *ComputeSet* takes a location l as an input and evaluates to a (possibly empty) set of locations.
$$\mathit{ComputeSet}(l) = \begin{cases} \{\} \ \text{ if } \mathit{first}(l) = \perp \\ \mathit{lfp}_w.(\{\mathit{first}(l)\} \cup \{\mathit{next}(l') : l' \in w | \mathit{next}(l') \neq \perp\}) \text{ otherwise} \end{cases} \tag{2.6}$$

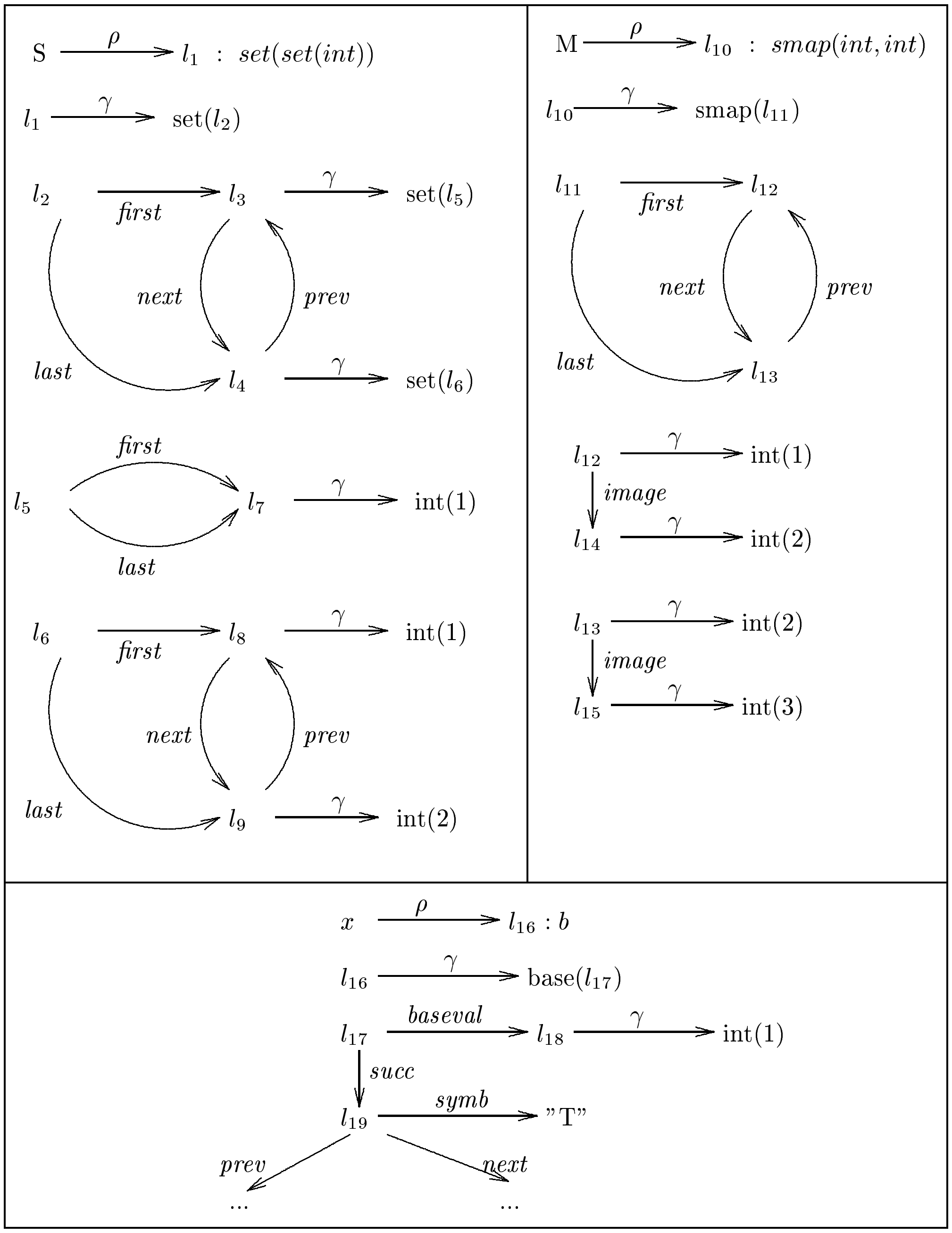


Figure 2.7: Implementation of S : *set*(*set*(*int*)), M : *smap*(*int*, *int*), and $x : b$ where $b < int$

where $lfp_w.f(w)$ denotes the least fixed point of function $f(w)$ with respect to parameter w, or in other words, the least value of w satisfying $f(w) = w$.

An informal explanation of Equation 2.6 is as follows. Let S be a set-valued variable such that $\rho(S) = l_S : set(\sigma)$, and $\gamma(l_S) = \text{set}(l)$. If $first(l) = \perp$, then set S is empty, and in this case function *ComputeSet* returns the empty set. Otherwise $first(l)$ is the location of an element of set S, and in this case function *ComputeSet* returns the smallest set of locations w such that $first(l) \in w$, and for all $l' \in w$, if $next(l') \neq \perp$, then $next(l') \in w$. It is not difficult to see that set w evaluates to the set containing the locations of all elements of set S. Similarly, if M is a map-valued variable such that $\rho(M) = l_M : \mathit{smap}(\sigma, \sigma')$, and $\gamma(l_M) = \text{smap}(l)$, then *ComputeSet*$(l)$ evaluates to the set containing locations of all the elements in the domain of map M. For example, in Figure 2.7, *ComputeSet*(l_2) evaluates to $\{l_3, l_4\}$, and *ComputeSet*(l_5) evaluates to $\{l_7\}$.

- The function *ComputeNodes* takes a location corresponding to an element x of some base type b, and evaluates to a set of locations such that each location corresponds either to a strongly based set that contains element x, or to a strongly based map whose domain contains element x.

$$\begin{array}{l} \mathit{ComputeNodes}(l) = \\ \quad \text{let } \gamma(l) = \text{base}(l') \text{ in} \\ \quad \left\{ \begin{array}{l} \{\} \text{ if } \mathit{succ}(l') = \bot \\ \mathit{lfp}_w.(\{\mathit{succ}(l')\} \cup \{\mathit{succ}(t) : t \in w | \mathit{succ}(t) \neq \bot\}) \text{ otherwise} \end{array} \right. \end{array} \tag{2.7}$$

Recall from Figure 2.6(b), that an element x of a base type b is implemented as a record containing a *key* field, and a field *succ* containing a pointer to a singly-linked list of nodes (linked together using *succ*), where each node corresponds either to a strongly based set containing x, or a strongly based map whose domain contains x. The function *ComputeNodes* evaluates to the set of locations corresponding to the nodes in this singly-linked list. For example, in Figure 2.7, *ComputeNodes*(l_{16}) evaluates to $\{l_{19}\}$.

- The function *StrongNext* takes a location corresponding to an element x of some base type b, and an identifier corresponding to either a set S of type *strong_set*(b) or a map M of type *strong_smap*(b, σ_1), and evaluates to the location of the next element in the doubly-linked list implementation of set S, or the domain of map M.

$$\begin{array}{rcl} \mathit{StrongNext}(l, v) & = & \text{let } s' = \mathit{ComputeNodes}(l) \text{ in} \\ & & \quad \text{let } t = \ni (\{t' \in s' | \mathit{symb}(t') = v\}) \text{ in} \\ & & \quad\quad \text{if } t = \bot \text{ then } \bot \text{ else } \mathit{next}(t) \end{array} \tag{2.8}$$

For example, in Figure 2.7, *StrongNext*$(l_{16},$"T"$)$ evaluates to the same location as location $\mathit{next}(l_{19})$, i.e. the location being pointed to by the arrow labelled *next* coming out of location l_{19}.

- The definition of function *StrongPrev* is similar except for the use *prev* instead of *next*. In Figure 2.7 *StrongPrev*$(l_{16},$"T"$)$ evaluates to the same location as location $\mathit{prev}(l_{19})$.

- The function *StrongImage* takes a location l corresponding to an element x of base type b, and an identifier v corresponding to a strongly based map M of type *strong_smap*(b, σ'), and evaluates to the location corresponding to the image of element x under map M.

$$\begin{array}{rcl} \mathit{StrongImage}(l, v) & = & \text{let } s' = \mathit{ComputeNodes}(l) \text{ in} \\ & & \quad \text{let } t = \ni (\{t' \in s' | \mathit{symb}(t') = v\}) \text{ in} \\ & & \quad\quad \text{if } t = \bot \text{ then } l_{om} \text{ else } \mathit{image}(t) \end{array} \tag{2.9}$$

- The function *IsElementOf* takes a location corresponding to an element x of some base type b, and an identifier v corresponding to a strongly based set S or map M, and tests for the membership of element x in set S, or in the domain of map M.

$$\begin{array}{rcl} \mathit{IsElementOf}(l, v) & = & \text{let } s' = \mathit{ComputeNodes}(l) \text{ in} \\ & & \quad \exists t' \in s' |\ \mathit{symb}(t') = v \end{array} \tag{2.10}$$

For example, in Figure 2.7, *IsElementOf*$(l_{16},$"T"$)$ evaluates to *true*.

- Let S be strongly based set with $\rho(S) = l_S : \mathit{strong\_set}(b)$ and $\gamma(l_S) = \text{strongset}(l)$, or M be a strongly based single-valued map with $\rho(M) = l_M : \mathit{strong\_smap}(b, \sigma')$ and $\gamma(l_M) = \text{strongsmap}(l)$. Function

ComputeStrongSet takes location l and evaluates to the set of locations corresponding to the elements of set S, or the domain of map M.

$$\begin{array}{l} \mathit{ComputeStrongSet}(l) = \\ \quad \left\{ \begin{array}{l} \{\} \text{ if } \mathit{first}(l) = \bot \\ \text{let } v = \mathit{symb}(l) \text{ in } \mathit{lfp}_w.(\{\mathit{first}(l)\} \cup \\ \quad \{\mathit{StrongNext}(l', v) : l' \in w \mid \mathit{StrongNext}(l', v) \neq \bot\}) \text{ otherwise} \end{array} \right. \end{array} \tag{2.11}$$

We are now ready to define the function *extract*.

Definition 2.3.1 The function *extract* takes an element of V_τ and evaluates to the corresponding value in $T[\tau]$, i.e. a value of type τ.

$$\begin{array}{rcl} \mathit{extract}(\text{int}(x)) & = & x \\ \mathit{extract}(\text{base}(l)) & = & \mathit{extract}(\gamma(\mathit{baseval}(l))) \\ \mathit{extract}(\text{set}(l)) & = & \{\mathit{extract}(\gamma(l')) : l' \in \mathit{ComputeSet}(l)\} \\ \mathit{extract}(\text{smap}(l)) & = & \{[\mathit{extract}(\gamma(l')), \mathit{extract}(\gamma(\mathit{image}(l')))] : \\ & & \quad l' \in \mathit{ComputeSet}(l)\} \\ \mathit{extract}(\text{strongset}(l)) & = & \{\mathit{extract}(\gamma(l')) : l' \in \mathit{ComputeStrongSet}(l)\} \\ \mathit{extract}(\text{strongsmap}(l)) & = & \{[\mathit{extract}(\gamma(l')), \mathit{extract}(\gamma(\mathit{image}(l')))] : \\ & & \quad l' \in \mathit{ComputeStrongSet}(l)\} \end{array}$$

Using Definition 2.3.1 it is easily verified that in Figure 2.7,

- $\mathit{extract}(\gamma(l_1))$ evaluates to $\{\{1\}, \{1, 2\}\}$,
- $\mathit{extract}(\gamma(l_{10}))$ evaluates to $\{[1, 2], [2, 3]\}$, and
- $\mathit{extract}(\gamma(l_{16})$ evaluates to 1.

We define the state of a Low SETL program (denoted by Σ) to be

$$\Sigma = [\rho, \gamma, \mathit{first}, \mathit{last}, \mathit{prev}, \mathit{next}, \mathit{succ}, \mathit{symb}, \mathit{image}, \mathit{baseval}].$$

The state Σ of a program can be used to associate values with all variables of a Low SETL program in the following way. Given a variable v, let $\rho(v)$ be the location-type pair $l : \tau$. Then, $\mathit{extract}(\gamma(l))$ evaluates to the value corresponding to variable v. Note that function *extract* makes use of the other maps in Σ, i.e. *first*, *last*, *prev*, *next*, *succ*, *symb*, *image*, and *baseval*.

In Sections 2.3.1,2.3.2, and 2.3.3 we specify the operational semantics for Low SETL expressions, boolean expressions, and commands. Given a state Σ, a Low SETL expression e evaluates to a pair comprising of a location l and a type τ such that the value $\mathit{extract}(\gamma(l))$ is a value of type τ. Similarly, given a state Σ, a boolean expression *be* evaluates to a boolean value *true* or *false*. The evaluation of expressions and boolean expressions is side-effect free, i.e. does not modify the state in which they are evaluated. On the other hand, the evaluation of commands leads to a transition from one state to another. For this reason, we find it convenient to express the semantics of expressions and boolean expressions as a big-step operational semantics (similar to a proof derivation), and the semantics for the evaluation of commands as a small-step operational semantics (i.e. as transitions from one configuration to another).

2.3.1 Operational Semantics for Expressions

The inference rules for the dynamic operational semantics of expressions appear in Figure 2.8. The notation $\boxed{\langle \Sigma, C, e \rangle \xrightarrow{exp} l : \tau}$ can be read as saying that *expression e evaluates to location l corresponding to a value of type τ in state Σ under the set of subtype constraints C*. The one exception to this is the rule $\langle \Sigma, C, e \rangle \xrightarrow{exp}$ *abort*, which indicates that the evaluation of expression e in state Σ causes an abort error. Informally, we can

$$\boxed{\langle \Sigma, C, e\rangle \xrightarrow{exp} l : \tau}$$

$$\langle \Sigma, C, v\rangle \xrightarrow{exp} \rho(v) \tag{2.12}$$

$$\langle \Sigma, C, \text{om}\rangle \xrightarrow{exp} l_{om} : \sigma \qquad \text{for all types } \sigma \tag{2.13}$$

$$\frac{\langle \Sigma, C, v\rangle \xrightarrow{exp} l : \mathit{set}(\sigma),\ \gamma(l) = \text{set}(s),\ \mathit{first}(s) = \bot}{\langle \Sigma, C, \ni v\rangle \xrightarrow{exp} l_{om} : \sigma} \tag{2.14}$$

$$\frac{\langle \Sigma, C, v\rangle \xrightarrow{exp} l : \mathit{set}(\sigma),\ \gamma(l) = \text{set}(s),\ \mathit{first}(s) \neq \bot}{\langle \Sigma, C, \ni v\rangle \xrightarrow{exp} \mathit{first}(s) : \sigma} \tag{2.15}$$

$$\frac{\langle \Sigma, C, v\rangle \xrightarrow{exp} l : \mathit{strong\_set}(b),\ \gamma(l) = \text{strongset}(s),\ \mathit{first}(s) = \bot}{\langle \Sigma, C, \ni v\rangle \xrightarrow{exp} l_{om} : b} \tag{2.16}$$

$$\frac{\langle \Sigma, C, v\rangle \xrightarrow{exp} l : \mathit{strong\_set}(b),\ \gamma(l) = \text{strongset}(s),\ \mathit{first}(s) \neq \bot}{\langle \Sigma, C, \ni v\rangle \xrightarrow{exp} \mathit{first}(s) : b} \tag{2.17}$$

$$\frac{\langle \Sigma, C, v\rangle \xrightarrow{exp} l : \mathit{set}(\sigma),\ \gamma(l) = \gamma(l_{om})}{\langle \Sigma, C, \ni v\rangle \xrightarrow{exp} \mathit{abort}} \tag{2.18}$$

$$\frac{\begin{array}{c}\langle \Sigma, C, v_2\rangle \xrightarrow{exp} l : \sigma,\ \gamma(l) \neq \gamma(l_{om}),\\ \langle \Sigma, C, v_1\rangle \xrightarrow{exp} l_1 : \mathit{smap}(\sigma, \sigma_1),\ \gamma(l_1) = \text{smap}(s)\end{array}}{\langle \Sigma, C, v_1(v_2)\rangle \xrightarrow{exp} \mathit{image}(l) : \sigma_1} \qquad \text{where } l \in \mathit{ComputeSet}(s) \tag{2.19}$$

$$\frac{\begin{array}{c}\langle \Sigma, C, v_2\rangle \xrightarrow{exp} l : b,\ \gamma(l) = \text{base}(l')\\ \langle \Sigma, C, v_1\rangle \xrightarrow{exp} l_1 : \mathit{strong\_smap}(b, \sigma), \gamma(l_1) = \text{strongsmap}(s)\\ t = \mathit{StrongImage}(l', \mathit{symb}(s))\end{array}}{\langle \Sigma, C, v_1(v_2)\rangle \xrightarrow{exp} t : \sigma} \tag{2.20}$$

Figure 2.8: Operational Semantics for Expressions (continued on next page)

think of the abort error being caused by expressions such as $\ni v$ where v evalautes to the value *undefined*, or by $f(v)$ where either f or v evaluate to the value *undefined*.

In Figure 2.8, Rule 2.12 can be read as saying that a variable v evaluates to $\rho(v)$[8]. Rule 2.13 should not be read as one rule but a rule schema saying that Low SETL expression om evaluates to the pair $l_{om} : \sigma$ for all types σ. Rules 2.14-2.18 define how the expression $\ni v$ is evaluated. Rules 2.14 and 2.16 say that if v is an empty set of type $\mathit{set}(\sigma)$ or an empty strongly based set of type $\mathit{strong\_set}(b)$, then $\ni v$ evaluates to

[8]Recall that the environment ρ maps variables to a location-type pair.

$$\frac{\begin{array}{c}\langle\Sigma, C, v_1\rangle \xrightarrow{exp} l_1 : \mathit{strong\_smap}(b,\sigma), \langle\Sigma, C, v_2\rangle \xrightarrow{exp} l_2 : b \\ \gamma(l_1) = \gamma(l_{om}) \text{ or } \gamma(l_2) = \gamma(l_{om})\end{array}}{\langle\Sigma, C, v_1(v_2)\rangle \xrightarrow{exp} \mathit{abort}} \tag{2.21}$$

$$\frac{\langle\Sigma, C, e\rangle \xrightarrow{exp} l : b,\ \gamma(l) = \gamma(l_{om}),\ b < \sigma \in C}{\langle\Sigma, C, e\rangle \xrightarrow{exp} l : \sigma} \tag{2.22}$$

$$\frac{\langle\Sigma, C, e\rangle \xrightarrow{exp} l : b,\ b < \sigma \in C,\ \gamma(l) = \text{base}(l')}{\langle\Sigma, C, e\rangle \xrightarrow{exp} \mathit{baseval}(l') : \sigma} \tag{2.23}$$

Figure 2.8: Operational Semantics for Expressions (continued from previous page)

$l_{om} : \sigma$, or $l_{om} : b$ respectively. Rules 2.15 and 2.17 correspond to the cases when the sets are not empty, in which case $\ni v$ evaluates to the first element of the set. According to Rule 2.18, if v evaluates to the value $\gamma(l_{om})$, i.e. *undefined*, then $\ni v$ causes an abort error. Rules 2.19-2.21 define how the expression $v_1(v_2)$ is evaluated. Rule 2.21 says that there is an abort error if either v_1 or v_2 evaluates to the value *undefined*.

According to Rule 2.19, if variable v_2 evaluates to a location l and type σ such that $\gamma(l) \neq \gamma(l_{om})$, and variable v_1 evaluates to location l_1 and type $\mathit{smap}(\sigma, \sigma_1)$ such that $\gamma(l_1) = \text{smap}(s)$, then the expression $v_1(v_2)$ evaluates to $\mathit{image}(l)$ only if the side condition $l \in \mathit{ComputeSet}(s)$ is satisfied. Note that if the side condition $l \in \mathit{ComputeSet}(s)$ is not satisfied, then there is no derivation for the evaluation of expression $v_1(v_2)$[9]. In the case of Rule 2.20, v_1 is a strongly based map, and v_2 is an element of a base type. The image of v_2 under v_1 can thus be computed using the function *StrongImage*. Note that in this case we don't have the side condition of Rule 2.19. Finally, Rules 2.22 and 2.23 define how an implementation of a value of type σ can be extracted from an implementation of a value of base type b, if base type b satisfies the subtype constraint $b < \sigma$.

2.3.2 Operational Semantics for Boolean Expressions

The dynamic operational semantics for the evaluation of boolean expressions is given in Figure 2.9. The notation $\boxed{\langle\Sigma, C, be\rangle \xrightarrow{bool} \mathit{boolval}}$ can be read as saying that *boolean expression be evaluates to the boolean value boolval in state* Σ *under the set of subtype constraints* C.

In Figure 2.9, Rules 2.24 and 2.25 define the evaluation of the boolean expression $v_1 \in v_2$. In the case of Rule 2.24, if v_1 is an element of some base type b, and v_2 is a strongly based set of type $\mathit{strong\_set}(b)$, then the boolean expression $v_1 \in v_2$ can be evaluated using the auxiliary function *IsElementOf*. According to Rule 2.25, if v_1 evaluates to the value *undefined*, then there is an abort error. Rules 2.26 and 2.27 define how the boolean expression $v_1 == v_2$ is evaluated. The equality test $v_1 == v_2$ is allowed only if both v_1 and v_2 are integers, or both are elements of some base type b. In both cases, if l_1 and l_2 are the locations corresponding to variables v_1 and v_2, then the expression $v_1 == v_2$ evaluates to true iff $\gamma(l_1)$ is equal to $\gamma(l_2)$. Rules 2.28-2.33 define how the boolean expressions $\mathit{IsEmptySet}(v)$ and $\mathit{IsEmptyMap}(v)$ are computed. In both cases, if v evaluates to *undefined*, an abort error is caused (Rules 2.30 and 2.33).

[9]If expression $v_1(v_2)$ appears inside a for loop of the form "for $v_2 \in \mathit{domain}(v_1)$ loop ... endloop", then the side condition for Rule 2.19 is guaranteed to be satisfied for all states Σ in which the expression $v_1(v_2)$ will have to evaluated. Such an evaluation of expression $v_1(v_2)$ corresponds to the special case of self-access as discussed in Section 2.2.4.

$$\boxed{\langle \Sigma, C, be \rangle \xrightarrow{bool} \mathit{boolval}}$$

$$\frac{\begin{array}{c}\rho(v_1) = l_1 : b,\ \rho(v_2) = l_2 : \mathit{strong\_set}(b) \\ \gamma(l_2) = \mathrm{strongset}(t),\ \gamma(l_1) = \mathrm{base}(s),\ \mathit{val} = \mathit{IsElementOf}(s, \mathit{symb}(t))\end{array}}{\langle \Sigma, C, v_1 \in v_2 \rangle \xrightarrow{bool} \mathit{val}} \tag{2.24}$$

$$\frac{\rho(v_1) = l_1 : b,\ \gamma(l_1) = \gamma(l_{om})}{\langle \Sigma, C, v_1 \in v_2 \rangle \xrightarrow{bool} \mathit{abort}} \tag{2.25}$$

$$\frac{\rho(v_1) = l_1 : \mathit{int},\ \rho(v_2) = l_2 : \mathit{int},\ \mathit{val} = (\gamma(l_1) == \gamma(l_2))}{\langle \Sigma, C, v_1 == v_2 \rangle \xrightarrow{bool} \mathit{val}} \tag{2.26}$$

$$\frac{\rho(v_1) = l_1 : b,\ \rho(v_2) = l_2 : b,\ \mathit{val} = (\gamma(l_1) == \gamma(l_2))}{\langle \Sigma, C, v_1 == v_2 \rangle \xrightarrow{bool} \mathit{val}} \tag{2.27}$$

$$\frac{\rho(v) = l : \mathit{set}(\sigma),\ \gamma(l) = \mathrm{set}(s),\ \mathit{val} = (\mathit{first}(s) == \perp)}{\langle \Sigma, C, \mathit{IsEmptySet}(v) \rangle \xrightarrow{bool} \mathit{val}} \tag{2.28}$$

$$\frac{\rho(v) = l : \mathit{strong\_set}(b),\ \gamma(l) = \mathrm{strongset}(s),\ \mathit{val} = (\mathit{first}(s) == \perp)}{\langle \Sigma, C, \mathit{IsEmptySet}(v) \rangle \xrightarrow{bool} \mathit{val}} \tag{2.29}$$

$$\frac{\rho(v) = l : \mathit{set}(\sigma), \gamma(l) = \gamma(l_{om})}{\langle \Sigma, C, \mathit{IsEmptySet}(v) \rangle \xrightarrow{bool} \mathit{abort}} \tag{2.30}$$

$$\frac{\rho(v) = l : \mathit{smap}(\sigma, \sigma'),\ \gamma(l) = \mathrm{smap}(s),\ \mathit{val} = (\mathit{first}(s) == \perp)}{\langle \Sigma, C, \mathit{IsEmptyMap}(v) \rangle \xrightarrow{bool} \mathit{val}} \tag{2.31}$$

$$\frac{\rho(v) = l : \mathit{strong\_smap}(b, \sigma'),\ \gamma(l) = \mathrm{strongsmap}(s),\ \mathit{val} = (\mathit{first}(s) == \perp)}{\langle \Sigma, C, \mathit{IsEmptyMap}(v) \rangle \xrightarrow{bool} \mathit{val}} \tag{2.32}$$

$$\frac{\rho(v) = l : \mathit{smap}(\sigma, \sigma'), \gamma(l) = \gamma(l_{om})}{\langle \Sigma, C, \mathit{IsEmptyMap}(v) \rangle \xrightarrow{bool} \mathit{abort}} \tag{2.33}$$

Figure 2.9: Operational Semantics for Boolean Expressions

2.3.3 Operational Semantics for Commands

In this section we specify a small-step operational semantics for the evaluation of commands. The small-step semantics is defined as a binary transition relation relating pairs of configurations. A configuration is a triple $\langle \Sigma, C, P \rangle$, where state Σ is called the data part of the configuration, and command sequence P is called the control part of the configuration, and C is the set of subtype constraints associated with command sequence

P. When the control part of the configuration is the empty program, we simply write the configuration as Σ. More precisely, a configuration is either

- a triple $\langle \Sigma, C, P \rangle$ consisting of a state Σ, a command sequence P, and a set of subtype constraints C, or
- a state Σ.

A configuration of the form Σ is called a terminating configuration since it indicates the termination of the execution of the program with Σ as the final state.

The rules for the evaluation of a command sequence are given in terms of a binary relation $\longrightarrow$ on configurations. A rule of the form $\boxed{\langle \Sigma, C, P \rangle \longrightarrow \langle \Sigma', C, P' \rangle}$ can be read as saying that a command sequence P in state Σ is transformed into a command sequence P' in state Σ' in one step of the computation. Similarly, a rule of the form $\boxed{\langle \Sigma, C, P \rangle \longrightarrow \Sigma'}$ can be read as saying that evaluation of a command sequence P in state Σ terminates in state Σ' in one step of the computation. In the case of operational semantics for command sequences, we will use *abort* to denote a special terminating state corresponding to the termination of the program as a result of an abort error.

The rules for the operational semantics of command sequences are given below. The relation $\longrightarrow$ is defined as the smallest relation closed under these rules. Note that the set of subtype constraints C does not change in any of the transition rules. This is because we work with a fixed set of subtype constraints for a given program. In each of the following rules, we assume that

$$\Sigma' = [\rho', \gamma', \mathit{first}', \mathit{last}', \mathit{prev}', \mathit{next}', \mathit{succ}', \mathit{symb}', \mathit{image}', \mathit{baseval}'].$$

Moreover, we assume that whenever the value of any of the primed maps ρ', γ', first', etc. is left unspecified, then its value remains unchanged.

Command $v := e$

$$\frac{\langle \Sigma, C, e \rangle \xrightarrow{exp} \mathit{abort}}{\langle \Sigma, C, v := e \rangle \longrightarrow \mathit{abort}} \tag{2.34}$$

Rule 2.34 says that evaluation of the command $v := e$ aborts if the evaluation of expression e aborts.

$$\begin{array}{c} \dfrac{\langle \Sigma, C, e \rangle \xrightarrow{exp} l : \tau,\ \ \rho(v) = l' : \tau}{\langle \Sigma, C, v := e \rangle \longrightarrow \Sigma'} \\ \text{where } \gamma' = \gamma[l' \mapsto \gamma(l)] \end{array} \tag{2.35}$$

Rule 2.35 says that if expression e evaluates to a location l of type τ, and v is a variable of type τ, then the evaluation of assignment $v := e$ in state Σ leads to the new state Σ', which is identical to Σ except for γ' whose new value is $\gamma[l' \mapsto \gamma(l)]$.

Command $v_1(v_2) := e$

$$\frac{\langle \Sigma, C, e \rangle \xrightarrow{exp} \mathit{abort}}{\langle \Sigma, C, v_1(v_2) := e \rangle \longrightarrow \mathit{abort}} \tag{2.36}$$

$$\frac{\rho(v_2) = l : \sigma,\ \ \gamma(l) = \gamma(l_{om})}{\langle \Sigma, C, v_1(v_2) := e \rangle \longrightarrow \mathit{abort}} \tag{2.37}$$

$$\frac{\rho(v_1) = l : \mathit{smap}(\sigma_1, \sigma_2),\ \gamma(l) = \gamma(l_{om})}{\langle \Sigma, C, v_1(v_2) := e\rangle \longrightarrow \mathit{abort}} \tag{2.38}$$

$$\frac{\rho(v_1) = l : \mathit{strong\_smap}(b, \sigma_2),\ \gamma(l) = \gamma(l_{om})}{\langle \Sigma, C, v_1(v_2) := e\rangle \longrightarrow \mathit{abort}} \tag{2.39}$$

Rules 2.36-2.39 say that the evaluation of the assignment $v_1(v_2) := e$ aborts if either the evaluation of expression e aborts, or if either v_1 or v_2 evaluates to the value *undefined*.

$$\frac{\begin{array}{c}\langle \Sigma, C, e\rangle \overset{exp}{\longrightarrow} l : \sigma_2,\ \gamma(l) = \gamma(l_{om})\\ \rho(v_2) = l_2 : \sigma_1,\ \gamma(l_2) \neq \gamma(l_{om})\\ \rho(v_1) = l_1 : \mathit{smap}(\sigma_1, \sigma_2),\ \gamma(l_1) = \mathrm{smap}(s)\\ s' = \mathit{ComputeSet}(s)\end{array}}{\langle \Sigma, C, v_1(v_2) := e\rangle \longrightarrow \Sigma} \quad \begin{array}{l}\text{where}\\ \forall t' \in s' | (\mathit{extract}(\gamma(t')) \neq \mathit{extract}(\gamma(l_2)))\end{array} \tag{2.40}$$

Rule 2.40 says that if expression e evaluates to the value *undefined*, and v_2 is not already in the domain of map v_1 (side-condition), then the evaluation of the command $v_1(v_2) := e$ causes no change in the state Σ.

$$\frac{\begin{array}{c}\langle \Sigma, C, e\rangle \overset{exp}{\longrightarrow} l : \sigma_2,\ \gamma(l) \neq \gamma(l_{om})\\ \rho(v_2) = l_2 : \sigma_1,\ \gamma(l_2) \neq \gamma(l_{om})\\ \rho(v_1) = l_1 : \mathit{smap}(\sigma_1, \sigma_2),\ \gamma(l_1) = \mathrm{smap}(s)\\ s' = \mathit{ComputeSet}(s),\ \mathit{first}(s) \neq \perp\\ l_3 = \mathit{newLocation}(),\ l_4 = \mathit{newLocation}()\end{array}}{\begin{array}{c}\langle \Sigma, C, v_1(v_2) := e\rangle \longrightarrow \Sigma'\\ \text{where } \gamma' = \gamma[l_3 \mapsto \gamma(l_2), l_4 \mapsto \gamma(l)],\\ \mathit{first}' = \mathit{first}[s \mapsto l_3],\\ \mathit{prev}' = \mathit{prev}[\mathit{first}(s) \mapsto l_3],\\ \mathit{next}' = \mathit{next}[l_3 \mapsto \mathit{first}(s)],\\ \mathit{image}' = \mathit{image}[l_3 \mapsto l_4]\end{array}} \quad \begin{array}{l}\text{where}\\ \forall t' \in s' | (\mathit{extract}(\gamma(t')) \neq \mathit{extract}(\gamma(l_2)))\end{array} \tag{2.41}$$

Rule 2.41 takes the case when expression e evaluates to a non-omega value, and v_2 is not already in the domain of map v_1. In this case the pair $[v_2, e]$ gets added to the map v_1. In Rule 2.41 we consider the case when map v_1 is originally not empty (i.e. $\mathit{first}(s) \neq \perp$). Another rule that takes care of the case when map v_1 is initially empty is similar, and therefore omitted.

$$\frac{\begin{array}{c}\langle \Sigma, C, e\rangle \overset{exp}{\longrightarrow} l : \sigma,\ \gamma(l) \neq \gamma(l_{om}),\ \rho(v_1) = l_1 : \mathit{strong\_smap}(b, \sigma),\ \gamma(l_1) = \mathrm{strongsmap}(m)\\ \rho(v_2) = l_2 : b,\ \gamma(l_2) = \mathrm{base}(s),\ s' = \mathit{ComputeNodes}(l_2)\\ t =\ni \{t' \in s' | \mathit{symb}(t') = \mathit{symb}(m)\},\ t \neq \perp,\ l_3 = \mathit{newLocation}()\end{array}}{\begin{array}{c}\langle \Sigma, C, v_1(v_2) := e\rangle \longrightarrow \Sigma'\\ \text{where } \gamma' = \gamma[l_3 \mapsto \gamma(l)]\\ \mathit{image}' = \mathit{image}[t \mapsto l_3]\end{array}} \tag{2.42}$$

Rule 2.42 defines the evaluation of command $v_1(v_2) := e$ when v_1 is a strongly based map, $v_2 \in \mathit{domain}(v_1)$, and expression e evaluates to a non-omega value. In this case, v_2 is implemented as a base record having a node corresponding to map v_1 in its linked list. The effect of $v_1(v_2) := e$ is to simply update the *image* pointer of this node to the new value.

$$\frac{\begin{array}{c}\langle \Sigma, C, e\rangle \xrightarrow{exp} l:\sigma,\ \gamma(l)=\gamma(l_{om}),\ \rho(v_1)=l_1 : \mathit{strong\_smap}(b,\sigma),\ \gamma(l_1)=\mathrm{strongsmap}(m)\\ \rho(v_2)=l_2:b,\ \gamma(l_2)=\mathrm{base}(s),\ s'=\mathit{ComputeNodes}(l_2)\\ t=\ni\{t'\in s' \mid \mathit{symb}(t')=\mathit{symb}(m)\},\ t\neq\bot,\ p=\ni\{t'\in s'\cup\{s\}\mid \mathit{succ}(t')=t\}\\ \gamma(\mathit{first}(m))\neq\gamma(l_2),\ \gamma(\mathit{last}(m)\neq\gamma(l_2)\\ s_p=\mathit{ComputeNodes}(\mathit{prev}(t)),\ t_p=\ni\{t'_p\in s_p\mid \mathit{symb}(t'_p)=\mathit{symb}(m)\}\\ s_n=\mathit{ComputeNodes}(\mathit{next}(t)),\ t_n=\ni\{t'_n\in s_n\mid \mathit{symb}(t'_n)=\mathit{symb}(m)\}\end{array}}{\begin{array}{c}\langle\Sigma,C,v_1(v_2):=e\rangle\longrightarrow\Sigma'\\ \text{where } \mathit{prev}'=\mathit{prev}[t_n\mapsto \mathit{prev}(t)]\\ \mathit{next}'=\mathit{next}[t_p\mapsto\mathit{next}(t)]\\ \mathit{succ}'=\mathit{succ}[p\mapsto\mathit{succ}(t)]\end{array}} \tag{2.43}$$

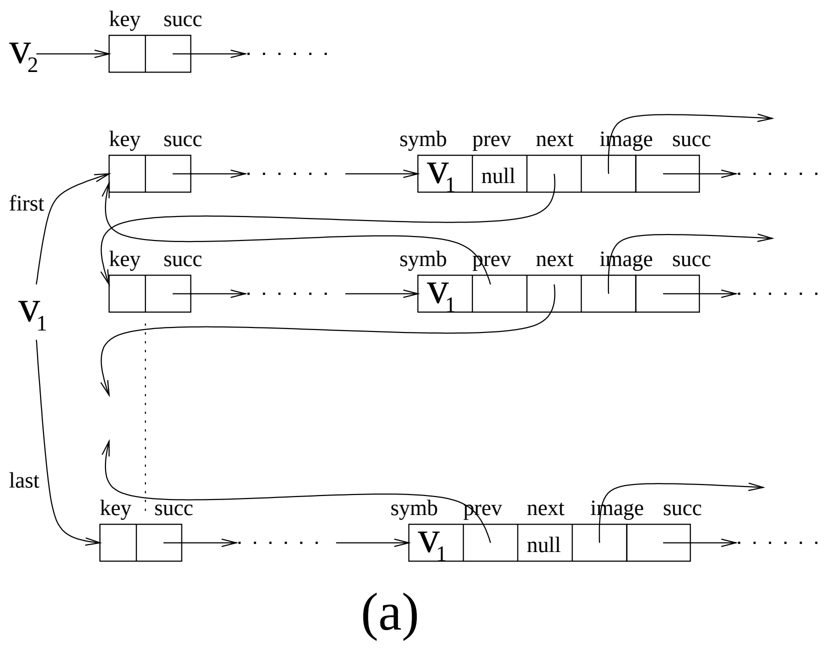
In Rule 2.43 we again have a strongly based map v_1, and $v_2 \in \mathit{domain}(v_1)$, but the difference is that e evaluates to the value *undefined*. The effect of $v_1(v_2) := e$ is therefore to remove v_2 from the domain of v_1. To do this, we need to do two things. First, we must remove the node corresponding to map v_1 from the linked list of the base record for v_2. Secondly, we must remove the base record for v_2 from the doubly linked list implementation of the domain of map v_1. In Rule 2.43 we consider the case when v_2 is neither the first, nor the last element of this linked list. The rules for the other three cases when v_2 is the first element of the linked list but not the last, or the last element of the linked list but not the first, or both the first and last element of the linked list are similar and therefore omitted.

$$\frac{\begin{array}{c}\langle \Sigma, C, e\rangle \xrightarrow{exp} l:\sigma,\ \gamma(l)=\gamma(l_{om})\\ \rho(v_1)=l_1 : \mathit{strong\_smap}(b,\sigma),\ \gamma(l_1)=\mathrm{strongsmap}(m)\\ \rho(v_2)=l_2:b,\ \gamma(l_2)=\mathrm{base}(s),\ \mathit{IsElementOf}(l_2,\mathit{symb}(m))=\mathit{false}\end{array}}{\langle\Sigma,C,v_1(v_2):=e\rangle\longrightarrow\Sigma} \tag{2.44}$$

In Rule 2.44 we have a strongly based map v_1 not having element v_2 in its domain, and the expression e evaluates to the value *undefined*. Thus, the command $v_1(v_2) := e$ causes no change in the state Σ.

$$\frac{\begin{array}{c}\langle \Sigma, C, e\rangle \xrightarrow{exp} l:\sigma,\ \gamma(l)\neq\gamma(l_{om})\\ \rho(v_1)=l_1 : \mathit{strong\_smap}(b,\sigma),\ \gamma(l_1)=\mathrm{strongsmap}(m),\ \mathit{first}(m)\neq\bot\\ \rho(v_2)=l_2:b,\ \gamma(l_2)=\mathrm{base}(s),\ \mathit{IsElementOf}(l_2,\mathit{symb}(m))=\mathit{false}\\ s_f=\mathit{ComputeNodes}(\mathit{first}(m)),\ t_f=\ni\{t'_f\in s_f\mid \mathit{symb}(t'_f)=\mathit{symb}(m)\}\\ l_3=\mathit{newLocation}(),\ l_4=\mathit{newLocation}(),\ l_5=\mathit{newLocation}()\end{array}}{\begin{array}{c}\langle\Sigma,C,v_1(v_2):=e\rangle\longrightarrow\Sigma'\\ \text{where } \gamma'=\gamma[l_3\mapsto\gamma(l_2),l_4\mapsto\gamma(l)]\\ \mathit{first}'=\mathit{first}[m\mapsto l_3]\\ \mathit{prev}'=\mathit{prev}[t_f\mapsto l_3]\\ \mathit{next}'=\mathit{next}[l_5\mapsto\mathit{first}(m)]\\ \mathit{image}'=\mathit{image}[l_5\mapsto l_4]\\ \mathit{succ}'=\mathit{succ}[s\mapsto l_5,l_5\mapsto\mathit{succ}(s)]\\ \mathit{symb}'=\mathit{symb}[l_5\mapsto\mathit{symb}(m)]\end{array}} \tag{2.45}$$

In Rule 2.45 we have a strongly-based map v_1 not having element v_2 in its domain. Furthermore, expression e evaluates to a non-omega value. Thus, the effect of $v_1(v_2) := e$, is to add the pair $[v_2, e]$ to map v_1. In Rule 2.45 we consider the case when map v_1 is not initially empty. Another rule for



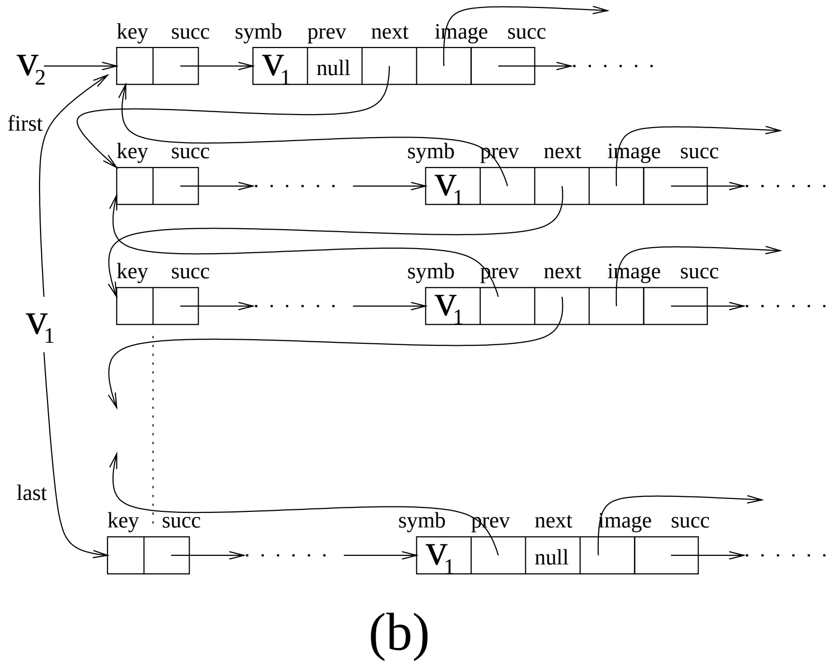


Figure 2.10: Pictorial representation of the data structures (a) before and (b) after the execution of the command $V_1(V_2) := e$ (corresponding to Rule 2.45)

the case when v_1 is initially empty is similar, and therefore omitted. This rule is best understood by looking at Figure 2.10 where we show a pictorial representation of the data structures before and after the execution of the command $v_1(v_2) := e$.

Command $v \ with := e$

$$\frac{\langle \Sigma, C, e\rangle \xrightarrow{exp} abort}{\langle \Sigma, C, v \ with := e\rangle \longrightarrow abort} \tag{2.46}$$

$$\frac{\langle \Sigma, C, e\rangle \xrightarrow{exp} l : \sigma,\ \gamma(l) = \gamma(l_{om})}{\langle \Sigma, C, v\ with := e\rangle \longrightarrow abort} \tag{2.47}$$

$$\frac{\rho(v) = l : set(\sigma),\ \gamma(l) = \gamma(l_{om})}{\langle \Sigma, C, v\ with := e\rangle \longrightarrow abort} \tag{2.48}$$

$$\frac{\rho(v) = l : strong\_set(b),\ \gamma(l) = \gamma(l_{om})}{\langle \Sigma, C, v\ with := e\rangle \longrightarrow abort} \tag{2.49}$$

Rules 2.46-2.49 say that the evaluation of the command $v\ with := e$ aborts if either the evaluation of expression e aborts, or either one of e or v evaluate to the value *undefined*.

$$\frac{\begin{array}{c}\rho(v) = l : set(\sigma),\ \gamma(l) = \mathrm{set}(s)\\ \langle \Sigma, C, e\rangle \xrightarrow{exp} l_1 : \sigma,\ \gamma(l_1) \neq \gamma(l_{om})\\ first(s) \neq \bot,\ s' = ComputeSet(s)\\ l_2 = newLocation()\end{array}}{\begin{array}{c}\langle \Sigma, C, v\ with := e\rangle \longrightarrow \Sigma'\\ \text{where } \gamma' = \gamma[l_2 \mapsto \gamma(l_1)]\\ first' = first[s \mapsto l_2]\\ prev' = prev[first(s) \mapsto l_2]\\ next' = next[l_2 \mapsto first(s)]\end{array}} \quad \begin{array}{l}\text{where}\\ \forall t \in s' | extract(\gamma(t)) \neq extract(\gamma(l_1))\end{array} \tag{2.50}$$

Rule 2.50 defines the evaluation of command $v\ with := e$ in the case when expression e evaluates to a location corresponding to a non-omega value, and the value corresponding to expression e is not already a member of set v(side-condition). In this case, the evaluation of the command $v\ with := e$ simply results in the addition of a new location having the same value as e to the linked list for set v. Rule 2.50 only takes care of the case when the set v is originally non-empty. Another similar rule takes care of the case when v is originally empty, and is omitted.

$$\frac{\begin{array}{c}\rho(v) = l : strong\_set(b),\ \gamma(l) = \mathrm{strongset}(s)\\ \langle \Sigma, C, e\rangle \xrightarrow{exp} l_1 : b,\ \gamma(l_1) = \mathrm{base}(t),\ IsElementOf(l_1, symb(s)) = true\end{array}}{\langle \Sigma, C, v\ with := e\rangle \longrightarrow \Sigma} \tag{2.51}$$

In Rule 2.51 v is a strongly based set, and the value corresponding to expression e is already a member of set v. Therefore, the evaluation of $v\ with := e$ causes no change to the state Σ.

$$\frac{\begin{array}{c}\rho(v) = l : strong\_set(b),\ \gamma(l) = \mathrm{strongset}(s)\\ \langle \Sigma, C, e\rangle \xrightarrow{exp} l_1 : b,\ \gamma(l_1) = \mathrm{base}(t),\ IsElementOf(l_1, symb(s)) = false\\ l_2 = newLocation(),\ l_3 = newLocation(),\ first(s) \neq \bot\\ s_f = ComputeNodes(first(s)),\ t_f = \ni \{t'_f \in s_f | symb(t'_f) = symb(s)\}\end{array}}{\begin{array}{c}\langle \Sigma, C, v\ with := e\rangle \longrightarrow \Sigma'\\ \text{where } \gamma' = \gamma[l_2 \mapsto \gamma(l_1)]\\ first' = first[s \mapsto l_2]\\ prev' = prev[t_f \mapsto l_2]\\ next' = next[l_3 \mapsto first(s)]\\ succ' = succ[t \mapsto l_3, l_3 \mapsto succ(t)]\\ symb' = symb[l_3 \mapsto symb(s)]\end{array}} \tag{2.52}$$

In Rule 2.52 v is a strongly-based set, and expression e evaluates to a value not already in v. Rule 2.52 takes care of the case when set v is originally non-empty. Another similar rule that takes care of the case when v is empty, is omitted.

Command v *less* $:= e$ The evaluation of the command v *less* $:= e$ aborts if either the evaluation of expression e aborts, or either one of v or e evaluates to the value *undefined*. The rules are similar to Rules 2.46-2.49, and are omitted.

$$\frac{\begin{array}{c}\rho(v) = l : \mathit{strong\_set}(b),\ \gamma(l) = \mathrm{strongset}(s) \\ \langle \Sigma, C, e\rangle \xrightarrow{exp} l_1 : b,\ \gamma(l_1) = \mathrm{base}(t), \mathit{IsElementOf}(l_1, \mathit{symb}(s)) = \mathit{false}\end{array}}{\langle \Sigma, C, v\ \mathit{less} := e\rangle \longrightarrow \Sigma} \tag{2.53}$$

Rule 2.53 takes care of the simple case when v is a strongly based set and expression e corresponds to a value that is not a member of set v.

$$\frac{\begin{array}{c}\rho(v) = l : \mathit{strong\_set}(b),\ \gamma(l) = \mathrm{strongset}(s) \\ \langle \Sigma, C, e\rangle \xrightarrow{exp} l_1 : b,\ \gamma(l_1) = \mathrm{base}(t), t' = \mathit{ComputeNodes}(l_1) \\ p =\ni \{p' \in t' | \mathit{symb}(p') = \mathit{symb}(s)\},\ p \neq \bot,\ q =\ni \{q' \in t' \cup \{t\} | \mathit{succ}(q') = p\} \\ \gamma(\mathit{first}(s)) \neq \gamma(l_1),\ \gamma(\mathit{last}(s)) \neq \gamma(l_1) \\ s_p = \mathit{ComputeNodes}(\mathit{prev}(p)),\ t_p = \{t'_p \in s_p | \mathit{symb}(t'_p) = \mathit{symb}(s)\} \\ s_n = \mathit{ComputeNodes}(\mathit{next}(p)),\ t_n = \{t'_n \in s_n | \mathit{symb}(t'_n) = \mathit{symb}(s)\}\end{array}}{\begin{array}{c}\langle \Sigma, C, v\ \mathit{less} := e\rangle \longrightarrow \Sigma' \\ \text{where } \mathit{prev}' = \mathit{prev}[t_n \mapsto \mathit{prev}(p)] \\ \mathit{next}' = \mathit{next}[t_p \mapsto \mathit{next}(p)] \\ \mathit{succ}' = \mathit{succ}[q \mapsto \mathit{succ}(p)]\end{array}} \tag{2.54}$$

In Rule 2.54 v is a strongly based set, and expression e evaluates to a location l_1 which corresponds to a value that is a member of set v. Rule 2.54 corresponds to the case when location l_1 corresponds to a base record that is neither the first, nor the last base record in the linked list for set v. The other rules are similar and therefore omitted.

Command v_1 *from* v_2 The evaluation aborts if v_2 evaluates to *undefined*. The rules for the aborted computation are similar to Rules 2.48 and 2.49, and are omitted.

$$\frac{\begin{array}{c}\rho(v_2) = l_2 : \mathit{set}(\sigma),\ \gamma(l) = \mathit{set}(s) \\ \rho(v_1) = l_1 : \sigma,\ \mathit{first}(s) = \bot\end{array}}{\begin{array}{c}\langle \Sigma, C, v_1\ \mathit{from}\ v_2\rangle \longrightarrow \Sigma' \\ \text{where } \gamma' = \gamma[l_1 \mapsto \gamma(l_{om})]\end{array}} \tag{2.55}$$

According to Rule 2.55, if v_2 is empty, then v_1 is set to om.

$$\frac{\begin{array}{c}\rho(v_2) = l_2 : \mathit{set}(\sigma),\ \gamma(l) = \mathit{set}(s) \\ \rho(v_1) = l_1 : \sigma,\ \mathit{first}(s) \neq \bot,\ \mathit{first}(s) = \mathit{last}(s)\end{array}}{\begin{array}{c}\langle \Sigma, C, v_1\ \mathit{from}\ v_2\rangle \longrightarrow \Sigma' \\ \text{where } \gamma' = \gamma[l_1 \mapsto \gamma(\mathit{first}(s))] \\ \mathit{first}' = \mathit{first}[s \mapsto \bot] \\ \mathit{last}' = \mathit{last}[s \mapsto \bot]\end{array}} \tag{2.56}$$

$$\frac{\begin{array}{c}\rho(v_2) = l_2 : \mathit{set}(\sigma),\ \gamma(l) = \mathit{set}(s)\\ \rho(v_1) = l_1 : \sigma,\ \mathit{first}(s) \neq \bot,\ \mathit{first}(s) \neq \mathit{last}(s)\end{array}}{\begin{array}{c}\langle \Sigma, C, v_1\ \mathit{from}\ v_2 \rangle \longrightarrow \Sigma' \\ \text{where } \gamma' = \gamma[l_1 \mapsto \gamma(\mathit{first}(s))] \\ \mathit{first}' = \mathit{first}[s \mapsto \mathit{next}(\mathit{first}(s))] \\ \mathit{prev}' = \mathit{prev}[\mathit{next}(\mathit{first}(s)) \mapsto \bot]\end{array}} \tag{2.57}$$

In Rule 2.56 v_2 evaluates to a set containing exactly one element which is assigned to v_1 and removed from v_2 (thereby making it empty). In Rule 2.57 v_2 evaluates to a set containing more than one element. The first element is removed from v_2 and assigned to v_1.

In Rules 2.55-2.57 set v_2 is unbased or weakly based. Three more rules corresponding to the case when v_2 is strongly based are similar, and are omitted.

Command *InitSet*(v)

$$\frac{\rho(v) = l : \mathit{set}(\sigma),\ \gamma(l) = \gamma(l_{om}),\ s = \mathit{newLocation}()}{\begin{array}{c}\langle \Sigma, C, \mathit{InitSet}(v) \rangle \longrightarrow \Sigma' \\ \text{where } \gamma' = \gamma[l \mapsto \mathrm{set}(s)] \\ \mathit{first}' = \mathit{first}[s \mapsto \bot] \\ \mathit{last}' = \mathit{last}[s \mapsto \bot]\end{array}} \tag{2.58}$$

$$\frac{\rho(v) = l : \mathit{strong\_set}(b),\ \gamma(l) = \gamma(l_{om}),\ s = \mathit{newLocation}()}{\begin{array}{c}\langle \Sigma, C, \mathit{InitSet}(v) \rangle \longrightarrow \Sigma' \\ \text{where } \gamma' = \gamma[l \mapsto \mathrm{strongset}(s)] \\ \mathit{first}' = \mathit{first}[s \mapsto \bot] \\ \mathit{last}' = \mathit{last}[s \mapsto \bot]\end{array}} \tag{2.59}$$

Rules 2.58 and 2.59 define how undefined sets are initialized to empty sets.

$$\frac{\rho(v) = l : \mathit{set}(\sigma),\ \gamma(l) = \mathrm{set}(s)}{\begin{array}{c}\langle \Sigma, C, \mathit{InitSet}(v) \rangle \longrightarrow \Sigma' \\ \text{where } \mathit{first}' = \mathit{first}[s \mapsto \bot] \\ \mathit{last}' = \mathit{last}[s \mapsto \bot]\end{array}} \tag{2.60}$$

Rule 2.60 defines the re-initialization of a set.

$$\frac{\rho(v) = l : \mathit{strong\_set}(b),\ \gamma(l) = \mathrm{strongset}(s),\ \mathit{first}(s) = \bot}{\langle \Sigma, C, \mathit{InitSet}(v) \rangle \longrightarrow \Sigma} \tag{2.61}$$

$$\frac{\begin{array}{c}\rho(v) = l : \mathit{strong\_set}(b),\ \gamma(l) = \mathrm{strongset}(s),\ \mathit{first}(s) \neq \bot \\ n = |\mathit{ComputeStrongSet}(s)|, a_1 = \mathit{first}(s), a_{i+1} = \mathit{StrongNext}(a_i) \text{ for all } i = 1 \ldots n-1 \\ \gamma(a_i) = \mathrm{base}(s_i),\ s'_i = \mathit{ComputeNodes}(a_i) \text{ for all } i = 1 \ldots n \\ t_i =\ni \{t \in s'_i | \mathit{symb}(t) = \mathit{symb}(s)\}, t_i \neq \bot \text{ for all } i = 1 \ldots n \\ p_i =\ni \{t \in s'_i \cup \{s_i\} | \mathit{succ}(t) = t_i\} \text{ for all } i = 1 \ldots n\end{array}}{\begin{array}{c}\langle \Sigma, C, \mathit{InitSet}(v) \rangle \longrightarrow \Sigma' \\ \text{where } \mathit{first}' = \mathit{first}[s \mapsto \bot] \\ \mathit{last}' = \mathit{last}[s \mapsto \bot] \\ \mathit{succ}' = \mathit{succ}[p_i \mapsto \mathit{succ}(t_i) \text{ for all } i = 1 \ldots n]\end{array}} \tag{2.62}$$

Rules 2.61 and 2.62 define the re-initialization of strongly based sets.

Command *InitMap*(v) The rules for the initialization and re-initialization of maps are similar to those for sets and are omitted.

Command Sequence $c; P$

$$\frac{\langle \Sigma, C, c\rangle \longrightarrow \mathit{abort}}{\langle \Sigma, C, c; P\rangle \longrightarrow \mathit{abort}} \tag{2.63}$$

$$\frac{\langle \Sigma, C, c\rangle \longrightarrow \Sigma'}{\langle \Sigma, C, c; P\rangle \longrightarrow \langle \Sigma', C, P\rangle} \tag{2.64}$$

$$\frac{\langle \Sigma, C, c\rangle \longrightarrow \langle \Sigma', C, P'\rangle}{\langle \Sigma, C, c; P\rangle \longrightarrow \langle \Sigma', C, P'; P\rangle} \tag{2.65}$$

Rules 2.63-2.65 define the evaluation of a command sequence $c; P$. In Rule 2.65 the control part of the new configuration is $P'; P$. It is not difficult to see that concatenating two command sequences in this way also gives a valid command sequence.

Command if be then P_1 else P_2

$$\frac{\langle \Sigma, C, be\rangle \xrightarrow{bool} \mathit{abort}}{\langle \Sigma, C, \text{ if } be \text{ then } P_1 \text{ else } P_2\rangle \longrightarrow \mathit{abort}} \tag{2.66}$$

$$\frac{\langle \Sigma, C, be\rangle \xrightarrow{bool} \mathit{true}}{\langle \Sigma, C, \text{ if } be \text{ then } P_1 \text{ else } P_2\rangle \longrightarrow \langle \Sigma, C, P_1\rangle} \tag{2.67}$$

$$\frac{\langle \Sigma, C, be\rangle \xrightarrow{bool} \mathit{false}}{\langle \Sigma, C, \text{ if } be \text{ then } P_1 \text{ else } P_2\rangle \longrightarrow \langle \Sigma, C, P_2\rangle} \tag{2.68}$$

Command while be loop P endloop

$$\frac{\langle \Sigma, C, be\rangle \xrightarrow{bool} \mathit{abort}}{\langle \Sigma, C, \text{ while } be \text{ loop } P \text{ endloop}\rangle \longrightarrow \mathit{abort}} \tag{2.69}$$

$$\frac{\langle \Sigma, C, be\rangle \xrightarrow{bool} \mathit{false}}{\langle \Sigma, C, \text{ while } be \text{ loop } P \text{ endloop}\rangle \longrightarrow \Sigma} \tag{2.70}$$

$$\frac{\langle \Sigma, C, be\rangle \xrightarrow{bool} \mathit{true}}{\langle \Sigma, C, \text{ while } be \text{ loop } P \text{ endloop}\rangle \longrightarrow \langle \Sigma, C, \ P; \text{while } be \text{ loop } P \text{ endloop}\rangle} \tag{2.71}$$

In Rule 2.71 the control part of the new configuration is of the form $P; c$. Once again, it is not difficult to see that appending a command at the end of a command sequence still gives a valid command sequence.

Command for $v_1 \in v_2$ loop P endloop

$$\frac{\rho(v_2) = l : \mathit{set}(\sigma),\ \gamma(l) = \gamma(l_{om})}{\langle \Sigma, C,\ \text{for } v_1 \in v_2 \text{ loop } P \text{ endloop}\rangle \longrightarrow \mathit{abort}} \tag{2.72}$$

$$\frac{\rho(v_2) = l : \mathit{set}(\sigma),\ \gamma(l) = \text{set}(s),\ \mathit{first}(s) = \bot}{\langle \Sigma, C,\ \text{for } v_1 \in v_2 \text{ loop } P \text{ endloop}\rangle \longrightarrow \langle \Sigma, C,\ \text{for } v_1 : \sigma \in [\,]\text{ loop } P \text{ endloop}\rangle} \tag{2.73}$$

$$\frac{\begin{array}{c}\rho(v_2) = l : \mathit{set}(\sigma),\ \gamma(l) = \text{set}(s),\ \mathit{first}(s) \neq \bot \\ n = |\mathit{ComputeSet}(s)|,\ a_1 = \mathit{first}(s), a_{i+1} = \mathit{next}(a_i) \text{ for all } i = 1, \ldots, n-1\end{array}}{\langle \Sigma, C,\ \text{for } v_1 \in v_2 \text{ loop } P \text{ endloop}\rangle \longrightarrow \langle \Sigma, C,\ \text{for } v_1 : \sigma \in [a_1, \ldots, a_n]\text{ loop } P \text{ endloop}\rangle} \tag{2.74}$$

$$\frac{\rho(v_2) = l : \mathit{strong\_set}(b),\ \gamma(l) = \gamma(l_{om})}{\langle \Sigma, C,\ \text{for } v_1 \in v_2 \text{ loop } P \text{ endloop}\rangle \longrightarrow \mathit{abort}} \tag{2.75}$$

$$\frac{\rho(v_2) = l : \mathit{strong\_set}(b),\ \gamma(l) = \text{strongset}(s),\ \mathit{first}(s) = \bot}{\langle \Sigma, C,\ \text{for } v_1 \in v_2 \text{ loop } P \text{ endloop}\rangle \longrightarrow \langle \Sigma, C,\ \text{for } v_1 : b \in [\,]\text{ loop } P \text{ endloop}\rangle} \tag{2.76}$$

$$\frac{\begin{array}{c}\rho(v_2) = l : \mathit{strong\_set}(b),\ \gamma(l) = \text{strongset}(s),\ \mathit{first}(s) \neq \bot \\ n = |\mathit{ComputeStrongSet}(s)|,\ a_1 = \mathit{first}(s), a_{i+1} = \mathit{StrongNext}(a_i) \text{ for all } i = 1, \ldots, n-1\end{array}}{\langle \Sigma, C,\ \text{for } v_1 \in v_2 \text{ loop } P \text{ endloop}\rangle \longrightarrow \langle \Sigma, C,\ \text{for } v_1 : b \in [a_1, \ldots, a_n]\text{ loop } P \text{ endloop}\rangle} \tag{2.77}$$

$$\begin{array}{l}\langle \Sigma, C,\ \text{for } v_1 : \sigma \in [a_1, \ldots, a_n] \text{ loop } P \text{ endloop}\rangle \longrightarrow \\ \qquad \langle \Sigma[\rho \mapsto \rho[v_1 \mapsto a_1 : \sigma]], C,\ \ P; \text{for } v_1 : \sigma \in [a_2, \ldots, a_n] \text{ loop } P \text{ endloop}\rangle\end{array} \tag{2.78}$$

$$\langle \Sigma, C,\ \text{for } v_1 : \sigma \in [\,] \text{ loop } P \text{ endloop}\rangle \longrightarrow \Sigma[\rho \mapsto \rho[v_1 \mapsto \bot]] \tag{2.79}$$

Note that "for $v_1 : \sigma \in [a_1, \ldots, a_n]$ loop P endloop" is not a valid Low SETL construct. However, for the sake of convenience, we will treat it as a valid Low SETL construct, although we disallow its use in the program that we begin with. Rules 2.78 and 2.79 define how the transition relation $\longrightarrow$ behaves in the presence of this new construct. It is also clear from Rules 2.78 and 2.79, that we can never get stuck in a configuration of the form $\langle \Sigma, C, \text{for } v_1 : \sigma \in [a_1, \ldots, a_n] \text{ loop } P \text{ endloop}\rangle$ (i.e. such a configuration will always lead to another configuration). This fact will be used later to prove that the derivation sequence for a certain class of programs (that we call well-typed programs) can never get stuck.

Command for $v_1 \in \mathit{domain}(v_2)$ loop P endloop

The rules for this case are similar to the previous case, and are omitted.

In this section, we have given a dynamic operational semantics for Low SETL expressions, boolean expressions, and commands. Our goal is to associate worst-case time complexities with the implementation of Low SETL operations on a pointer machine model of computation. In the next section, we re-visit the precise definition of a pointer machine (as given by Tarjan [105]) and describe how the Low SETL operational semantics can be translated into an efficient pointer machine implementation.

2.4 Implementing Low SETL on a Pointer Machine

The following is the original definition of a *pointer machine* taken directly from Tarjan's JCSS article [105].

> A pointer machine consists of a *memory* and a finite number of *registers*. The registers are of two types: *data registers* and *pointer registers*. The memory consists of a finite but expandable pool of records. Each record consists of a finite number of fields, each of which is either a *data field* or a *pointer field*. Each field has an identifying *name*. All records are identical in structure; that is, they contain the same fields.
>
> A pointer machine manipulates *data* and *pointers*. A pointer either specifies a particular record or is null (ϕ). Each pointer register and pointer field can store one pointer. Data can be of any kind whatsoever (integers, logical values, strings, real numbers, vectors etc.). Each data register and data field can store one datum.
>
> A *program* for a pointer machine consists of a sequence of *instructions*, numbered consecutively from one. Each instruction is one of the following eight types. The last instruction of every program is **halt**. Execution and running time of pointer machines are defined in the obvious way; we charge one unit of time per machine instruction executed.
>
> Each r below denotes a pointer register, each s denotes a data register, each t denotes a register of any type, and each n denotes a field name.

$r \leftarrow \phi$	Place a null pointer in register r.
$t_1 \leftarrow t_2$	(t_1 and t_2 must be of the same type). Place the contents of register t_2 in register t_1, erasing what was there previously.
$t \leftarrow n(r)$	(n and t must be of the same type). Place the contents of the n field of the record specified by the contents of r into register t, erasing what was there previously. (If r contains ϕ, this instruction does nothing.)
$n(r) \leftarrow t$	(n and t must be of the same type). Place the contents of t into the n field of the record specified by the contents of r, erasing what was there previously. (If r contains ϕ, this instruction does nothing.)
$s_1 \leftarrow s_2 \theta s_3$	Combine the data in registers s_2 and s_3 by applying the operation θ. Store the result in s_1, erasing what was there previously.
create r	Create a new record (not specified by any existing pointer) and place a pointer to it in r. All fields of the new record initially contain a special value called *undefined* (Λ).
halt	Cease execution.
if *condition* **then go to** i	If the condition is true, then transfer control to instruction i. If the condition is false, do nothing.

> Each condition in an **if** instruction is one of the following types.

true	Always true
$t_1 == t_2$	(t_1 and t_2 must be of the same type). True if the contents of t_1 and t_2 are the same.
$p(s_1, s_2)$	True if the contents of s_1 and s_2 satisfy predicate p, where p is any predicate on data.

Given the above definition of a pointer machine, implementing Low SETL on a pointer machine is relatively straightforward. One way of doing this is to model each distinct location $l \in V_{loc}$ and each distinct variable by a record in the memory. Each record has 12 fields having the names ρ_1, ρ_2, γ_1, γ_2, *first*, *last*, *prev*, *next*, *succ*, *image*, *baseval*, *symb*. Of these, ρ_2, γ_1, and *symb* are data fields and the others are pointer fields. A record corresponding to variable v has the value Λ in all fields except ρ_1, and ρ_2. If $\rho(v) = l : \tau$, then the

Operation	Time
$v := e$	$O(1)$
$v\ \mathit{with/less} := e$	$O(1)$
$v_1(v_2) := e$	$O(1)$
$v_1\ \mathit{from}\ v_2$	$O(1)$
$\mathit{InitSet}(v)$	$O(1)$ if v is unbased or weakly based, and $O(\lvert v\rvert + 1)$ if v is strongly based
$\mathit{InitMap}(v)$	$O(1)$ if v is unbased or weakly based, and $O(\lvert \mathit{domain}(v)\rvert + 1)$ if v is strongly based

Table 2.1: Time Complexities of Low SETL operations

field ρ_1 contains the pointer to the record corresponding to location l, and field ρ_2 contains τ. A record corresponding to location l has the value Λ in the fields ρ_1, and ρ_2. If $\gamma(l) = \mathit{undefined}$, then fields γ_1, and γ_2 also contain Λ, but if $\gamma(l) = \mathit{label}(l_1)$ (where *label* could be one of int, set, smap, strongset, strongsmap, or, base), then field γ_1 contains *label* and field γ_2 contains the pointer to the record corresponding to location l_1. The other fields *first*, *last*, *prev*, *next*, *succ*, *image*, *baseval*, and *symb* correspond to the respective maps in the obvious way. The initial set of pointer registers contains a distinct register for each distinct variable, containing a pointer to the record corresponding to that variable.

From the operational semantics of expressions given in Figure 2.8, it is easily verified that the application of each of the Rules 2.12-2.23 can be done in $O(1)$ time. The only problematic case might be Rule 2.19, in which case the actual application of the rule is implementable in $O(1)$ time but the verification of the side condition $l \in \mathit{ComputeSet}(s)$ may not be implementable in $O(1)$ time. For now, let us assume that Rule 2.19 is implemented without actually verifying the side condition. We will be able to prove later that for the subset of programs we are interested in (called well-typed programs), the verification of this side condition will in fact be unnecessary, and can be safely omitted. It is also easily verified that for all expressions e, the size of the operational semantics derivation, i.e. the number of rule applications in the derivation, is bounded by a constant[10]. Thus, we infer that every operational semantics derivation for an expression e can be implemented in $O(1)$ time on a pointer machine. Similarly, it is easily verified from Figure 2.9 that every derivation for a boolean expression *be* can also be implemented in $O(1)$ time on a pointer machine.

From the operational semantics of commands given in Section 2.3.3, it is again easy to verify that the application of Rules 2.34- 2.60 can be done in $O(1)$ time. Once again, we ignore the implementation costs of the side conditions since we will show later that for well-typed programs, the satisfiability of the side conditions is guaranteed and hence their verification can be safely omitted. From Rules 2.61 and 2.62 we see that the implementation of the operation $\mathit{InitSet}(v)$ where v is a strongly based set takes time proportional to the number of elements in set v. Similarly, the time taken to implement $\mathit{InitMap}(v)$ for a strongly based map v would take time proportional to the number of elements in the domain of map v.

Thus, we see that either the evaluation of an associative access operation takes $O(1)$ time on a pointer machine, or its evaluation gets stuck. For example, consider the element deletion operation $v\ \mathit{less} := x$. From the operational semantics (Rules 2.53 and 2.54) we see that the evaluation of the command $v\ \mathit{less} := e$ proceeds only if v is a strongly based set and e evaluates to an element of a base type. If v is indeed a strongly based set, the element deletion operation can be performed in $O(1)$ time. Otherwise, the evaluation (i.e. the operational semantics derivation) gets stuck. Table 2.1 summarizes the time complexities of Low SETL commands assuming that the evaluation does not get stuck. The time complexity of execution of "if *be* then P_1 else P_2" is the maximum of the time complexities of P_1 and P_2. In a for loop of the form "for $v_1 \in v_2$ loop P endloop", the time taken to iterate over all elements of set v_2 is proportional to the number of elements in set v_2. Therefore the time complexity of the for-loop is $O(|v_2| + 1) + \Sigma_{i=1}^{|v_2|} t_i$, where t_i is the time taken for the i^{th} iteration. There is no a priori bound on the number of iterations of the while and its time complexity can only be determined by conventional algorithmic methods.

We have defined the Low SETL operational semantics in such a way that if a Low SETL program cannot perform an associative access operation in $O(1)$ time, then its evaluation (i.e. operational semantics

[10]a simple case analysis reveals that any derivation can have at most 6 rule applications

derivation) gets stuck. Therefore, if we could prove that the evaluation of a given Low SETL program P cannot get stuck, then we can infer that the time complexity of execution of each associative access operation in P is worst-case $O(1)$ time on a pointer machine. In order to prove that the evaluation of a given Low SETL program P cannot get stuck, we make use of a type system. In Section 2.5 we define a type system for Low SETL and prove that if a Low SETL program is well-typed, and if state Σ is consistent with the type derivation for the program (in a sense to be made precise later), then the operational semantics derivation for the Low SETL program starting in state Σ can never get stuck. Thus, we show that the well-typedness of a Low SETL program ensures that every associative access operation in the program can be implemented in $O(1)$ time on a pointer machine.

2.5 Static Semantics (Type System)

The static semantics (type system) makes use of a type environment TE which is a map from variables to types, i.e.

$$TE : \mathit{Vars} \mapsto \mathit{Type}.$$

2.5.1 Static Semantics for Expressions

The static semantics for the typing of expressions are given in Figure 2.11. The notation $\boxed{TE, C \vdash e : \tau}$ can be read as saying that *the given occurrence of expression e in program P has type τ under the type environment TE and the set of subtype constraints C*. Thus, the type derivation $TE, C \vdash e : \tau$ deals with the type of a particular occurrence of expression e in the context of a fixed program P. Therefore, it would be more precise to use the notation $\boxed{TE, C \vdash_P e_l : \tau}$, where the subscript l in e_l refers to the label of expression e (allowing this occurrence of expression e to be distinguished from other occurrences), and the subscript P in $\vdash_P$ indicates that the type derivation is in the context of program P. For the rest of this chapter, unless otherwise stated, we will assume that the type derivations are in the context of some fixed Low SETL program P. Therefore, the subscript P in $\vdash_P$ will be omitted. Similarly, unless there is any confusion as to which occurrence of an expression we are talking about, we will also safely omit the expression labels in the type rules to be given later in this section.

Rule 2.80 says that the type of an occurrence of variable v as an expression[11] is given by $TE(v)$. According to Rules 2.81 and 2.82, if v is a set of type $set(\sigma)$, expression $\ni v$ is of type σ, and if v is of type $strong\_set(b)$ (for some base type b), then expression $\ni v$ is of type b.

Rule 2.83 has an important side condition. According to Rule 2.83, if variable v_1 is a single-valued map of type $smap(\sigma_1, \sigma_2)$, then an occurrence of expression $v_1(v_2)$ is well-typed, only if the occurrence of this expression appears inside a loop of the form "for $v_2 \in domain(v_1)$ loop ... endloop", and v_2 is of type σ_1. Note that for the same type environment TE, an occurrence of expression $v_1(v_2)$ that does not appear inside a for-loop of the form "for $v_2 \in domain(v_1)$ loop ... endloop", would not be well-typed. Thus, it is possible for one occurrence of an expression to be well-typed, and for another to be not well-typed under the same type environment TE. Recall that Rule 2.83 corresponds to the case of self-access discussed in Section 2.2.4.

According to Rule 2.84, if v_1 is a strongly based map, and v_2 is of the appropriate base type, then the expression $v_1(v_2)$ is well-typed. Rule 2.85 just says that Low SETL expression om is of any type σ. Finally, Rule 2.86 is a type coercion rule, which says that if expression e is of base type b satisfying the subtype constraint $b < \sigma$, then expression e is also of type σ.

2.5.2 Static Semantics for Boolean Expressions

The static semantics for the boolean expressions are given in Figure 2.12. According to Rule 2.87, the boolean expression $v_1 \in v_2$ is well-typed only if v_1 is an element of some base type b and v_2 is a strongly based set of type $strong\_set(b)$. According to Rules 2.88 and 2.89, the boolean expression $v_1 == v_2$ is well-typed only

[11] Note that if variable v occurs on the left hand side of an assignment (e.g. $v := e$), then this occurrence of v is not as an expression

$$\boxed{TE, C \vdash e : \tau}$$

$$TE, C \vdash v : TE(v) \tag{2.80}$$

$$\frac{TE, C \vdash v : \mathit{set}(\sigma)}{TE, C \vdash \ni v : \sigma} \tag{2.81}$$

$$\frac{TE, C \vdash v : \mathit{strong\_set}(b)}{TE, C \vdash \ni v : b} \tag{2.82}$$

$$\frac{\begin{array}{c} TE, C \vdash v_1 : \mathit{smap}(\sigma_1, \sigma_2) \\ TE, C \vdash v_2 : \sigma_1 \end{array}}{TE, C \vdash v_1(v_2) : \sigma_2} \tag{2.83}$$
where expression $v_1(v_2)$ appears inside
for $v_2 \in \mathit{domain}(v_1)$ loop ... endloop

$$\frac{\begin{array}{c} TE, C \vdash v_1 : \mathit{strong\_smap}(b, \sigma) \\ TE, C \vdash v_2 : b \end{array}}{TE, C \vdash v_1(v_2) : \sigma} \tag{2.84}$$

$$TE, C \vdash \text{om} : \sigma \text{ for all types } \sigma \tag{2.85}$$

$$\frac{\begin{array}{c} TE, C \vdash e : b \\ (b < \sigma \in C) \end{array}}{TE, C \vdash e : \sigma} \tag{2.86}$$

Figure 2.11: Static Semantics for Expressions

if both v_1 and v_2 are both of type *int*, or both v_1 and v_2 are of some base type b. Rules 2.90 and 2.91 say that the boolean expression *IsEmptySet*(v) is well typed if v is an unbased, weakly based or strongly based set. Rules 2.92 and 2.93 are the corresponding rules of *IsEmptyMap*(v).

2.5.3 Static Semantics for Commands

The static semantics for the commands is given in Figure 2.13. Most of the rules (with the possible exception of Rules 2.95 and 2.97) are straightforward. Rules 2.95 and 2.97 require special explanation. As previously mentioned, the Low SETL type system is somewhat non-standard in the following sense. For most standard type systems it is possible to automatically perform type verification. This is not possible for the Low SETL type system because of the side conditions of Rules 2.95 and 2.97. Recall that the type derivation $TE, C \vdash c$ should be understood as the type derivation for a particular occurrence of command c in the context of a fixed program P. The side conditions of Rules 2.95, and 2.97 cannot be verified automatically because they requires a proof that for all possible invlocations of program P, expression e cannot evaluate to a value already in set v or in the domain of smap v_1.

$$\boxed{TE, C \vdash be : bool}$$

$$\frac{\begin{array}{c} TE, C \vdash v_1 : b \\ TE, C \vdash v_2 : strong\_set(b) \end{array}}{TE, C \vdash v_1 \in v_2 : bool} \tag{2.87}$$

$$\frac{\begin{array}{c} TE, C \vdash v_1 : int \\ TE, C \vdash v_2 : int \end{array}}{TE, C \vdash v_1 == v_2 : bool} \tag{2.88}$$

$$\frac{\begin{array}{c} TE, C \vdash v_1 : b \\ TE, C \vdash v_2 : b \end{array}}{TE, C \vdash v_1 == v_2 : bool} \tag{2.89}$$

$$\frac{TE, C \vdash v : set(\sigma)}{TE, C \vdash IsEmptySet(v) : bool} \tag{2.90}$$

$$\frac{TE, C \vdash v : strong\_set(b)}{TE, C \vdash IsEmptySet(v) : bool} \tag{2.91}$$

$$\frac{TE, C \vdash v : smap(\sigma_1, \sigma_2)}{TE, C \vdash IsEmptyMap(v) : bool} \tag{2.92}$$

$$\frac{TE, C \vdash v : strong\_smap(b, \sigma_2)}{TE, C \vdash IsEmptyMap(v) : bool} \tag{2.93}$$

Figure 2.12: Static Semantics for Boolean Expressions

However we feel that the use of such side conditions in the type rules for Low SETL is justified because we don't consider Low SETL as a language to be used for manual programming, but instead as an intermediate language to which programs written in High SETL and SQ$^+$ are translated. Later, we define type systems for High SETL and SQ$^+$, and specify translations from High SETL and SQ$^+$ to Low SETL along with a proof that translations of well-typed High SETL and SQ$^+$ programs always generate well-typed Low SETL programs. Thus, Low SETL programs generated from well-typed High SETL and SQ$^+$ programs are guaranteed a priori to be well-typed, and therefore do not need to be type-checked. For example, consider the case of the High SETL expression $\{x \in S | K(x)\}$ (called a *set comprehension* expression), which evaluates to the set of elements x in S that satisfy the boolean-valued predicate $K(x)$. The High SETL code fragment

```
T := { x ∈ S | K(x) }
```

is translated into the following Low SETL implementation.

```
InitSet(T);
for x ∈ S loop
    if K(x) then
```

$$\boxed{TE, C \vdash P}$$

$$\frac{\begin{array}{c} TE, C \vdash e : \tau \\ TE(v) = \tau \end{array}}{TE, C \vdash v := e} \tag{2.94}$$

$$\frac{\begin{array}{c} TE, C \vdash e : \sigma' \\ TE(v_1) = \mathit{smap}(\sigma, \sigma'),\ TE(v_2) = \sigma \end{array}}{\begin{array}{c} TE, C \vdash v_1(v_2) := e \\ \text{where one can prove } v_2 \text{ can never be an element} \\ \text{of } \mathit{domain}(v_1) \text{ prior to the execution of this command} \end{array}} \tag{2.95}$$

$$\frac{\begin{array}{c} TE, C \vdash e : \sigma' \\ TE(v_1) = \mathit{strong\_smap}(b, \sigma'),\ TE(v_2) = b \end{array}}{TE, C \vdash v_1(v_2) := e} \tag{2.96}$$

$$\frac{\begin{array}{c} TE, C \vdash e : \sigma \\ TE(v) = \mathit{set}(\sigma) \end{array}}{\begin{array}{c} TE, C \vdash v \ \mathit{with} := e \\ \text{where one can prove that the value that } e \text{ evaluates to can never} \\ \text{be an element of set } v \text{ prior to the execution of this command} \end{array}} \tag{2.97}$$

$$\frac{\begin{array}{c} TE, C \vdash e : b \\ TE(v) = \mathit{strong\_set}(b) \end{array}}{TE, C \vdash v \ \mathit{op} := e} \tag{2.98}$$

$$\frac{\begin{array}{c} TE, C \vdash v_1 : \sigma \\ TE, C \vdash v_2 : \mathit{set}(\sigma) \end{array}}{TE, C \vdash v_1 \ \mathit{from} \ v_2} \tag{2.99}$$

$$\frac{\begin{array}{c} TE, C \vdash v_1 : b \\ TE, C \vdash v_2 : \mathit{strong\_set}(b) \end{array}}{TE, C \vdash v_1 \ \mathit{from} \ v_2} \tag{2.100}$$

Figure 2.13: Static Semantics for Commands (continued on the next page)

```
        T with:= x
    endif
endloop
```

Since T is initialized to the empty set before the loop, and set S, by definition, cannot contain multiple copies of any element, we can easily infer that whenever the control flow reaches command `T with:= x`, element x cannot be a member of set T.

For readers who are still uncomfortable with the side conditions on Rules 2.95 and 2.97, we propose another way of looking at the type system. Consider an alternate type system which is the same as the one

$$\frac{TE(v) = \mathit{set}(\tau)}{TE, C \vdash \mathit{InitSet}(v)} \tag{2.101}$$

$$\frac{TE(v) = \mathit{smap}(\tau, \sigma)}{TE, C \vdash \mathit{InitMap}(v)} \tag{2.102}$$

$$\frac{\begin{array}{c} TE, C \vdash c \\ TE, C \vdash P \end{array}}{TE, C \vdash c; P} \tag{2.103}$$

$$\frac{\begin{array}{c} TE, C \vdash be : \mathit{bool} \\ TE, C \vdash P_1 \\ TE, C \vdash P_2 \end{array}}{TE, C \vdash \text{ if } be \text{ then } P_1 \text{ else } P_2} \tag{2.104}$$

$$\frac{\begin{array}{c} TE, C \vdash be : \mathit{bool} \\ TE, C \vdash P \end{array}}{TE, C \vdash \text{ while } be \text{ loop } P \text{ endloop}} \tag{2.105}$$

$$\frac{TE(v_2) = \mathit{set}(\sigma),\ TE[v_1 \mapsto \sigma], C \vdash P}{TE, C \vdash \text{ for } v_1 \in v_2 \text{ loop } P \text{ endloop}} \tag{2.106}$$

$$\frac{TE(v_2) = \mathit{strong\_set}(b),\ TE[v_1 \mapsto b], C \vdash P}{TE, C \vdash \text{ for } v_1 \in v_2 \text{ loop } P \text{ endloop}} \tag{2.107}$$

$$\frac{TE(v_2) = \mathit{smap}(\sigma, \sigma'),\ TE[v_1 \mapsto \sigma], C \vdash P}{TE, C \vdash \text{ for } v_1 \in \mathit{domain}(v_2) \text{ loop } P \text{ endloop}} \tag{2.108}$$

$$\frac{TE(v_2) = \mathit{strong\_smap}(b, \sigma),\ TE[v_1 \mapsto b], C \vdash P}{TE, C \vdash \text{ for } v_1 \in \mathit{domain}(v_2) \text{ loop } P \text{ endloop}} \tag{2.109}$$

Figure 2.13: (continued) Static Semantics for Commands

presented here except that Rules 2.95 and 2.97 do not have any side conditions. Now, we have a type system for which type checking can be done automatically. However, our final result, i.e. that the well-typedness of a Low SETL program guarantees the implementation of associative access operations in $O(1)$ time, is no longer valid. Instead, we can prove that if a Low SETL program is well-typed, and if it can be shown that for each application of Rule 2.95 expression e never evaluates to a value already in the domain of smap v_1, and for every application of Rule 2.97 expression e never evaluates to a value already in set v, then all associative access operations in the program can be performed in $O(1)$ time[12]. However, this way of looking

[12]From a pragmatic point of view, it would be possible to have a type-checker for such a type system, that in addition to type-checking, would also produce a list of assertions that would need to be proven in order to guarantee the correctness of an implementation in which all associative access operations are performed in $O(1)$ time.

at the type system would not significantly change any of the other results in this thesis.

2.6 Consistency between A State Σ and A Type Environment TE

The dynamic semantics of Low SETL make use of a state Σ and the static semantics make use of a type environment TE. In this section, we define a notion of consistency between a state Σ and a type environment TE. A state Σ associates values with variables, and a type environment associates types with variables. A state Σ may be thought of as being consistent with a type environment TE if the value associated with any variable v under state Σ is of type $TE(v)$. To be more precise, a state Σ associates not just values, but also specific implementations with each variable. Therefore, our definition of consistency turns out to be a little more complex since it also incorporates the requirement that all implementations be well-formed.

Our definition requires the introduction of a store typing ST which is a map from locations to types, i.e. $ST : V_{loc} \mapsto \mathit{Type}$. Such a use of a store typing ST to relate the dynamic and static semantics of an imperative language with a store is common (for example [108]). We say that a state Σ, a store typing ST, a type environment TE and a set of subtype constraints C are well-formed with respect to each other (denoted by $\models \Sigma, C, ST, TE$) iff the following conditions hold.

1. All variables are consistently typed in environment ρ and type environment TE, i.e.

$$\begin{array}{l} \mathit{domain}(TE) \subseteq \mathit{domain}(\rho) \wedge \\ (\forall x \in \mathit{domain}(TE))\ (\rho(x) = l : \tau \implies (ST(l) = \tau \ \wedge\ TE(x) = \tau \ \wedge\ l \in \mathit{domain}(\gamma))). \end{array}$$

2. The type of the value corresponding to a location l matches the type $ST(l)$, i.e.

$$\begin{array}{l} \mathit{domain}(\gamma) = \mathit{domain}(ST) \text{ and} \\ (\forall l \in \mathit{domain}(\gamma))\ ((\gamma(l) \neq \gamma(l_{om})) \implies \mathit{extract}(\gamma(l)) \in T[ST(l)]). \end{array}$$

3. The labels int, base, set, strongset, smap, strongsmap should correctly match the store type ST for each location, i.e.

$$\begin{array}{ll} \multicolumn{2}{l}{(\forall l \in \mathit{domain}(\gamma))} \\ (& (\gamma(l) = \mathrm{int}(x) \implies ST(l) = \mathit{int}) \\ \wedge & (\gamma(l) = \mathrm{set}(l_1) \implies ST(l) = \mathit{set}(\sigma) \text{ for some } \sigma) \\ \wedge & (\gamma(l) = \mathrm{smap}(l_1) \implies ST(l) = \mathit{smap}(\sigma_1, \sigma_2) \text{ for some } \sigma_1, \sigma_2) \\ \wedge & (\gamma(l) = \mathrm{base}(l_1) \implies ST(l) = b \text{ for some base type } b) \\ \wedge & (\gamma(l) = \mathrm{strongset}(l_1) \implies ST(l) = \mathit{strong\_set}(b) \text{ for some base type } b) \\ \wedge & (\gamma(l) = \mathrm{strongsmap}(l_1) \implies ST(l) = \mathit{strong\_smap}(b, \sigma) \text{ for some } b \text{ and } \sigma)) \end{array}$$

4. The sets should not have a member with the value *undefined*, should not have duplicate elements, and the doubly linked list implementation of the set must be well-formed, i.e.

$$\begin{array}{l} (\forall l \in \mathit{domain}(\gamma)) \\ \quad (\gamma(l) = \mathrm{set}(s) \implies \\ \quad (\mathit{first}(s) \neq \bot \implies (\mathit{prev}(\mathit{first}(s)) = \bot \ \wedge\ \mathit{next}(\mathit{last}(s)) = \bot)) \ \wedge \\ \quad (s' = \mathit{ComputeSet}(s) \implies \\ \qquad ((\forall t \in s')\ (\gamma(t) \neq \gamma(l_{om}))) \ \wedge \\ \qquad (\forall t_1, t_2 \in s')\ (t_1 \neq t_2 \iff \mathit{extract}(\gamma(t_1)) \neq \mathit{extract}(\gamma(t_2))) \wedge \\ \qquad\qquad\qquad (t_1 = \mathit{prev}(t_2) \iff t_2 = \mathit{next}(t_1))))). \end{array}$$

5. Similarly, the maps must be well-formed, i.e.

$$\begin{array}{l}
(\forall l \in \mathit{domain}(\gamma)) \\
\quad (\gamma(l) = \mathrm{smap}(s) \implies \\
\quad (\mathit{first}(s) \neq \bot \implies (\mathit{prev}(\mathit{first}(s)) = \bot \ \wedge\ \mathit{next}(\mathit{last}(s)) = \bot)) \ \wedge \\
\quad (s' = \mathit{ComputeSet}(s) \implies \\
\qquad ((\forall t \in s')\ (\gamma(t) \neq \gamma(l_{om})) \ \wedge\ (\mathit{image}(t) \neq \bot) \ \wedge\ (\gamma(\mathit{image}(t)) \neq \gamma(l_{om}))) \ \wedge \\
\qquad (\forall t_1, t_2 \in s')\ (t_1 \neq t_2 \iff \mathit{extract}(\gamma(t_1)) \neq \mathit{extract}(\gamma(t_2))) \wedge \\
\qquad\qquad\qquad (t_1 = \mathit{prev}(t_2) \iff t_2 = \mathit{next}(t_1))))).
\end{array}$$

6. Each distinct base record must correspond to a distinct value. The singly linked list of nodes for this based record must not contain more than one node with the same identifier. Moreover, if the singly linked list contains a node with an identifier v, then v must either be a strongly based set containing this element, or a strongly based map containing this element in its domain.

$(\forall l, l_1, l_2 \in \mathit{domain}(\gamma))$

- $\mathit{ST}(l) = b \implies (\exists l' \in V_{loc})\ (\gamma(l) = \mathrm{base}(l') \ \wedge\ \mathit{baseval}(l') \neq \bot \ \wedge\ \mathit{baseval}(l') \neq l_{om})$
- $(\gamma(l_1) = \mathrm{base}(s_1) \ \wedge\ \gamma(l_2) = \mathrm{base}(s_2)) \implies (s_1 \neq s_2 \iff \mathit{extract}(\gamma(l_1)) \neq \mathit{extract}(\gamma(l_2)))$
- $(\mathit{ST}(l) = b \ \wedge\ s = \mathit{ComputeNodes}(l)) \implies (\forall t_1, t_2 \in s)(t_1 \neq t_2 \iff \mathit{symb}(t_1) \neq \mathit{symb}(t_2))$
- $(\mathit{ST}(l) = b \wedge s = \mathit{ComputeNodes}(l)) \implies ((\forall t \in s)\ (\exists l' \in \mathit{domain}(\gamma))\ ((\gamma(l') = \mathrm{strongset}(s') \vee \gamma(l') = \mathrm{strongsmap}(s')) \wedge \mathit{symb}(s') = \mathit{symb}(t) \wedge (\exists t' \in \mathit{ComputeStrongSet}(s'))(\gamma(t') = \gamma(l))))$

7. The strongly based sets should be well-formed, i.e.

$$\begin{array}{l}
(\forall l \in \mathit{domain}(\gamma)) \\
\quad (\mathit{ST}(l) = \mathit{strong\_set}(b) \wedge \gamma(l) = \mathrm{strongset}(s) \wedge s' = \mathit{ComputeStrongSet}(s)) \implies \\
\quad (\forall t \in s')\ (\mathit{ST}(t) = b \ \wedge\ (\exists t' \in \mathit{ComputeNodes}(t))\ (\mathit{symb}(t') = \mathit{symb}(s) \ \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{prev}(t') = \bot \iff \mathit{first}(s) = t \ \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{next}(t') = \bot \iff \mathit{last}(s) = t)) \\
\quad (\forall t_1, t_2 \in s')\ (\gamma(t_1) \neq \gamma(t_2) \ \wedge\ ((\exists t_1' \in \mathit{ComputeNodes}(t_1))\ (\mathit{next}(t_1') = t_2)) \iff \\
\qquad\qquad\qquad\qquad ((\exists t_2' \in \mathit{ComputeNodes}(t_2))\ (\mathit{prev}(t_2') = t_1)))
\end{array}$$

8. The strongly based maps should be well-formed, i.e.

$$\begin{array}{l}
(\forall l \in \mathit{domain}(\gamma)) \\
\quad (\mathit{ST}(l) = \mathit{strong\_smap}(b, \sigma) \wedge \gamma(l) = \mathrm{strongsmap}(s) \wedge s' = \mathit{ComputeStrongSet}(s)) \\
\quad \implies \\
\quad (\forall t \in s')\ (\mathit{ST}(t) = b \ \wedge\ (\exists t' \in \mathit{ComputeNodes}(t))\ (\mathit{symb}(t') = \mathit{symb}(s) \ \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{image}(t') \neq \bot \ \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \gamma(\mathit{image}(t')) \neq \gamma(l_{om}) \ \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{prev}(t') = \bot \iff \mathit{first}(s) = t \ \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{next}(t') = \bot \iff \mathit{last}(s) = t)) \\
\quad (\forall t_1, t_2 \in s')\ (\gamma(t_1) \neq \gamma(t_2) \ \wedge\ ((\exists t_1' \in \mathit{ComputeNodes}(t_1))\ (\mathit{next}(t_1') = t_2)) \iff \\
\qquad\qquad\qquad\qquad ((\exists t_2' \in \mathit{ComputeNodes}(t_2))\ (\mathit{prev}(t_2') = t_1)))
\end{array}$$

In Section 2.7 we use this notion of consistency between a state Σ and a type environment TE to formulate a notion of consistency between the static and dynamic semantics, which leads to a proof that the execution of a well-typed program cannot get stuck.

2.7 Execution of a Well-typed Program Cannot Get Stuck

Our goal is to prove that if P is a Low SETL program such that $TE, C \vdash P$, and state Σ, and store typing ST are such that $\models \Sigma, C, ST, TE$, then the derivation sequence starting from $\langle \Sigma, C, P \rangle$ can never get stuck. In order to prove this, we first prove theorems stating the consistency of static and dynamic semantics of expressions and boolean expressions. Following that, we outline a proof of consistency of the static and dynamic semantics for commands, which leads to a proof that the derivation sequence for well-typed program, starting from any suitable state Σ that is consistent with the type environment TE, cannot get stuck. Therefore, on the basis of the discussion at the end of Section 2.4, we can infer that the well-typedness of a program guarantees an $O(1)$ time implementation of all its associative access operations.

Theorem 2.7.1 (Consistency of Static and Dynamic Semantics for Expressions)

- *If* $\models \Sigma, C, ST, TE$, *and* $TE, C \vdash e : \tau$, *and* $\langle \Sigma, C, e \rangle \xrightarrow{exp} l : \tau'$, *then* $(\gamma(l) = \gamma(l_{om})) \vee (extract(\gamma(l)) \in T[\tau])$

Proof: The proof follows by a simple rule induction on the rules for the dynamic semantics of expressions[13]. We consider here only the cases for Rules 2.12, 2.19, and 2.23. The other cases can be handled similarly, and are omitted.

Rule 2.12: $\langle \Sigma, C, v \rangle \xrightarrow{exp} \rho(v)$

Let type environment TE, and store typing ST be such that $\models \Sigma, C, ST, TE$. There are only two possible type derivations for v.

The first possible derivation is

$$TE, C \vdash v : TE(v)$$

Let $\rho(v) = l : \tau'$. If $\gamma(l) = \gamma(l_{om})$, then we are done. Otherwise, let $TE(v) = \tau$. Then, from $\models \Sigma, C, ST, TE$, we have $\tau = \tau' = ST(l)$ and $extract(\gamma(l)) \in T[ST(l)]$, as required.

The second possible derivation for v is

$$\frac{\begin{array}{c} TE, C \vdash v : b \\ (b < \sigma \in C) \end{array}}{TE, C \vdash v : \sigma}$$

This derivation requires the sub-derivation $TE, C \vdash v : b$ which must necessarily follow from Rule 2.80, since we do not allow subtype constraints of the form $b_1 < b_2$, where both b_1 and b_2 are base types. Therefore, $TE(v)$ must equal b. Again, let $\rho(v) = l : \tau'$. If $\gamma(l) = \gamma(l_{om})$, we are done. Otherwise, $\models \Sigma, C, ST, TE$ implies that $ST(l) = b = \tau'$ and $extact(\gamma(l)) \in T[b]$. Furthermore, $b < \sigma \in C$ implies that $T[b] = T[\sigma]$. Thus, $extract(\gamma(l)) \in T[\sigma]$, as required.

Rule 2.19:

$$\frac{\begin{array}{c} \langle \Sigma, C, v_2 \rangle \xrightarrow{exp} l : \sigma,\ \gamma(l) \neq \gamma(l_{om}), \\ \langle \Sigma, C, v_1 \rangle \xrightarrow{exp} l_1 : smap(\sigma, \sigma'), \\ \gamma(l_1) = \text{smap}(s) \end{array}}{\langle \Sigma, C, v_1(v_2) \rangle \xrightarrow{exp} image(l) : \sigma'} \quad \text{where } l \in ComputeSet(s)$$

[13]it may also be thought of as induction on the depth of the dynamic evaluation

Let type environment TE, and store typing ST be such that $\models \Sigma, C, ST, TE$. Derivations of the form $TE, C \vdash v_1(v_2) : \tau$ can only be made only through Rules 2.83, 2.84, or 2.86.

Let us first consider the case when Rule 2.83 (as shown below) is used to get a derivation of the form $TE, C \vdash v_1(v_2) : \tau$.

$$\frac{\begin{array}{c} TE, C \vdash v_1 : \mathit{smap}(\sigma_1, \sigma_2) \\ TE, C \vdash v_2 : \sigma_1 \end{array}}{TE, C \vdash v_1(v_2) : \sigma_2}$$

where expression $v_1(v_2)$ appears inside for $v_2 \in \mathit{domain}(v_1)$ loop ... endloop

By the inductive hypothesis, we get that $\mathit{extract}(\gamma(l_1)) \in T[\mathit{smap}(\sigma_1, \sigma_2)]$ and $\mathit{extract}(\gamma(l)) \in T[\sigma_1]$. Using the fact that $\gamma(l_1) = \mathrm{smap}(s)$, we get that

$$\begin{aligned} \mathit{extract}(\gamma(l_1)) &= \mathit{extract}(\mathrm{smap}(s)) \\ &= \{[\mathit{extract}(\gamma(t)), \mathit{extract}(\gamma(\mathit{image}(t)))] : \\ &\qquad t \in \mathit{ComputeSet}(s)\}. \end{aligned}$$

Since, $l \in \mathit{ComputeSet}(s)$, we see that

$$[\mathit{extract}(\gamma(l)), \mathit{extract}(\gamma(\mathit{image}(l)))] \in \mathit{extract}(\gamma(l_1)).$$

Furthermore,

$$\mathit{extract}(\gamma(l_1)) \in T[\mathit{smap}(\sigma_1, \sigma_2)] \implies \mathit{extract}(\gamma(l_1)) \subseteq T[\sigma_1] \times T[\sigma_2].$$

Thus, we get that $\mathit{extract}(\gamma(\mathit{image}(l))) \in T[\sigma_2]$, as required.

Next, we show that it is impossible for the derivation $TE, C \vdash v_1(v_2) : \tau$ to follow from Rule 2.84, i.e.

$$\frac{\begin{array}{c} TE, C \vdash v_1 : \mathit{strong\_smap}(b, \sigma_2) \\ TE, C \vdash v_2 : b \end{array}}{TE, C \vdash v_1(v_2) : \sigma_2}$$

Let the derivation $TE, C \vdash v_1(v_2) : \tau$ follow from Rule 2.84, if possible. In this case there must be a sub-derivation $TE, C \vdash v_1 : \mathit{strong\_smap}(b, \sigma_2)$, which must necessarily follow from Rule 2.80 (since we can't have subtype constraints for the form $b < \mathit{strong\_smap}(b, \sigma_2)$). Therefore, $TE(v_1)$ must equal $\mathit{strong\_smap}(b, \sigma_2)$. Then, using the fact that $\models \Sigma, C, ST, TE$, we get $\rho(v_1) = l' : \mathit{strong\_smap}(b, \sigma_2)$, for some location l'. However, this would mean that it is impossible to have a derivation of the form

$$\langle \Sigma, C, v_1 \rangle \xrightarrow{exp} l_1 : \mathit{smap}(\sigma, \sigma'),$$

which is a contradiction.

Finally, we consider the case when the derivation $TE, C \vdash v_1(v_2) : \tau$ follows from Rule 2.86. In this case, the result follows from an argument very similar to the one for Rule 2.83, and is therefore omitted.

Rule 2.23:

$$\frac{\langle \Sigma, C, e \rangle \xrightarrow{exp} l : b,\ b < \sigma \in C,\ \gamma(l) = \mathrm{base}(l')}{\langle \Sigma, C, e \rangle \xrightarrow{exp} \mathit{baseval}(l') : \sigma}$$

Let type environment TE, and store typing ST be such that $\models \Sigma, C, ST, TE$, and let $TE, C \vdash e : \tau$ be any type derivation for expression e. Using the inductive hypothesis, we see that $extract(\gamma(l)) \in T[\tau]$. Furthermore, using the fact that $\gamma(l) = \text{base}(l')$, we see that

$$extract(\gamma(l)) = extract(\text{base}(l')) = extract(\gamma(baseval(l'))).$$

Thus, $extract(\gamma(baseval(l'))) \in T[\tau]$, as required.

□

Theorem 2.7.2 (Consistency of Static and Dynamic Semantics for Boolean Expressions)

- *If* $\models \Sigma, C, ST, TE$, *and* $TE, C \vdash be : bool$, *and* $\langle \Sigma, C, be \rangle \xrightarrow{bool} val$, *then* val *must be either abort, or true, or false.*

Proof Idea: The proof based on the use of Theorem 2.7.1 and an exhaustive case analysis for derivations of the form $\langle \Sigma, C, be \rangle \xrightarrow{bool} val$ is relatively straightforward, and is omitted. □

The next goal is to prove that the static and dynamic semantics for commands are also consistent, which will finally lead us to a proof that if P is a well-typed program, and Σ is an initial state consistent with the typing of the program, then the derivation sequence

$$\langle \Sigma, C, P \rangle \longrightarrow \langle \Sigma_1, C, P_1 \rangle \longrightarrow \ldots \tag{2.110}$$

can never get stuck. In other words, the derivation sequence of a well-typed program is either infinite or ends in a terminating configuration, but can never end in a stuck configuration.

The conventional strategy (presented originally by Felliesen and Wright in the seminal paper [115]) to prove such a result requires the following three steps

1. Define a notion of well-formedness of a configuration.
2. Prove that well-formed configurations can only lead to well-formed configurations, i.e. a derivation sequence starting from a well-formed configuration may only contain well-formed configurations.
3. Prove that stuck configurations are inherently not well-formed

Then, it would logically follow that a derivation sequence starting from a well-formed configuration cannot lead to a stuck configuration. The strategy that we shall employ will be very similar to the above strategy, the only difference being Step 3, where instead of proving that all stuck configurations are inherently not well-formed, we shall prove the contrapositive, i.e. a well-formed configuration cannot be a stuck configuration.

2.7.1 Definition of Well-formedness of a Configuration

In this section, we formulate a precise definition of *well-formedness* of a configuration $\langle \Sigma', C, P' \rangle$ relative to a well-typed program P. In this, and the later sections in this chapter, we will always assume that we are talking in the context of a certain Low SETL program P having a type derivation $TE, C \vdash P$. The definition of well-formedness will also be relative to the well-typed program P.

First, we define a labeling for a command sequence (i.e. an assignment of labels to every occurrence of a command in the command sequence) so that we can distinguish between distinct occurrences of identical commands in a command sequence. A label l is a tuple of integers $[t_1, \ldots t_n]$. The length of a label l of an occurrence of a command c is 1 more than the nesting depth of c inside other commands. For example, a command at the outermost level in a program will have a label of length 1. A command inside a while loop at the outermost level will have a label of length 2.

We also define three operations on labels. The first operation $inc(l)$ returns a label by incrementing the last integer in label l, i.e.

$$inc([t_1, t_2, \ldots, t_n]) = [t_1, t_2, \ldots, (t_n + 1)] \tag{2.111}$$

The next operation $trunc(l, i)$ is defined for labels of length greater than i and returns a label by removing the last i integers from label l, i.e.

$$trunc([t_1, \ldots, t_n], i) = [t_1, \ldots, t_{n-i}] \text{ where } n > i \tag{2.112}$$

Finally, we define the function $extend(l, t)$ that returns a new label by appending integer t to the end of label l, i.e.

$$extend([t_1, \ldots, t_n], t) = [t_1, \ldots, t_n, t] \tag{2.113}$$

We also use $extend(l, t, t')$ as a shortcut for $extend(extend(l, t), t')$.

From now on, we assume that labels are a part of the command itself. Furthermore, given an occurrence of a command c, we use $L(c)$ to denote the label of c. For a sequence of one or more commands P', we use $L(P')$ to denote the label of the first command of P', i.e. if $P' \equiv c_1$, or $P' \equiv c_1; P_1$, then $L(P') = L(c_1)$.

Below, we also define two notions of what it means for a labeling of a command sequence to be Strongly Consistent, and Weakly Consistent.

Definition 2.7.3 Strongly Consistent Labeling

- A labeling of a command sequence $P \equiv c$ is said to be Strongly Consistent if the labeling of command c is Strongly Consistent.
- A labeling of a command sequence $P \equiv c'; P'$ is said to be Strongly Consistent if the labelings of both c' and P' are Strongly Consistent, and $L(P') = inc(L(c'))$.
- A labeling for a commands $v := e$, $v\ op := e$, $v_1(v_2) := e$, v_1 *from* v_2, *InitSet*(v), or *InitMap*(v) is always Strongly Consistent.
- A labeling for a command ($c' \equiv$ "if *be* then P_1 else P_2") is said to be Strongly Consistent if the labelings for P_1 and P_2 are Strongly Consistent, and

  $$L(P_1) = extend(L(c'), 1, 1), \text{ and } L(P_2) = extend(L(c'), 2, 1).$$

- A labeling for a command ($c' \equiv$ "while *be* loop P endloop") is said to be strongly consistent if the labeling for P is strongly consistent, and

  $$L(P) = extend(L(c'), 1).$$

  A similar definition applies to the for-loop.

It can be easily see that a Strongly Consistently labelled command sequence P must have distinct labels for all distinct occurrences of commands. The Low SETL program shown along with the labels for each command has a Strongly Consistent Labelling.

```
[1]: InitSet(U);
[2]: InitSet(V);
[3]: for x ∈ S loop
     [3,1]: if x ∈ T then
            [3,1,1,1]: U with:= x;
            else
            [3,1,2,1]: V with:= x;
            endif
     endloop
```

Definition 2.7.4 Weakly Consistent Labeling

- A labeling of a command sequence $P \equiv c$ is said to be Weakly Consistent if the labeling of command c is Strongly Consistent.
- A labeling of a command sequence $P \equiv c'; P'$ is said to be Weakly Consistent if it is either Strongly Consistent, or the labeling of c' is Strongly Consistent, the labeling of P' is Weakly Consistent, and

$$L(P') = \mathit{trunc}(L(c'), i), \text{ or } L(P') = \mathit{inc}(\mathit{trunc}(L(c'), i))$$

for some integer $i > 0$.

Every Strongly Consistent labeling is also Weakly Consistent, but the converse is not true. In the case of a Weakly Consistent labeling, distinct occurrences of syntactically identical commands may have the same label. The labeling of the command sequence shown below is Weakly Consistent but not Strongly Consistent.

```
[3,1]: if x ∈ T then
       [3,1,1,1]: U with:= x;
       else
       [3,1,2,1]: V with:= x;
       endif
[3]: for x ∈ [ a_2, a_3, ..., a_n ] loop
     [3,1]: if x ∈ T then
            [3,1,1,1]: U with:= x;
            else
            [3,1,2,1]: V with:= x;
            endif
     endloop
```

A closer look at the two examples shown above will reveal the relationship between a Strongly Consistent and a Weakly Consistent labeling. The code in the second example could be the control part of a configuration after a derivation sequence of four steps, starting from a configuration whose control part is the program in the first example. Note that in the second example, distinct occurrences of the same command `U with:= x` have the same label `[3,1,1,1]`.

Note that any occurrence of a for-loop of the form "for $v_1 \in [a_1, \ldots, a_n]$ loop P endloop" must arise from some original for-loop of the form "for $v_1 \in v_2$ loop P endloop", or "for $v_1 \in \mathit{domain}(v_2)$ loop P endloop". The label for this newly generated for-loop is taken to be the same as the label of the original for-loop from which it was generated using Rules 2.73, 2.74, 2.76, and 2.77.

Now we can give a precise definition of a well-formed configuration $\langle \Sigma', C, P' \rangle$ relative to a well-typed program P.

Definition 2.7.5 Well-formed Configuration
Let P be a Low SETL program with a Strongly Consistent labeling such that $TE, C \vdash P$, and let state Σ, and store typing ST be such that $\models \Sigma, C, ST, TE$. We call configuration $\langle \Sigma, C, P \rangle$ the start configuration. Let c_1 be the first command in the sequence of commands P' (i.e. $P' \equiv c_1$, or $P' \equiv c_1; P_1$). Then, we say that the configuration $\langle \Sigma', C, P' \rangle$ is well-formed relative to program P if

1. The configuration $\langle \Sigma', C, P' \rangle$ is reachable by a derivation sequence from the start configuration, i.e.

$$\langle \Sigma, C, P \rangle \xrightarrow{*} \langle \Sigma', C, P' \rangle.$$

2. P' has a Weakly Consistent labeling.
3. The type derivation $TE, C \vdash P$ contains a sub-derivation of the form $TE', C \vdash c_1$ for some type environment TE'.
4. $\exists ST' \supseteq ST$ such that $\models \Sigma', C, ST', TE'$.

Next we show that a derivation sequence starting from a well-formed configuration can only lead to other well-formed configurations, and that the starting configuration $\langle \Sigma, C, P \rangle$ is well-formed.

2.7.2 Well-formed Configurations lead to Well-formed Configurations

We first prove that the control parts of all configurations seen in a derivation sequence starting from the start configuration must have a Weakly Consistent labeling.

Lemma 2.7.6 *If command sequence P' has a Weakly Consistent labeling, and $\langle \Sigma', C, P' \rangle \longrightarrow \langle \Sigma'', C, P'' \rangle$, then*

1. *Command Sequence P'' must also have a Weakly Consistent labeling.*
2. *the set of labels used in Command Sequence P'' is a subset of those used in Command Sequence P'.*
3. *if $l_1 = L(P')$, and $l_2 = L(P'')$, then*

$$l_2 = inc(trunc(l_1, i)) \ \vee \ l_2 = trunc(l_1, i) \text{ for some integer } i \geq 0 \ \vee$$
$$l_2 = extend(l_1, 1) \ \vee \ l_2 = extend(l_1, 1, 1) \ \vee \ l_2 = extend(l_1, 2, 1)$$

Proof: The proof follows by a simple structural induction on the command sequence P'. Suppose, the command sequence P' is just a single command c. Then the only possible derivations of the form $\langle \Sigma', C, c \rangle \longrightarrow \langle \Sigma'', C, P'' \rangle$ must use one of the Rules 2.67, 2.68, 2.71, 2.73, 2.74, 2.76, 2.77, or 2.78. In each case, it is easily verified that if command c has a Strongly Consistent labeling (which must be true for command sequence P' to have a Weakly Consistent labeling), then so does the command sequence P''. It is also easily verified that the set of labels used in command sequence P'' can only be a subset of those used in command sequence P'. In the case of Rules 2.67, and 2.68, the label l_2 is either $extend(l_1, 1, 1)$ or $extend(l_1, 2, 1)$. In the case of Rules 2.71, and 2.78, $l_2 = extend(l_1, 1)$. In the case of Rules 2.73, 2.74, 2.76, and 2.77, $l_2 = l_1 (= trunc(l_1, 0))$.

The second case is when command sequence P' is of the form $c_1; P_1$. In that case, the only possible derivation of the form $\langle \Sigma', C, P' \rangle \longrightarrow \langle \Sigma'', C, P'' \rangle$ must use one of the Rules 2.64, or 2.65. In the case of Rule 2.64, P'' is the same as P_1 which must have a Weakly Consistent labeling since command sequence P' has a Weakly Consistent labeling. Furthermore, by the definition of a Weakly Consistent labeling, it must be the case that $l_2 = inc(l_1)$, or $l_2 = trunc(l_1, i)$ or $l_2 = inc(trunc(l_1, i))$ for some integer $i > 0$. In the case of Rule 2.65, we need to have a derivation of the form $\langle \Sigma', C, c_1 \rangle \longrightarrow \langle \Sigma'', C, P_1' \rangle$, which could again only come from Rules 2.67, 2.68, 2.71, 2.73, 2.74, 2.76, 2.77, or 2.78. In each case, it is easily verified that command sequence P'' has a Weakly Consistent Labeling and that the set of labels used in command sequence P'' can only be a subset of those used in command sequence P'. For example, consider the case when the derivation of $\langle \Sigma', C, c_1 \rangle \longrightarrow \langle \Sigma'', C, P_1' \rangle$ uses Rule 2.67. In this case, the Strongly Consistent labeling for c_1 implies a Strongly Consistent labeling for Command Sequence P_1'. Furthermore, the label of every command in P_1' must have label $extend(L(c_1), 1)$ as prefix. Then, it can be shown that if $c_1; P_1$ has a weakly consistent labeling, then so does $P_1'; P_1 \equiv P''$.

Also, the only possible values for label l_2 in the second case ($P' \equiv c_1; P_1$) are $extend(l_1, 1)$, $extend(l_1, 1, 1)$, $extend(l_1, 2, 1)$, or l_1. $\square$

Corollary 2.7.7 *If program P has a Strongly Consistent labeling, and*

$$\langle \Sigma, C, P \rangle \xrightarrow{*} \langle \Sigma', C, P' \rangle,$$

then Command Sequence P' must have a Weakly Consistent Labeling, and the set of labels in command sequence P' is a subset of those in program P.

Next we prove two lemmas and a theorem about the consistency of the static and dynamic semantics of commands, that will be used to prove that well-formed configurations can only lead to well-formed configurations.

Lemma 2.7.8 *Let P be a well-typed Low SETL program having a strongly consistent labeling such that $TE, C \vdash P$. Let c be a command occurring in program P. Then the type derivation $TE, C \vdash P$ must include a sub-derivation of the form $TE', C \vdash c$ for some type environment TE'.*

Proof Idea: The proof follows by a simple structural induction on the program P. $\square$

Lemma 2.7.9 *Let P be a well-typed Low SETL program having a strongly consistent labeling such that $TE, C \vdash P$. Let c be the first command in program P, i.e. program P is either of the form c, or of the form $c; P'$ for some command sequence P'. Then, the type derivation $TE, C \vdash P$ must include a sub-derivation of the form $TE, C \vdash c$.*

Proof Idea: The proof again follows by a simple structural induction on the program P. $\square$

Theorem 2.7.10 (Consistency of Static and Dynamic Semantics for Commands)
Let P be a well-typed Low SETL program having a strongly consistent labeling such that $TE, C \vdash P$. Let state Σ, and a store typing ST be such that $\models \Sigma, C, ST, TE$, and

$$\langle \Sigma, C, P\rangle \xrightarrow{*} \langle \Sigma', C, P'\rangle,$$

From Corollary 2.7.7, $L(P')$ must be the label of some command c in program P. Also from Lemma 2.7.8, the derivation $TE, C \vdash P$ must include a sub-derivation of the form $TE', C \vdash c$. Then, there exists a store typing $ST' \supseteq ST$ such that $\models \Sigma', C, ST', TE'$.

Proof: The proof follows by a long but relatively straightforward induction on the length of the derivation sequence. The base case of the induction is the derivation sequence of length 0, which follows trivially from Lemma 2.7.9. Let us consider a derivation sequence of length n, i.e.,

$$\langle \Sigma, C, P\rangle \xrightarrow{n-1} \langle \Sigma', C, P'\rangle \longrightarrow \langle \Sigma'', C, P''\rangle.$$

We consider the two possible cases where P' can either be of type c_1' or $c_1'; P_1'$.

Case $P' \equiv c_1'$**:** In this case the derivation step $\langle \Sigma', C, P'\rangle \longrightarrow \langle \Sigma'', C, P''\rangle$ must use one of the Rules 2.67, 2.68, 2.71, 2.73, 2.74, 2.76, 2.77 or 2.78. We consider two of the cases below.

- Rule 2.67: In this case, c_1' must be of the form "if be then P_1 else P_2", where $\langle \Sigma', C, be\rangle \xrightarrow{exp} true$. In this case, $P'' = P_1$. Let the type sub-derivation for command c_1' be $TE', C \vdash c_1'$. Then, it is not difficult to prove that this sub-derivation must further contain the sub-derivation $TE', C \vdash P_1$. Moreover, by Lemma 2.7.6 and the definition of a weakly consistent labeling, we see that the labeling of P_1 must be strongly consistent. Then, if c_2' is the first command of P_1, the type sub-derivation for P_1 must further contain a sub-derivation of the form $TE', C \vdash c_2'$. Furthermore, the resulting configuration $\langle \Sigma'', C, P''\rangle$ is the configuration $\langle \Sigma', C, P_1\rangle$. Thus, we just need to prove that $\exists ST'' \supseteq ST'$ such that $\models \Sigma', C, ST'', TE'$. From our induction hypothesis, we get $\models \Sigma', C, ST', TE'$, and choosing ST'' to be ST', we get the result.

  The cases for Rules 2.68 and 2.71 are similar to that for Rule 2.67. The cases for Rules 2.73, 2.74, 2.76 and 2.77 are trivial.
- Rule 2.78: Let c_1' be of the form "for $v_1 \in [a_1, \ldots, a_n]$ loop P_1 endloop". In this case, the type sub-derivation $TE', C \vdash c_1'$ must include a sub-derivation $TE'[v_1 \mapsto \sigma], C \vdash P_1$. So, we have $\Sigma'' = \Sigma'[\rho' \mapsto \rho'[v_1 \mapsto a_1 : \sigma]]$, and $TE'' = TE'[v_1 \mapsto \sigma]$. Using the fact $\models \Sigma', C, ST', TE'$ and choosing $ST'' = ST'$, it is easy to show that $\models \Sigma'', C, ST'', TE''$.

Case $P' \equiv c_1'; P_1'$**:** In this case the derivation step $\langle \Sigma', C, P'\rangle \longrightarrow \langle \Sigma'', C, P''\rangle$ must use either Rule 2.64 or Rule 2.65. In the case of Rule 2.65, the arguments from the preceding case (Case $P' \equiv c_1'$) suffice to prove the result. Let us consider the case if Rule 2.64 is used. In this case, we must have $\langle \Sigma', C, c_1'\rangle \longrightarrow \Sigma''$.

We can first consider the case when command c_1' is one of $v := e$, $v_1(v_2) := e$, $v\ op := e$, $v_1\ from\ v_2$, *InitSet*(v), or *InitMap*(v). In this case, it is not difficult to prove that the label $L(P_1')$ of P_1' must either be of the form $inc(trunc(l, i))$ or $trunc(l, i)$ for some integer i. Let c_2' be the first command of P_1'. It is not difficult to prove that the type sub-derivation for command c_2' in the original type derivation must be of the form $TE'', C \vdash c_2'$, where $domain(TE'') \subseteq domain(TE')$, and $\forall v \in domain(TE'') : TE''(v) = TE'(v)$. Moreover, $domain(TE') - domain(TE'')$ can either be empty (i.e. $TE'' = TE'$) or contain at most one variable which must be some loop iteration variable. In either case, it is not difficult to show that $\exists ST'' \supseteq ST'$ such that $\models \Sigma'', C, ST'', TE''$.

The other case to consider is when command c_1' is either a for or a while loop. In this case the derivation step $\langle \Sigma', C, c_1' \rangle \longrightarrow \Sigma''$ must use either Rule 2.70 or Rule 2.79. In either case, an argument similar to the one in the previous paragraph suffices to prove the result.

□

Theorem 2.7.11 ***Well-formed Configurations only lead to well-formed configurations***
Let configuration $\langle \Sigma', C, P' \rangle$ be a well-formed configuration relative to program P, and let $\langle \Sigma', C, P' \rangle \longrightarrow \langle \Sigma'', C, P'' \rangle$. Then configuration $\langle \Sigma'', C, P'' \rangle$ is well-formed.

Proof: We just need to verify that the four conditions given in Definition 2.7.5 are satisfied for the new configuration $\langle \Sigma'', C, P'' \rangle$. The first condition follows from

$$\langle \Sigma, C, P \rangle \xrightarrow{*} \langle \Sigma', C, P' \rangle \longrightarrow \langle \Sigma'', C, P'' \rangle.$$

The second condition follows from Lemma 2.7.6. The third condition follows from Lemmas 2.7.6 and 2.7.8. The fourth condition follows from Theorem 2.7.10. □

Theorem 2.7.12 *The start configuration $\langle \Sigma, C, P \rangle$ is well-formed.*

Proof: Once again we just need to verify that the four conditions in Definition 2.7.5 are satisfied. The first condition is trivially satisfied. The second condition follows from the fact that P has a Strongly Consistent labeling. The third condition follows from Lemma 2.7.9, and the fourth condition is satisfied from the assumptions about the start configuration. □

2.7.3 Well-formed Configurations cannot be Stuck

Lemma 2.7.13 *Let P be a well-typed Low SETL program having a strongly consistent labeling such that $TE, C \vdash P$. Let state Σ, and a store typing ST be such that $\models \Sigma, C, ST, TE$. Let $\langle \Sigma, C, P \rangle \xrightarrow{n} \langle \Sigma', C, P' \rangle$ where we use $\xrightarrow{n}$ to denote n steps of the derivation sequence. Let $L(P') = l' = [t_1, \ldots t_n]$, and let label l be a proper prefix of label l' such that there exists some command c in the original program having label l. Then, there exists an integer $n_1 < n$ such that*

1. *$\langle \Sigma, C, P \rangle \xrightarrow{n_1} \langle \Sigma_1, C, P_1 \rangle \xrightarrow{n-n_1} \langle \Sigma', C, P' \rangle$ where $L(P_1) = l$.*
2. *For every configuration $\langle \Sigma_i, C, P_i \rangle$ along the path $\langle \Sigma_1, C, P_1 \rangle \xrightarrow{n-n_1} \langle \Sigma', C, P' \rangle$, label l is a proper prefix of label $L(P_i)$.*

Proof Idea: The proof follows easily from Lemma 2.7.6. □

Lemma 2.7.14 *Let P be a well-typed Low SETL program having a strongly consistent labeling such that $TE, C \vdash P$. Let state Σ, and a store typing ST be such that $\models \Sigma, C, ST, TE$. Let program P contain a for loop "for $v_1 \in domain(v_2)$ loop ... endloop" where $TE(v_2) = smap(\sigma_1, \sigma_2)$. Let command c be some command inside this for loop. Let $\langle \Sigma, C, P \rangle \xrightarrow{*} \langle \Sigma', C, P' \rangle$ where $L(P') = L(c)$. Then,*

- $\gamma'(\rho'(v_2)) = \text{smap}(s)$ *for some location s such that $\rho'(v_1) \in ComputeSet(s)$.*

Proof Sketch: Let $l = L(c) = L(P')$, and let

$$l' = L(\text{for } v_1 \in \mathit{domain}(v_2) \text{ loop } \ldots \text{ endloop}).$$

Since P has a strongly consistent labeling, label l' must be a proper prefix of label l. Then, by Lemma 2.7.13, there must be an intermediate configuration $\langle \Sigma_1, C, P_1 \rangle$ such that $L(P_1) = l'$, and such that all subsequent configurations have label l' as a proper prefix. It is not difficult to prove that the property

$$\gamma'(\rho'(v_2)) = \text{smap}(s) \text{ for some location } s \text{ such that } \rho'(v_1) \in \mathit{ComputeSet}(s)$$

holds for all such configurations, using the fact that no command inside the for loop "for $v_1 \in \mathit{domain}(v_2)$ loop ... endloop" may cause modification to the values of either v_1 or v_2. $\Box$

Theorem 2.7.15 *Let P be a well-typed Low SETL program having a strongly consistent labeling such that $TE, C \vdash P$. Let state Σ, and a store typing ST be such that $\models \Sigma, C, ST, TE$. Then the derivation sequence starting from configuration $\langle \Sigma, C, P \rangle$ cannot get stuck. In other words, if*

$$\langle \Sigma, C, P \rangle \overset{*}{\longrightarrow} \langle \Sigma', C, P' \rangle,$$

then there exists a configuration $\langle \Sigma'', C, P'' \rangle$ such that

$$\langle \Sigma', C, P' \rangle \longrightarrow \langle \Sigma'', C, P'' \rangle, \text{ or}$$

$$\langle \Sigma', C, P' \rangle \longrightarrow \Sigma'', \text{ or } \langle \Sigma', C, P' \rangle \longrightarrow \mathit{abort}.$$

Proof: The proof is based on an easily verifiable fact that if the configuration $\langle \Sigma, C, c \rangle$ is not a stuck configuration, then the configuration $\langle \Sigma, C, c; P \rangle$ is also not a stuck configuration. Thus, in order to show that a configuration $\langle \Sigma', C, P' \rangle$ is not stuck, it suffices to show that the configuration $\langle \Sigma', C, c'_1 \rangle$ is not stuck, where command c'_1 is the first command of command sequence P' ($P' \equiv c'_1$, or $P' \equiv c'_1; P'_1$). Since configuration $\langle \Sigma', C, P' \rangle$ is well-formed, the type derivation $TE, C \vdash P$ must contain a type sub-derivation of the form $TE', C \vdash c'_1$, and $\exists ST' \supseteq ST$ such that $\models \Sigma', C, ST', TE'$.

We prove that the configuration $\langle \Sigma', C, c'_1 \rangle$ cannot be stuck by a case analysis on command c'_1.

- $c'_1 \equiv v_1 \mathit{ from } v_2$, or $c'_1 \equiv \mathit{InitSet}(v)$, or $c'_1 \equiv \mathit{InitMap}(v)$

  Using the fact that $TE', C \vdash c'_1$, and $\models \Sigma', C, ST', TE'$, it is easy to verify that the configuration $\langle \Sigma', C, c'_1 \rangle$ cannot be stuck.

- $c'_1 \equiv v := e$, or $c'_1 \equiv v_1(v_2) := e$, or $c'_1 \equiv v \mathit{ with} := e$, or $c'_1 \equiv v \mathit{ less} := e$

  In this case, we first prove that the evaluation of expression e cannot get stuck in state Σ', i.e either $\langle \Sigma', C, e \rangle \overset{exp}{\longrightarrow} l : \tau$, or $\langle \Sigma', C, e \rangle \overset{exp}{\longrightarrow} \mathit{abort}$. If expression e is of the form v, om, or $\ni v$, it is easily verified from Rules 2.12, 2.13, 2.14, 2.15, 2.16, 2.17 and 2.18 that the evaluation of expression e in state Σ' cannot get stuck. The remaining case is when expression e is of the form $v_1(v_2)$. In this case, the type derivation $TE', C \vdash v_1(v_2) : \tau$ must use either Rule 2.83, or Rule 2.84. In the case of Rule 2.84, v_1 is a strongly based map and it is again easy to check that the derivation cannot get stuck. In the case of Rule 2.83, the side condition ensures that expression $v_1(v_2)$ appears inside a for-loop of the form "for $v_2 \in \mathit{domain}(v_1)$ loop ... endloop", and therefore, we can make use of Lemma 2.7.14 to check that evaluation of expression e does cannot get stuck.

  Now that we know that the evaluation of expression e cannot get stuck, we need to make sure that the evaluation of the command c'_1 can also not get stuck. The cases for $c'_1 \equiv v := e$, $c'_1 \equiv v \mathit{ with} := e$, and $c'_1 \equiv v \mathit{ less} := e$ are easily verified. In the case of $c'_1 \equiv v_1(v_2) := e$, we again look at the type derivation $TE', C \vdash c'_1$, which must make use of either Rule 2.95, or 2.96. In case the derivation uses Rule 2.96, it is again easy to check that the evaluation of command c'_1 doesn't get stuck. In the case of Rule 2.95, the side-condition ensures that the value that v_2 evaluates to cannot already be an element of $\mathit{domain}(v_1)$. As a result, we know that the side-condition of Rules 2.40, and 2.41 (or the other corresponding rule which was omitted) are satisfied, which ensures that one of these rules is applicable.

- $c'_1 \equiv$ if be then P_1 else P_2, or $c'_1 \equiv$ while be loop P_1 endloop

  As in the previous case, we first need to prove that the evaluation of the boolean expression be cannot get stuck in state Σ'. This follows easily from the fact that the evaluation of the expressions inside the boolean expressions does not get stuck. Once we know that the evaluation of be doesn't get stuck, it is trivial to verify that configuration $\langle \Sigma', C, c'_1 \rangle$ is not stuck.

- $c'_1 \equiv$ for $v_1 \in v_2$ loop P endloop, or $c'_1 \equiv$ for $v_1 \in \mathit{domain}(v_2)$ loop P endloop, or $c'_1 \equiv$ for $v_1 \in [a_1, \ldots a_n]$ loop P endloop

  In all three cases, it is easily verified that configuration $\langle \Sigma', C, c'_1 \rangle$ cannot be stuck.

$\square$

Chapter 3

High SETL

3.1 Introduction

In this thesis, we present three successively more abstract algorithm specification languages called Low SETL, High SETL, and SQ$^+$, and show how these languages can help in algorithm explanation and discovery. In Chapter 2, we defined Low SETL, the lowest level (i.e. least abstract) specification language, and used a type system to make Low SETL computationally transparent with respect to a pointer machine model of computation. In this chapter, we present the next higher level specification language, High SETL.

High SETL is a statically typed, imperative, executable superset of Low SETL augmented with such abstract operations as set comprehension, quantification, and a variety of operations on sets and binary relations. Computational transparency is obtained for High SETL by implementing it in Low SETL. Since only well-typed Low SETL programs are computationally transparent, we need to ensure that the Low SETL implementations of High SETL are well-typed. We do this by defining a new type system for High SETL and defining a translation from High SETL to Low SETL such that the translation of well-typed High SETL programs is always guaranteed to generate well-typed Low SETL.

The cost of computing High SETL expressions may be determined in two ways. One way is to determine the cost of fresh evaluation of High SETL expressions by looking at a direct implementation in Low SETL. The other way is to use finite differencing transformations [70, 68], in which *repeated* evaluation of costly High SETL expressions is replaced by their less expensive incremental counterparts, in order to reduce the *cumulative* cost of the repeated evaluation of these expressions. The cost of these differential calculations are determined by associating precise amortized complexities with an *eager strategy* for maintaining equality invariants $v = E(x_1, \dots, x_n)$[1] with respect to worst-case sequences of modifications to variables $x_1, \dots, x_n$. By an *eager strategy*, we mean that each time a modification to any of the variables $x_1, \dots, x_k$ occurs, variable v is updated to re-establish the invariant.

In this chapter, we will only consider the cost of fresh evaluation of High SETL expressions. The second approach, i.e., the differential (incremental) computation of High SETL expressions will be considered in Chapter 6.

3.2 Extended Type System For Low SETL

In Chapter 2, we defined a type system for Low SETL in which the set of types included special types called base types, integers, booleans, sets, single-valued maps, strongly based sets and strongly based single-valued maps. We now extend the Low SETL type system to additionally include multi-valued maps, and fixed-length tuples (or k-tuples). The new set of types *Type* is given by the types derivable from τ in the Grammar in 3.1.

[1]where $E(x_1, \dots, x_n)$ denotes a High SETL expression with input variables $x_1, \dots, x_n$

$$\begin{array}{lll} \tau & ::= & bool \mid \sigma \mid strong\_set(b) \mid strong\_smap(b,\sigma) \mid strong\_mmap(b,\sigma) \mid \tau_1 \times \tau_2 \times \ldots \times \tau_k \\ \sigma & ::= & int \mid b \mid set(\sigma_1) \mid smap(\sigma_1,\sigma_2) \mid mmap(\sigma_1,\sigma_2) \mid \sigma_1 \times \sigma_2 \times \ldots \times \sigma_k \end{array} \tag{3.1}$$

We define a set of types *ComparableType* to be the subset of *Type*, whole elements can be compared for equality in $O(1)$ time. The set *ComparableType* contains types derivable from μ in the Grammar in 3.2.

$$\mu ::= int \mid b \mid \mu_1 \times \mu_2 \times \ldots \times \mu_k \tag{3.2}$$

Values of type *int* and a base type b can be compared for equality in $O(1)$ times. Assuming that value of each of the type μ_1, μ_2, ..., and μ_k are comparable for equality in $O(1)$ time, and k is a finite constant, it is clear that values of type $\mu_1 \times \mu_2 \times \ldots \times \mu_k$ can also be compared for equality in $O(1)$ time.

We extend Low SETL with expressions $f\{x\}$ and $t[i]$ corresponding to multi-valued maps and fixed-length tuples. Expression $f\{x\}$ returns the image of element x under a multi-valued map f, i.e. the set $\{v : \exists[u,v] \in f \mid u = x\}$, and expression $t[i]$ returns the i^{th} component of the k-tuple t if i is a literal constant between 1 and k. We also extend Low SETL with commands $f\{x\} := e$, $t[i] := e$, and $InitTuple(t,k)$, where k is a literal constant. Command $InitTuple(t,k)$ initializes t to a tuple of length k each of whose entries is initially om (undefined). The extension of the dynamic and static semantics, and the extension of the proof of computational transparency of Low SETL in the presence of these additional types, expressions and commands is straightforward, and is omitted.

3.3 High SETL

High SETL is a superset of Low SETL containing abstract expressions such as set comprehension, quantified expressions, etc. Figure 3.1 shows the syntax for High SETL boolean expressions K, expressions E (which from now onwards are assumed to include boolean expressions), commands c, and command sequences P. It is easy to verify from the syntax of High SETL that it is a superset of Low SETL.

An informal explanation of High SETL constructs is as follows. The boolean expression $\exists x \in s \mid K$ (respectively $\forall x \in s \mid K$) evaluates to *true* if there exists some element x in set s satisfying the boolean predicate K (respectively, if all elements x in set s satisfy predicate K). The High SETL operators $\cup$, $\cap$, $-$, $\uplus$ are the set union, intersection, difference, and disjoint union operators respectively. The operator $/$ is the map-override operator, i.e. if f and g are two maps, then expression f/g evaluates the map h such that $h(x) = g(x)$ whenever $x \in domain(g)$, and $h(x) = f(x)$ otherwise. The expression $\#s$ evaluates to the cardinality of set s. Given a map f, expression $ToSet(f)$ evaluates to a set of pairs representation of map f. Similarly, given a set of pairs s, expression $ToMap(s)$ evaluates to a map representation of the set of pairs s. Expressions $domain(f)$ and $range(f)$ evaluate to sets containing elements in the domain and range of map f respectively. Expression $f \circ g$ evaluates to a map which equals the composition of maps f and g. Expression $f|_s$ evaluates to a map h where $h(x) = f(x)$ if $x \in s$, and $h(x) =$ om otherwise. Expression $f[s]$ evaluates to the set $\cup_{x \in s} f\{x\}$. Expression f^{-1} evaluates to the inverse of map f. Note that the subset of High SETL containing only High SETL expressions is purely functional, i.e. does not have any side-effects.

The High SETL expressions $s \uplus t$, $\uplus_{x \in s} E$ and High SETL command $v \uplus := E$ require a special mention. A High SETL program is considered erroneous if variables s and t in the expression $s \uplus t$ can evaluate to sets which are non-disjoint. It is the responsibility of the programmer to prove that whenever the expression $s \uplus t$ is evaluated during the execution of the program, s and t must evaluate to disjoint sets. The same disjointness condition also applies to expression $\uplus_{x \in s} E$, and command $v \uplus := E$.

In this chapter we define a new static type-system for High SETL, and define a translation from High SETL to Low SETL. Moreover, we prove that the translation of all non-erroneous[2] well-typed High SETL programs leads only to well-typed Low SETL programs. Note that the well-typedness of a High SETL

[2]in the sense that all disjointness conditions in expressions $s \uplus t$, $\uplus_{x \in s} E$, and command $v \uplus := E$ should be satisfied at run-time

$s, t, f, g, u, x, x_1, \ldots, v, v_1, \ldots$: Variable Names
$E, E_1, E_2, \ldots$: High SETL expressions
$K, K_1, K_2, \ldots$: High SETL boolean expressions
$c, c_1, \ldots$: High SETL Commands
$P, P_1, \ldots$: High SETL Command Sequences
op : $\mathit{with} \mid \mathit{less} \mid \cup \mid \cap \mid - \mid \uplus \mid /$

$$\begin{array}{lll}
K & ::= & s \in t \mid s == t \mid \mathit{IsEmptySet}(s) \mid \mathit{IsEmptyMap}(f) \mid \\
 & & (\exists x \in s | K_1) \mid (\forall x \in s | K_1) \mid \text{if } K_1 \text{ then } K_2 \text{ else } K_3 \text{ endif} \mid \\
 & & \neg K_1 \mid K_1 \wedge K_2 \mid K_1 \vee K_2 \mid \text{Let } v = E \text{ in } K \\
 & & \\
E & ::= & K \mid \\
 & & x \mid \ni s \mid f(x) \mid f\{x\} \mid u[i] \mid \text{om} \mid \\
 & & s \cup t \mid s \cap t \mid s - t \mid s \times t \mid \#s \mid s \uplus t \mid \mathit{ToSet}(f) \mid \mathit{ToMap}(s) \mid \\
 & & \mathit{domain}(f) \mid \mathit{range}(f) \mid f \circ g \mid (f|_s) \mid f[s] \mid f^{-1} \mid f/g \mid \\
 & & \cup_{x \in s} E \mid \cap_{x \in s} E \mid \uplus_{x \in s} E \mid \text{Let } v = E_1 \text{ in } E_2 \mid \text{if } K \text{ then } E_1 \text{ else } E_2 \text{ endif} \mid \\
 & & \{E : x_1 \in E_1, x_2 \in E_2, \ldots, x_n \in E_n \mid K\} \\
 & & \\
P & ::= & c \mid c; P \\
 & & \\
c & ::= & v := E \mid v_1(v_2) := E \mid v_1\{v_2\} := E \mid v[i] := E \mid v \; op := E \mid \\
 & & v_1 \; \mathit{from} \; v_2 \mid \mathit{InitSet}(v) \mid \mathit{InitMap}(v) \mid \mathit{InitTuple}(v, i) \mid \\
 & & \text{if } K \text{ then } P_1 \text{ else } P_2 \text{ endif} \mid \text{while } K \text{ loop } P \text{ endloop} \mid \\
 & & \text{for } v_1 \in v_2 \text{ loop } P \text{ endloop} \mid \\
 & & \text{for } v_1 \in \mathit{domain}(v_2) \text{ loop } P \text{ endloop}
\end{array}$$

Figure 3.1: Syntax of High SETL

program does not guarantee that it is non-erroneous. It is the responsibility of the programmer to prove that the program is non-erroneous. If the program is non-erroneous, then the well-typedness of a High SETL program guarantees that the translation from High SETL to Low SETL will always generate well-typed Low SETL programs.

In Section 3.4 we present a set of type rules for High SETL expressions. In Section 3.5 we define a translation from *well-typed High SETL expressions* to Low SETL and prove that the generated Low SETL implementations are guaranteed to be well-typed. In Section 3.6 we present a set of type rules for High SETL commands, and in Section 3.7 we define a translation from *well-typed High SETL programs* to Low SETL and again prove that the generated Low SETL implementations are guaranteed to be well-typed.

3.4 Type System for High SETL Expressions

The type rules for High SETL expressions are given below. We use type judgments of the form $TE, C \vdash_H E : \tau$ for a High SETL expression E, where TE is a type environment (map from variables to types), and C is a set of subtype constraints. We say that a High SETL expression E is well-typed in the environment TE with the set of subtype constraints C, if there exists a type derivation for $TE, C \vdash_H E : \tau$.

Although the type judgments for the High SETL type system look similar to those for Low SETL, there is one important difference between the two. A type judgment $TE, C \vdash_H E : \tau$ for a High SETL expression is independent of the context that expression E appears in. On the other hand, a Low SETL type derivation is always applied in the context of a particular Low SETL program. The Low SETL type system involves judgments of the form $TE, C \vdash_P P'$, which means the command sequence P' (appearing inside a Low SETL program P) is well-typed in the context of program P. The context (i.e. the program P) is important because of the presence of type rules such as

$$\frac{\begin{array}{c} TE, C \vdash e : \sigma \\ TE(v) = set(\sigma) \end{array}}{TE, C \vdash v \ with := e} \quad \begin{array}{l} \text{where one can prove that the value that } e \text{ evaluates to can never} \\ \text{be an element of set } v \text{ prior to the execution of this command} \end{array} \tag{3.3}$$

in which the side-condition must hold in the context of program P. Thus, the well-typedness of a particular command sequence P' is not independent of the context (program P) that it appears in. For example, it is possible that $TE, C \vdash_{P_1} P'$, while $TE, C \not\vdash_{P_2} P'$, i.e. the command sequence P' is well-typed in the context of a Low SETL program P_1 but not well-typed in the context of another Low SETL program P_2. A Low SETL program P is well-typed if there exists a type derivation for $TE, C \vdash_P P$ (i.e. the entire command sequence P is well-typed in the context of program P). This context dependence in the Low SETL type system makes the system non-compositional. By eliminating this context dependence from the High SETL type system, we get a type system that is compositional[3].

The type inference rules for High SETL expressions are given below. Rules 3.4-3.10 are similar to the type rules for Low SETL expressions.

$$TE, C \vdash_H v : TE(v) \tag{3.4}$$

$$\frac{TE, C \vdash_H v : set(\sigma)}{TE, C \vdash_H \ni v : \sigma} \tag{3.5}$$

$$\frac{TE, C \vdash_H v : strong\_set(b)}{TE, C \vdash_H \ni v : b} \tag{3.6}$$

$$\frac{\begin{array}{c} TE, C \vdash_H v_1 : strong\_smap(b, \sigma) \\ TE, C \vdash_H v_2 : b \end{array}}{TE, C \vdash_H v_1(v_2) : \sigma} \tag{3.7}$$

$$\frac{\begin{array}{c} TE, C \vdash_H v_1 : strong\_mmap(b, \sigma) \\ TE, C \vdash_H v_2 : b \end{array}}{TE, C \vdash_H v_1\{v_2\} : set(\sigma)} \tag{3.8}$$

$$\frac{TE, C \vdash_H v_1 : \sigma_1 \times \sigma_2 \times \ldots \times \sigma_k}{TE, C \vdash_H v_1[i] : \sigma_i} \quad \begin{array}{l} \text{where } i \text{ is an integer} \\ \text{literal between 1 and } k \end{array} \tag{3.9}$$

$$TE, C \vdash_H \text{om} : \sigma \text{ for all types } \sigma \tag{3.10}$$

[3]A side-effect of defining a compositional type system for High SETL is that although syntactically High SETL is a superset of Low SETL, every well-typed Low SETL program is not necessarily a well-typed High SETL programs. However this does not imply that Low SETL programs can express programs that cannot be expressed in High SETL. Both Low SETL and High SETL are Turing Complete, and therefore are equally expressive. It just means that certain well-typed Low SETL programs are not well-typed according to the High SETL type system, but that there exist other well-typed High SETL programs that are semantically equivalent to these Low SETL programs.

$$\frac{\begin{array}{c} TE, C \vdash_H s : \mathit{set}(b) \text{ or } \mathit{strong\_set}(b) \\ TE, C \vdash_H t : \mathit{set}(b) \text{ or } \mathit{strong\_set}(b) \end{array}}{TE, C \vdash_H s \cup t : \mathit{set}(b) \text{ and } \mathit{strong\_set}(b)} \tag{3.11}$$

Rule 3.11 is a short form for the following eight different rules in which the types of s, t, and $s \cup t$ may be either $\mathit{set}(b)$, or $\mathit{strong\_set}(b)$. Rule 3.11 should be read as saying that both types $\mathit{set}(b)$ and $\mathit{strong\_set}(b)$ are acceptable for expression $s \cup t$ to be well-typed if each of the sets s and t is a weakly or strongly based set of type $\mathit{set}(b)$ or $\mathit{strong\_set}(b)$.

$$\frac{\begin{array}{c} TE, C \vdash_H s : \mathit{set}(b) \\ TE, C \vdash_H t : \mathit{set}(b) \end{array}}{TE, C \vdash_H s \cup t : \mathit{set}(b)}$$

$$\frac{\begin{array}{c} TE, C \vdash_H s : \mathit{strong\_set}(b) \\ TE, C \vdash_H t : \mathit{set}(b) \end{array}}{TE, C \vdash_H s \cup t : \mathit{set}(b)}$$

$$\frac{\begin{array}{c} TE, C \vdash_H s : \mathit{set}(b) \\ TE, C \vdash_H t : \mathit{strong\_set}(b) \end{array}}{TE, C \vdash_H s \cup t : \mathit{set}(b)}$$

$$\frac{\begin{array}{c} TE, C \vdash_H s : \mathit{strong\_set}(b) \\ TE, C \vdash_H t : \mathit{strong\_set}(b) \end{array}}{TE, C \vdash_H s \cup t : \mathit{set}(b)}$$

$$\frac{\begin{array}{c} TE, C \vdash_H s : \mathit{set}(b) \\ TE, C \vdash_H t : \mathit{set}(b) \end{array}}{TE, C \vdash_H s \cup t : \mathit{strong\_set}(b)}$$

$$\frac{\begin{array}{c} TE, C \vdash_H s : \mathit{strong\_set}(b) \\ TE, C \vdash_H t : \mathit{set}(b) \end{array}}{TE, C \vdash_H s \cup t : \mathit{strong\_set}(b)}$$

$$\frac{\begin{array}{c} TE, C \vdash_H s : \mathit{set}(b) \\ TE, C \vdash_H t : \mathit{strong\_set}(b) \end{array}}{TE, C \vdash_H s \cup t : \mathit{strong\_set}(b)}$$

$$\frac{\begin{array}{c} TE, C \vdash_H s : \textit{strong\_set}(b) \\ TE, C \vdash_H t : \textit{strong\_set}(b) \end{array}}{TE, C \vdash_H s \cup t : \textit{strong\_set}(b)}$$

As in the case of Rule 3.11, we will use such short forms wherever possible.

$$\frac{\begin{array}{c} TE, C \vdash_H s : \textit{set}(b_1 \times b_2 \times \ldots \times b_n) \\ TE, C \vdash_H t : \textit{set}(b_1 \times b_2 \times \ldots \times b_n) \end{array}}{TE, C \vdash_H s \cup t : \textit{set}(b_1 \times b_2 \times \ldots \times b_n)} \qquad \text{where } n \geq 2 \tag{3.12}$$

According to Rule 3.12, if s and t are both sets of type $\textit{set}(b_1 \times b_2 \times \ldots \times b_n)$, then expression $s \cup t$ is a well-typed High SETL expression of the same type.

$$\frac{\begin{array}{c} TE, C \vdash_H s : \textit{set}(b) \text{ or } \textit{strong\_set}(b) \\ TE, C \vdash_H t : \textit{set}(b) \text{ or } \textit{strong\_set}(b) \end{array}}{TE, C \vdash_H s \cap t : \textit{set}(b) \text{ and } \textit{strong\_set}(b)} \tag{3.13}$$

$$\frac{\begin{array}{c} TE, C \vdash_H s : \textit{set}(b_1 \times b_2 \times \ldots \times b_n) \\ TE, C \vdash_H t : \textit{set}(b_1 \times b_2 \times \ldots \times b_n) \end{array}}{TE, C \vdash_H s \cap t : \textit{set}(b_1 \times b_2 \times \ldots \times b_n)} \qquad \text{where } n \geq 2 \tag{3.14}$$

$$\frac{\begin{array}{c} TE, C \vdash_H s : \textit{set}(b) \text{ or } \textit{strong\_set}(b) \\ TE, C \vdash_H t : \textit{set}(b) \text{ or } \textit{strong\_set}(b) \end{array}}{TE, C \vdash_H s - t : \textit{set}(b) \text{ and } \textit{strong\_set}(b)} \tag{3.15}$$

$$\frac{\begin{array}{c} TE, C \vdash_H s : \textit{set}(b_1 \times b_2 \times \ldots \times b_n) \\ TE, C \vdash_H t : \textit{set}(b_1 \times b_2 \times \ldots \times b_n) \end{array}}{TE, C \vdash_H s - t : \textit{set}(b_1 \times b_2 \times \ldots \times b_n)} \qquad \text{where } n \geq 2 \tag{3.16}$$

Rules 3.13 and 3.15 are short form for eight different rules, and say that if both sets s and t are weakly or strongly based sets of type $\textit{set}(b)$ or $\textit{strong\_set}(b)$, then the expressions $s \cap t$ and $s - t$ are also well-typed. Similarly, according to Rules 3.14 and 3.16, if s and t are sets of type $\textit{set}(b_1 \times b_2 \times \ldots \times b_n)$, then expressions $s \cap t$ and $s - t$ are well-typed High SETL expressions of the same type.

$$\frac{\begin{array}{c} TE, C \vdash_H s : \textit{set}(\sigma_1) \\ TE, C \vdash_H t : \textit{set}(\sigma_2) \end{array}}{TE, C \vdash_H s \times t : \textit{set}(\sigma_1 \times \sigma_2)} \tag{3.17}$$

$$\frac{\begin{array}{c} TE, C \vdash_H s : \textit{strong\_set}(b_1) \\ TE, C \vdash_H t : \textit{set}(\sigma_2) \end{array}}{TE, C \vdash_H s \times t : \textit{set}(b_1 \times \sigma_2)} \tag{3.18}$$

$$\frac{\begin{array}{c} TE, C \vdash_H s : \textit{set}(\sigma_1) \\ TE, C \vdash_H t : \textit{strong\_set}(b_2) \end{array}}{TE, C \vdash_H s \times t : \textit{set}(\sigma_1 \times b_2)} \tag{3.19}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : \mathit{strong\_set}(b_1) \\ TE, C \vdash_{H} t : \mathit{strong\_set}(b_2) \end{array}}{TE, C \vdash_{H} s \times t : \mathit{set}(b_1 \times b_2)} \tag{3.20}$$

$$\frac{TE, C \vdash_{H} s : \mathit{set}(\sigma) \text{ or } \mathit{strong\_set}(b)}{TE, C \vdash_{H} \#s : \mathit{int}} \tag{3.21}$$

Rule 3.21 is short form for two different rules.

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : \mathit{set}(\sigma) \\ TE, C \vdash_{H} t : \mathit{set}(\sigma) \end{array}}{TE, C \vdash_{H} s \uplus t : \mathit{set}(\sigma)} \qquad \text{where } \sigma \text{ is not a base type} \tag{3.22}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : \mathit{set}(b) \text{ or } \mathit{strong\_set}(b) \\ TE, C \vdash_{H} t : \mathit{set}(b) \text{ or } \mathit{strong\_set}(b) \end{array}}{TE, C \vdash_{H} s \uplus t : \mathit{set}(b) \text{ and } \mathit{strong\_set}(b)} \tag{3.23}$$

Rule 3.23 is short form for eight different rules.

$$\frac{TE, C \vdash_{H} f : \mathit{smap}(\sigma_1, \sigma_2)}{TE, C \vdash_{H} \mathit{ToSet}(f) : \mathit{set}(\sigma_1 \times \sigma_2)} \tag{3.24}$$

$$\frac{TE, C \vdash_{H} f : \mathit{strong\_smap}(b_1, \sigma_2)}{TE, C \vdash_{H} \mathit{ToSet}(f) : \mathit{set}(b_1 \times \sigma_2)} \tag{3.25}$$

$$\frac{TE, C \vdash_{H} f : \mathit{mmap}(\sigma_1, \sigma_2)}{TE, C \vdash_{H} \mathit{ToSet}(f) : \mathit{set}(\sigma_1 \times \sigma_2)} \tag{3.26}$$

$$\frac{TE, C \vdash_{H} f : \mathit{strong\_mmap}(b_1, \sigma_2)}{TE, C \vdash_{H} \mathit{ToSet}(f) : \mathit{set}(b_1 \times \sigma_2)} \tag{3.27}$$

$$\frac{TE, C \vdash_{H} s : \mathit{set}(b_1 \times \sigma_2)}{TE, C \vdash_{H} \mathit{ToMap}(s) : \mathit{mmap}(b_1, \sigma_2) \text{ and } \mathit{strong\_mmap}(b_1, \sigma_2)} \tag{3.28}$$

Rule 3.28 is a short form for two different rules.

$$\frac{TE, C \vdash_{H} f : \mathit{smap}(\sigma_1, \sigma 2)}{TE, C \vdash_{H} \mathit{domain}(f) : \mathit{set}(\sigma_1)} \qquad \text{where } \sigma_1 \text{ is not a base type} \tag{3.29}$$

$$\frac{TE, C \vdash_H f : \mathit{smap}(b_1, \sigma_2) \text{ or } \mathit{strong\_smap}(b_1, \sigma_2)}{TE, C \vdash_H \mathit{domain}(f) : \mathit{set}(b_1) \text{ and } \mathit{strong\_set}(b_1)} \tag{3.30}$$

Rule 3.30 is a short form for four rules.

$$\frac{TE, C \vdash_H f : \mathit{mmap}(\sigma_1, \sigma 2)}{TE, C \vdash_H \mathit{domain}(f) : \mathit{set}(\sigma_1)} \qquad \text{where } \sigma_1 \text{ is not a base type} \tag{3.31}$$

$$\frac{TE, C \vdash_H f : \mathit{mmap}(b_1, \sigma_2) \text{ or } \mathit{strong\_mmap}(b_1, \sigma_2)}{TE, C \vdash_H \mathit{domain}(f) : \mathit{set}(b_1) \text{ and } \mathit{strong\_set}(b_1)} \tag{3.32}$$

Rule 3.32 is a short form for four rules.

$$\frac{TE, C \vdash_H f : \mathit{smap}(\sigma_1, b_2) \text{ or } \mathit{strong\_smap}(b_1, b_2)}{TE, C \vdash_H \mathit{range}(f) : \mathit{set}(b_2) \text{ and } \mathit{strong\_set}(b_2)} \tag{3.33}$$

Rule 3.33 is short form for four rules.

$$\frac{TE, C \vdash_H f : \mathit{mmap}(\sigma_1, b_2) \text{ or } \mathit{strong\_mmap}(b_1, b_2)}{TE, C \vdash_H \mathit{range}(f) : \mathit{set}(b_2) \text{ and } \mathit{strong\_set}(b_2)} \tag{3.34}$$

Rule 3.34 is short form for four rules.

$$\frac{\begin{array}{c} TE, C \vdash_H g : \mathit{smap}(\sigma_1, b_2) \\ TE, C \vdash_H f : \mathit{smap}(b_2, \sigma_3) \text{ or } \mathit{strong\_smap}(b_2, \sigma_3) \end{array}}{TE, C \vdash_H f \circ g : \mathit{smap}(\sigma_1, \sigma_3)} \qquad \text{where } \sigma_1 \text{ is not a base type} \tag{3.35}$$

Rule 3.35 is short form for two rules.

$$\frac{\begin{array}{c} TE, C \vdash_H g : \mathit{smap}(b_1, b_2) \text{ or } \mathit{strong\_smap}(b_1, b_2) \\ TE, C \vdash_H f : \mathit{smap}(b_2, \sigma_3) \text{ or } \mathit{strong\_smap}(b_2, \sigma_3) \end{array}}{TE, C \vdash_H f \circ g : \mathit{smap}(b_1, \sigma_3) \text{ and } \mathit{strong\_smap}(b_1, \sigma_3)} \tag{3.36}$$

Rule 3.36 is short form for eight rules.

$$\frac{\begin{array}{c} TE, C \vdash_H g : \mathit{mmap}(\sigma_1, b_2) \\ TE, C \vdash_H f : \mathit{smap}(b_2, b_3) \text{ or } \mathit{strong\_smap}(b_2, b_3) \end{array}}{TE, C \vdash_H f \circ g : \mathit{mmap}(\sigma_1, b_3)} \qquad \text{where } \sigma_1 \text{ is not a base type} \tag{3.37}$$

Rule 3.37 is short form for two rules. Note that in Rule 3.37, it is very important that the elements of the range of f be of some base type b_3 rather than any type σ_3. The reason can be understood as follows. Consider an element x_1 belonging to the domain of map g. Since map g is multi-valued, the expression $g\{x\}$ may evaluate to a set containing more than one element. Suppose $g\{x\}$ evaluates to the set $\{y_1, y_2\}$.

Suppose both y_1 and y_2 are in the domain of map f. Then, by definition, the image of element x in map $f \circ g$ is $\{f(y_1), f(y_2)\}$ if $f(y_1) \neq f(y_2)$, and $\{f(y_1)\}$ if $f(y_1) = f(y_2)$. Therefore, the values $f(y_1)$ and $f(y_2)$ must be efficiently comparable for equality. In general, if $g\{x\}$ evaluates to $\{y_1, \ldots, y_n\}$, we will need to construct a set $\{f(y_1), \ldots, f(y_n)\}$. In order to do this efficiently, we require that the elements of the range of map f (i.e. elements like $f(y_1)$, $f(y_2)$, etc.) be of some base type. Similar requirements are true for Rules 3.38, 3.41 and 3.42 below.

$$\frac{\begin{array}{c} TE, C \vdash_H g : \mathit{mmap}(b_1, b_2) \text{ or } \mathit{strong\_mmap}(b_1, b_2) \\ TE, C \vdash_H f : \mathit{smap}(b_2, b_3) \text{ or } \mathit{strong\_smap}(b_2, b_3) \end{array}}{TE, C \vdash_H f \circ g : \mathit{mmap}(b_1, b_3) \text{ and } \mathit{strong\_mmap}(b_1, b_3)} \tag{3.38}$$

Rule 3.38 is short form for eight rules.

$$\frac{\begin{array}{c} TE, C \vdash_H g : \mathit{smap}(\sigma_1, b_2) \\ TE, C \vdash_H f : \mathit{mmap}(b_2, \sigma_3) \text{ or } \mathit{strong\_mmap}(b_2, \sigma_3) \end{array}}{TE, C \vdash_H f \circ g : \mathit{mmap}(\sigma_1, \sigma_3)} \qquad \text{where } \sigma_1 \text{ is not a base type} \tag{3.39}$$

Rule 3.39 is short form for two rules.

$$\frac{\begin{array}{c} TE, C \vdash_H g : \mathit{smap}(b_1, b_2) \text{ or } \mathit{strong\_smap}(b_1, b_2) \\ TE, C \vdash_H f : \mathit{mmap}(b_2, \sigma_3) \text{ or } \mathit{strong\_mmap}(b_2, \sigma_3) \end{array}}{TE, C \vdash_H f \circ g : \mathit{mmap}(b_1, \sigma_3) \text{ and } \mathit{strong\_mmap}(b_1, \sigma_3)} \tag{3.40}$$

Rule 3.40 is short form for eight rules.

$$\frac{\begin{array}{c} TE, C \vdash_H g : \mathit{mmap}(\sigma_1, b_2) \\ TE, C \vdash_H f : \mathit{mmap}(b_2, b_3) \text{ or } \mathit{strong\_mmap}(b_2, b_3) \end{array}}{TE, C \vdash_H f \circ g : \mathit{mmap}(\sigma_1, b_3)} \qquad \text{where } \sigma_1 \text{ is not a base type} \tag{3.41}$$

Rule 3.41 is short form for two rules.

$$\frac{\begin{array}{c} TE, C \vdash_H g : \mathit{mmap}(b_1, b_2) \text{ or } \mathit{strong\_mmap}(b_1, b_2) \\ TE, C \vdash_H f : \mathit{mmap}(b_2, b_3) \text{ or } \mathit{strong\_mmap}(b_2, b_3) \end{array}}{TE, C \vdash_H f \circ g : \mathit{mmap}(b_1, b_3) \text{ and } \mathit{strong\_mmap}(b_1, b_3)} \tag{3.42}$$

Rule 3.42 is short form for eight rules.

$$\frac{\begin{array}{c} TE, C \vdash_H f : \mathit{smap}(b, \sigma) \text{ or } \mathit{strong\_smap}(b, \sigma) \\ TE, C \vdash_H s : \mathit{set}(b) \text{ or } \mathit{strong\_set}(b) \end{array}}{TE, C \vdash_H (f|_s) : \mathit{smap}(b, \sigma) \text{ and } \mathit{strong\_smap}(b, \sigma)} \tag{3.43}$$

$$\frac{\begin{array}{c} TE, C \vdash_H f : \mathit{mmap}(b, \sigma) \text{ or } \mathit{strong\_mmap}(b, \sigma) \\ TE, C \vdash_H s : \mathit{set}(b) \text{ or } \mathit{strong\_set}(b) \end{array}}{TE, C \vdash_H (f|_s) : \mathit{mmap}(b, \sigma) \text{ and } \mathit{strong\_mmap}(b, \sigma)} \tag{3.44}$$

Rules 3.43 and 3.44 are short forms for eight rules each.

$$\frac{\begin{array}{c}TE, C \vdash_H f : smap(b_1, b_2) \text{ or } strong\_smap(b_1, b_2) \\ TE, C \vdash_H s : set(b_1) \text{ or } strong\_set(b_1)\end{array}}{TE, C \vdash_H (f[s]) : set(b_2) \text{ and } strong\_set(b_2)} \tag{3.45}$$

$$\frac{\begin{array}{c}TE, C \vdash_H f : mmap(b_1, b_2) \text{ or } strong\_mmap(b_1, b_2) \\ TE, C \vdash_H s : set(b_1) \text{ or } strong\_set(b_1)\end{array}}{TE, C \vdash_H (f[s]) : set(b_2) \text{ and } strong\_set(b_2)} \tag{3.46}$$

Rules 3.45 and 3.46 are short forms for eight rules each.

$$\frac{TE, C \vdash_H f : smap(\sigma_1, b_2)}{TE, C \vdash_H f^{-1} : mmap(b_2, \sigma_1) \text{ and } strong\_mmap(b_2, \sigma_1)} \tag{3.47}$$

Rule 3.47 is short form for two rules.

$$\frac{TE, C \vdash_H f : strong\_smap(b_1, b_2)}{TE, C \vdash_H f^{-1} : mmap(b_2, b_1) \text{ and } strong\_mmap(b_2, b_1)} \tag{3.48}$$

Rule 3.48 is short form for two rules.

$$\frac{TE, C \vdash_H f : mmap(\sigma_1, b_2)}{TE, C \vdash_H f^{-1} : mmap(b_2, \sigma_1) \text{ and } strong\_mmap(b_2, \sigma_1)} \tag{3.49}$$

Rule 3.49 is short form for two rules.

$$\frac{TE, C \vdash_H f : strong\_mmap(b_1, b_2)}{TE, C \vdash_H f^{-1} : mmap(b_2, b_1) \text{ and } strong\_mmap(b_2, b_1)} \tag{3.50}$$

Rule 3.50 is short form for two rules.

$$\frac{\begin{array}{c}TE, C \vdash_H f : smap(b_1, \sigma_2) \text{ or } strong\_smap(b_1, \sigma_2) \\ TE, C \vdash_H g : smap(b_1, \sigma_2) \text{ or } strong\_smap(b_1, \sigma_2)\end{array}}{TE, C \vdash_H f/g : smap(b_1, \sigma_2) \text{ and } strong\_smap(b_1, \sigma_2)} \tag{3.51}$$

Rule 3.51 is short form for eight rules.

$$\frac{\begin{array}{c}TE, C \vdash_H f : mmap(b_1, \sigma_2) \text{ or } strong\_mmap(b_1, \sigma_2) \\ TE, C \vdash_H g : mmap(b_1, \sigma_2) \text{ or } strong\_mmap(b_1, \sigma_2)\end{array}}{TE, C \vdash_H f/g : mmap(b_1, \sigma_2) \text{ and } strong\_mmap(b_1, \sigma_2)} \tag{3.52}$$

Rule 3.52 is short form for eight rules.

$$\frac{\begin{array}{c} TE, C \vdash_{H} E : b \\ (b < \sigma \in C) \end{array}}{TE, C \vdash_{H} E : \sigma} \tag{3.53}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} E_1 : \tau_1 \\ TE[v \mapsto \tau_1], C \vdash_{H} E_2 : \tau_2 \end{array}}{TE, C \vdash_{H} \text{Let } v = E_1 \text{ in } E_2 : \tau_2} \tag{3.54}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} K : \mathit{bool} \\ TE, C \vdash_{H} E_1 : \tau \\ TE, C \vdash_{H} E_2 : \tau \end{array}}{TE, C \vdash_{H} \text{if } K \text{ then } E_1 \text{ else } E_2 \text{ endif} : \tau} \tag{3.55}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : \mathit{set}(\sigma) \\ TE[x \mapsto \sigma], C \vdash_{H} E : \mathit{set}(b_1) \text{ or } \mathit{strong\_set}(b_1) \end{array}}{TE, C \vdash_{H} \cup_{x \in s} E : \mathit{set}(b_1) \text{ and } \mathit{strong\_set}(b_1)} \tag{3.56}$$

Rule 3.56 is short form for four different rules.

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : \mathit{set}(\sigma) \\ TE[x \mapsto \sigma], C \vdash_{H} E : \mathit{set}(b_1 \times b_2 \times \ldots \times b_n) \end{array}}{TE, C \vdash_{H} \cup_{x \in s} E : \mathit{set}(b_1 \times b_2 \times \ldots \times b_n)} \qquad \text{where } n \geq 2 \tag{3.57}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : \mathit{set}(\sigma) \\ TE[x \mapsto \sigma], C \vdash_{H} E : \mathit{set}(b_1) \text{ or } \mathit{strong\_set}(b_1) \end{array}}{TE, C \vdash_{H} \cap_{x \in s} E : \mathit{set}(b_1) \text{ and } \mathit{strong\_set}(b_1)} \tag{3.58}$$

Rule 3.58 is short form for four different rules.

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : \mathit{set}(\sigma) \\ TE[x \mapsto \sigma], C \vdash_{H} E : \mathit{set}(b_1 \times b_2 \times \ldots \times b_n) \end{array}}{TE, C \vdash_{H} \cap_{x \in s} E : \mathit{set}(b_1 \times b_2 \times \ldots \times b_n)} \qquad \text{where } n \geq 2 \tag{3.59}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : \mathit{set}(\sigma) \\ TE[x \mapsto \sigma], C \vdash_{H} E : \mathit{set}(\sigma') \end{array}}{TE, C \vdash_{H} \uplus_{x \in s} E : \mathit{set}(\sigma')} \tag{3.60}$$

$$\begin{array}{c} \dfrac{\begin{array}{c} (\forall i = 1, \ldots, n)\ TE[(\forall j = 1, \ldots, i-1)\ x_j \mapsto t_j], C \vdash_{H} E_i : \mathit{set}(\sigma_i) \text{ or } \mathit{strong\_set}(b_i) \\ TE[(\forall i = 1, \ldots, n)\ x_i \mapsto t_i], C \vdash_{H} K : \mathit{bool} \end{array}}{TE, C \vdash_{H} \{[x_1, \ldots, x_n] : x_1 \in E_1, \ldots, x_n \in E_n \mid K\} : \mathit{set}(t_1 \times \ldots \times t_n)} \\ \\ \text{where } t_j \text{ is } \sigma_j \text{ or } b_j \text{ depending on whether the type of} \\ E_j \text{ is } \mathit{set}(\sigma_j) \text{ or } \mathit{strong\_set}(b_j) \end{array} \tag{3.61}$$

$$\frac{\begin{array}{c}(\forall i = 1,\dots,n)\ TE[(\forall j = 1,\dots,i-1)\ x_j \mapsto t_j], C \vdash_H E_i : \mathit{set}(\sigma_i) \text{ or } \mathit{strong\_set}(b_i) \\ TE[(\forall i = 1,\dots,n)\ x_i \mapsto t_i], C \vdash_H E : \sigma \\ TE[(\forall i = 1,\dots,n)\ x_i \mapsto t_i], C \vdash_H K : \mathit{bool}\end{array}}{TE, C \vdash_H \{[[x_1,\dots,x_n], E] : x_1 \in E_1,\dots,x_n \in E_n \mid K\} : \mathit{smap}(t_1 \times \dots \times t_n, \sigma)} \tag{3.62}$$

where t_j is σ_j or b_j depending on whether the type of E_j is $\mathit{set}(\sigma_j)$ or $\mathit{strong\_set}(b_j)$

$$\frac{\begin{array}{c}(\forall i = 1,\dots,n)\ TE[(\forall j = 1,\dots,i-1)\ x_j \mapsto t_j], C \vdash_H E_i : \mathit{set}(\sigma_i) \text{ or } \mathit{strong\_set}(b_i) \\ TE[(\forall i = 1,\dots,n)\ x_i \mapsto t_i], C \vdash_H E : b \\ TE[(\forall i = 1,\dots,n)\ x_i \mapsto t_i], C \vdash_H K : \mathit{bool}\end{array}}{TE, C \vdash_H \{E : x_1 \in E_1,\dots,x_n \in E_n \mid K\} : \mathit{set}(b) \text{ and } \mathit{strong\_set}(b)} \tag{3.63}$$

where t_j is σ_j or b_j depending on whether the type of E_j is $\mathit{set}(\sigma_j)$ or $\mathit{strong\_set}(b_j)$

$$\frac{\begin{array}{c}(\forall i = 1,\dots,n)\ TE[(\forall j = 1,\dots,i-1)\ x_j \mapsto t_j], C \vdash_H E_i : \mathit{set}(\sigma_i) \text{ or } \mathit{strong\_set}(b_i) \\ TE[(\forall i = 1,\dots,n)\ x_i \mapsto t_i], C \vdash_H E : b \\ TE[(\forall i = 1,\dots,n)\ x_i \mapsto t_i], C \vdash_H E' : b' \\ TE[(\forall i = 1,\dots,n)\ x_i \mapsto t_i], C \vdash_H K : \mathit{bool}\end{array}}{TE, C \vdash_H \{[E, E'] : x_1 \in E_1,\dots,x_n \in E_n \mid K\} : \mathit{mmap}(b, b') \text{ and } \mathit{strong\_mmap}(b, b')} \tag{3.64}$$

where t_j is σ_j or b_j depending on whether the type of E_j is $\mathit{set}(\sigma_j)$ or $\mathit{strong\_set}(b_j)$

$$\frac{\begin{array}{c}TE, C \vdash_H E_1 : \mathit{set}(b_1) \text{ or } \mathit{strong\_set}(b_1) \\ TE[x \mapsto b_1], C \vdash_H E : \sigma \\ TE[x \mapsto b_1], C \vdash_H K : \mathit{bool}\end{array}}{TE, C \vdash_H \{[x, E] : x \in E_1\} : \mathit{strong\_smap}(b_1, \sigma)} \tag{3.65}$$

$$\frac{\begin{array}{c}TE, C \vdash_H E_1 : \mathit{set}(\sigma_1) \\ TE[x_1 \mapsto \sigma_1], C \vdash_H E_2 : \mathit{set}(\sigma_2) \text{ or } \mathit{strong\_set}(b_2) \\ TE[x_1 \mapsto \sigma_1, x_2 \mapsto t_2], C \vdash_H K : \mathit{bool}\end{array}}{TE, C \vdash_H \{[x_1, x_2] : x_1 \in E_1, x_2 \in E_2 \mid K\} : \mathit{mmap}(\sigma_1, t_2)} \tag{3.66}$$

where t_2 is σ_2 or b_2 depending on whether the type of E_2 is $\mathit{set}(\sigma_2)$ or $\mathit{strong\_set}(b_2)$

$$\frac{\begin{array}{c}TE, C \vdash_H E_1 : \mathit{set}(b_1) \text{ or } \mathit{strong\_set}(b_1) \\ TE[x_1 \mapsto \sigma_1], C \vdash_H E_2 : \mathit{set}(\sigma_2) \text{ or } \mathit{strong\_set}(b_2) \\ TE[x_1 \mapsto \sigma_1, x_2 \mapsto t_2], C \vdash_H K : \mathit{bool}\end{array}}{TE, C \vdash_H \{[x_1, x_2] : x_1 \in E_1, x_2 \in E_2 \mid K\} : \mathit{strong\_mmap}(b_1, t_2)} \tag{3.67}$$

where t_2 is σ_2 or b_2 depending on whether the type of E_2 is $\mathit{set}(\sigma_2)$ or $\mathit{strong\_set}(b_2)$

$$\frac{\begin{array}{c}(\forall i = 1, \ldots, n)\ TE[(\forall j = 1, \ldots, i-1)\ x_j \mapsto t_j], C \vdash_H E_i : \mathit{set}(\sigma_i) \text{ or } \mathit{strong\_set}(b_i) \\ TE[(\forall i = 1, \ldots, n)\ x_i \mapsto t_i], C \vdash_H E : \mu \\ TE[(\forall i = 1, \ldots, n)\ x_i \mapsto t_i], C \vdash_H K : \mathit{bool}\end{array}}{TE, C \vdash_H \{E : x_1 \in E_1, \ldots, x_n \in E_n \mid K\} : \mathit{set}(\mu)} \tag{3.68}$$

where t_j is σ_j or b_j depending on whether the type of E_j is $\mathit{set}(\sigma_j)$ or $\mathit{strong\_set}(b_j)$

Recall that μ represents the set of types *ComparableType* (given by the Grammar in 3.2) that consist of a subset of *Type* whose elements can be compared for equality in $O(1)$ time. Note that Rule 3.68 is different from Rule 3.63, since it does not require that expression E evaluate to a value of a base type. We just require that expression E evaluate to a value of *ComparableType*. Since every base type b is a *ComparableType*, Rule 3.68 is actually a strict generalization of Rule 3.63. However, the two rules (3.63 and 3.68) have been separated because they correspond to different Low SETL implementations having different time complexities. As will be seen in Section 3.8.1, the Low SETL implementation corresponding to Rule 3.63 is more efficient than that of Rule 3.68.

$$\frac{\begin{array}{c}TE, C \vdash_H s : b \\ TE, C \vdash_H t : \mathit{strong\_set}(b)\end{array}}{TE, C \vdash_H s \in t : \mathit{bool}} \tag{3.69}$$

$$\frac{\begin{array}{c}TE, C \vdash_H s : \mu \\ TE, C \vdash_H t : \mu\end{array}}{TE, C \vdash_H s == t : \mathit{bool}} \tag{3.70}$$

As in the case of Rule 3.68, μ represents the set of types *ComparableType*.

$$\frac{TE, C \vdash_H s : \mathit{set}(\sigma) \text{ or } \mathit{strong\_set}(b)}{TE, C \vdash_H \mathit{IsEmptySet}(s) : \mathit{bool}} \tag{3.71}$$

Rule 3.71 is short form for two rules

$$\frac{TE, C \vdash_H f : \mathit{smap}(\sigma_1, \sigma_2) \text{ or } \mathit{strong\_smap}(b_1, \sigma_2)}{TE, C \vdash_H \mathit{IsEmptyMap}(f) : \mathit{bool}} \tag{3.72}$$

Rule 3.72 is short form for two rules.

$$\frac{TE, C \vdash_H f : \mathit{mmap}(\sigma_1, \sigma_2) \text{ or } \mathit{strong\_mmap}(b_1, \sigma_2)}{TE, C \vdash_H \mathit{IsEmptyMap}(f) : \mathit{bool}} \tag{3.73}$$

Rule 3.73 is short form for two rules.

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : \mathit{set}(\sigma) \\ TE[x \mapsto \sigma], C \vdash_{H} K : \mathit{bool} \end{array}}{TE, C \vdash_{H} (\exists x \in s \mid K) : \mathit{bool}} \tag{3.74}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : \mathit{strong\_set}(b) \\ TE[x \mapsto b], C \vdash_{H} K : \mathit{bool} \end{array}}{TE, C \vdash_{H} (\exists x \in s \mid K) : \mathit{bool}} \tag{3.75}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : \mathit{set}(\sigma) \\ TE[x \mapsto \sigma], C \vdash_{H} K : \mathit{bool} \end{array}}{TE, C \vdash_{H} (\forall x \in s \mid K) : \mathit{bool}} \tag{3.76}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : \mathit{strong\_set}(b) \\ TE[x \mapsto b], C \vdash_{H} K : \mathit{bool} \end{array}}{TE, C \vdash_{H} (\forall x \in s \mid K) : \mathit{bool}} \tag{3.77}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} K_1 : \mathit{bool} \\ TE, C \vdash_{H} K_2 : \mathit{bool} \\ TE, C \vdash_{H} K_3 : \mathit{bool} \end{array}}{TE, C \vdash_{H} \text{if } K_1 \text{ then } K_2 \text{ else } K_3 \text{ endif} : \mathit{bool}} \tag{3.78}$$

$$\frac{TE, C \vdash_{H} K : \mathit{bool}}{TE, C \vdash_{H} \neg K : \mathit{bool}} \tag{3.79}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} K_1 : \mathit{bool} \\ TE, C \vdash_{H} K_2 : \mathit{bool} \end{array}}{TE, C \vdash_{H} (K_1 \wedge K_2) : \mathit{bool}} \tag{3.80}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} K_1 : \mathit{bool} \\ TE, C \vdash_{H} K_2 : \mathit{bool} \end{array}}{TE, C \vdash_{H} (K_1 \vee K_2) : \mathit{bool}} \tag{3.81}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} E : \tau \\ TE[v \mapsto \tau], C \vdash_{H} K : \mathit{bool} \end{array}}{TE, C \vdash_{H} \text{Let } v = E \text{ in } K : \mathit{bool}} \tag{3.82}$$

3.5 Low SETL Implementations of Well-Typed High SETL Expressions

In this section we define how well-typed High SETL expressions can be implemented in Low SETL. The implementations of well-typed High SETL expressions are defined inductively on the type derivation of the expression. Different type derivations correspond to different implementations. Moreover, we prove that the Low SETL code generated from well-typed High SETL is always well-typed (in the Low SETL type system). This proof is also by induction on the type derivation of the High SETL expression.

Recall from our discussion at the beginning of Section 3.4 that the Low SETL type system is non-compositional because the type rules only make sense in the context of a particular program. As a result, the following simple inductive argument for proving that Low SETL implementations of well-typed High SETL expressions are also well-typed does not suffice. The simple inductive argument is as follows. Let P_1 and P_2 be Low SETL implementations corresponding to the High SETL commands $v_1 := E_1$, and $v_2 := E_2$ respectively (i.e. P_1 and P_2 are used to evaluate expressions E_1 and E_2 respectively, and assign the results to v_1 and v_2 respectively). For simplicity, assume that v_1 does not appear in E_1 and v_2 does not appear in E_2. Let the inductive hypothesis be that both programs P_1 and P_2 are well-typed. i.e. $TE, C \vdash_{P_1} P_1$, and $TE, C \vdash_{P_2} P_2$. Now consider the Low SETL implementation P_3 of the High SETL command $v_2 :=$ (Let $v_1 := E_1$ in E_2). As we shall see later in this section, P_3 is simply the concatenation of programs P_1 and P_2, i.e. $P_1; P_2$. The goal of our inductive argument is to prove that program P_3 is well-typed, i.e. $TE, C \vdash_{P_3} P_3$. In order to do this, we would need to prove that $TE, C \vdash_{P_3} P_1$, and $TE, C \vdash_{P_3} P_2$. At this point we get stuck because the inductive hypothesis $TE, C \vdash_{P_1} P_1$ and $TE, C \vdash_{P_2} P_2$ is insufficient to prove that $TE, C \vdash_{P_3} P_1$ and $TE, C \vdash_{P_3} P_2$ since the Low SETL type system is not compositional.

To get around this difficulty, we use an inductive argument that proves a stronger result about the well-typedness of the Low SETL implemenations of well-typed High SETL expressions. The stronger result is as follows. Let P_1 be the Low SETL implementation of an arbitrary High SETL expression E. We will prove that command sequence P_1 is well-typed in the context of all Low SETL programs P that contain P_1. The formal statement of this claim is made in Theorem 3.5.1. In the proof of Theorem 3.5.1, we simultaneously define the Low SETL implementation of expression E and prove the well-typedness property of this implementation by induction on the type derivation of expression E.

Theorem 3.5.1 *Let E be a High SETL expression such that $TE, C \vdash_H E : \tau$. Let P_1 be the Low SETL code fragment evaluating expression E, and assigning the result to variable $\mathtt{v}'$ (assumed to be distinct from all variables appearing in expression E including its bound variables). Let P_1 be the sequence of commands $c_1, c_2, \ldots, c_n$, and let P_2 be any Low SETL program containing P_1 as a sub-program, i.e. containing the consecutive sequence of commands $c_1, c_2, \ldots, c_n$. Assume that the disjointness constraints imposed by the appearance of operator $\uplus$ in expression E are satisfied in all execution instances of program P_2*[4]*. Then,*

$$\exists TE' \supseteq TE \text{ such that } TE', C \vdash_{P_2} c_i, \text{ for all } i = 1, \ldots n.$$

In particular, if P_2 is P_1, then $TE, C \vdash_{P_1} P_1$, i.e. the Low SETL program P_1 implementing the High SETL command $\mathtt{v}' := E$ is well-typed.

The Low SETL code fragment P_1 evaluating expression E may use some temporary variables in addition to variable $\mathtt{v}'$ (to which the result is assigned), and the variables appearing in expression E. Moreover, $domain(TE') - domain(TE)$ contains only these temporary variables, the bound variables appearing in expression E, and the variable $\mathtt{v}'$. Furthermore, $TE'(\mathtt{v}') = \tau$.

Proof: We shall use rule induction (or induction on the length of the type derivation) for both

1. defining the Low SETL implementation of High SETL expressions, and
2. proving the well-typedness of the generated Low SETL code sequence in the context of any program P_2 containing the code sequence.

[4]For example, if E is the expression $s \uplus t$, then assume that for all execution instances of program P_2, s and t evalaute to disjoint sets whenever the control reaches command c_1

Consider the last rule (any one of Rules 3.4-3.82) that is used in the type derivation of $TE, C \vdash_H E : \tau$.

- Rules 3.4-3.10

  In the case of Rules 3.4-3.10, the High SETL expression E is also a Low SETL expression, and the Low SETL implementation of $\mathtt{v}' := E$ (where $\mathtt{v}'$ is distinct from all variables occurring in expression E) is simply

  ```
  v' := E;
  ```

  Let P_2 be any command sequence containing the command $\mathtt{v}' := E$, and choose $TE' = TE[\mathtt{v}' \mapsto \tau]$. In the case of each of the Rules 3.4-3.10, it is easy to verify that $TE', C \vdash_{P_2} \mathtt{v}' := E$. Moreover, $domain(TE') - domain(TE)$ contains only the variable $\mathtt{v}'$, and $TE'(\mathtt{v}') = \tau$.

- Rule 3.11

  As mentioned earlier, Rule 3.11 is short form for eight different rules. Consider the four rules represented by:

  $$\frac{\begin{array}{c} TE, C \vdash_H s : set(b) \text{ or } strong\_set(b) \\ TE, C \vdash_H t : set(b) \text{ or } strong\_set(b) \end{array}}{TE, C \vdash_H s \cup t : strong\_set(b)}$$

  The Low SETL implementation of $\mathtt{v}' := s \cup t$ for these rules is

  ```
  InitSet(v');
  for x ∈ s loop
      v' with:= x
  endloop;
  for y ∈ t loop
      v' with:= y
  endloop
  ```

  Let the above Low SETL implementation be denoted by the command sequence $c_1; c_2; c_3$, where c_1 is the command $\mathtt{InitSet}(\mathtt{v}')$, and commands c_2 and c_3 are the two for-loops. Let P_2 be any Low SETL program containing the sequence of commands $c_1; c_2; c_3$. Choosing $TE' = TE[\mathtt{v}' \mapsto strong\_set(b)]$, it is not difficult to prove that

  $$TE', C \vdash_{P_2} c_i \quad \text{for } i = 1, 2, 3.$$

  Moreover, $domain(TE') - domain(TE) = \mathtt{v}'$, and $TE'(\mathtt{v}') = strong\_set(b)$.

  The other four rules corresponding to Rule 3.11 are represented by

  $$\frac{\begin{array}{c} TE, C \vdash_H s : set(b) \text{ or } strong\_set(b) \\ TE, C \vdash_H t : set(b) \text{ or } strong\_set(b) \end{array}}{TE, C \vdash_H s \cup t : set(b)}$$

  In this case the Low SETL implementation for $\mathtt{v}' := s \cup t$ is

```
InitSet(v');
InitSet(temp);
for x ∈ s loop
    temp with:= x;
endloop;
for y ∈ t loop
    temp with:= y
endloop;
for z ∈ temp loop
    v' with:= temp;
endloop
```

where `temp` is a newly generated temporary variable.

Let the above code fragment be the command sequence $c_1; c_2; c_3; c_4; c_5$, and once again let P_2 be any Low SETL program containing this sequence of commands. Choosing $TE' = TE[\mathtt{v}' \mapsto set(b), \mathtt{temp} \mapsto strong\_set(b)]$, once again it is easy to show that

$$TE', C \vdash_{P_2} c_i \quad \text{for } i = 1, \ldots, 5.$$

Moreover, the only variables in $domain(TE') - domain(TE)$ are $\mathtt{v}'$ and `temp`, which is a newly generated temporary variable, and $TE'(\mathtt{v}') = set(b)$.

- Rule 3.12

  We first consider the case when sets s and t are of type $set(b_1 \times b_2)$. We present some pseudo-code below that can easily be transformed into a Low SETL implementation of $\mathtt{v}' := s \cup t$.

```
InitSet(v');
InitSet(temp1);    --  temp1: set((b1 x b2) x int)
for x ∈ s loop
    temp1 with:= [x,1]    --  we can prove that [x,1] is not in temp1
endloop;
for y ∈ t loop
    temp1 with:= [y,2]    --  we can prove that [y,2] is not in temp1
endloop;
InitSet(temp2);    --  temp2: mmap(b1-strong, (b1 x b2) x int)
for z ∈ temp1 loop
    temp2{z[1][1]} with:= z    --  we can prove that z is not
                               --  in temp2{z[1][1]}
endloop;
for u ∈ domain(temp2) loop
    InitSet(temp3);    --  temp3: set(b2-strong)
    for w ∈ temp2{u} loop
        if w[1][2] ∉ temp3 then
            v' with:= w[1];    -- we can prove that w[1] is not in v'
            temp3 with:= w[1][2]
        endif
    endloop
endloop
```

The comments in the above pseudo-code indicate the types of the temporary variables `temp1`, `temp2`, and `temp3` that are required to prove the well-typedness of the Low SETL program. The rest of the proof is long but straightforward, and is omitted. The other types $set(b_1 \times b_2 \times \ldots \times b_n)$ where $n > 2$ can also be handled similarly, and are omitted.

- Rule 3.13

  Once again Rule 3.13 is short form for eight rules. Consider the rule

$$\frac{\begin{array}{c} TE, C \vdash_{_H} s : \mathit{set}(b) \\ TE, C \vdash_{_H} t : \mathit{set}(b) \end{array}}{TE, C \vdash_{_H} s \cap t : \mathit{set}(b)}$$

  The Low SETL implementation of $\mathtt{v}' := s \cap t$ corresponding to this rule is

```
InitSet(v');
InitSet(temp);
for x ∈ s loop
    temp with:= x;
endloop;
for y ∈ t loop
    if y ∈ temp then
        v' with:= y;
    endif;
endloop
```

  where `temp` is a newly generated temporary variable.

  Let the above code fragment be denoted by the command sequence $c_1; c_2; c_3; c_4$, and let P_2 be any Low SETL program containing this code sequence. Choosing $TE' = TE[\mathtt{v}' \mapsto \mathit{set}(b), \mathtt{temp} \mapsto \mathit{strong\_set}(b)]$, it is again easy to prove that

$$TE', C \vdash_{P_2} c_i \quad \text{for all } i = 1, \ldots, 4.$$

  Moreover, the only variables in $\mathit{domain}(TE') - \mathit{domain}(TE)$ are $\mathtt{v}'$ and `temp`, which is a newly generated temporary variable, and $TE'(\mathtt{v}') = \mathit{set}(b)$.

  Another rule corresponding to Rule 3.13 is

$$\frac{\begin{array}{c} TE, C \vdash_{_H} s : \mathit{set}(b) \\ TE, C \vdash_{_H} t : \mathit{strong\_set}(b) \end{array}}{TE, C \vdash_{_H} s \cap t : \mathit{set}(b)}$$

  In this case the Low SETL implementation is

```
InitSet(v');
for x ∈ s loop
    if x ∈ t then
        v' with:= x
    endif
endloop
```

  In this case we chose $TE' = TE[\mathtt{v}' \mapsto \mathit{set}(b)]$, and the rest of the proof follows easily.

  The Low SETL implementation and the proof for the other six rules corresponding to Rule 3.13 are similar to the two cases considered above, and are omitted.

- Rule 3.14

  The proof for this case uses a combination of the ideas used to prove the cases for Rules 3.12 and 3.13, and is omitted.

- Rule 3.15

  Once again Rule 3.15 is short form for eight rules. Consider the rule

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : \mathit{set}(b) \\ TE, C \vdash_{H} t : \mathit{set}(b) \end{array}}{TE, C \vdash_{H} s - t : \mathit{strong\_set}(b)}$$

The Low SETL implementation for $\mathtt{v}' = s - t$ in this case is

```
InitSet(v');
for x ∈ s loop
    v' with:= x
endloop;
for y ∈ t loop
    v' less:= y
endloop
```

Choosing $TE' = TE[\mathtt{v}' \mapsto \mathit{strong\_set}(b)]$, the rest of the proof follows easily.

Another rule corresponding to Rule 3.15 is

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : \mathit{set}(b) \\ TE, C \vdash_{H} t : \mathit{strong\_set}(b) \end{array}}{TE, C \vdash_{H} s - t : \mathit{set}(b)}$$

In this case the Low SETL implementation is

```
InitSet(v);
for x ∈ s loop
    if ¬(x ∈ t) then
        v with:= x
    endif
endloop
```

Choosing $TE' = TE[\mathtt{v}' \mapsto \mathit{set}(b)]$, the rest of the proof follows easily. The implementations and proof for the other six rules corresponding to Rule 3.15 are similar, and are omitted.

- Rule 3.16

  The proof for this case uses a combination of the ideas used to prove the cases for Rules 3.12 and 3.15, and is omitted.

- Rules 3.17-3.20

  In each of these cases the Low SETL implementation of $\mathtt{v}' := s \times t$ is

```
InitSet(v');
for x ∈ s loop
    for y ∈ t loop
        InitTuple(temp,2);
        temp[1] := x;
        temp[2] := y;
        v' with:= temp
    endloop
endloop
```

where `temp` is a newly generated temporary variable.

In the case of Rule 3.17, choosing $TE' = TE[\mathtt{v}' \mapsto set(\sigma_1 \times \sigma_2), \mathtt{temp} \mapsto \sigma_1 \times \sigma_2]$, the rest of the proof follows easily. Similarly, in the case of Rule 3.18, choose $TE' = TE[\mathtt{v}' \mapsto set(b_1 \times \sigma_2), \mathtt{temp} \mapsto b_1 \times \sigma_2]$, in the case of Rule 3.19, choose $TE' = TE[\mathtt{v}' \mapsto set(\sigma_1 \times b_2), \mathtt{temp} \mapsto \sigma_1 \times b_2]$, and in the case of Rule 3.20, choose $TE' = TE[\mathtt{v}' \mapsto set(b_1 \times b_2), \mathtt{temp} \mapsto b_1 \times b_2]$. The proofs follow easily and are omitted.

- Rule 3.21

  The Low SETL implementation and proof are both straightforward and omitted.

- Rules 3.22-3.23

  The Low SETL implementation of $\mathtt{v}' := s \uplus t$ corresponding to Rule 3.22 is

```
InitSet(v');
for x ∈ s loop
    v' with:= x
endloop;
for y ∈ t loop
    v' with:= y
endloop
```

Let the above Low SETL code-fragment be denoted by the code-sequence $c_1; c_2; c_3$, and let P_2 be any Low SETL program containing the above code-sequence. Moreover, we assume that all executions of program P_2 satisfy the disjointness of sets s and t whenever control reaches command c_1. In this case, choosing TE' to be $TE[\mathtt{v}' \mapsto set(\sigma)]$, it is easily shown that

$$TE', C \vdash_{P_2} c_i \quad \text{for } i = 1, 2, 3.$$

The rest of the proof and other cases corresponding to Rule 3.23 are straightforward, and are omitted.

- Rules 3.24-3.27

  The Low SETL implementation of $\mathtt{v}' := ToSet(f)$; corresponding to Rule 3.24 is

```
InitSet(v');
for x ∈ domain(f) loop
    InitTuple(temp,2);
    temp[1] := x;
    temp[2] := f(x);
    v' with:= temp
endloop
```

Choosing $TE' = TE[\mathtt{v}' \mapsto set(\sigma_1 \times \sigma_2), \mathtt{temp} \mapsto \sigma_1 \times \sigma_2]$, the rest of the proof follows easily. Rules 3.25-3.27 can be handled similarly, and are omitted.

- Rule 3.28

  Rule 3.28 is short form for two rules. Consider the case for the following rule

$$\frac{TE, C \vdash_H s : set(b_1 \times \sigma_2)}{TE, C \vdash_H \mathit{ToMap}(s) : strong\_mmap(b_1, \sigma_2)}$$

  The Low SETL implementation of $\mathtt{v}' := \mathit{ToMap}(s)$ corresponding to the above rule is

```
InitMap(v');
for x in s loop
    temp1 := x[1];
    temp2 := x[2];
    temp3 := v'{temp1};
    if temp3 == om then  -- i.e. temp1 not in domain(v')
        InitSet(temp3);
    endif
    temp3 with:= temp2;
    v'{temp1} := temp3;
endloop
```

  Choosing TE' to be

$$TE[\mathtt{v}' \mapsto strong\_mmap(b_1, \sigma_2), \mathtt{temp1} \mapsto b_1, \mathtt{temp2} \mapsto \sigma_2, \mathtt{temp3} \mapsto set(\sigma_3)],$$

  the rest of the proof follows easily. The second rule corresponding to Rule 3.28 can be handled similarly, and is omitted.

- Rules 3.29-3.34

  The Low SETL implementations and the corresponding proofs for Rules 3.29-3.34 are straightforward, and are omitted.

- Rules 3.35-3.42

  Rules 3.35-3.42 correspond to forty type rules. We will look at the illustrative case of just one rule

$$\frac{\begin{array}{c}TE, C \vdash_H g : mmap(\sigma_1, b_2) \\ TE, C \vdash_H f : strong\_mmap(b_2, b_3)\end{array}}{TE, C \vdash_H f \circ g : mmap(\sigma_1, b_3)}$$

  The Low SETL implementation of $\mathtt{v}' := f \circ g$ corresponding to the above rule is

```
InitMap(v');
InitSet(temp1);
InitSet(temp4);
for x ∈ domain(g) loop
    -- sets temp1 and temp4 are always empty on entry to the loop
    temp2 = g{x};
    for y ∈ temp2 loop
        temp3 = f{y};
        if temp3 != om then
            for z ∈ temp3 loop
                temp1 with:= z
            endloop
        endif
    endloop
    if !IsEmpty(temp1) then
        for w ∈ temp1 loop
            temp4 with:= w
        endloop
        v'{x} := temp4;
        InitSet(temp4);
        InitSet(temp1)
    endif
endloop
```

Choosing $TE' = TE[\texttt{v}' \mapsto \mathit{mmap}(\sigma_1, b_3),\ \texttt{temp1} \mapsto \mathit{strong\_set}(b_3),\ \texttt{temp4} \mapsto \mathit{set}(b_3),\ \texttt{temp2} \mapsto \mathit{set}(b_2),\ \texttt{temp3} \mapsto \mathit{set}(b_3)]$, the rest of the proof is not difficult. The other cases are similar, and are omitted.

- Rules 3.43-3.52

  The Low SETL implementations and proofs corresponding to Rules 3.43-3.52 are straightforward, and are omitted.

- Rule 3.53

  The proof and Low SETL implementation for this case is also straightforward, and is also omitted.

- Rule 3.54

$$\frac{\begin{array}{c} TE, C \vdash_H E_1 : \tau_1 \\ TE[v \mapsto \tau_1], C \vdash_H E_2 : \tau_2 \end{array}}{TE, C \vdash_H \text{Let } v = E_1 \text{ in } E_2 : \tau_2} \qquad \text{where } v \notin \mathit{domain}(TE)$$

By the induction hypothesis, let P_{E_1} be the Low SETL code fragment implementing $\texttt{v} := E_1$ corresponding to the type derivation of $TE, C \vdash_H E_1 : \tau_1$, and let P_{E_2} be the Low SETL code fragment implementing $\texttt{v}' := E_2$ corresponding to the type derivation of $TE[\texttt{v} \mapsto \tau_1], C \vdash_H E_2 : \tau_2$. Then the Low SETL implementation of $\texttt{v}' :=$ Let $v = E_1$ in E_2 corresponding to the type derivation of $TE, C \vdash_H$ Let $v = E_1$ in $E_2 : \tau_2$ is simply the concatenation of the code fragments P_{E_1}, and P_{E_2}, i.e. $P_{E_1}; P_{E_2}$.

Let P_{E_1} be the command sequence $c_1; \ldots c_n$, and let P_{E_2} be the command sequence $c'_1; \ldots c'_m$. Let P_1 denote the concatenation of programs P_{E_1} and P_{E_2}, i.e. the code sequence $c_1; \ldots c_n; c'_1; \ldots c'_m$. Let P_2 be any Low SETL program containing P_1 as a sub-program. Since P_{E_1} and P_{E_2} are themselves sub-programs of P_1, they must also be sub-programs of P_2. Therefore, it follows from the induction hypothesis that $\exists\, TE'_1 \supseteq TE$ such that

$$TE'_1, C \vdash_{P_2} c_i \quad \text{for all } i = 1, \ldots, n,$$

and that $\exists\, TE'_2 \supseteq TE[\texttt{v} \mapsto \tau_1]$ such that

$$TE'_2, C \vdash_{P_2} c'_j \quad \text{for all } j = 1, \ldots, m.$$

Moreover, $\mathit{domain}(TE'_1) - \mathit{domain}(TE)$ contains only temporary and bound variables besides the variable $\mathtt{v}$, and $TE'_1(\mathtt{v}) = \tau_1$. Similarly, $\mathit{domain}(TE'_2) - \mathit{domain}(TE[\mathtt{v} \mapsto \tau_1])$ also contains only temporary and bound variables besides the variable $\mathtt{v}'$, and $TE'_2[\mathtt{v}'] = \tau_2$. By ensuring that the temporary variables in the P_{E_1} are different from those in P_{E_2} and also renaming the bound variables such that all bound variables are distinct, it is easy to prove that $TE' = TE'_1 \cup TE'_2$ is also a valid type-environment. Furthermore, it is easy to prove that

$$TE', C \vdash_{P_2} c_i \text{ for all } i = 1, \dots, n,$$

and that

$$TE', C \vdash_{P_2} c'_j \text{ for all } j = 1, \dots, m.$$

Finally, we see that $\mathit{domain}(TE') - \mathit{domain}(TE)$ only contains the variables $\mathtt{v}'$, $\mathtt{v}$ (which is the bound variable of the High SETL expression "Let $v = E_1$ in E_2"), and other temporary variables. Finally, since $TE'_2[\mathtt{v}'] = \tau_2$, it follows that $TE'[\mathtt{v}'] = \tau_2$.

- Rule 3.55

$$\frac{\begin{array}{c} TE, C \vdash_H K : \mathit{bool} \\ TE, C \vdash_H E_1 : \tau \\ TE, C \vdash_H E_2 : \tau \end{array}}{TE, C \vdash_H \text{if } K \text{ then } E_1 \text{ else } E_2 \text{ endif} : \tau}$$

By the induction hypothesis, let P_K be the Low SETL code fragment implementing $v_K := K$ corresponding to the type derivation of $TE, C \vdash_H K : \mathit{bool}$, let P_{E_1} be the Low SETL code fragment implementing $\mathtt{v}' := E_1$ corresponding to the type derivation of $TE, C \vdash_H E_1 : \tau$, and let P_{E_2} be the Low SETL code fragment implementing $\mathtt{v}' := E_2$ corresponding to the type derivation of $TE, C \vdash_H E_2 : \tau$. Let P_K be the command sequence $c_1; c_2; \dots c_l$, let P_{E_1} be the command sequence $c'_1; c'_2; \dots c'_m$, and let P_{E_2} be the command sequence $c''_1; c''_2; \dots ; c''_n$. Then the Low SETL implementation of $\mathtt{v}' :=$ if K then E_1 else E_2 endif corresponding to the type derivation of $TE, C \vdash_H$ if K then E_1 else E_2 endif : τ is

$$P_K; \text{if } (v_K == \mathit{true}) \text{ then } P_{E_1} \text{ else } P_{E_2} \text{ endif}, \quad \text{or, in other words}$$

$$c_1; c_2; \dots c_l; \text{if } (v_K == \mathit{true}) \text{ then } c'_1; c'_2; \dots c'_m \text{ else } c''_1; c''_2; \dots ; c''_n \text{ endif}$$

We use P_1 to denote the above code fragment. Let P_2 be any Low SETL program containing P_1 as a sub-program. Then P_2 also contains P_K, P_{E_1}, and P_{E_2} as sub-programs. Therefore, $\exists TE'_1 \supseteq TE$ such that

$$TE'_1, C \vdash_{P_2} c_i \text{ for all } i = 1, \dots, l,$$

$\exists TE'_2 \supseteq TE$ such that

$$TE'_2, C \vdash_{P_2} c'_i \text{ for all } i = 1, \dots, m,$$

and $\exists TE'_3 \supseteq TE$ such that

$$TE'_3, C \vdash_{P_2} c''_i \text{ for all } i = 1, \dots, n.$$

By ensuring that all the temporary variables generated in the implementation of K, E_1, and E_2 are distinct, and also ensuring that all the bound variables in K, E_1 and E_2 are also distinct, it is easy to prove that $TE' = TE'_1 \cup TE'_2 \cup TE'_3$ is a valid type environment, $TE'(v_K) = bool$, and $TE'(\mathtt{v}') = \tau$. Then, it follows that

$$TE', C \vdash_{P_2} c_i \ \text{ for all } i = 1, \dots, l,$$

and

$$TE', C \vdash_{P_2} \text{if } (v_K == \mathit{true}) \text{ then } c'_1; c'_2; \dots c'_m \text{ else } c''_1; c''_2; \dots ; c''_n \text{ endif.}$$

- Rule 3.56

  Rule 3.56 is short form for 4 rules. Consider the case of the following rule

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : set(\sigma) \\ TE[x \mapsto \sigma], C \vdash_{H} E : set(b) \end{array}}{TE, C \vdash_{H} \cup_{x \in s} E : strong\_set(b)}$$

  Let P_E be the Low SETL implementation of $v_E := E$. Then, the following is the Low SETL implementation of $\mathtt{v}' := \cup_{x \in s} E$.

```
InitSet(v');
for x ∈ s loop
    $P_E$;
    for y ∈ $v_E$ loop
        v' with:= y
    endloop
endloop
```

  The rest of the proof is straightforward, and is omitted.

- Rule 3.57

  Let us consider the case when $n = 2$, i.e. the rule is

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : set(\sigma) \\ TE[x \mapsto \sigma], C \vdash_{H} E : set(b_1 \times b_2) \end{array}}{TE, C \vdash_{H} \cup_{x \in s} E : set(b_1 \times b_2)}$$

  Let P_E be the Low SETL implementation of $v_E := E$. We present some pseudo-code below that can easily be transformed into a Low SETL implementation of $\mathtt{v}' = \cup_{x \in s} E$.

```
InitSet(v');
InitSet(temp1);    --  temp1: set((b1 x b2) x $\sigma$)
for x ∈ s loop
    $P_E$;
    for y ∈ $v_E$ loop
        temp1 with:= [y,x]    --  we can prove that [y,x] is not in temp1
    endloop
endloop;
InitSet(temp2);    --  temp2: mmap(b1-strong, (b1 x b2) x $\sigma$)
for z ∈ temp1 loop
```

```
    temp2{z[1][1]} with:= z     --  we can prove that z is not
                                --  in temp2{z[1][1]}
endloop;
for u ∈ domain(temp2) loop
    InitSet(temp3);    --  temp3: set(b2-strong)
    for w ∈ temp2{u} loop
        if w[1][2] ∉ temp3 then
            v' with:= w[1];    -- we can prove that w[1] is not in v'
            temp3 with:= w[1][2]
        endif
    endloop
endloop
```

The comments in the above pseudo-code indicate the types of the temporary variables `temp1`, `temp2`, and `temp3` that are required to prove the well-typedness of the Low SETL program. The rest of the proof is long but straightforward, and is omitted. The other types $set(b_1 \times b_2 \times \ldots \times b_n)$ where $n > 2$ can also be handled similarly, and are omitted.

- Rules 3.58-3.60

  The proofs are similar to the proofs for Rules 3.56 and 3.57, and are omitted.

- Rule Schemas 3.61-3.64 and Rules 3.65-3.67

  Consider the rule schema

$$\frac{\begin{array}{c}(\forall i = 1, \ldots, n)\ TE[(\forall j = 1, \ldots, i-1)\ x_j \mapsto t_j], C \vdash_H E_i : set(\sigma_i) \text{ or } strong\_set(b_i) \\ TE[(\forall j = 1, \ldots, n)\ x_j \mapsto t_j], C \vdash_H K : bool\end{array}}{TE, C \vdash_H \{[x_1, \ldots, x_n] : x_1 \in E_1, \ldots, x_n \in E_n \mid K\} : set(t_1 \times \ldots \times t_n)}$$

where t_j is σ_j or b_j depending on whether type of E_j is $set(\sigma_j)$ or $strong\_set(b_j)$

Let P_{E_i} denote the Low SETL implementation of the High SETL assignment $v_{E_i} := E_i$ corresponding to the type derivation of $TE[(\forall j = 1, \ldots, i-1)\ x_j \mapsto t_j], C \vdash_H E_i : set(\sigma_i)$ or $strong\_set(b_i)$. Let P_K denote the Low SETL implementation of $v_K := K$ corresponding to the type derivation of $TE[(\forall j = 1, \ldots, n)\ x_j \mapsto t_j], C \vdash_H K : bool$. Then, the Low SETL implementation of $v' := \{[x_1, \times, x_n] : x_1 \in E_1, \ldots, x_n \in E_n \mid K\}$ corresponding to the above type derivation is

```
InitSet(v');
$P_{E_1}$;
for $x_1$ ∈ $v_{E_1}$ loop
    $P_{E_2}$;
    for $x_2$ ∈ $v_{E_2}$ loop
        ⋮
             $P_{E_n}$;
             for $x_n$ ∈ $v_{E_n}$ loop
                 $P_K$;
                 if ($v_K$ == true) then
                     v' with:= [$x_1$, ..., $x_n$]
                 endif
             endloop
        ⋮
    endloop
endloop
```

Let P_1 denote the above code fragment, and let P_2 be any Low SETL program containing the code fragment P_1. The well-typedness of the above Low SETL code fragment may be proved as follows. From $TE[(\forall j = 1, \ldots, n)\ x_j \mapsto t_j], C \vdash_H K : \mathit{bool}$, it follows that

$$\exists TE'_K \supseteq TE[(\forall j = 1, \ldots, n)\ x_j \mapsto t_j] \text{ such that } TE'_K, C \vdash_{P_2} P_K.$$

Choosing $TE'_{E_n} = TE'_K \cup \{[\mathtt{v}' \mapsto \mathit{set}(t_1 \times \ldots \times t_n), (\forall j = 1, \ldots, n) x_j \mapsto t_j]\}$, it is easy to show that

$$TE'_{E_n}, C \vdash_{P_2} (P_K; \text{if } v_K == \mathit{true} \text{ then } P_E; \mathtt{v}'\ \mathit{with} := [x_1, \ldots, x_n] \text{ endif}).$$

Similarly, it can also be shown that

$$TE'_{E_n}[x_n \mapsto \bot, v_{E_n} \mapsto \mathit{set}(\sigma_n) \text{ or } \mathit{strong\_set}(b_n)], C \vdash_{P_2}$$
$$(\text{for } x_n \in v_{E_n} \text{ loop } \ldots \text{ endloop}).$$

Proceeding in this fashion, it is easy to prove that

$$\exists TE' \supseteq TE \text{ such that } TE', C \vdash_{P_2} P_1,$$

and that $TE'(\mathtt{v}') = \mathit{set}(t_1 \times \ldots \times t_n)$.

The proof for Rule Schemas 3.62-3.64 and Rules 3.65-3.67 are based on the same idea and are omitted.

- Rule Schema 3.68

  The only difference in this case from Rule Schema 3.63 is that expression E is not guaranteed to evaluate to value of a base type. Let P_E be the Low SETL implementation of $v_E := E$. Then, the implementation of $v' := \{E : x_1, \in E_1, \ldots, x_n \in E_n \mid K\}$ is the same as that in the case of Rule Schema 3.61 above except that the assignment $v' := [x_1, \times, x_n]$ inside the innermost look is replaced by the following code fragment.

```
$P_E$;
temp1 = true;
for temp2 ∈ v' loop
    if (temp2 == $v_E$) then
        temp1 = false
    endif
endloop;
if (temp1) then
    v' with:= $v_E$
endif
```

  Note that since both variables `temp2` and v_E will be of some type μ, therefore, the boolean expression `temp2` $==$ v_E will be well-typed. Moreover, v_E is added to set $\mathtt{v}'$ only after verifying that it isn't already a member. Thus, the command $\mathtt{v}'$ `with` $:= v_E$ in the new implementation is well-typed. The rest of the proof is similar to proof for the case of Rule Schema 3.61, and is omitted.

- Rules 3.69-3.73

  The proof and the Low SETL implementations corresponding to these rules are trivial, and are omitted.

- Rules 3.74-3.77

  Consider the rule

$$\frac{\begin{array}{c} TE, C \vdash_{H} s : set(\sigma) \\ TE[x \mapsto \sigma], C \vdash_{H} K : bool \end{array}}{TE, C \vdash_{H} (\exists x \in s \mid K) : bool}$$

Let P_K denote the Low SETL implementation of $v_K := K$ corresponding to the type derivation of $TE[x \mapsto \sigma], C \vdash_{H} K : bool$. The Low SETL implementation of $\mathtt{v'} := (\exists x \in s \mid K)$ corresponding to the above type derivation is

```
v' := false;
for $x$ ∈ s loop
    if (v' == false) then
        $P_K$;
        v' = $v_K$
    endif
endloop
```

The rest of the proof follows easily and is omitted. The other cases corresponding to Rules 3.75, 3.76, and 3.77 are similar, and are also omitted.

- Rules 3.78-3.82

  This case of Rule 3.78 is similar to that of Rule 3.55 and is omitted. Similarly, the case of Rule 3.82 is similar to that of Rule 3.54, and is also omitted. Rules 3.79, 3.80, and 3.81 are also simple, and omitted.

□

3.6 Type System for High SETL Commands

In this section we give type rules for High SETL commands. We use type inference rules of the form $TE, C \vdash_{H} P$, where P is a sequence of one or more High SETL commands. A High SETL command sequence P is said to well-typed in a type environment TE under the set of subtype constraints C if there exists a type derivation for $TE, C \vdash_{H} P$. We define Low SETL implementations for well-typed High SETL programs, and prove that well-typed High SETL programs lead to well-typed Low SETL programs.

$$\frac{\begin{array}{c} TE, C \vdash_{H} E : \tau \\ TE(v) = \tau \end{array}}{TE, C \vdash_{H} v := E} \tag{3.83}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} E : \sigma_2 \\ TE(v_1) = \mathit{strong\_smap}(b_1, \sigma_2), TE(v_2) = b_1 \end{array}}{TE, C \vdash_{H} v_1(v_2) := E} \tag{3.84}$$

$$\frac{\begin{array}{c} TE, C \vdash_{H} E : set(\sigma_2) \\ TE(v_1) = \mathit{strong\_mmap}(b_1, \sigma_2), TE(v_2) = b_1 \end{array}}{TE, C \vdash_{H} v_1\{v_2\} := E} \tag{3.85}$$

$$\frac{\begin{array}{c}TE, C \vdash_{H} E : \sigma_i \\ TE(v) = \sigma_1 \times \sigma_2 \times \ldots \times \sigma_n\end{array}}{TE, C \vdash_{H} v[i] := E} \tag{3.86}$$
where i is an integer literal between 1 and n

$$\frac{\begin{array}{c}TE, C \vdash_{H} E : b \\ TE(v) = \mathit{strong\_set}(b)\end{array}}{TE, C \vdash_{H} v\ \mathit{with} := E} \tag{3.87}$$

$$\frac{\begin{array}{c}TE, C \vdash_{H} E : b \\ TE(v) = \mathit{strong\_set}(b)\end{array}}{TE, C \vdash_{H} v\ \mathit{less} := E} \tag{3.88}$$

$$\frac{\begin{array}{c}TE, C \vdash_{H} E : \mathit{set}(b) \text{ or } \mathit{strong\_set}(b) \\ TE(v) = \mathit{strong\_set}(b)\end{array}}{TE, C \vdash_{H} v \cup := E} \tag{3.89}$$

$$\frac{\begin{array}{c}TE, C \vdash_{H} E : \mathit{set}(b) \text{ or } \mathit{strong\_set}(b) \\ TE(v) = \mathit{set}(b) \text{ or } \mathit{strong\_set}(b)\end{array}}{TE, C \vdash_{H} v \cap := E} \tag{3.90}$$

$$\frac{\begin{array}{c}TE, C \vdash_{H} E : \mathit{set}(b) \text{ or } \mathit{strong\_set}(b) \\ TE(v) = \mathit{strong\_set}(b)\end{array}}{TE, C \vdash_{H} v - := E} \tag{3.91}$$

Rules 3.89-3.91 are all short forms for two rules each.

$$\frac{\begin{array}{c}TE, C \vdash_{H} E : \mathit{set}(\sigma) \\ TE(v) = \mathit{set}(\sigma)\end{array}}{TE, C \vdash_{H} v \uplus := E} \tag{3.92}$$

$$\frac{\begin{array}{c}TE, C \vdash_{H} E : \mathit{smap}(b_1, \sigma_2) \text{ or } \mathit{strong\_smap}(b_1, \sigma_2) \\ TE(v) = \mathit{strong\_smap}(b_1, \sigma_2)\end{array}}{TE, C \vdash_{H} v\ / := E} \tag{3.93}$$

$$\frac{\begin{array}{c}TE, C \vdash_{H} E : \mathit{mmap}(b_1, \sigma_2) \text{ or } \mathit{strong\_mmap}(b_1, \sigma_2) \\ TE(v) = \mathit{strong\_mmap}(b_1, \sigma_2)\end{array}}{TE, C \vdash_{H} v\ / := E} \tag{3.94}$$

Rule 3.93 and 3.94 are both short forms for two rules each.

$$\frac{TE(v) = \mathit{set}(\sigma) \text{ or } \mathit{strong\_set}(b)}{TE, C \vdash_{H} \mathit{InitSet}(v)} \tag{3.95}$$

Rule 3.95 is short form for two rules.

$$\frac{TE(v) = \mathit{smap}(\sigma_1, \sigma_2) \text{ or } \mathit{strong\_smap}(b_1, \sigma_2) \text{ or } \mathit{mmap}(\sigma_1, \sigma_2) \text{ or } \mathit{strong\_mmap}(b_1, \sigma_2)}{TE, C \vdash_H \mathit{InitMap}(v)} \tag{3.96}$$

Rule 3.96 is short form for four rules.

$$\begin{array}{c}\dfrac{TE(v) = \sigma_1 \times \sigma_2 \times \ldots \times \sigma_i}{TE, C \vdash_H \mathit{InitTuple}(v, i)} \\ \text{where } i \text{ is some integer literal greater than 1}\end{array} \tag{3.97}$$

$$\frac{\begin{array}{c}TE, C \vdash_H K : \mathit{bool} \\ TE, C \vdash_H P_1 \\ TE, C \vdash_H P_2\end{array}}{TE, C \vdash_H \text{if } K \text{ then } P_1 \text{ else } P_2 \text{ endif}} \tag{3.98}$$

$$\frac{\begin{array}{c}TE, C \vdash_H K : \mathit{bool} \\ TE, C \vdash_H P\end{array}}{TE, C \vdash_H \text{while } K \text{ loop } P \text{ endloop}} \tag{3.99}$$

$$\frac{TE(v_2) = \mathit{set}(\sigma), TE[v_1 \mapsto \sigma], C \vdash_H P}{TE, C \vdash_H \text{for } v_1 \in v_2 \text{ loop } P \text{ endloop}} \tag{3.100}$$

$$\frac{TE(v_2) = \mathit{strong\_set}(b), TE[v_1 \mapsto b], C \vdash_H P}{TE, C \vdash_H \text{for } v_1 \in v_2 \text{ loop } P \text{ endloop}} \tag{3.101}$$

$$\frac{TE(v_2) = \mathit{smap}(\sigma_1, \sigma_2), TE[v_1 \mapsto \sigma_1], C \vdash_H P}{TE, C \vdash_H \text{for } v_1 \in \mathit{domain}(v_2) \text{ loop } P \text{ endloop}} \tag{3.102}$$

$$\frac{TE(v_2) = \mathit{strong\_smap}(b_1, \sigma_2), TE[v_1 \mapsto b_1], C \vdash_H P}{TE, C \vdash_H \text{for } v_1 \in \mathit{domain}(v_2) \text{ loop } P \text{ endloop}} \tag{3.103}$$

$$\frac{TE(v_2) = \mathit{mmap}(\sigma_1, \sigma_2), TE[v_1 \mapsto \sigma_1], C \vdash_H P}{TE, C \vdash_H \text{for } v_1 \in \mathit{domain}(v_2) \text{ loop } P \text{ endloop}} \tag{3.104}$$

$$\frac{TE(v_2) = \mathit{strong\_mmap}(b_1, \sigma_2), TE[v_1 \mapsto b_1], C \vdash_H P}{TE, C \vdash_H \text{for } v_1 \in \mathit{domain}(v_2) \text{ loop } P \text{ endloop}} \tag{3.105}$$

$$\frac{\begin{array}{c}TE, C \vdash_H c \\ TE, C \vdash_H P\end{array}}{TE, C \vdash_H c; P} \tag{3.106}$$

3.7 Low SETL Implementation of High SETL Programs

In this section we define Low SETL implementations for High SETL commands. Moreover, we also prove that well-typed High SETL programs are translated into well-typed Low SETL programs. Once again, we use rule induction (or induction on the type derivation for the command sequence) for both

1. defining Low SETL implementations for High SETL command sequences, and

2. for proving that the generated Low SETL programs are well-typed.

Theorem 3.7.1 *Let P be a High SETL command sequence such that $TE, C \vdash_H P$. Let P_1 be the Low SETL code fragment implementing program P. Let P_1 be the sequence of commands $c_1; c_2; \ldots c_n$. Let P_2 be any Low SETL program containing the command sequence P_1 as a sub-program. Assume that the disjointness constraints imposed by the appearance of operator $\uplus$ in program P are satisfied in all execution instances of program P_2. Then,*

$$\exists TE' \supseteq TE \text{ such that } TE', C \vdash_{P_2} c_i \text{ for all } i = 1, \ldots, n.$$

In particular, if P_2 is the same as P_1, then $TE', C \vdash_{P_1} P_1$, i.e. the Low SETL program P_1 implementing the well-typed High SETL program P is also well-typed.

Moreover, $domain(TE') - domain(TE)$ contains only newly generated temporary variables, and the bound variables appearing in the High SETL expressions in command sequence P.

Proof Idea: The proof follows by a very simple rule induction. We just show a few sample Low SETL implementations of High SETL commands. Consider Rule 3.83

$$\frac{\begin{array}{c} TE, C \vdash_H E : \tau \\ TE(v) = \tau \end{array}}{TE, C \vdash_H v := E}$$

Let P_E be the Low SETL implementation of $v_E := E$ corresponding the the type derivation of $TE, C \vdash_H E : \tau$. Then, the Low SETL implementation of the High SETL command $v := E$ is

```
$P_E$;
v := $v_E$
```

Similarly, consider Rule 3.89

$$\frac{\begin{array}{c} TE, C \vdash_H E : set(b) \\ TE, C \vdash_H v : strong\_set(b) \end{array}}{TE, C \vdash_H v \cup := E}$$

Let P_E be the Low SETL implementation of $v_E := E$ corresponding the the type derivation of $TE, C \vdash_H E : set(b)$. Then the Low SETL implementation of the High SETL command $v \cup := E$ is

```
$P_E$;
for $x$ $\in$ $v_E$ loop
    v with:= x;
endloop
```

The proof of well-typedness of the Low SETL implementation P_1 in the context of an arbitrary Low SETL program P_2 is simple, and is omitted. □

3.8 Time Complexity

The well-typedness of a Low SETL program guarantees that all of its primitive set-theoretic operations, including operations involving associative access, can be performed in worst-case $O(1)$ time on a pointer machine without the use of hashing. This $O(1)$ time implementation guarantee for primitive operations may be useful for computing the time-complexity of a Low SETL program, but is not sufficient to guarantee that the time-complexity of all well-typed Low SETL programs can be computed. In fact, it is easy to prove that well-typed Low SETL is *Turing Complete*, and therefore, even the termination of arbitrary Low SETL programs is undecidable. The well-typedness of Low SETL programs simply allows us to use traditional methods of algorithmic analysis, such as those used for programming languages like Pascal, C, etc. to compute the worst-case time-complexities.

High SETL, being a superset of Low SETL, is also Turing-Complete. Therefore, in general, it is not possible to determine the time complexity of an arbitrary High SETL program. However, the functional subset of High SETL is a subset of the language or primitive recursive functions [31]. In this section we show how to systematically estimate the time complexity of evaluation of the functional subset of High SETL. As a result, the task of manually computing the time complexity of High SETL programs is considerably simpler than computing the time complexity of Low SETL programs by traditional bean counting arguments. In Chapter 4 we will look at a database query optimization algorithm in which queries in a subset of the relational calculus [27] (called RCS) formulated as abstract but inefficient High SETL expressions will be transformed into equivalent High SETL expressions that can be computed in time linear in the sum of the input and output sizes of the database queries. The ability to systematically analyze High SETL expressions for time complexity turns out to be extremely useful for proving the linear time result for the database query language RCS.

In Section 3.8.1 we show how to compute worst-case time complexities for arbitrary High SETL expressions, and in Section 3.8.2, we show how to compute time complexities for High SETL command sequences that do not contain any while loops. In order to compute time complexities of arbitrary High SETL programs, we rely on traditional methods of algorithmic analysis to estimate bounds on the number of iterations of each while loop, and use these bounds to compute the time complexity of the whole program.

3.8.1 Time Complexity of High SETL Expressions

The time complexity of implementation of a well-typed High SETL expression depends on the type derivation of the expression. Since different type derivations of High SETL expressions may correspond to different Low SETL implementations, different type derivations also correspond to different time complexities. Once again we use rule induction (or induction on the length of the type derivation) to define a method for computing the time-complexity of an arbitrary well-typed High SETL expression E.

- Expressions v, $\ni v$, $v_1(v_2)$, $v\{v_2\}$, $v_1[i]$, om derived using Rules 3.4-3.10: In each of these cases, expression E is itself a Low SETL expression, and can be computed in $O(1)$ time.

- Expression $s \cup t$ derived using Rule 3.11 or Rule 3.12: It is easy to show that the expression $s \cup t$ can be computed in $O(\#s + \#t)$ time.

- Expression $s \cap t$ derived using Rule 3.13 or Rule 3.14: As mentioned earlier, Rule 3.13 is short form for eight distinct rules. For those rules in which set s is strongly based, the expression $s \cap t$ can be evaluated in $O(\#t)$ time. Similarly, if set t is strongly based, the evaluation takes $O(\#s)$ time. If both s and t are not strongly based, then the evaluation takes $O(\#s + \#t)$ time.

  In the case of Rule 3.14 the time complexity of computing $s \cap t$ is $O(\#s + \#t)$.

- Expression $s - t$ derived using Rule 3.15 or Rule 3.16: In the case of Rule 3.15, it is easy to verify that depending on whether set t is strongly based or not in the actual rule applied, the evaluation of expression $s - t$ takes either $O(\#s)$ time, or $O(\#s + \#t)$ time.

  In the case of Rule 3.16 the time complexity of computing $s - t$ is $O(\#s + \#t)$.

- Expression $s \times t$ derived using Rules 3.17-3.20: In each case, expression $s \times t$ can be computed in time $O(\#s \times \#t)$ time.
- Expression $\#s$ derived using Rule 3.21: $O(\#s)$ time.
- Expression $s \uplus t$ derived using Rules 3.22-3.23: $O(\#s + \#t)$ time.
- Expression $\mathit{ToSet}(f)$ derived using Rules 3.24-3.27: $O(\#f)$ time.
- Expression $\mathit{ToMap}(s)$ derived using Rules 3.28: $O(\#s)$ time.
- Expression $\mathit{domain}(f)$ derived using Rules 3.29-3.32: $O(\#\mathit{domain}(f))$ time.
- Expression $\mathit{range}(f)$ derived using Rules 3.33-3.34: $O(\#f)$ time.
- Expression $f \circ g$ derived using Rules 3.35-3.38: In each of these cases the time complexity is $O(\#g)$ if map f is strongly based, and $O(\#f + \#g)$ otherwise.
- Expression $f \circ g$ derived using Rule 3.39-3.40: In each of these cases the time complexity is $O(\#g + \#f \circ g)$ if map f is strongly based, and $O(\#g + \#\mathit{domain}(f) + \#f \circ g)$ otherwise.
- Expression $f \circ g$ derived using Rule 3.41-3.42: In each of the cases the time complexity is $O(\#g + (\#f \times \#g))$ if map f is strongly based, and $O(\#g + \#f + (\#f \times \#g))$ otherwise. However, if one of the maps f or g is one-one, then the time complexity is $O(\#g + \#f \circ g)$ if map f is strongly based, and $O(\#g + \#f + \#f \circ g)$ otherwise.
- Expressions $f|_s$ and $f[s]$ derived using Rules 3.43-3.46: In each of these cases the time complexity is $O(\#f)$ if set s is strongly based, and $O(\#f + \#s)$ otherwise.
- Expression f^{-1} derived using Rules 3.47-3.50: $O(\#f)$ time.
- Expression f/g derived using Rules 3.51-3.52: $O(\#f + \#g)$ time.
- Expression E derived using Rule 3.53: The derivation for $TE, C \vdash_H E : \sigma$ contains a sub-derivation for $TE, C \vdash_H E : b$. If we use $\mathit{Cost}(E_b)$ to denote the time complexity of computing expression E of type b, then the cost of computing expression E of type σ is $\mathit{Cost}(E_b) + O(1)$.
- Expression "Let $v = E_1$ in E_2" derived using Rule 3.54: Let $\mathit{Cost}(E_1)$ and $\mathit{Cost}(E_2)$ denote the cost of computing expressions E_1 and E_2 respectively. Then the cost of computing expression "Let $v = E_1$ in E_2" is $\mathit{Cost}(E_1) + \mathit{Cost}(E_2)[v \mapsto E_1]$, where $\mathit{Cost}(E_2)[v \mapsto E_1]$ is the expression $\mathit{Cost}(E_2)$ with all occurrences of v replaced by E_1.
- Expression "if K then E_1 else E_2 endif" derived using Rule 3.55: Let $\mathit{Cost}(K)$, $\mathit{Cost}(E_1)$, and $\mathit{Cost}(E_2)$ denote the costs of computing expressions K, E_1, and E_2 respectively. The cost of computing expression "if K then E_1 else E_2 endif" is $\mathit{Cost}(K) + \mathit{max}(\mathit{Cost}(E_1), \mathit{Cost}(E_2))$.
- Expressions $\cup_{x \in s} E$, $\cap_{x \in s} E$, or $\uplus_{x \in s} E$ derived using Rules 3.56-3.60: Let the cost of computing expression E be denoted by $\mathit{Cost}(E)$. Then, the cost of computing the expressions $\cup_{x \in s} E$, $\cap_{x \in s} E$, or $\uplus_{x \in s} E$ is $\Sigma_{x \in s}(\mathit{Cost}(E) + \#E)$.
- Expression $\{[x_1, \ldots, x_n] : x_1 \in E_1, \ldots, x_n \in E_n \mid K\}$ derived using Rule Schema 3.61: Let $\mathit{Cost}(E_i)$, and $\mathit{Cost}(K)$ denote the cost of computing expression E_i (for $i = 1, \ldots, n$), and expression K. The cost of computing expression $\{[x_1, \ldots, x_k] : x_1 \in E_1, \ldots, x_n \in E_n \mid K\}$ is defined by the following recursive equations.

  If $n = 1$, then the Cost of computing expression $\{[x_1] : x_1 \in E_1 \mid K\}$ is

$$\mathit{Cost}(E_1) + \Sigma_{x_1 \in E_1}(\mathit{Cost}(K) + O(1)).$$

For $n > 1$, the cost of computing expression $\{[x_1, \ldots, x_n] : x_1 \in E_1, \ldots, x_n \in E_n \mid K\}$ is

$$Cost(E_1) + \Sigma_{x_1 \in E_1} Cost(\{[x_2, \ldots, x_n] : x_2 \in E_2, \ldots, x_n \in E_n \mid K\}),$$

where $Cost(\{[x_2, \ldots, x_n] : x_2 \in E_2, \ldots, x_n \in E_n \mid K\})$ is the cost of computing expression $\{[x_2, \ldots, x_n] : x_2 \in E_2, \ldots, x_n \in E_n \mid K\}$.

The cost of expressions derived from Rule Schemas 3.62-3.64 and Rules 3.65-3.67 can be computed similarly.

- Expression $\{E : x_1 \in E_1, \ldots, x_n \in E_n \mid K\}$ derived using Rule Schema 3.68: Let *Output* denote the final value of the expression $\{E : x_1 \in E_1, \ldots, x_n \in E_n \mid K\}$. Then the cost of computing this expression is the cost computed by the recursive equations in the previous case along with an additional factor of $\#Output$.

- Expressions $s \in t$, $s == t$, $IsEmptySet(s)$, $IsEmptyMap(f)$ derived using Rules 3.69-3.73: $O(1)$ time

- Expressions $(\exists x \in S \mid K)$ and $(\forall x \in s \mid K)$ derived using Rules 3.74-3.77: Let $Cost(K)$ denote the time complexity of computing expression K. Then the time complexity of computing $(\exists x \in S \mid K)$ or $(\forall x \in s \mid K)$ is $O(\#s) \times Cost(K)$.

  The complexities corresponding to Rules 3.78-3.82 can be computed in a similar fashion, and are omitted.

3.8.2 Time Complexity of Some High SETL Commands

In Section 3.8.1 we saw that the evaluation of every High SETL expression is guaranteed to terminate, and that the time-complexity of evaluation can be accurately computed. The evaluation of all High SETL programs that do not contain while loops, is also guaranteed to terminate, and their time-complexities can also be accurately estimated.

Let P denote a well-typed High SETL command sequence not containing any while loops. Once again, we use rule induction (or induction on the length of the type derivation) to define a method for computing the time-complexity of command sequence P.

- Commands $v := E$, $v_1(v_2) := E$, $v_1\{v_2\} := E$, $v[i] := E$, $v\ with := E$, $v\ less := E$ derived using Rules 3.83-3.88: Let $Cost(E)$ denote the time complexity of computing Expression E. Then, the time complexity of executing the above commands is $Cost(E) + O(1)$ time.

- Command $v \cup := E$ derived using Rule 3.89: $Cost(E) + O(\#E)$ time.

- Command $v \cap := E$ derived using Rule 3.90: $Cost(E) + O(\#v + \#E)$ time.

- Command $v - := E$ derived using Rule 3.91: $Cost(E) + O(\#E)$ time.

- Command $v \uplus := E$ derived using Rule 3.92: $Cost(E) + O(\#E)$ time.

- Command $v\ / := E$ derived using Rules 3.93-3.94: $Cost(E) + O(\#E)$ time.

- Command $InitSet(v)$ or $InitMap(v)$ derived using Rules 3.95-3.96: If the set or map is strongly based, the time complexity is $O(\#v)$ or $O(\#domain(v))$ for sets and maps respectively, and $O(1)$ otherwise.

- Command $InitTuple(v, i)$ derived using Rule 3.97: $O(1)$ time.

- Command "if K then P_1 else P_2 endif" derived using Rule 3.98 where P_1 and P_2 do not contain any while loops: Let $Cost(k)$, $Cost(P_1)$, and $Cost(P_2)$ denote the costs of computing expression K, and executing P_1 and P_2 respectively. Then, the time complexity of executing "if K then P_1 else P_2 endif" is $Cost(K) + max(Cost(P_1), Cost(P_2))$ time.

- Command "for $v_1 \in v_2$ loop P endloop" derived using Rules 3.100-3.101 where P does not contain any while loop: $\Sigma_{v_1 \in v_2} Cost(P)$ time, where $Cost(P)$ denotes the cost of executing P. Alternately, the complexity can be taken to be $O(\#v_2) \times Cost(P)$ time, where $Cost(P)$ denotes the maximum cost of executing P for any element v_1 in set v_2.

- Command "for $v_1 \in domain(v_2)$ loop P endloop" derived using Rules 3.102-3.105 where P does not contain any while loop: $\Sigma_{v_1 \in domain(v_2)} Cost(P)$, or $O(\#domain(v_2)) \times Cost(P)$ time.

Chapter 4

Speedup of Linear Time Fragment of Willards Relational Calculus Subset

4.1 Introduction

In the previous chapters we defined two algorithm specification languages Low SETL and High SETL. We proved that well-typed Low SETL programs can be translated into pointer machine ([57, pages 462-463], [105, 83, 106, 7]) implementations in which each associative access operation is implemented in $O(1)$ time. We also defined a translation from well-typed High SETL to well-typed Low SETL, and showed how to compute the time complexity of the functional subset of High SETL in a systematic manner. In this chapter we show how the use of High SETL as an algorithm specification language can lead to an improved database query optimization algorithm.

The problem is to compile a subset RCS of Relational Calculus defined by Willard ([110]) in a novel way so that efficient run-time query performance is guaranteed. Willard gives an algorithm to compile each query q belonging to RCS so that it executes in $O(n \log^d n + o)$ steps and $O(n)$ space, where n and o are respectively the input and output set sizes, and d is a parameter associated with the syntax of query q. Willard's time bounds are based on the assumption that hashing unit-space data takes unit time.

In this chapter we show how queries in the expected linear-time fragment of Willard's RCS (which we call LRCS), can be implemented in worst-case linear time. We first show that each LRCS query can be expressed as a well-typed High SETL expression. The direct Low SETL implementations of these well-typed High SETL specifications correspond to naive and inefficient implementations of LRCS queries. The time complexity of execution of such naive implementations is typically a high degree polynomial in the size of the input. We show how to transform these naive High SETL specifications into semantically-equivalent, but more efficiently computable High SETL expressions whose worst-case time complexity is linear in the sum of the sizes of the input and output sizes. The benefits of using High SETL expressions as LRCS specifications are two-fold. Firstly, the complexity analysis of High SETL expressions is simplified because of the use of more algebraic reasoning rather than low level counting arguments. Secondly, the use of High SETL allows the transformation process to be guided by efficiency considerations, i.e. in each step of the transformation process, the LRCS query is transformed into a semantically equivalent, but more efficiently computable High SETL expression. Thus, we get shorter and simpler proofs of correctness, and analysis of time complexity.

The improvement to Willard's result comes from the use of the High SETL type system. The fact that the final High SETL implementation of each LRCS query is well-typed guarantees that each hash operation used to implement associative access operations in Willard's RCS can be simulated in worst-case $O(1)$ time on a pointer machine. This improves Willard's expected linear time result to worst-case linear time, and demonstrates how the type system can be used as a tool to obtain an algorithmic speedup.

The results in this chapter were first presented at the IFIP TC2 conference on Algorithmic Languages and Calculi [43].

4.1.1 Background

The work presented here follows a long-term investigation of Willard's Relational Calculus Subset (RCS), a database query language proposed in his thesis [110], and developed further in [111] and [112], a part of which appeared in JCSS [113]. Under the assumption that hashing unit-space data takes unit time, Willard defined broad subclasses of Relational Calculus queries that could be executed in linear space and low expected time $O(n \log^d n + o)$, where n and o are the input and output set sizes, and d is a parameter associated with the syntax of the queries.

Our interest in Willard's fascinating work is motivated by a number of issues. RCS is a rare comprehensive investigation of query translation for the Relational Calculus [27]. Willard's work is one of the earliest investigations that link a substantial language to low order run-time query complexity for a main-memory model. Willard's investigation is also one of the earliest nontrivial examples of output-sensitive algorithmic analysis. Although other languages that are bound to low order run-time complexities (e.g., [54] , [5], [22] , and [4]) may have greater expressive power, RCS is unusual in its richness and its run-time requirements.

4.1.2 Overview

We make two contributions that integrate solutions to algorithmic and software problems. First we detail how to implement LRCS i.e. Willard's linear-time fragment of RCS, using semantics-preserving source program transformations. Second, we show how type theory can be used to improve the run-time performance of LRCS from linear expected time to linear worst-case time.

A problem shared by the database and algorithmics communities is the lack of notation and notational calculi to formally map perspicuous problem specifications into efficient implementations. Our approach to obtain an implementation of LRCS is to use High SETL, a wide spectrum set-theoretic language capable of specifying both high-level user queries and their efficient (but still abstract) implementations. Program transformations that preserve familiar set-theoretic semantics are used to derive efficient implementations from high-level queries.

By itself Willard's compilation of RCS queries is a model of sophistication that works in two phases. The first phase, called the decomposition phase, transforms a query into a sequence of efficient lower level queries. The second phase, which we call the back-end implementation, transforms each efficient query into a sequence of element-at-a-time operations (e.g., loads, stores, retrievals), many of which require hashing.

We abide by most of Willard's ideas in the first phase, and contribute an abstract implementation in the more convenient set-theoretic language High SETL. It is in the back-end implementation where we use well-typed High SETL expressions to get efficient implementations without the use of hashing. The abstraction provided by High SETL helps reuse redundant low level constructions found in Willard's proofs, thereby giving a more perspicuous implementation. Moreover, we get a simple time complexity analysis that replaces Willard's repeated low level counting arguments by simpler logical and algebraic reasoning.

We go on to show that Willard's time bound (for run-time query execution) in the expected case can be achieved in the worst case without degrading space utilization. This is achieved by demonstrating that the translated form of each query is a well-typed High SETL program. The well-typedness of the final High SETL implementation of the LRCS query guarantees the real-time simulation of primitive set operations on a pointer machine, which means that each primitive set operation (*e.g.* membership testing $x \in S$) can be implemented in unit worst-case time on a pointer machine. The time complexity analysis of the generated High SETL program immediately reveals that each query in LRCS can be translated into pointer machine code that runs in linear worst-case time.

4.2 Definition of LRCS

LRCS includes two kinds of queries, namely *count-queries*

$$Q \;=\; \{\; [x_1, \#\{x_2 \in X_2 \mid e(x_1, x_2)\}] : \; x_1 \in X_1\}, \tag{4.1}$$

which maps each element $x_1 \in X_1$ into the number of elements $x_2 \in X_2$ that satisfy predicate $e(x_1, x_2)$, and *find-queries*

$$\begin{aligned} Q \;=\;& \{\, [x_1, \ldots\, x_n] \in \times_{p=1}^{n} X_p \;|\Theta_{n+1} x_{n+1} \in X_{n+1}, \ldots, \Theta_m x_m \in X_m \\ & e(x_1, \ldots, x_m) \,\}, \end{aligned} \tag{4.2}$$

which evaluates a subset of an n-way cross product of sets $X_1, \ldots, X_n$, qualified by an arbitrary number $(m - n)$ of leading bounded quantifiers $\Theta_{n+1} \ldots \Theta_m$, and satisfying the predicate $e(x_1, \ldots, x_m)$. The predicates $e(x_1, x_2)$ and $e(x_1, \ldots, x_m)$ are formed from atomic predicates (to be defined later) and logical connectives *or*, *and*, and *not*. Willard's result is surprising as he shows that a very broad class of these queries that satisfy an acyclicity condition (also to be described later) can be efficiently computed without actually evaluating the cross product, in time proportional to just the sum of the cardinalities of the input sets and the final output set.

The sets X_1 and X_2 in Query 4.1 and sets $X_1, \ldots, X_m$ in Query 4.2 are inputs to the query. We assume that the query is preceded by a *read* statement which reads in the input. This allows us to assume without loss of generality that each set X_i contains elements of some base type b_i. If the input values corresponding to the elements of set X_1 are of type σ_1, then we add the subtype constraint $b_1 < \sigma_1$ to the set of subtype constraints C, and assume that X_1 is of type *set*(b_1). If there are two (or more) subtype constraints $b_i < \sigma$, and $b_j < \sigma$ for some type σ, then we eliminate the second constraint $b_j < \sigma$ and replace base type b_j with base type b_i.

The variables $x_1, \ldots, x_m$ are called the free variables of predicate e within the Queries 4.1 and 4.2. Predicate e may be an arbitrary collection of the following atomic predicates and their negations connected by the boolean connectives $\wedge$, and $\vee$

1. Boolean constants *true* and *false*.

2. Equality and inequality predicates (called *joins* and *anti-joins*) *e.g.* $f_1(x) = f_2(y)$, $f_1(x) \neq f_2(y)$, where x and y are free variables of predicate e. For each such join or anti-join we assume that expressions $f_1(x)$ and $f_2(x)$ are of some base type b.

3. Unary Comparison Predicates *e.g.* $f_1(x) = c$, $f_1(x) \neq c$, $f_1(x) < c$, $f_1(x) = f_2(x)$ and $f_1(x) < f_2(x)$, where x is a free variable of predicate e and c is any constant. For unary predicates involving the constant c, we assume that expression $f_1(x)$ is of some base type b satisfying the subtype constrain $b < \sigma$, where constant c is type σ. Moreover, we assume that the operations $=$, $\neq$, $<$ etc. can be performed in $O(1)$ time for elements of type σ (for example, σ may be *int*). Similarly, for the unary comparison predicate $f_1(x) = f_2(x)$, we assume that $f_1(x)$ and $f_2(x)$ are both of some base type b. For the predicate $f_1(x) < f_2(x)$, we additionally assume that base type b satisfies a subtype constraint $b < \sigma$ where elements of type σ may be compared using the operator $<$ in $O(1)$ time.

4. Unary List Predicates *e.g.* $\exists y \in Y\ (f_1(y) = f_2(x))$ and $\forall y \in Y\ (f_1(y) = f_2(x))$, where x is a free variable of predicate e. We assume that set Y is of type *set*(b) for some base type b, and expressions $f_1(x)$ and $f_2(x)$ are also of some base type b' (which may or may not be the same as base type b).

5. Tabular Predicates *e.g.* $\exists z \in Z\ (f_1(x) = f_2(z)\ \wedge\ f_3(y) = f_4(z))$, where x and y are free variables of predicate e. Again, we assume that set Z is of type *set*(b) for some base type b, and expressions $f_1(x)$ and $f_2(z)$ are of some base type b' (which may or may not be the same as base type b), and $f_3(y)$ and $f_4(z)$ of some base type b'' (which may or may not be the same as b' or b). Furthermore, all such expressions $f_i(w)$ appearing in tabular predicates are assumed to be *few-to-one i.e.* the sizes of the pre-images of f_i are uniformly bounded by a constant.

We also assume that each expression $f_i(w)$ that appears in an atomic predicate can be evaluated in $O(1)$ time.

The input to Query 4.1 are the sets X_1, X_2 and the sets appearing in the unary list or tabular predicates appearing in predicate $e(x_1, x_2)$. Similarly, the input to Query 4.2 are the sets $X_1, \ldots, X_m$ and the sets appearing in unary list or tabular predicates in appearing in predicate $e(x_1, \ldots, x_m)$. Assume that type environment TE maps each such input set X_i to its type $set(b_i)$ for some base type b_i, and the set of subtype constraints C contains the subtype constraints corresponding to these base types. Then, it is easily verified that each predicate $e(x_1, x_2)$ or $e(x_1, \ldots x_m)$ is well-typed i.e.

$$TE, C \vdash_H e(x_1, x_2) : bool \text{ and } TE, C \vdash_H e(x_1, \ldots, x_m) : bool$$

The time complexity of evaluating the equality, inequality, and unary comparison predicates is $O(1)$ time, and the time complexity of evaluating unary list predicates such as $\exists y \in Y\ (f(x) = g(y))$ is $O(\#Y)$, and tabular predicates such as $\exists z \in Z\ (f(z) = g(x)\ \wedge\ f'(z) = g'(y))$ is $O(\#Z)$. Thus, the time complexity of evaluation of predicate $e(x_1, \ldots, x_n)$ is linear in the sum of the sizes of sets that appear in the unary list and tabular predicates in e. Let t_e denote the time complexity of evaluation of predicate e.

It is easy to verify that the expression on the right hand side of Query 4.1 is a well-typed High SETL expression of type $strong\_smap(b_1, int)$, i.e.

$$TE, C \vdash_H \{\ [x_1, \#\{x_2 \in X_2 \mid e(x_1, x_2)\}] :\ x_1 \in X_1\} : strong\_smap(b_1, int) \tag{4.3}$$

and that the expression on the right hand side of Query 4.2 is a well-typed High SETL expression of type $set(b_1 \times b_2 \times \ldots b_n)$, i.e.

$$\begin{aligned} TE, C \vdash_H \quad & \{\ [x_1, \ldots\ x_n] \in \times_{p=1}^{n} X_p \mid \Theta_{n+1} x_{n+1} \in X_{n+1}, \ldots, \Theta_m x_m \in X_m \\ & e(x_1, \ldots, x_m)\ \} : set(b_1 \times b_2 \times \ldots b_n) \end{aligned} \tag{4.4}$$

All count-queries of the form of Query 4.1 are valid LRCS queries. However, we restrict the class of LRCS queries of the form of Query 4.2 in the following way. We define the query graph for Find-query (4.2) as having variables $x_1, \ldots, x_m$ as vertices, and having an edge from x_k to x_l *iff* there is a two-variable atomic predicate on x_k and x_l in the expression $e(x_1, \ldots, x_m)$, where x_k appears to the *left* of x_l in query (4.2). Find-query (4.2) is in LRCS *iff* the query graph for the query is a forest. In other words the in-degree of every vertex should be no more than 1.

The time complexity of a naive implementation of Query 4.1 is $O(\#X_1 \times \#X_2 \times t_e)$, and that of Query 4.2 is $O(\#X_1 \times \#X_2 \times \ldots \#X_m \times t_e)$, where t_e denotes the time complexity of evaluation of the predicate e. In this chapter we will show that these queries can be transformed into implementations that run in time linear in the sum of the input and output sizes.

4.3 Linear-Time Implementation of LRCS Queries

We will now prove that all LRCS *count-queries* and *find-queries* (satisfying the condition that the query graph is a forest) can be implemented to run in linear time (*i.e. linear in the sum of the sizes of their input and output sets*).

In order to do this, we shall first show how to decompose the general find-query (4.2) into a finite number of count-queries and a union $\cup_{1=1}^{t} Q_i$ where t depends on the predicate $e(x_1, \ldots, x_m)$ but is independent of the input sets X_1, X_2, etc. , and each query Q_i is another find-query with no leading quantifiers, and whose predicate is just a conjunction of atomic predicates and their negations. Next we turn each such find-query Q_i into a nest of joins, $\bowtie_{j=1}^{q_i} P_{ij}$ where q_i is once again a constant independent of the input, and each query P_{ij} is of the form

$$\{[x_1, x_2] \in X_1 \times X_2 \mid e(x_1, x_2)\ \}, \tag{4.5}$$

where predicate $e(x_1, x_2)$ is just a conjunction of atomic predicates and their negations. We shall call queries such as (4.5) *simple find-queries.* Willard's decomposition ensures that the sizes of the intermediate outputs P_{ij} are no bigger than the sizes of the outputs Q_i, which is very important for obtaining the linear-time result. This part constitutes the front-end or the decomposition phase of Willard's algorithm.

In the decomposition phase, we have adhered to the essentials of Willard's decomposition steps but have simplified his arguments and fleshed out details for an actual implementation. In the back-end phase however, we make a more significant contribution. We give semantics-preserving transformations that transform *count-queries* and *simple find-queries* into simpler queries that are shown to be well-typed and hence, can be implemented in worst-case linear time on a pointer machine.

4.3.1 Quantifier Elimination

Let Q denote the general find-query

$$\begin{array}{l}\{\ [x_1, x_2, \ldots, x_n]\ \in \times_{p=1}^{n} X_p \mid \Theta_{n+1} x_{n+1} \in X_{n+1}, \ldots, \Theta_m x_m \in X_m \\ \quad e(x_1, \ldots, x_m)\ \}.\end{array} \tag{4.6}$$

Assume that Query Q is a valid LRCS query (i.e. the query graph of Q is a forest) and is also a well-typed High SETL expression, i.e.

$$TE, C \vdash_H Q : set(b_1 \times b_2 \times \ldots b_n).$$

In this section we show how to transform Query Q into a sequence of count-queries and a quantifier-free find-query of the form

$$\{\ [x_1, x_2, \ldots, x_n]\ \in \times_{p=1}^{n} X_p \mid e'(x_1, \ldots, x_n)\}. \tag{4.7}$$

In the following discussion, we abbreviate $\exists x_i \in X_i$ and $\forall x_i \in X_i$ to $\exists x_i$ and $\forall x_i$ respectively. The goal is to simplify the query by successively eliminating each quantifier starting from the innermost to the outermost. First, consider the case when the innermost (rightmost) quantifier is an *existential* quantifier. We use the following two rules :

1. $\exists x_i\ (f(x_i)\ \vee g(x_i)) \equiv \exists x_i\ f(x_i)\ \vee\ \exists x_i\ g(x_i)$.
2. $\exists x_i\ (f(x_i) \wedge h) \equiv \exists x_i\ f(x_i)\ \wedge\ h$, if x_i does not appear free in h.

As defined earlier, the predicate $e(x_1, \ldots, x_m)$ is just a collection of atomic predicates and their negations connected with the boolean connectives $\wedge$ and *vee*. Converting predicate $e(x_1, \ldots, x_m)$ into *Disjunctive Normal Form (DNF)*, we get:

$$e(x_1, \ldots, x_m)\ \equiv e_1(x_1, \ldots, x_m)\ \vee\ e_2(x_1, \ldots, x_m)\ \vee\ \ldots e_l(x_1, \ldots, x_m),$$

where each predicate e_i is just a conjunction of atomic predicates and their negations. Then by Rule 1, the predicate

$$\exists x_m\ e(x_1, \ldots, x_m)$$

is equivalent to

$$\exists x_m\ e_1(x_1, \ldots, x_m)\ \vee\ \ldots\ \vee\ \exists x_m\ e_l(x_1, \ldots, x_m).$$

We now exploit an important property that arises from the fact that the query graph of Query Q is a forest. The property is that in predicate $e(x_1, \ldots, x_m)$, variable x_m can occur in atomic predicates with at most one other variable. Since all other variables in Q occur to the left of variable x_m, there must be an in-coming edge into vertex corresponding to variable x_m from vertices of all other variables x_i which occur in an atomic predicate with variable x_m. Since x_m can have an in-degree of at-most 1, it can therefore appear in an atomic predicate in e with at-most one other variable.

Suppose the in-degree of x_m is 1, *i.e.* the query graph has exactly one edge coming into x_m, say from x_k. This implies that all atomic predicates involving x_m are either single–variable atomic predicates, or two–variable atomic predicates involving x_k and x_m. The case when the in-degree for x_m is 0, *i.e.* it appears only in single–variable atomic predicates is a trivial special case of the case being considered, and is omitted.

For each predicate e_i for $i = 1, \ldots, l$, we separate the atomic predicates into those involving variable x_m and those that do not. Then, applying Rule 2 we get

$$\exists x_m\ e_i(x_1, \ldots, x_m) \equiv e_i^1(x_1, \ldots, x_{m-1})\ \wedge\ \exists x_m\ e_i^2(x_k, x_m)$$

where $e_i^2(x_k, x_m)$ is a conjunction of one and two–variable atomic predicates involving variables x_k and x_m, and the expression $e_i^1(x_1, \ldots, x_{m-1})$ does not contain any atomic predicates involving variable x_m.

Next, we compute the count-query

$$C_i \;=\; \{\; [x_k, \#\{x_m \in X_m : e_i^2(x_k, x_m)\}\;] \;:\; x_k \in X_k \;\},$$

and replace predicate $\exists x_m \; e_i^2(x_k, x_m)$ by the equivalent $C_i(x_k) > 0$. Computing such count-queries for each expression e_i and replacing each predicate $\exists x_m \; e_i^2(x_k, x_m)$ by the equivalent $C_i(x_k) > 0$, for $i = 1, \dots, l$, we get the following equivalent version Q_1 of query Q.

$$\begin{array}{lll} Q_1 = & \text{let} & \\ & & C_1 = \{\; [x_k, \#\{\; x_m \in X_m : e_1^2(x_k, x_m)\}] : x_k \in X_k \;\} \\ & & \vdots \\ & & C_l = \{\; [x_k, \#\{\; x_m \in X_m : e_l^2(x_k, x_m)\}] : x_k \in X_k \;\} \\ & \text{in} & \\ & & \{\; [x_1, \dots, x_n] \in \times_{p=1}^{n} X_p \;| \Theta_{n+1} x_{n+1} \in X_{n+1}, \dots, \Theta_{m-1} x_{m-1} \in X_{m-1} \\ & & \quad ((e_1(x_1, \dots, x_{m-1}) \;\wedge\; C_1(x_k) > 0) \;\vee\; \dots \;\vee \\ & & \quad (e_l(x_1, \dots, x_{m-1}) \;\wedge\; C_l(x_k) > 0)) \;\} \end{array}$$

Note that we are using the expression

$$\begin{array}{lll} Q_1 = & \text{let} & \\ & & C_1 = \dots \\ & & C_2 = \dots \\ & & \vdots \\ & & C_l = \dots \\ & \text{in} & \\ & & \dots \end{array}$$

as an abbreviation of the High SETL expression

$$\begin{array}{lll} Q_1 = & \text{let} & \\ & & C_1 = \dots \\ & \text{in} & \\ & \text{let} & \\ & & C_2 = \dots \\ & & \vdots \\ & \text{in} & \\ & \text{let} & \\ & & C_l = \dots \\ & \text{in} & \\ & & \dots \end{array}$$

Using the fact that Query Q is a well-typed High SETL expression, it is easy to prove that each count-query C_i is also well-typed, i.e.

$$TE, C \vdash_H C_i : \mathit{strong\_smap}(b_k, \mathit{int}),$$

and therefore, that Query Q_1 is also well-typed, i.e.

$$TE, C \vdash_H Q_1 : \mathit{set}(b_1 \times b_2 \times \dots b_n).$$

Thus, the well-typed Query Q is transformed into a query Q_1 having one less quantifier than Q.

The other case in which the innermost quantifier to be eliminated is a *universal* quantifier is handled similarly except that the expression $e(x_1, \dots, x_m)$ is first converted into *Conjunctive Normal Form (CNF).*This process of eliminating the innermost quantifier is applied repeatedly until all leading quantifiers are eliminated. As a result, we are left with a sequence of count-queries followed by one find-query having no leading quantifiers (which we call a quantifier-free find-query).

Theorem 4.3.1 *If LRCS count-queries and quantifier-free find-queries are linear-time queries, then the general LRCS find-query is also a linear-time query*

Proof: The transformation described above transforms the original LRCS find-query into a finite number of count-queries followed by a quantifier-free find-query. The number of count-queries generated is some constant that depends on the syntax of the query (i.e. on the number of sets $X_1, \ldots, X_m$ and on the predicate $e(x_1, \ldots, x_m)$), but is independent of the database size (i.e. the size and the actual values of the input sets). The number of quantifiers in the query is also independent of the database size. Hence the total number of count-queries generated during the quantifier elimination stage is some constant. Furthermore, each count-query is *input-bounded i.e.* the size of the output is bounded by the size of the input. Thus if each count-query is a *linear-time query*, then its time complexity of implementation is asymptotically bounded by the size of its input. The query-graph for the generated quantifier-free find-query is a subgraph of the query-graph of the original find-query, and therefore, is also a forest. Assuming that all LRCS quantifier-free find-queries are also linear-time queries, then the time complexity of implementation of the generated quantifier-free find-query is asymptotically bounded by the sum of the sizes of its input and output. Thus, the time complexity of implementation of the original find-query is linear in the sum of the sizes of its input and output. Hence, under the assumption that count-queries and quantifier-free find-queries are linear-time queries, we see that general LRCS find-queries are also linear-time queries. $\Box$

Thus, the problem of showing that all LRCS queries are linear-time queries is reduced to showing that all LRCS count-queries and quantifier-free find-queries are linear-time queries. Consider the following quantifier–free find-query

$$Q' \;=\; \{\; [x_1, x_2, \ldots, x_n] \;\in \times_{p=1}^{n} X_p \mid e'(x_1, \ldots, x_n) \;\}.$$

Converting $e'(x_1, \ldots, x_n)$ to *DNF*, we get

$$e'(x_1, \ldots, x_n) \;\equiv\; e_1(x_1, \ldots, x_n) \;\vee\; \ldots \;\vee\; e_j(x_1, \ldots, x_n).$$

From Rule 1, it follows that $Q' \;=\; \cup_{i=1}^{j}\; Q_i$, where for all $i = 1, \ldots, j$, Query Q_i is given by

$$Q_i \;=\; \{\; [x_1, x_2, \ldots, x_n] \;\in \times_{p=1}^{n} X_p \mid e_i(x_1, \ldots, x_n) \;\},$$

and each such query Q_i is a quantifier-free find-query whose predicate is just a conjunction of atomic predicates and their negations. If Query Q' is a valid LRCS query, then it must be well-typed, i.e. there exists a type environment TE, and a set of subtype constraints C such that

$$TE, C \vdash_{H} Q' : set(b_1 \times b_2 \times \ldots \times b_n).$$

Then, it is easy to show that for all $i = 1, \ldots, j$

$$TE, C \vdash_{H} Q_i : set(b_1 \times b_2 \times \ldots \times b_n).$$

It follows from the High SETL type system that the High SETL expression $\cup_{i=1}^{j}\; Q_i$ is also well-typed, i.e.

$$TE, C \vdash_{H} \cup_{i=1}^{j}\; Q_i : set(b_1 \times b_2 \times \ldots \times b_n),$$

and that the time complexity of performing the actual union is asymptotically bounded by just the sum of the sizes of the outputs of each query Q_i, i.e. $\Sigma_{i=1}^{j} \# Q_j$. Moreover, $\# Q_i \leq \# Q'$ for each individual query Q_i, and the number j of these queries is a constant independent of the database size. Therefore, if each query Q_i is a linear-time query, then it follows that the time taken to compute query Q' by computing the equivalent $\cup_{i=1}^{j} Q_i$ is also linear in the sum of the sizes of its input and output.

Thus, the problem of showing that all quantifier-free find-queries are linear-time queries is reduced to showing that all quantifier-free find-queries, whose predicates are just conjunctions of atomic predicates and their negations, are linear-time queries. In the next section we describe another transformation that reduces the problem further to just proving that all simple find-queries (i.e. two-variable qunatifier-free find-queries whose predicate is just a conjunction of atomic predicates and their negations) are linear-time queries.

4.3.2 Join Decomposition

We now look at queries of the form

$$Q \;=\; \{\; [x_1, x_2, \ldots, x_n] \;\in \times_{p=1}^{n} X_p \mid e(x_1, \ldots, x_n) \;\} \tag{4.8}$$

where predicate $e(x_1, \ldots, x_n)$ is a conjunction of atomic predicates and their negations.

Definition 4.3.2 Projection: Given a set Q (Equation 4.8) containing a subset of $\times_{i=1}^{n} X_i$, the projection of Q onto sets $X_1, \ldots, X_j$, denoted by $\Pi_{X_1,\ldots,X_j} Q$, is a subset of $\times_{i=1}^{j} X_i$ consisting of the restriction of the tuples of Q to the sets $X_1, \ldots, X_j$; *i.e.*,

$$\begin{aligned}\Pi_{X_1,\ldots,X_j} Q = \quad & \{[x_1, x_2, \ldots, x_j] \in \times_{p=1}^{j} X_p \mid \exists x_{j+1} \in X_{j+1}, \ldots, \exists x_n \in X_n \\ & \quad e(x_1, \ldots, x_n) \;\}\end{aligned}$$

Definition 4.3.3 Natural Join: Let Q_1 be a subset of $\times_{i=1}^{k} X_i$ and Q_2 be a subset of $\times_{i=k}^{j} X_i$. The natural join of Q_1 and Q_2 with respect to set X_k, denoted by $Q_1 \bowtie_{X_k} Q_2$, is a subset of $\times_{i=1}^{j} X_i$ resulting from putting together tuples in Q_1 and Q_2 that have the same value from set X_k; *i.e.*,

$$Q_1 \bowtie_{X_k} Q_2 = \{[x_1, \ldots, x_j] \in \times_{i=1}^{j} X_i \mid [x_1, \ldots, x_k] \in Q_1 \;\wedge\; [x_k, \ldots, x_j] \in Q_2\}$$

The following two lemmas are helpful in transforming query Q into a more efficient implementation.

Lemma 4.3.4 *Let Q be the query defined in (4.8) and $T = \Pi_{X_1,\ldots,X_j} Q$ where $j \le n$. Then $\#T \le \#Q$.*

The proof is trivial, and is omitted.

Lemma 4.3.5 *Let Q be the query defined in (4.8). Let T_1, T_2, and T_3 be given by*

$$\begin{aligned} T_1 &= \Pi_{X_1,\ldots,X_j} Q, \\ T_2 &= \Pi_{X_1,\ldots,X_k} Q, \\ T_3 &= \Pi_{X_k,\ldots,X_j} Q, \end{aligned}$$

where $1 \le k \le j$. Furthermore, let predicate $e(x_1, \ldots, x_n)$ be decomposable into a conjunction of predicates $e_1(x_1, \ldots, x_k)$ and $e_2(x_k, \ldots, x_n)$, i.e.

$$e(x_1, \ldots, x_n) = e_1(x_1, \ldots, x_k) \;\wedge\; e_2(x_k, \ldots, x_n).$$

This is possible if predicate $e(x_1, \ldots, x_n)$ does **not** *contain any two–variable atomic predicates on variables u and v where $u \in \{x_1, \ldots, x_{k-1}\}$ and $v \in \{x_{k+1}, \ldots, x_n\}$. Then, T_1 is the natural join of T_2 and T_3 with respect to set X_k, i.e. $T_1 = T_2 \bowtie_{X_k} T_3$.*

Proof: Proving $T_1 \subseteq (T_2 \bowtie_{X_k} T_3)$ is trivial, and is omitted.

To prove that $(T_2 \bowtie_{X_k} T_3) \subseteq T_1$, consider a tuple $[x'_1, x'_2, \ldots, x'_j] \in (T_2 \bowtie_{X_k} T_3)$. Then, it follows from Definition 4.3.3 that $[x'_1, \ldots, x'_k] \;\in\; T_2$, and $[x'_k, \ldots, x'_j] \;\in\; T_3$. By Definition 4.3.2, $[x'_1, \ldots, x'_k] \in T_2$ implies that

$$\exists x'_{k+1}, \ldots, x'_n \in (\times_{p=k+1}^{n} X_p) \; (e_1(x'_1, \ldots, x'_k) \;\wedge\; e_2(x'_k, \ldots x'_n)).$$

Therefore, the tuple $[x'_1, \ldots, x'_k]$ must satisfy the predicate e_1, i.e. $e_1(x'_1, \ldots, x'_k)$ must be true. By a similar reasoning, $[x'_k, \ldots, x'_j] \in T_3$ implies that

$$\exists x'_{j+1}, \ldots, x'_n \in (\times_{p=j+1}^{n} X_p) \; e_2(x'_k, \ldots, x'_j, x'_{j+1}, \ldots, x'_n).$$

Hence, it follows that

$$\exists x'_{j+1}, \ldots, x'_n \in (\times_{p=j+1}^{n} X_p)\ e(x'_1, \ldots, x'_n).$$

From Definition 4.3.2 it follows that $[x'_1, \ldots, x'_j] \in T_1$, thus completing the proof that $T_2 \bowtie_{X_k} T_3 \subseteq T_1$. $\square$

Since Query Q given by Equation 4.8 is a valid LRCS query, its query graph must be a forest. It is not difficult to prove that for every directed edge (x_k, x_j) in the forest, there exists a permutation $[x_k, x_j, x_{p_1}, \ldots, x_{p_{n-2}}]$ of variables $[x_1, x_2, \ldots, x_n]$, such that the query graph for the query

$$\begin{aligned} Q_{(x_k,x_j)} \ = \ \{\ & [x_k, x_j] \in X_k \times X_j \mid \exists x_{p_1} \in X_{p_1}, \ldots, \exists x_{p_{n-2}} \in X_{p_{n-2}} \\ & e(x_1, \ldots, x_n)\ \} \end{aligned} \tag{4.9}$$

is also a forest. In fact such an ordering of variables may be obtained as follows. Take the tree T containing the directed edge (x_k, x_j) from the forest, and reverse the direction of edges along the path from x_k to the root of this tree. As a result, vertex x_k becomes the new root of tree T. Now, an appropriate sequence of variables $x_{p_1}, \ldots, x_{p_{n-2}}$ may be selected by first taking the variables other than x_k and x_j from tree T in the order of breadth-first search from root x_k, and then taking all the variables from all other trees in the forest also in breadth-first search order. With this sequence of variables $x_{p_1}, \ldots, x_{p_{n-2}}$, it is easy to verify that the query-graph for Query $Q_{(x_k,x_j)}$ given by Equation 4.9 is a forest. Moreover, if the original query Q is a well-typed High SETL expression, then so is Query $Q_{(x_k,x_j)}$.

We generate such a query $Q_{(x,y)}$ for each directed edge (x, y) in the query graph. The number of edges in the forest is bounded by n, the number of sets $X_1, \ldots, X_n$, but is independent of the database size. For each edge (x_k, x_j), the value of $Q_{(x_k,x_j)}$ is the projection of the value of Query Q to sets X_k and X_j, i.e. $Q_{(x,y)} = \Pi_{X,Y} Q$. Therefore, it follows that $\#Q_{(x_k,x_j)} \leq \#Q$ for each edge (x_k, x_j).

We now perform a sequence of *natural joins* (i.e. the $\bowtie$ operator) on these queries $Q_{(x_k,x_j)}$ to compute Query Q. We select one query corresponding to a leaf-edge (i.e. an edge one of whose vertices is a leaf of the forest), and select the second query corresponding to another edge which shares a common vertex with the previously selected leaf-edge, and replace the two edges with a single edge that represents the natural join of these two queries. It is easy to verify that on taking the natural join of these two queries, the conditions of Lemma 4.3.5 are satisfied, and thus, the result of the natural join is also a projection of Q. We continue this process till we are just left with disconnected edges in the forest. Taking the cross-product of the sets corresponding to these edges, we get Q.

Since each of the the results of the natural joins are projections of Q (from Lemma 4.3.5), the size of each the intermediate outputs from the natural joins is bounded by the size $\#Q$ of the final output.

Theorem 4.3.6 *If each of the queries $Q_{(x,y)}$ is a linear-time query, and* **join** *is a linear-time query, then Q is a linear-time query.*

Proof: Q is computed by first computing each of the queries $Q_{(x,y)}$, and then performing *join* operations in a suitable order such that the conditions of Lemma 4.3.5 are satisfied for each join. If each $Q_{(x,y)}$ is a linear-time query, then its running time is asymptotically bounded by the sum of its input and output sizes, and hence by the sum of the input size and the size of the final output Q (because $\#Q_{(x,y)} \leq \#Q$). Also the linear running time for the *joins* implies that their running times are asymptotically bounded by the sum of *their* input and output sizes, and hence the size of the final output because the intermediate input and output sizes for each join are bounded by the size of the final output Q. Thus, the time to compute Q is asymptotically bounded by the sum of sizes of the input sets and the size of the final output, thereby proving that Q is a linear-time query. $\square$

Since each Query $Q_{(x_k,x_j)}$ is a valid LRCS query, we may perform the quantifier elimination described in Section 4.3.1 to transform it into a sequence of count queries followed by a simple find-query, i.e. a two-variable quantifier-free find-query whose predicate is just a sequence of conjuncts of atomic predicates and their negations. Furthermore a natural join operation is just a very small variation of the *simple find* query

$$\{[x', y'] \in X' \times Y' \mid f(x') = g(y')\ \}.$$

Thus, the problem of showing that all well-typed queries of the form of Query Q given by Equation 4.8 are linear-time queries is reduced to the problem of showing that all well-typed count-queries and simple find-queries are linear-time queries.

In the next section we prove that all well-typed count-queries and simple find-queries are linear-time queries. Then, from the previous discussion, it will follow that all valid LRCS queries are linear-time queries.

4.4 Count-Queries And Simple Find-Queries

In this section we focus our attention on count-queries

$$Q_c = \{[x, \#\{y \in Y \mid e(x,y)\ \}] : x \in X\}, \tag{4.10}$$

and simple find-queries

$$Q_{sf} = \{[x,y] \in X \times Y \mid e(x,y)\ \}, \tag{4.11}$$

where predicate $e(x,y)$ is just a conjunction of atomic predicates and their negations. We restrict our attention to well-typed queries, i.e. queries for which there exists a type environment TE, and a set of subtype constraints C such that $TE(X) = set(b_1)$ and $TE(Y) = set(b_2)$ for some base types b_1 and b_2, and

$$TE, C \vdash_H \{[x, \#\{y \in Y \mid e(x,y)\ \}] : x \in X\} : strong\_smap(b_1, int),$$

or

$$TE, C \vdash_H \{[x,y] \in X \times Y \mid e(x,y)\ \} : set(b_1 \times b_2).$$

Willard showed that count-queries and simple find-queries could be implemented in average-case linear time by studying increasingly more general variants of the predicate $e(x,y)$ appearing in (4.10) and (4.11), which he called classes E-1 through E-8 of enactment expressions. He gave algorithms to decompose queries in a more general class E-(i+1) into queries in class E-(i). He described low level operations to construct indexes and other auxiliary data structures used to compute the simple find queries efficiently. Similar constructions were used to calculate several different forms of simple queries, and were described in detail each time a different case was considered. The low level constructions were analyzed by detailed and sometimes complex counting arguments repeated for each different case.

Our use of well-typed High SETL expressions to express these queries provides us with larger, more abstract building blocks that are easily combined by composition and parameter substitution to form queries. This allows us to replace low level counting arguments by more algebraic and logical reasoning in our algorithmic analysis. Consequently, we can focus on the key ideas involved in decomposing difficult queries into simpler queries in terms of high-level transformations that are easy to understand and prove correct. Furthermore, the transformations themselves and the order in which they are applied have been carefully tailored to give a linear time implementation. Since the type system ensures that each associative access operation can be implemented to run in $O(1)$ time on a pointer machine, we get a worst-case linear time implementation for LRCS.

We now present a series of theorems that together lead to the proof that well-typed count-queries and simple find-queries are linear-time queries.

Theorem 4.4.1 *All well-typed count-queries and simple-find queries whose predicate $e(x,y)$ is just a conjunct of an arbitrary*[1] *number of anti-join predicates are linear-time queries.*

Proof: We shall consider the two cases for count-queries and simple find-queries separately.

[1]but a finite constant independent of the database size

Case 1: Count-queries The proof follows by induction on the number of anti-join predicates.

The base case is when the number of anti-join predicates is 0, i.e. the predicate $e(x, y)$ is *true*. In this case Count-query Q_c given by

$$Q_c = \{[x, \#\{y \in Y \mid \mathit{true}\}] : x \in X\}$$

is equivalent to the query Q' given by

$$\begin{array}{ll} Q' = & \text{let} \\ & \quad n = \#Y \\ & \text{in} \\ & \quad \{[x, n] : x \in X\} \end{array}$$

If Query Q_c is well-typed, i.e. $TE, C \vdash_H Q_c : \mathit{strong\_smap}(b_1, \mathit{int})$, then it is easy to show that Query Q' is also well-typed, i.e. $TE, C \vdash_H Q' : \mathit{strong\_smap}(b_1, \mathit{int})$. The time complexity to compute Q' is $O(\#Y + \#X)$. Thus, Q' is a linear-time query.

Now let us consider the case of a count-query whose predicate is a conjunction of $t + 1$ anti-joins. Thus, Query Q_c is given by

$$Q_c = \{[x, \#\{y \in Y \mid K(x, y) \ \wedge \ (f(x) \neq g(y))\}] : x \in X\},$$

where $K(x, y)$ represents the other t anti-join predicates. Assume that Query Q_c is well-typed, i.e.

$$TE, C \vdash_H Q_c : \mathit{strong\_smap}(b_1, \mathit{int}). \tag{4.12}$$

Recall that for each anti-join predicate $f(x) \neq g(y)$, both expressions $f(x)$ and $g(y)$ must be of some base type b_3, i.e.

$$TE[x \mapsto b_1], C \vdash_H f(x) : b_3, \text{ and } TE[y \mapsto b_2], C \vdash_H g(y) : b_3. \tag{4.13}$$

Query Q_c can be transformed into the following equivalent query.

$$\begin{array}{ll} Q' = & \text{let} \\ & \quad Q_1 = \{[f(x), x] : x \in X\} \\ & \quad Q_2 = \{[g(y), y] : y \in Y \mid Q_1\{g(y)\} \neq \text{om}\} \\ & \quad Q_3 = \uplus_{z \in domain(Q_2)} \{[x, \#\{y \in Q_2\{z\} \mid K(x, y)\}] : x \in Q_1\{z\}\} \\ & \quad Q_4 = \{[x, \#\{y \in Y \mid K(x, y)\}] : x \in X\} \\ & \text{in} \\ & \quad \{[x, Q_4(x) - Q_3(x)] : x \in X \mid Q_3(x) \neq \text{om}\} \uplus \\ & \quad \{[x, Q_4(x)] : x \in X \mid Q_3(x) = \text{om}\} \end{array}$$

First, let us prove that Query Q' is in fact equivalent to Query Q_c. Query Q_1 computes the inverse map of expression f restricted to set X, and Query Q_2 computes the inverse map of expression g restricted to those $y \in Y$ for which $\exists x \in X \ (f(x) = g(y))$. Then, it follows that for all $x \in domain(Q_3)$, $Q_3(x) = \#\{y \in Y \mid K(x, y) \ \wedge \ (f(x) = g(y))\}$, and for all $x \notin domain(Q_3)$, the $\#\{y \in Y \mid K(x, y) \ \wedge \ (f(x) = g(y))\} = 0$. The rest of the proof is trivial and is omitted.

Next, let us prove that Query Q' is also a well-typed High SETL expression. From the well-typedness of Query Q_c given by Derivation 4.12, and the types of $f(x)$, and $g(y)$ given by the derivations in 4.13, it is easy to prove that

$$TE, C \vdash_H Q_1 : \mathit{strong\_mmap}(b_3, b_1).$$

Next, we can prove that

$$\begin{aligned}&TE[Q_1 \mapsto \mathit{strong\_mmap}(b_3, b_1)], C \vdash_H Q_2 : \mathit{strong\_mmap}(b_3, b_2),\\ &TE[Q_1 \mapsto \mathit{strong\_mmap}(b_3, b_1), Q_2 \mapsto \mathit{strong\_mmap}(b_3, b_2)], C \vdash_H\\ &\quad Q_3 : \mathit{strong\_smap}(b_1 : \mathit{int}), \text{ and}\\ &TE[Q_1 \mapsto \mathit{strong\_mmap}(b_3, b_1), Q_2 \mapsto \mathit{strong\_mmap}(b_3, b_2),\\ &\quad Q_3 \mapsto \mathit{strong\_smap}(b_1 : \mathit{int})], C \vdash_H Q_4 : \mathit{strong\_smap}(b_1 : \mathit{int})\end{aligned}$$

Finally, it follows from the above derivations that

$$TE, C \vdash_H Q' : \mathit{strong\_smap}(b_1, \mathit{int}),$$

thus proving that Query Q' is a well-typed High SETL expression.

Finally, we consider the time complexity of computing Q'. The costs of computing Q_1 and Q_2 are $O(\#X)$ and $O(\#Y)$ respectively. For computing Q_3, we compute multiple count-queries each of whose predicate is a conjunct of t anti-joins. By our inductive hypothesis, each of these count-queries are linear-time queries. Therefore, the cost of computing Q_3 is

$$\Sigma_{z \in domain(Q_2)} O(\#Q_1\{z\} + \#Q_2\{z\}) = O(\#X + \#Y).$$

Query Q_4 is also a count query whose predicate contains t anti-joins, and therefore by the inductive hypothesis, the cost of computing Q_4 is $O(\#X+\#Y)$. Then, it follows easily that the cost of computing Q' is $O(\#X + \#Y)$, thus proving that a count-query whose predicate is a conjunct of $t + 1$ anti-joins is also a linear-time query.

Case 2: Simple Find-queries The proof once again follows by induction on the number of anti-join predicates.

The base case is when the number of anti-join predicates is 0. In this case, the simple find-query Q_{sf} is

$$Q_{sf} = \{[x, y] \in X \times Y \mid \mathit{true}\},$$

and its time complexity is $O(\#X + \#Y + \#Q_{sf})$, where $\#Q_{sf}$ is the size of the output. Thus, the query is a linear-time query.

Now consider the case when predicate $e(x, y)$ of Query 4.11 is a conjunction of t anti-join predicates. Let Q_{sf} be given by

$$Q_{sf} = \{[x, y] \in X \times Y \mid \wedge_{i=1}^{t} f_i(x) \neq g_i(y)\}.$$

We assume that Q_{sf} is well-typed High SETL expression, i.e.

$$TE, C \vdash_H Q_{sf} : \mathit{set}(b_1 \times b_2).$$

For each anti-join predicate $f_i(x) \neq g_i(y)$, we assume that both $f_i(x)$ and $g_i(y)$ are of some base type b'_i. Then, Query Q_{sf} can be transformed into the equivalent

$$\begin{aligned}Q' = \;& \text{let}\\ &\quad n = \#Y\\ &\quad C_i = \{[f_i[x], x] : x \in X\} \text{ for all } i = 1, \ldots, t\\ &\quad D_i = \{[g_i[y], y] : y \in Y\} \text{ for all } i = 1, \ldots, t\\ &\quad \mathit{Crit}_i = \{z \in \mathit{domain}(D_i) \mid \#D_i\{z\} \geq n/2t\} \text{ for all } i = 1, \ldots, t\\ &\quad X_0 = \{x \in X \mid \wedge_{i=1}^{t} f_i(x) \notin \mathit{Crit}_i\}\\ &\quad Q_0 = \{[x, y] \in X_0 \times Y \mid \wedge_{i=1}^{t} f_i(x) \neq g_i(y)\}\\ &\quad Q_i = \uplus_{z \in \mathit{Crit}_i} \{[x, y] \in C_i\{z\} \times (Y - D_i\{z\}) \mid\\ &\quad\qquad \wedge_{j=1, j \neq i}^{t} f_j(x) \neq g_j(y)\} \text{ for all } i = 1, \ldots, t\\ & \text{in}\\ &\quad Q_0 \cup Q_1 \cup \ldots \cup Q_t\end{aligned}$$

First, let us prove that Query Q' is in fact equivalent to Query Q_{sf}. Each query C_i is the inverse map of function f_i restricted to set X, and each query D_i is the inverse map of function g_i restricted to set Y. Therefore, $Crit_i$ contains those elements z in the range of function g_i such that the size of the pre-image of z under g_i is greater than or equal to $\#Y/2t$, i.e. $\#g_i^{-1}\{z\} \geq \#Y/2t$. Set X_0 is the set of those x in X such that for all $i = 1, \ldots, t$, $g_i(x)$ does not belong to $Crit_i$, i.e.

$$X_0 = \{x \in X \mid \forall i = 1, \ldots, t\ (g_i(x) \notin Crit_i)\}.$$

Let us define $X_i = \{x \in X \mid g_i(x) \in Crit_i\}$ for all $i = 1, \ldots, t$. Then, it follows that

$$X = X_0 \cup X_1 \cup \ldots \cup X_t,$$

and that

$$Q_i = \{[x, y] \in X_i \times Y \mid \wedge_{j=1}^{t} f_j(x) \neq g_j(y)\}, \text{for all } i = 0, \ldots, t.$$

Then, it is obvious that $Q_0 \cup Q_1 \cup \ldots \cup Q_t$ is equivalent to Query Q_{sf}.

Next we prove that Query Q' is a well-typed High SETL expression. The following sequence of derivations give an outline of how to get a type-derivation for expression Q'.

$$TE, C \vdash_H C_i : \mathit{strong\_mmap}(b'_i, b_1) \text{ (for all } i = 1, \ldots, t)$$

$$TE, C \vdash_H D_i : \mathit{strong\_mmap}(b'_i, b_2) \text{ (for all } i = 1, \ldots, t)$$

$$TE[D_i \mapsto \mathit{strong\_mmap}(b'_i, b_2)], C \vdash_H \\ Crit_i : \mathit{strong\_set}(b'_i) \text{ (for all } i = 1, \ldots, t)$$

$$TE[\forall j = 1, \ldots, t\ Crit_j \mapsto \mathit{strong\_set}(b'_j)], C \vdash_H X_0 : \mathit{set}(b_1)$$

$$TE[X_0 \mapsto \mathit{set}(b_1)], C \vdash_H Q_0 : \mathit{set}(b_1 \times b_2)$$

$$TE[Crit_i \mapsto \mathit{strong\_set}(b'_i), C_i \mapsto \mathit{strong\_mmap}(b'_i, b_1), \\ D_i \mapsto \mathit{strong\_mmap}(b'_i, b_2)], C \vdash_H Q_i : \mathit{set}(b_1 \times b_2)$$

$$TE[\forall j = 0, \ldots, t\ Q_j \mapsto \mathit{set}(b_1 \times b_2)], C \vdash_H Q_0 \cup Q_1 \cup \ldots \cup Q_t : \mathit{set}(b_1 \times b_2)$$

Finally, we consider the time complexity of computing Query Q'. The time complexity of computing each query C_i and D_i is $O(\#X)$ and $O(\#Y)$ respectively. The cost of computing each query $Crit_i$ is $O(\Sigma_{z \in domain(D_i)} D_i\{z\}) = O(\#Y)$. The cost of computing Query X_0 is $O(\#X)$. The cost of computing Q_0 is $O(\#X_0 \times \#Y)$. Now, we make use of an interesting argument to prove that the cost of computing Q_0 is in fact $O(\#Q_0)$, and since $Q_0 \subseteq Q'$, the cost of computing Q_0 is also $O(\#Q')$. For all $x \in X_0$, we know that $g_i(x) \notin Crit_i$ for all $i = 1, \ldots, t$. It then follows that

$$\text{for all } x \in X_0,\ \#\{y \in Y \mid f_i(x) = g_i(y)\} \leq \#Y/2t.$$

This implies that

$$\text{for all } x \in X_0,\ \#\{y \in Y \mid \vee_{i=1}^{t} f_i(x) = g_i(y)\} \leq \#Y/2,$$

which can be reformulated as

$$\text{for all } x \in X_0,\ \#\{y \in Y \mid \wedge_{i=1}^{t} f_i(x) \neq g_i(y)\} \geq \#Y/2,$$

Therefore, it follows that $\#Q_0 \geq (\#X_0 \times \#Y)/2$. Thus, the cost of computing Query Q_0 is $O(\#Q_0)$, and therefore also $O(\#Q')$. Consider the cost of computing each Q_i. From a simple application of the pigeon-hole principle, it follows that $\#Crit_i \leq 2t$ for all i. Then, each Q_i is a disjoint union of at-most $2t$ (a constant) simple find-queries each of which only has $t-1$ anti-join predicates. By our, inductive hypothesis, each of these queries are linear-time queries, and therefore the time complexity of each of these queries is $O(\#X + \#Y + \#Q')$. Finally, the cost of the final union $Q_0 \cup Q_1 \cup \ldots Q_t$ is $O(\#Q')$ (since once again, t is just a constant). Thus, the total cost of computing Q' is $O(\#X + \#Y + \#Q')$, and therefore, it is a linear-time query[2]

This ends the proof of Theorem 4.4.1. $\square$

Theorem 4.4.2 *All well-typed count-queries and simple find-queries whose predicate $e(x,y)$ is just a conjunction of an arbitrary number of anti-join and equi-join predicates are linear-time queries.*

Proof: The proof follows by induction on the number of equi-join predicates. We first consider the case for simple find-queries.

The base case is when the number of equi-join predicates is 0. Then, it follows from Theorem 4.4.1, that the simple find-query containing only anti-join predicates is a linear-time query.

Now, let's consider the case of a simple find-query whose predicate $e(x,y)$ is a conjunction of an arbitrary number of anti-join predicates and $t+1$ equi-join predicates. Let the query Q_{sf} be given by

$$Q_{sf} = \{[x,y] \in X \times Y \mid K(x,y) \ \wedge \ f(x) = g(y)\},$$

where the predicate $K(x,y)$ denotes a conjunction of an arbitrary number of anti-join predicates and t equi-join predicates. Assume that query Q_{sf} is well-typed, i.e.

$$TE, C \vdash_H Q_{sf} : set(b_1 \times b_2).$$

Recall that for each equi-join predicate $f(x) = g(y)$, both expressions $f(x)$ and $g(y)$ must be of some base type b_3, i.e.

$$TE[x \mapsto b_1], C \vdash_H f(x) : b_3, \text{ and } TE[y \mapsto b_2], C \vdash_H g(y) : b_3.$$

Query Q_{sf} can be transformed into the following equivalent query.

$$\begin{array}{rl} Q' = & \text{let} \\ & \quad Q_1 = \{[f(x), x] : x \in X\} \\ & \quad Q_2 = \{[g(y), y] : y \in Y \mid Q_1\{g(y)\} \neq \text{om}\} \\ & \text{in} \\ & \quad \biguplus_{z \in domain(Q_2)} \{[x,y] \in Q_1\{z\} \times Q_2\{z\} \mid K(x,y)\} \end{array}$$

First, let us prove that Query Q' is in fact equivalent to Query Q_{sf}. Query Q_1 computes the inverse map of expression f restricted to set X, and Q_2 computes the inverse map of expression g restricted to those $y \in Y$ for which $\exists x \in X \ f(x) = g(y)$. Therefore, it follows that for all $z \in domain(Q_2)$, any pair $[x,y] \in Q_1\{z\} \times Q_2\{z\}$ satisfies $f(x) = g(y)$. Moreover, for every pair $[x,y] \in X \times Y$ satisfying $f(x) = g(y)$, there exists a z in $domain(Q_2)$ such that $x \in Q_1\{z\}$ and $y \in Q_2\{z\}$. Then, it easily follows that Query Q' is equivalent to Q_{sf}.

Next, we prove that Query Q' is well-typed. The following sequence of type-derivations give an outline of how to get a type-derivation of Query Q'.

$$TE, C \vdash_H Q_1 : strong\_mmap(b_3, b_1)$$

$$TE[Q_1 \mapsto strong\_mmap(b_3, b_1)], C \vdash_H Q_2 : strong\_mmap(b_3, b_2)$$

$$\begin{array}{l} TE[Q_1 \mapsto strong\_mmap(b_3, b_1), Q_2 \mapsto strong\_mmap(b_3, b_2)], C \vdash_H \\ \quad \biguplus_{z \in domain(Q_2)} \{[x,y] \in Q_1\{z\} \times Q_2\{z\} \mid K(x,y)\} : set(b_1 \times b_2) \end{array}$$

[2]Note, that the constant factor involved in $O(\#X + \#Y + \#Q')$ is $2^t \times (t!)^2$, which for out purposes is a constant, since t is a constant.

Finally, we consider the time complexity of computing Q'. The cost of computing Q_1 and Q_2 is $O(\#X)$ and $O(\#Y)$ respectively. By the inductive hypothesis, each query $\{[x,y] \in Q_1\{z\} \times Q_2\{z\} \mid K(x,y)\}$ is a linear-time query since $K(x,y)$ contains only t equi-join predicates and any number of anti-join predicates. Let $Output(z)$ denote the output of query $\{[x,y] \in Q_1\{z\} \times Q_2\{z\} \mid K(x,y)\}$. Then, the cost of computing $\biguplus_{z \in domain(Q_2)}\{[x,y] \in Q_1\{z\} \times Q_2\{z\} \mid K(x,y)\}$ is

$$\Sigma_{z \in domain(Q_2)} O(\#Q_1\{z\} + \#Q_2\{z\} + Output(z)) = O(\#X + \#Y + \#Q').$$

Thus, the total cost of computing Q' is $O(\#X + \#Y + \#Q')$, and therefore Q' is a linear-time query. The proof for the case of count-queries is similar, and is omitted. $\square$

Theorem 4.4.3 *All well-typed count-queries and simple find-queries whose predicate $e(x,y)$ is a conjunction of an arbitrary number of anti-join, equi-join or tabular predicates are linear-time queries.*

Proof: We first consider the case of simple find-queries. Let Q_{sf} be a simple-find query containing t tabular predicates and any number of anti-join or equi-join predicates. If t is 0, then it follows from Theorem 4.4.2 that Q_{sf} is a linear-time query. If $t > 0$, then let the query Q_{sf} be given by

$$\begin{aligned} Q_{sf} = \{[x,y] \in X \times Y \mid\ & K(x,y) \wedge \\ & \wedge_{i=1}^{t} \exists z \in Z_i\ (f_i'(z) = g_i'(x)\ \wedge\ f_i''(z) = g_i''(y))\}, \end{aligned}$$

where $K(x,y)$ is the conjunction of an arbitrary number of equi-join and anti-join predicates. Assume that Query Q_{sf} is well-typed, i.e.

$$TE, C \vdash_H Q_{sf} : set(b_1 \times b_2).$$

Recall that for all tabular predicates of the form $\exists z \in Z_i\ (f_i'(z) = g_i'(x)\ \wedge\ f_i''(z) = g_i''(y))$, each set Z_i must contain elements of some base type, each expression $f_i'(z)$ and $g_i'(x)$ must be of some base type b_i', each expression $f_i''(z)$ and $g_(''y)$ must be of some base type b_i'', and that all expressions $f_i'(z), g_i'(x), f_i''(z)$, and $g_(''y)$ must be few-to-one. Also recall that a map h is few to one, if for any element z, the number of elements in the pre-image of z under map h is bounded by a constant c, i.e $\#h^{-1}\{z\} \leq c$, for some constant c.

Query Q_{sf} is transformed into the following equivalent query Q'.

$$\begin{aligned} Q' = \ & \text{let} \\ & \quad Q_i = \{[f_i'(z), f_i''(z)] : z \in Z_i\} \text{ for all } i = 1, \ldots t \\ & \quad X_1 = \{[g_1'(x), x] : x \in X\} \\ & \quad Y_1 = \{[g_1''(y), y] : y \in Y\} \\ & \text{in} \\ & \quad \biguplus_{[p,q] \in Q_1}\{[x,y] \in X_1\{p\} \times Y_1\{q\} \mid K(x,y) \wedge \\ & \qquad\qquad \wedge_{i=2}^{t}\ \exists w \in Q_i\{g_i'(x)\}\ (g_i''(y) = w)\} \end{aligned}$$

First, let us prove that Query Q' is in fact equivalent to Query Q_{sf}. Each query Q_i contains that pairs $[f_i'(z), f_i''(z)]$ for every z in set Z_i. From the definition of X_1 and Y_1, it follows that for all $[p,q]$ in Q_1, if there exists a pair $[x,y]$ in $X_1\{p\} \times Y_1\{q\}$, then there exists a z in Z_1 such that $f_1'(z) = g_1'(x) = p$ and $f_1''(z) = g_1''(y) = q$. Moreover, $g_i''(y) \in Q_i\{g_i'(x)\}$ is also equivalent to $\exists z \in Z_i\ (f_i'(z) = g_i'(x)\ \wedge\ f_i''(z) = g_i''(y))$. Thus, it follows that Q' is equivalent to Q_{sf}.

Next, let us prove that Query Q' is well-typed. The following sequence of type-derivations give an outline of how to get a type-derivation for Query Q'.

$$TE, C \vdash_H Q_i : strong\_mmap(b_i', b_i'')$$

$$TE, C \vdash_H X_1 : strong\_mmap(b_1', b_1)$$

$$TE, C \vdash_H Y_1 : strong\_mmap(b_1'', b_2)$$

$$\begin{aligned} & TE[(\forall i = 1, \ldots t\ Q_i \mapsto strong\_mmap(b_i', b_i'')), X_1 \mapsto strong\_mmap(b_1', b_1), \\ & \quad Y_1 \mapsto strong\_mmap(b_1'', b_2)], C \vdash_H \{[x,y] \in X_1\{p\} \times Y_1\{q\} \mid K(x,y) \wedge \\ & \qquad \wedge_{i=2}^{t}\ \exists w \in Q_i\{g_i'(x)\}\ (g_i''(y) = w)\} : set(b_1 \times b_2) \end{aligned}$$

Finally, we consider the time complexity of computing Q'. The time complexity of computing each Q_i is $O(\#Z_i)$. The time complexity of computing X_1 and Y_1 are $O(\#X)$ and $O(\#Y)$ respectively. Since expressions $g_1'(x)$ and $g_1''(y)$ are few-to-one, it follows that for any p and q, the sizes of $X_1\{p\}$ and $Y_1\{q\}$ are bounded by a constant c, i.e. the size of $X_1\{p\} \times Y_1\{q\}$ is bounded by c^2, which is also a constant. Moreover, since each expression $f_i''(z)$ is also few-to-one, each map Q_i must also be one-to-few. Thus, for any z, the size of $Q_i\{z\}$ is also bounded by a constant. Thus, it follows that the total time to evaluate

$$\uplus_{[p,q]\in Q_1}\{[x,y] \in X_1\{p\} \times Y_1\{q\} \mid K(x,y) \;\wedge\; \wedge_{i=2}^{t} \exists w \in Q_i\{g_i'(x)\}\ (g_i''(y) = w)\}$$

is $O(\#Z_1)$. Thus, the time complexity of evaluation of Q' is $O(\#X + \#Y + \#Z_1 + \#Z_2 + \ldots + \#Z_t)$, and therefore Q' is a linear-time query. The proof for count-queries is similar, and is omitted. $\Box$

Theorem 4.4.4 *All well-typed count-queries and simple find-queries whose predicate $e(x,y)$ is just a conjunction of an arbitrary number of anti-join, equi-join, tabular and negated tabular predicates are linear-time queries.*

Proof: We consider the cases of count-queries and simple find-queries separately.

Case 1: Count-queries The proof follows by an induction on the number of negated tabular predicates.

The base case is when the number of negated tabular predicates is 0. In this case, it follows from Theorem 4.4.3 that the query is a linear-time query.

Next, we consider the case, when the query has $t+1$ negated tabular predicates. Let Q_c be the count-query given by

$$\begin{aligned} Q_c = \{[x, \#\{y \in Y \mid\ & K(x,y) \wedge \\ & \neg\exists z \in Z\ (f'(z) = g'(x) \;\wedge\; f''(z) = g''(y))\}] : x \in X\}, \end{aligned}$$

where predicate $K(x,y)$ is a conjunction of t negated tabular predicates and an arbitrary number of anti-join, equi-join, and tabular predicates. Assume that Query Q_c is well-typed, i.e.

$$TE, C \vdash_H Q_c : \mathit{strong\_smap}(b_1, \mathit{int}).$$

Query Q_c is transformed into the following equivalent query Q'.

$$\begin{aligned} Q' = \quad & \text{let} \\ & \quad Q_1 = \{[x, \#\{y \in Y \mid K(x,y)\}] : x \in X\} \\ & \quad Q_2 = \{[x, \#\{y \in Y \mid K(x,y) \wedge \\ & \qquad\quad \exists z \in Z\ (f'(z) = g'(x) \;\wedge\; f''(z) = g''(y))\}] : x \in X\} \\ & \text{in} \\ & \quad \{[x, Q_1(x) - Q_2(x)] : x \in X\} \end{aligned}$$

The proof that Query Q' is equivalent to Query Q_c is straightforward, and is omitted. The out-line of the type-derivation of query Q' can be seen from the following type-derivations

$$TE, C \vdash_H Q_1 : \mathit{strong\_smap}(b_1, \mathit{int})$$

$$TE, C \vdash_H Q_2 : \mathit{strong\_smap}(b_1, \mathit{int})$$

$$\begin{aligned} & TE[Q_1 \mapsto \mathit{strong\_smap}(b_1, \mathit{int}), Q_2 \mapsto \mathit{strong\_smap}(b_1, \mathit{int})], C \vdash_H \\ & \quad \{[x, Q_1(x) - Q_2(x)] : x \in X\} : \mathit{strong\_smap}(b_1, \mathit{int}) \end{aligned}$$

Finally, we consider the time complexity of Query Q'. Both queries Q_1 and Q_2 have only t negated tabular predicates (and an arbitrary number of anti-join, equi-join, and tabular predicates). Therefore, by the inductive hypothesis, both Q_1 and Q_2 are linear-time queries, and thus can be computed in $O(\#X + \#Y)$ time. It easily follows that Q' is also a linear-time query.

Case 2: Simple find-queries Once again the proof follows by induction on the number of negated tabular predicates.

The base case for 0 negated tabular predicates follows from Theorem 4.4.3.

Consider the case when we have $t+1$ negated tabular predicates. Let the Query Q_{sf} be given by

$$\begin{aligned} Q_{sf} = \{[x,y] \in X \times Y \mid & K(x,y) \wedge \\ & \neg\exists z \in Z\ (f'(z) = g'(x)\ \wedge\ f''(z) = g''(y))\}, \end{aligned}$$

where $K(x,y)$ is a conjunction of t negated tabular predicates and any number of anti-join, equi-join and tabular predicates. Assume that Query Q_{sf} is well-typed, i.e.

$$TE, C \vdash_H Q_{sf} : set(b_1 \times b_2).$$

Recall that for all negated tabular predicates of the form $\neg\exists z \in Z\ (f'(z) = g'(x)\ \wedge\ f''(z) = g''(y))$, the expressions $f'(z)$, and $g'(x)$ must be of some base type b_3, and $f''(z)$ and $g''(y)$ must be of some base type b_4.

Query Q_{sf} may be transformed into the following equivalent query Q'.

$$\begin{aligned} Q' = \ & \text{let} \\ & \quad Q_1 = \{[g'(x), x] : x \in X\} \\ & \quad Q_2 = \{[g''(y), y] : y \in Y\} \\ & \quad Q_3 = \{[f'(z), f''(z)] : z \in Z\} \\ & \text{in} \\ & \quad \{[x,y] \in X \times Y \mid K(x,y)\} - \uplus_{[p,q] \in Q_3}(Q_1\{p\} \times Q_2\{q\}) \end{aligned}$$

The proof that Q' is equivalent to Q_{sf} is straightforward and is omitted. The outline of the type-derivation for Q' can be seen from the following type-derivations

$$TE, C \vdash_H Q_1 : strong\_mmap(b_3, b_1)$$

$$TE, C \vdash_H Q_2 : strong\_mmap(b_4, b_2)$$

$$TE, C \vdash_H Q_3 : mmap(b_3, b_4)$$

$$\begin{aligned} & TE[Q_1 \mapsto strong\_mmap(b_3, b_1), Q_2 \mapsto strong\_mmap(b_4, b_2), \\ & \quad Q_3 \mapsto mmap(b_3, b_4)], C \vdash_H \{[x,y] \in X \times Y \mid K(x,y)\} - \\ & \quad \uplus_{[p,q] \in Q_3}(Q_1\{p\} \times Q_2\{q\}) : set(b_1 \times b_2) \end{aligned}$$

Finally, we consider the time complexity of computing Q'. The costs of computing Q_1, Q_2, and Q_3 are $O(\#X)$, $O(\#Y)$, and $O(\#Z)$ respectively. Let Q_4 denote the query $\{[x,y] \in X \times Y \mid K(x,y)\}$. Since expression $K(x,y)$ contains t negated tabular predicates, it follows from the inductive hypothesis that Query Q_4 can be computed in $O(\#X + \#Y + \#Q_4)$ time. Also, since expressions $g'(x)$ and $g''(y)$ are few-to-one, it follows that for any pair $[p,q]$, this size of $Q_1\{p\} \times Q_2\{q\}$ is bounded by some constant. Thus, the size of $\uplus_{[p,q] \in Q_3}(Q_1\{p\} \times Q_2\{q\})$ is $O(\#Z)$, and also that $\#Q_4 - \#Q' = O(\#Z)$, or in other words $\#Q_4 = O(\#Q' + \#Z)$. Therefore, it follows that Query Q' is also a linear-time query.

□

Theorem 4.4.5 *All count-queries and simple find-queries whose predicate $e(x,y)$ is just a conjunction of an arbitrary number of anti-join, equi-join, tabular, negated tabular, unary list predicates or negated unary list predicates are linear-time queries.*

Proof: We consider the case of simple-find queries first. The proof follows by induction on the number of unary list and negated unary list predicates.

The base case is when the number of unary list predicates and negated unary list predicates is 0. In this case, it follows from Theorem 4.4.1 that the simple find-query is a linear-time query.

Now, let us consider the case when Query Q_{sf} has $t+1$ unary list and negated unary list predicates. Let Q_{sf} be given by the query

$$Q_{sf} = \{[x,y] \in X \times Y \mid K(x,y) \ \wedge \ U(x)\},$$

where predicate $K(x,y)$ contains t unary list and negated unary list predicates, and an arbitrary number of anti-join, equi-join, tabular, and negated tabular predicates, and predicate $U(x)$ is a unary list or a negated unary list predicate. Let us consider the case when $U(x)$ is the predicate $\exists z \in Z \ (f(z) = g(x))$. Also assume that Query Q_{sf} is well-typed, i.e.

$$TE, C \vdash_H Q_{sf} : set(b_1 \times b_2),$$

and that expressions $f(z)$ and $g(x)$ are of some base type b_3 for $z \in Z$ and $x \in X$.

Query Q_{sf} can be transformed into the following equivalent query Q'.

$$\begin{array}{rl} Q' = & \text{let} \\ & \quad Q_1 = \{f(z) : z \in Z\} \\ & \quad X_1 = \{x \in X \mid g(x) \in Q_1\} \\ & \text{in} \\ & \quad \{[x,y] \in X_1 \times Y \mid K(x,y)\} \end{array}$$

The proofs that Query Q' is equivalent to Query Q_{sf}, and that Query Q' is well-typed are straight-forward and are omitted. From the inductive hypothesis, it follows that the query $\{[x,y] \in X_1 \times Y \mid K(x,y)\}$ is a linear-time query. Then, it follows easily that Q' is also a linear-time query.

The cases for other kinds of unary list and negated unary list predicates, and for count-queries are similar, and are omitted. □

Theorem 4.4.6 *All count-queries and simple find-queries are linear-time queries.*

Proof: We consider the case of simple find-queries first. The proof follows by induction on the number of unary and negated unary predicates in the predicate $e(x,y)$ of the simple find-query Q_{sf}.

The base case is when there are 0 unary and negated unary predicates. In this case, it follows from Theorem 4.4.5 that Query Q_{sf} is a linear-time query.

Now consider the case when $e(x,y)$ contains $t+1$ unary or negated unary predicates. Let Q_{sf} be given by the query

$$Q_{sf} = \{[x,y] \in X \times Y \mid K(x,y) \ \wedge \ U(x)\},$$

where predicate $K(x,y)$ contains t unary or negated unary predicates and an arbitrary number of other kinds of predicates, and predicate $U(x)$ is a unary or negated unary predicate involving x. Assume that Query Q_{sf} is well-typed, i.e.

$$TE, C \vdash_H Q_{sf} : set(b_1 \times b_2).$$

Query Q_{sf} can be transformed into the following equivalent query Q'.

$$\begin{array}{rl} Q' = & \text{let} \\ & \quad X_1 = \{x \in X \mid U(x)\} \qquad \text{in} \\ & \quad \{[x,y] \in X_1 \times Y \mid K(x,y)\} \end{array}$$

The proofs that Query Q' is equivalent to Query Q_{sf}, and that Query Q' is well-typed are straight-forward and are omitted. From the inductive hypothesis, it follows that the query $\{[x,y] \in X_1 \times Y \mid K(x,y)\}$ is a linear-time query. Then, it follows easily that Q' is also a linear-time query. □

4.5 Summary

In Section 4.3 we reduced the problem of showing that all LRCS count-queries and find-queries are linear-time queries to the problem of showing that all well-typed count-queries and *simple* find-queries are linear-time queries. In Section 4.4 we presented a sequence of theorems (Theorems 4.4.1-4.4.6) that finally led to a proof that all count-queries, and simple find-queries are linear-time queries. Thus, we proved that each LRCS query can be implemented in worst-case linear time with respect to the sum of the sizes of its input and the output.

In this chapter we have demonstrated how High SETL can be used as an algorithm specification language to shorten, simplify, and, finally, speedup an algorithmic result. Our LRCS translation is simple enough to consider a practical project to mechanically check its correctness. Of course, its correctness must include verification of the linear run-time complexity of translated queries. It would be interesting to find out if typability can be inferred mechanically or semi–automatically, in order to make our type system a more useful tool for algorithmic speedup.

Chapter 5

SQ$^+$

5.1 Introduction

In Chapters 2 and 3, we defined two algorithm specification languages Low SETL and High SETL. Low SETL is an imperative language with built-in finite sets and maps, and a small set of primitive set-theoretic operations for manipulating sets and maps. High SETL extends Low SETL with more abstract set-theoretic operations such as union, intersection, map-composition, etc. and more abstract expressions such as set-comprehension expressions. In Chapter 2 we showed that the well-typedness of a Low SETL program guarantees that it can be transformed into a pointer machine implementation in which each associative access operation can be performed in worst-case $O(1)$ time without the use of hashing. In Chapter 3 we defined a translation from High SETL to Low SETL and proved that the translation of well-typed High SETL programs always leads to well-typed Low SETL programs.

In this chapter we present a more abstract, functional set-theoretic language SQ$^+$ (where SQ stands for *Set Queries* and the + indicates its extension with *least and greatest fixed points*). Since SQ$^+$ is a functional language, we shall use the terms "SQ$^+$ program" and "SQ$^+$ function" inter-changeably. Similarly, we shall use the terms "High SETL expression" and "High SETL function" inter-changeably. SQ$^+$ comprises of the functional subset of High SETL (i.e. High SETL expressions) augmented with the least and greatest fixed point operators. The language SQ$^+$ presented here is only slightly different from the version presented by Cai and Paige [13]. However, one significant difference is that the language presented here is statically typed whereas the language presented by Cai and Paige was dynamically typed. Cai and Paige showed that their language was capable of expressing any partial-recursive function in a fixed point normal form. We show that our statically typed version can also do the same. One of the major contributions of Cai and Paige was the development of a non-deterministic iterative schema called *dominated convergence*, that together with finite differencing [70, 68, 72] was used for computing fixed points of monotone functions very efficiently. In Section 5.5 we will see how dominated convergence and finite differencing can be used to get efficient High SETL implementations of SQ$^+$ programs.

In this chapter we present our statically typed version of SQ$^+$, and define an operational semantics for typed SQ$^+$ in terms of High SETL. Unfortunately, we are not able to guarantee that all SQ$^+$ programs will be transformed into equivalent High SETL implementations. In general, even the problem of determining whether a fixed point exists is undecidable (see [13]). To get around this problem, we define a translation from SQ$^+$ to High SETL which is guaranteed to be correct only if certain semantic conditions hold for the expression whose fixed point is being computed. For example, one such semantic condition is that the expression should be monotonic. We present theorems which state sufficient conditions under which our translation from SQ$^+$ is guaranteed to produce a High SETL program that computes the least/greatest fixed point.

Although, we are not precisely able to characterize the subset of SQ$^+$ that can be transformed into equivalent High SETL implementations, we can at-least prove that the subset of SQ$^+$ programs that satisfy the semantic conditions that are sufficient to ensure a correct translation into High SETL, can express all partially recursive functions, i.e. is Turing Complete. We also define a type system for SQ$^+$, and prove that

$s, t, f, g, x, x_1, \ldots, v, v_1, \ldots$: Variable Names
$E, E_1, E_2, \ldots$: SQ$^+$ expressions
$K, K_1, K_2, \ldots$: Boolean valued SQ$^+$ expressions

$$\begin{array}{lcl} K & ::= & s \in t \mid s == t \mid \mathit{IsEmptySet}(s) \mid \mathit{IsEmptyMap}(f) \mid \\ & & (\exists x \in s \mid K_1) \mid (\forall x \in s \mid K_1) \mid \text{if } K_1 \text{ then } K_2 \text{ else } K_3 \text{ endif} \mid \\ & & \neg K_1 \mid K_1 \wedge K_2 \mid K_1 \vee K_2 \mid \text{let } v = E \text{ in } K \end{array}$$

$$\begin{array}{lcl} E & ::= & K \\ & & v \mid \ni v \mid v_1(v_2) \mid v_1\{v_2\} \mid v_1[i] \mid \text{om} \\ & & s \cup t \mid s \cap t \mid s - t \mid s \times t \mid \#s \mid s \uplus t \mid \mathit{ToSet}(f) \mid \mathit{ToMap}(s) \mid \\ & & \mathit{domain}(f) \mid \mathit{range}(f) \mid f \circ g \mid (f|_s) \mid f[s] \mid f^{-1} \mid f/g \mid \\ & & \cup_{x \in s} E \mid \cap_{x \in s} E \mid \uplus_{x \in s} E \mid \text{let } v = E_1 \text{ in } E_2 \mid \text{if } K \text{ then } E_1 \text{ else } E_2 \text{ endif} \mid \\ & & \{E : x_1 \in E_1, x_2 \in E_2, \ldots, x_n \in E_n \mid K\} \\ & & \text{LFP}_{\leq,w}(E,s) \mid \text{GFP}_{\leq,w}(E,s) \end{array}$$

Figure 5.1: Syntax of SQ$^+$

our translation of well-typed SQ$^+$ programs always leads to well-typed High SETL programs.

The High SETL implementations of SQ$^+$ specifications are based on the idea of computing Tarski iteration sequences [107, 29] , and may sometimes be inefficient. In [13], a non-deterministic iterative schema, called dominated convergence, was presented, which is a generalization of the "chaotic iteration" found in Kildall [56], Tanenbaum [103], and Cousot and Cousot [29] (restricted to finite iteration). It was also shown that dominated convergence could be adapted to a wide range of contexts to synthesize efficient algorithms and to provide succinct transformational correctness proofs. In this chapter we will describe sufficient conditions under which this non-deterministic algorithm schema may be adapted to give alternate (and possibly more efficient) well-typed High SETL implementations of SQ$^+$ functions.

5.2 Definition of SQ$^+$

SQ$^+$ is the functional subset of High SETL augmented with the least and greatest fixed point operators. Thus, every High SETL expression is also an SQ$^+$ expression. Let $\leq$ be a partial order. If E is an SQ$^+$ expression, then the expression $\text{LFP}_{\leq,w}(E,s)$ and $\text{GFP}_{\leq,w}(E,s)$ are also SQ$^+$ expressions and denote respectively the least fixed point of expression E with respect to variable s greater than or equal to w, and the greatest fixed point of expression E with respect to variable s less than or equal to w, where *least* and *greatest* are taken relative to the partial order $\leq$. If we omit the w in $\text{LFP}_{\leq,w}$ and $\text{GFP}_{\leq,w}$, then the w is by default taken to be respectively the minimum and the maximum element of the partial order $\leq$. The syntax of SQ$^+$ is given in Figure 5.1. As in the case of Low SETL and High SETL, language SQ$^+$ also contains ordinary arithmetic expressions of the form $E_1 + E_2$, $E_1 - E_2$, etc. that have been omitted in Figure 5.1 for the sake of brevity.

Unlike Low SETL and High SETL, SQ$^+$ does not contain imperative constructs such as assignment, and, the for and while loops. An interesting question arises about the expressiveness of SQ$^+$. Recall that both Low SETL and High SETL are Turing Complete. However, if we eliminate the *while* loop from Low and High SETL, the resulting languages are no longer Turing Complete. The functional subset of High SETL by itself is not Turing Complete, and in fact, is a subset of the language of primitive recursive functions [31]. Thus, the elimination of all imperative constructs from High SETL limits its expressive power. However, we can show that this loss of expressiveness can be made up for by adding the least and greatest fixed point operators, and that the resulting language SQ$^+$ is also Turing Complete.

In the next section, we briefly go over some preliminaries that are required to state and prove theorems outlining sufficient conditions under which the fixed points of SQ$^+$ expressions are guaranteed to exist, and can be correctly computed using our translation from SQ$^+$ to High SETL.

5.3 Preliminaries

We first review a few basic definitions and concepts from lattice theory (taken from [13]) that underlie the main results. This background material may be found in any introductory text on lattice theory; for example, Birkhoff [8], or Gratzer [45]. Next we present some basic theory underlying fixed point computation that is later used to give an operational semantics to SQ$^+$ in terms of High SETL.

5.3.1 Definitions

A *poset* $(L, \leq)$ is a reflexive, transitive, anti-symmetric binary relation $\leq$ on a set L. A poset $(L, \leq)$ has a minimal element y iff $\forall x \in L(x \leq y \implies x = y)$, and a minimum element $\mathbf{0}$ iff $\forall x \in L(\mathbf{0} \leq x)$. Maximal and maximum elements are defined analogously. We use $\mathbf{1}$ to denote the maximum element. A chain for a poset $(L, \leq)$ is a strictly increasing or decreasing sequence of elements of L. A poset $(L, \leq)$ is said to have an *ascending* (respectively *descending*) chain condition, abbreviated ACC (respectively DCC), iff there are no infinite increasing(decreasing) chains in L. Let $w \in L$. An element $a \in L$ is w^+*-finite* if the set $\{x \in L | w \leq x \leq a\}$ satisfies ACC. Similarly, a is w^-*-finite* if the set $\{x \in L | a \leq x \leq w\}$ satisfies DCC. Let $a, b, c \in L$. If $a \leq c$, and $b \leq c$, then c is an upper bound for a and b. An upper bound c for a and b is a least upper bound if all other upper bounds x of a and b are greater than or equal to c. The least upper bound of a and b is also called the *join* of a and b, and is denoted by $a \vee b$. If $a \vee b$ is defined and belongs to L for all $a, b \in L$, then $(L, \leq)$ is called a *join semilattice*. Lower bounds, greatest lower bound (also called *meet* and denoted by $\wedge$), and meet semilattices are defined analogously. If a poset is both a join and a meet semilattice, then it is a *lattice*.

Let $f : T \longrightarrow Q$ be a function from poset $(T, \leq)$ to poset $(Q, \leq')$. Function f is said to be monotone (respectively antimonotone) if for every two elements $x, y \in domain(f)$ such that $x \leq y$, it is the case that $f(x) \leq' f(y)$ (respectively $f(y) \leq' f(x)$). If poset $(Q, \leq')$ is the same as poset $(T, \leq)$, then we say that f is *inflationary* (respectively *deflationary*) at x if $x \leq f(x)$ (respectively $f(x) \leq x$). Function f is inflationary (deflationary) if it is inflationary (deflationary) at every point in its domain.

Next we describe a basic theory of fixed points. For succinctness, we only describe results for least fixed points. Dual results hold for the greatest fixed points.

5.3.2 Basic Theory

Most of the transformations from SQ$^+$ to High SETL are based on the following theorem and corollary, which can be derived from Tarski's more general Theorem [107] or its constructive reformulation due to Cousot and Cousot [29].

Theorem 5.3.1 *(Paige and Henglein [67]) Let $(T, \leq)$ be a poset with a unique minimum element designated $\mathbf{0}$. Let $f : T \longrightarrow T$ be a monotone computable function. Then the set $\{f^i(\mathbf{0}) : i = 0, 1, \ldots\}$ is finite iff there exists an integer $k \geq 0$ such that $f^k(\mathbf{0}) = LFP_{\leq}(f(x), x)$.*

Corollary 5.3.2 *(Cai and Paige [13]) Let $(T, \leq)$ be a poset. Let $f : T \longrightarrow T$ be a monotone computable function, $w \in T$, and $w \leq f(w)$. Then the set $\{f^i(w) : i = 0, 1, \ldots\}$ is finite iff there exists an integer $k \geq 0$ such that $f^k(w) = LFP_{\leq, w}(f(x), x)$.*

Theorem 5.3.3 *(Cai and Paige [13]) Let $(T, \leq)$ be a poset. Let $f : T \longrightarrow T$ be a monotone computable function, $w \in T$, and $w \leq f(w)$. Then the set $\{f^i(w) : i = 0, 1, \ldots\}$ is finite if any one of the following conditions holds:*

1. *either of the sets $\{x \in T | w \leq x\}$ or $\{x \in range(f) | w \leq x\}$ is finite.*
2. *either of the sets $\{x \in T | w \leq x\}$ or $\{x \in range(f) | w \leq x\}$ satisfies ACC.*

$$\frac{TE, C \vdash_H E : \tau}{TE, C \vdash_S E : \tau} \tag{5.2}$$

$$\frac{\begin{array}{c} TE, C \vdash_S w : \tau \\ TE[x \mapsto \tau], C \vdash_S E : \tau \end{array}}{TE, C \vdash_S \mathrm{LFP}_{\leq, w}(E, x) : \tau} \tag{5.3}$$

$$\frac{\begin{array}{c} TE, C \vdash_S w : \tau \\ TE[x \mapsto \tau], C \vdash_S E : \tau \end{array}}{TE, C \vdash_S \mathrm{GFP}_{\leq, w}(E, x) : \tau} \tag{5.4}$$

Figure 5.2: Type Rules for SQ^+

3. *f has a w^+-finite fixed point greater than or equal to w either with respect to $(T, \leq)$ or to the poset $(range(f), \leq)$.*

4. *The poset $(T, \leq)$ is a join semilattice and function f has the form $f(x) = x \vee g(x)$, where the set $\{g(x) : x \in T | w \leq x\}$ is finite.*

5. *The poset $(T, \leq)$ is a join semilattice and function f has the form $f(x) = x \vee g(x)$, g is monotone, and the set $\{g(x) : x \in T | w \leq x\}$ satisfies ACC.*

If f is monotone and inflationary at w, then according to Corollary 5.3.2, a straightforward algorithm to compute $\mathrm{LFP}_{\leq, w}(f)$ initializes p to w, and then repeatedly computes the new value of p by assigning $f(p)$ to p until p does not change. Theorem 5.3.3 gives sufficient conditions under which such an algorithm is guaranteed to terminate with the correct answer.

In the next section, we define a static type system for SQ^+. Next, we define an operational semantics for typed SQ^+ queries in terms of High SETL. We prove that the High SETL implementations of well-typed SQ^+ queries are also well-typed. Using the theory developed in this section, we are able to infer that if the conditions of Corollary 5.3.2 and Theorem 5.3.3 are met, then the High SETL implementations are guaranteed to correctly compute the least/greatest fixed point.

5.4 Type System and Operational Semantics for SQ^+

The set of types *Type* is the same as that for High SETL, i.e. the types derivable from τ in Grammar 5.1.

$$\begin{array}{lcl} \tau & ::= & \mathit{bool} \mid \sigma \mid \mathit{strong\_set}(b) \mid \mathit{strong\_smap}(b, \sigma) \mid \mathit{strong\_mmap}(b, \sigma) \mid \tau_1 \times \tau_2 \times \ldots \times \tau_k \\ \sigma & ::= & \mathit{int} \mid b \mid \mathit{set}(\sigma_1) \mid \mathit{smap}(\sigma_1, \sigma_2) \mid \mathit{mmap}(\sigma_1, \sigma_2) \mid \sigma_1 \times \sigma_2 \times \ldots \times \sigma_k \end{array} \tag{5.1}$$

5.4.1 Type Rules for SQ^+

In Figure 5.2, we give type inference rules for SQ^+ using judgments of the form $TE, C \vdash_S E : \tau$. We say that an SQ^+ expression E is well-typed in type environment TE with a set of subtype constraints C, if there exists a type-derivation for the judgment $TE, C \vdash_S E : \tau$. Rule 5.2 says that every well-typed High SETL expression is a well-typed SQ^+ expression. Rules 5.3 and 5.4 say that if w is a well-typed SQ^+ expression of type τ, and assuming that variable x is of type τ, expression E is also a well-typed SQ^+ expression of type τ, then the expressions $\mathrm{LFP}_{\leq, w}(E, x)$ and $\mathrm{GFP}_{\leq, w}(E, x)$ are also well-typed SQ^+ expressions of type τ.

$$\begin{array}{rcl}
Eq_{int}(x,y) & = & (x == y) \\
Eq_b(x,y) & = & (x == y) \text{ for any base type } b \\
Eq_{\sigma_1 \times \ldots \times \sigma_n}(x,y) & = & Eq_{\sigma_1}(x[1],y[1]) \wedge \ldots \wedge Eq_{\sigma_n}(x[n],y[n]) \\
Eq_{set(\sigma)}(x,y) & = & (\#x == \#y) \wedge (\forall z \in x \mid \exists z' \in y \mid Eq_\sigma(z,z')) \\
Eq_{strong\_set(b)}(x,y) & = & (\#x == \#y) \wedge (\forall z \in x \mid \exists z' \in y \mid Eq_b(z,z')) \\
Eq_{smap(\sigma_1,\sigma_2)}(x,y) & = & \text{let } u = \mathit{ToSet}(x) \text{ in let } v = \mathit{ToSet}(y) \text{ in } Eq_{set(\sigma_1 \times \sigma_2)}(u,v) \\
Eq_{strong\_smap(b_1,\sigma_2)}(x,y) & = & \text{let } u = \mathit{ToSet}(x) \text{ in let } v = \mathit{ToSet}(y) \text{ in } Eq_{set(b_1 \times \sigma_2)}(u,v) \\
Eq_{mmap(\sigma_1,\sigma_2)}(x,y) & = & \text{let } u = \mathit{ToSet}(x) \text{ in let } v = \mathit{ToSet}(y) \text{ in } Eq_{set(\sigma_1 \times \sigma_2)}(u,v) \\
Eq_{strong\_mmap(b_1,\sigma_2)}(x,y) & = & \text{let } u = \mathit{ToSet}(x) \text{ in let } v = \mathit{ToSet}(y) \text{ in } Eq_{set(b_1 \times \sigma_2)}(u,v)
\end{array}$$

Figure 5.3: Definition of $Eq_\tau(x,y)$

5.4.2 High SETL implementations of SQ$^+$

In this section we will ascribe an operational semantics to SQ$^+$ in terms of High SETL. We will also show that well-typed SQ$^+$ functions have well-typed High SETL implementations. In order to implement SQ$^+$ programs in High SETL, we need to define a class of High SETL expressions Eq_τ (defined in Figure 5.3) inductively on τ. For each type τ, the functionality of the High SETL expression $Eq_\tau(x,y)$ is to compare variables x and y of type τ for equality. It is easy to prove that if $TE = \{[x,\tau],[y,\tau]\}$ and C contains subtype constraints corresponding to the base types appearing in type τ, then $TE, C \vdash_H Eq_\tau(x,y) : \mathit{bool}$, i.e. for all types τ and variables x and y of type τ, the High SETL expression $Eq_\tau(x,y)$ is well-typed.

Now, we are ready to define the High SETL implementations of well-typed SQ$^+$ programs. Let E be a well-typed SQ$^+$ expression having a type derivation for the judgment $TE, C \vdash_S E : \tau$. We define the High SETL implementation of $v := E$ by rule induction on the type derivation of expression E.

- Case Rule 5.2: In this case since E is itself a well-typed High SETL expression, the command `v := E` is a valid High SETL implementation.
- Case Rule 5.3: By inductive hypothesis, assume that P_w is a High SETL implementation of $x := w$, and that P_E is a High SETL implementation of $x := E$. The High SETL implementation of $x :=$ LFP$_{\leq,w}(E,x)$ (shown below) is obtained by first assigning w to x, and repeatedly assigning E to x until x does not change.

```
$P_w$;    -- x := w
xprev := x;
$P_E$;    -- x := E
while $\neg$ $Eq_\tau$(x,xprev) loop
    xprev := x;
    $P_E$;    -- x := E
endloop
```

- Case Rule 5.4: This case is the dual case of Rule 5.3.

Moreover, we can show that High SETL implementations of well-typed SQ$^+$ programs are well-typed (Theorem 5.4.1).

Theorem 5.4.1 *Let E be a well-typed SQ$^+$ program having a type derivation for the judgment $TE, C \vdash_S E$: τ. Let P_E be the High SETL implementation of $v := E$. Then, there exists a type environment $TE' \supseteq TE$ with $TE'(v) = \tau$ such that $TE', C \vdash_H P_E$.*

Proof: The proof follows by a simple rule-induction on the type derivation for the judgment $TE, C \vdash_S E : \tau$. □

5.4.3 A Note on the Expressiveness of SQ$^+$

In Section 5.4 we defined High SETL implementations for SQ$^+$ expressions. We proved that these implementations are guaranteed to compute the least or greatest fixed point if the conditions of Corollary 5.3.2 and Theorem 5.3.3 are satisfied. Although, we cannot make such a guarantee if the conditions of Corollary 5.3.2 and Theorem 5.3.3 are not satisfied, we can at least show that every partially-computable function can be expressed as a well-typed SQ$^+$ program that computes the least fixed point of a monotonic and inflationary function i.e. a function satisfying the conditions of Corollary 5.3.2.

Theorem 5.4.2 *Every partially computable function $f : N \longrightarrow N$ over the natural numbers N can be expressed as a well-typed SQ$^+$ program $LFP_{\leq,w}(E,x)$ where expression E is monotonic and inflationary at w.*

Proof Sketch: Cai and Paige prove a similar result in the Appendix of [13] by showing that a Turing Machine can be simulated by an SQ$^+$ program. It only remains to show that the Turing Machine may be simulated by a **well-typed** SQ$^+$ program. It is not difficult to prove that the SQ$^+$ program that Cai and Paige use to simulate the Turing Machine is in fact a well-typed SQ$^+$ program. $\square$

5.4.4 Time Complexity of SQ$^+$ programs

Since SQ$^+$ is Turing Complete, the problem of determining whether an implementation of an SQ$^+$ program terminates is undecidable. However, if the conditions of Theorem 5.3.3 are satisfied, then the High SETL implementation of the SQ$^+$ program is guaranteed to terminate. Using Theorem 5.3.3, it is possible to bound the number of iterations of the outermost while loop in the High SETL implementation. Let us re-examine the 5 conditions of Theorem 5.3.3 in order to compute a bound on the number of iterations of the High SETL implementation.

1. either of the sets $\{x \in T | w \leq x\}$ or $\{x \in \mathit{range}(f) | w \leq x\}$ is finite: In this case the number of iterations of the outermost while loop is bounded by $\#\{x \in T | w \leq x\}$ or $\#\{x \in \mathit{range}(f) | w \leq x\}$ respectively.

2. either of the sets $\{x \in T | w \leq x\}$ or $\{x \in \mathit{range}(f) | w \leq x\}$ satisfies ACC: In this case the number of iterations of the outermost while loop is bounded by the size of the largest ascending chain in $\{x \in T | w \leq x\}$ or $\{x \in \mathit{range}(f) | w \leq x\}$.

3. f has a w^+-*finite* fixed point greater than or equal to w either with respect to T or to the poset $(\mathit{range}(f), \leq)$: Let a be the w^+-*finite* fixed point greater than or equal to w. In this case the number of iterations of the outermost while loop is bounded by the size of the largest ascending chain in $\{x \in T | w \leq x \leq a\}$ or $\{x \in \mathit{range}(f) | w \leq x \leq a\}$.

4. The poset $(T, \leq)$ is a join semilattice and function f has the form $f(x) = x \vee g(x)$, where the set $\{g(x) : x \in T | w \leq x\}$ is finite: In this case the number of iterations of the outermost while loop is bounded by $\#\{g(x) : x \in T | w \leq x\}$.

5. The poset $(T, \leq)$ is a join semilattice and function f has the form $f(x) = x \vee g(x)$, where the set $\{g(x) : x \in T | w \leq x\}$ satisfies ACC: In this case the number of iterations of the outermost while loop is bounded by the size of the largest ascending chain in $\{g(x) : x \in T | w \leq x\}$.

Let x_f denote $\text{LFP}_{\leq,w}(E,x)$. Then, if C_E is the cost of computing expression E, and C_{Eq} is the cost of comparing two values of type τ that are no larger than the size of x_f for equality, then the cost of computing $\text{LFP}_{\leq,w}(E,x)$ is bounded by $O(n \times (C_E + C_{Eq}))$ where n is an upper bound on the number of iterations of the while loop.

Expression	Parameters that satisfy properties			
	Monotone	Antimonotone	Inflationary	Deflationary
$x \in s$	s			
$y \in f\{x\}$	f			
$\exists x \in s \mid k(x)$	s and k			
$\forall x \in s \mid k(x)$	k	s		
$\neg k$		k		
$k \wedge p$	k and p			k and p
$k \vee p$	k and p		k and p	
$f\{x\}$	f			
$s \cup t$	s and t		s and t	
$s \cap t$	s and t			s and t
$s - t$	s	t		s
$s \times t$	s and t			
$\#s$	s			
$domain(f)$	f			
$range(f)$	f			
$f[s]$	f and s			
f^{-1}	f			
$\{x \in s \mid k(x)\}$	s and k			s

Table 5.1: Basic monotone, antimonotone, inflationary, deflationary functions

5.4.5 Pragmatic Considerations

The correctness of the High SETL implementation of SQ$^+$ programs depends on properties such as monotonicity, which are in general undecidable (see [13]). One practical approach to proving these properties is to define stronger syntactically defined decidable properties that imply these undecidable semantic properties. This can be done by specifying properties using a formal system of pattern-directed inductive definitions similar to Sintzoff's method of valuations [96].

Inflationary and Deflationary

Although these properties are undecidable, a recursive class of inflationary and deflationary SQ$^+$ expressions can be generated by composition from a predefined collection of basic inflationary and deflationary functions (See Table 5.1) by using the rule that the composition of two inflationary functions is inflationary and that the composition of two deflationary functions is deflationary. Moreover, the function $f(s) = s \vee g(s)$ is always inflationary, and the function $f(s) = s \wedge g(s)$ is always deflationary.

f	g	$f \circ g$
inflationary	inflationary	inflationary
deflationary	deflationary	deflationary

Monotone and Antimonotone

We can recognize a large class of monotone and antimonotone computable functions as follows:

1. Basic monotone and antimonotone functions shown in Table 5.1.
2. Composition:

f	g	$f \circ g$
monotone	monotone	monotone
antimonotone	monotone	antimonotone
monotone	antimonotone	antimonotone
antimonotone	antimonotone	monotone

3. if $f(x,y)$ is monotone (respectively antimonotone) in each parameter, then $g(x) = f(x,x)$ is monotone (respectively antimonotone) in x.

4. if $f(x,y)$ is a function with a finite range, is monotone in each parameter x and y, and inflationary in x, then the function $g(w,y) = \mathrm{LFP}_{\leq,w}(f(x,y),x)$ and $h(w,y) = \mathrm{GFP}_{\leq,w}(f(x,y),x)$ are monotone in w and y and have finite ranges.

5.5 Dominated Convergence

The iterative High SETL implementation of SQ$^+$ programs defined in Section 5.4.2 may be highly inefficient because of the potentially costly redundancy in the recomputation of expression E in each iteration. The following theorem provides another form of non-deterministic iteration (called Dominated Convergence), which when used in conjunction with finite differencing [70, 68], can lead to much more efficient implementations of SQ$^+$ programs.

Theorem 5.5.1 *(Cai and Paige [13]) Let $(T, \leq)$ be a poset. Let $f : T \longrightarrow T$ be a monotone computable function, $w \in T$, and $w \leq f(w)$. Let $s_0, \dots, s_i, \dots$ be any sequence such that*

- $s_0 = w$;
- $s_{i+1} \in \{x \in T | s_i \leq x \leq f(s_i)\}, i = 0, 1, \dots$

Then we conclude the following:

1. *If there exists an integer $k \geq 0$ such that $s_k = f(s_k)$, then $s_k = LFP_{\leq,w}(f)$.*
2. *If $LFP_{\leq,w}(f)$ is w^+-finite, and if $s_i < s_{i+1}$ whenever $s_i \neq f(s_i)$, then there exists a $k \geq 0$ such that $s_k = f(s_k)$.*

It is easy to see that the sequence $(w, f(w), f^2(w), \dots)$ is just a special case of the sequence $(s_0, s_1, s_2, \dots)$. Cai and Paige [13] formalized the way in which general sequences $s_0, s_1, s_2, \dots$ satisfying the conditions of Theorem 5.5.1 may be generated in the following way.

Let $(T, \leq)$ be a poset and S be a non-empty set. Let $\Delta : T \times T \longrightarrow 2^S$ (called the workset function) be such that $\Delta(q,p) = \{\} \iff q \leq p$ for all $[q,p] \in T \times T$. Let $\delta : T \times S \longrightarrow T$ (called the increment function) be such that $p \leq \delta(p,z)$ for all $[p,z] \in T \times S$. The two functions Δ and δ are said to be feasible relative to each other if

$$\forall z \in \Delta(q,p)(p \ < \ \delta(p,z) \ \leq \ p \vee q).$$

Then, it is not difficult to prove that if function f satisfies the conditions of Theorem 5.5.1, $\mathrm{LFP}_{\leq,w}(f)$ is w^+*-finite*, and functions Δ and δ are feasible relative to each other, then $\mathrm{LFP}_{\leq,w}(f)$ may be computed by the following piece of code

```
p := w;
while ∃z ∈ Δ(f(p), p) loop
    p := δ(p, z);                                   (5.5)
end loop
```

It is easy to prove that the successive values assigned to p in Program 5.5 form a sequence $s_0, s_1, s_2, \dots$ satisfying the conditions of Theorem 5.5.1. Then it follows from the fact that $\mathrm{LFP}_{\leq,w}(f)$ is finite that Program 5.5 terminates with the value $\mathrm{LFP}_{\leq,w}(f)$. We illustrate the above ideas through the following two examples.

Example 1: (Graph Reachability) The problem is to find the set of vertices s reachable along paths in a directed graph G from an arbitrary set of vertices w. Let Graph G be represented by the set of vertices v and the finite set of edges e, where each edge is a pair of vertices. Let e be implemented as a multi-valued map, so that for each vertex x, the term $e\{x\}$ represents the set of vertices adjacent to x. Recall that $e[s] = \cup_{x \in s} e\{x\}$. Then, the graph reachability problem can be expressed as the following SQ$^+$ program

$$\mathrm{LFP}_{\subseteq, w}(s \cup e[s], s) \tag{5.6}$$

Since $\cup$ is the *join* operator for the finite lattice $(2^v, \subseteq)$, the function $s \cup e[s]$ is inflationary and Condition 1 of Theorem 5.3.3 is satisfied.. The function $s \cup e[s]$ is also monotonic in s. Thus, it follows that Function 5.6 may be computed by the following program.

```
s := w;
sprev := s;
s := s $\cup$ e[s];
while s $\neq$ sprev loop
    sprev := s;
    s := s $\cup$ e[s]
endloop
```

Alternately, it can be seen that the functions $\Delta(q,p) = q - p$ and $\delta(p,z) = p \; with \; z$ (choosing T to be 2^v and S to be v) are respectively workset and increment functions (because $q \subseteq p \implies q - p = \{\}$ and $p \subseteq p \; with \; z$) that are feasible relative to each other. It follows from Theorem 5.5.1 and the preceding discussion that the following is another implementation of graph reachability.

```
s := w;
while $\exists$ z $\in$ (s $\cup$ e[s]) - s loop
    s with:= z
endloop
```

Example 2: (Cycle Detection) Once again we assume that a directed graph is represented by a set of vertices v and multi-valued map e such that for any vertex x, $e\{x\}$ is the set of vertices adjacent to x. Graph G contains a cycle iff the largest set of vertices $s \subseteq v$ each containing a successor belonging to s is non-empty. This can be expressed as the following SQ$^+$ program

$$\neg \mathit{IsEmptySet}(\mathrm{GFP}_{\subseteq}((s - \{x \in s | \mathit{IsEmptySet}(e\{x\} \cap s)\}), s)), \tag{5.7}$$

or as the following more readable specification

$$\mathrm{GFP}_{\subseteq}((s - \{x \in s | e\{x\} \cap s = \{\}\ \}), s) \neq \{\}.$$

Once again the lattice $(2^v, \subseteq)$ is finite and therefore Condition 1 of Theorem 5.3.3 is satisfied. The function $(s - \{x \in s | e\{x\} \cap s = \{\}\ \})$ is monotone in s and deflationary. Therefore, it follows that Program 5.7 may be computed by the following program.

```
s := v;
sprev := s;
s -:= { x $\in$ s | IsEmptySet(e{x} $\cap$ s)};
while (s $\neq$ sprev) loop
    sprev := s;
    s -:= { x $\in$ s | IsEmptySet(e{x} $\cap$ s)}
endloop
```

Alternately, it can be seen that the functions $\Delta(q,p) = p - q$ and $\delta(p,z) = p\ less\ z$ (choosing T to be 2^v and S to be v) are respectively workset and increment functions (because $p \subseteq q \implies p - q = \{\}$ and $p\ less\ z \subseteq p$) that are feasible relative to each other. It follows from Theorem 5.5.1 and the preceding discussion that the following is another implementation of cycle testing.

```
s := v;
while ∃ z ∈ { x ∈ s | IsEmptySet(e{x} ∩ s)} loop
    s less:= z
endloop
```

5.6 Concluding Remarks

In this chapter we have defined a statically typed version of SQ$^+$, and defined an operational semantics for a subset of SQ$^+$ in terms of High SETL. We proved that the subset of SQ$^+$ with operational semantics is Turing Complete. We have also described how a non-deterministic iteration strategy called dominated convergence (Cai and Paige [13]) can be used for computing least and greatest fixed points.

In order to use dominated convergence, we need the workset and increment functions Δ and δ that are feasible relative to each other. Unfortunately, there is no mechanical way of generating a suitable pair of such functions. We shall partially address this problem in the next chapter, where we shall see how to discover this pair of functions for an interesting class of SQ$^+$ expressions.

In the examples above, we saw the different implementations using either the ordinary Tarski iteration or dominated convergence. We have still not made it clear how dominated convergence can be more efficient than the Tarski iteration. It turns out that dominated convergence when used together with finite differencing, can lead to much more efficient implementations of SQ$^+$ programs. We shall see this in detail in the next chapter where we define a **linear-time** subset of SQ$^+$, i.e a language that comprises of a subset of SQ$^+$ expressions which can all be guaranteed to have linear time[1] implementations using dominated convergence and finite differencing.

[1]linear in the sum of the sizes of the input and the output

Chapter 6

A Linear Time Language

6.1 Introduction

In Chapter 5, we defined a functional language SQ$^+$ containing the functional subset of High SETL and least and greatest fixed point operators. We gave an operational semantics to SQ$^+$ in terms of High SETL, and showed that if certain semantic properties are satisfied, then the High SETL implementations of SQ$^+$ programs are guaranteed to correctly compute the least/greatest fixed point. The naive translation of SQ$^+$ to High SETL is based on the computation of Tarski iteration sequences. We also described how a non-deterministic iteration schema, called dominated convergence, could be used as an alternative to Tarski iteration.

In this chapter we define a language $L_{IO} \subseteq$ SQ$^+$ containing expressions that are guaranteed to have implementations with time complexity linear in the sum of their *input and output sizes.* This language is based on a similar linear-time language presented by Cai and Paige in [12]. We start by defining simple expressions that have linear-cost implementations. Next, we define rules which determine when the composition of linear-cost expressions is also of linear cost. In these cases, we only consider the *static* complexity of expressions, i.e. the cost of fresh evaluation of expressions from scratch. The next step is to determine which fixed point expressions can also be computed in linear time. Unfortunately, the naive evaluation of SQ$^+$ expressions using Tarski iteration is too expensive, and most SQ$^+$ expressions cannot be computed in linear time using Tarski iteration. The class of SQ$^+$ expressions that can be computed in linear time is greatly expanded by using finite differencing together with dominated convergence.

Finite differencing involves substituting the fresh evaluation of expressions in each iteration by their more inexpensive incremental counterparts. We determine the cost of differential calculations by associating precise amortized complexities with an eager strategy for maintaining equality invariants $e = f(x_1, \dots, x_k)$ within worst-case sequences of modifications to variables $x_1, \dots, x_k$; i.e. each time a modification to $x_1, \dots, x_k$ occurs, variable e is updated to re-establish the invariant. This amortized complexity of maintaining the invariant is called the *dynamic complexity* of the expression $f(x_1, \dots, x_k)$. We start by associating precise amortized complexities with the maintenance of basic invariants. Next, we define closure rules that allow us to determine the cost of maintaining collections of interdependent invariants. Thus, we are able to define a rich class of invariants that can be maintained differentially with precise amortized complexities.

We also show how dominated convergence, together with differential evaluation of expressions, leads to more efficient algorithms for a large class of problems involving fixed point computations. In order to demonstrate the expressiveness of L_{IO}, we consider examples of simple textbook algorithmic problems such as Graph Reachability, Cycle Testing, Constant Propagation etc. in this chapter, and show that all these problems are expressible in L_{IO}, and therefore, can be transformed into linear-time implementations.

Later in this thesis, we go beyond textbook examples by tackling two sophisticated algorithmic problems and show how the use of L_{IO} helps in explaining and improving existing algorithms. In Chapter 7, we deal with the problem of solving fixed-points of equations on transition systems. We demonstrate how the use of SQ$^+$ as a specification language leads to the derivation of a linear-time algorithm that is much easier to understand than the original linear-time algorithm proposed by Arnold and Crubille [5]. This example serves to illustrate the usefulness of SQ$^+$ as a tool for understanding complex algorithms involving fixed point computations. In Chapter 8, we deal with the problem of computing intra-procedural point-wise may-alias information for an imperative programming language like C. We demonstrate how the use of SQ$^+$ as a specification language for the original problem leads to the discovery of a new $O(N^3)$ (where N is the size of the program) time algorithm for computing may-alias information, which significantly improves the previously best known $O(N^5)$ time algorithm [51].

6.2 Terminology and Notation

In Section 6.3, we define a subset L_{IO} of typed SQ$^+$ containing expressions which are guaranteed to have worst-case linear time[1] implementations. For the rest of this chapter, we shall use the terms *expression* and *function* inter-changeably. We shall use the term *Linear Cost Expression* to denote an expression in L_{IO}.

We shall find it convenient to make the free variables $x_1, \ldots, x_k$ of an expression E explicit by representing expression E as $E(x_1, \ldots, x_k)$. For example, expression $E_1 = s \cup t$ will be represented as $E_1(s,t)$, and expression $E_2 = s \cup e[s]$ will be represented as $E_2(s,e)$. Note that the representation of an arbitrary expression E having a free variable x as $E(x)$ should not to be confused with the SQ$^+$ map application $f(x)$ where f is a single-valued map. In order to avoid confusion, we shall use lower case letters for all variables in an SQ$^+$ program, and upper case letters otherwise.

Every expression in L_{IO} is represented by a quadruple $\langle TE, C, E, \tau\rangle$, where

- TE, the type environment is a map from variables to types,
- C, the set of subtype constraints, is admissible[2], and contains subtype constraints for all base types appearing in TE.
- Expression E is a well-typed SQ$^+$ expression of type τ under type environment TE, and the set of subtype constraints C, i.e. there exists a type derivation for the type judgment

$$TE, C \vdash_s E : \tau.$$

Sometimes we will not explicitly state the type environment TE, set of subtype constraints C, and type τ with a linear-cost expression, but will just annotate the free variables occurring in the expression with their respective types. For example, when we say that expression $E(x_1 : \tau_1, \ldots, x_k : \tau_k) : \tau$ is a linear-cost expression, we will mean that $\langle TE, C, E, \tau\rangle$ is a linear-cost expression, where $TE = [x_1 : \tau_1, \ldots, x_k : \tau_k]$, C is any admissible set of subtype constraints containing subtype constraints for all base types appearing in TE, and τ is the type of the output.

6.3 Linear-Cost Language

In this section we define the linear-cost language L_{IO}. We start by defining simple linear-cost expressions in Section 6.3.1. In Section 6.3.2 we describe sufficient conditions under which linear-cost expressions may be composed to form other linear-cost expressions. For these expressions, a static complexity measure is used. In Section 6.3.3, we look at the cost of computing expressions involving fixed-point computations.

6.3.1 Simple Linear-Cost Expressions

In Table 6.1 we define some simple linear-cost expressions. As mentioned in the previous section, a linear-cost expression E must be a well-typed SQ$^+$ expression associated with a type environment TE, an admissible set of subtype constraints for base types appearing in TE, and a type τ. In Table 6.1 we include both type

[1]linear in the sum of the input and output sizes

[2]a set of subtype constraints is admissible if every base type is associated with a unique constraint, and the subtype constraint graph is acyclic

Expr. E	**Type Environment** TE	**Type** τ	**Cost**
$s \in t$	$s : b,\ t : strong\_set(b)$	$bool$	$O(1)$
$s == t$	$s : \mu,\ t : \mu$	$bool$	$O(1)$
$IsEmptySet(s)$	$s : set(\sigma)$ or $strong\_set(b)$	$bool$	$O(1)$
$IsEmptyMap(f)$	$f : smap(\sigma_1, \sigma_2)$ or $mmap(\sigma_1, \sigma_2)$ or $strong\_smap(b_1, \sigma_2)$ or $strong\_mmap(b_1, \sigma_2)$	$bool$	$O(1)$
$\neg K_1$	TE_1	$bool$	$O(1)$
$K_1 \wedge K_2$	$TE_1 \cup TE_2$ assuming it is a valid type env.	$bool$	$O(1)$
$K_1 \vee K_2$	$TE_1 \cup TE_2$ assuming it is a valid type env.	$bool$	$O(1)$
$\exists x \in s \mid K_1(x)$	$TE_1 \cup \{s : set(\sigma)$ or $strong\_set(b)\}$	$bool$	$O(\#s)$
$\forall x \in s \mid K_1(x)$	$TE_1 \cup \{s : set(\sigma)$ or $strong\_set(b)\}$	$bool$	$O(\#s)$
$\ni s$	$s : set(\sigma)$	σ	$O(1)$
$\ni s$	$s : strong\_set(b)$	b	$O(1)$
$f(x)$	$f : strong\_smap(b, \sigma_1),\ x : b$	σ_1	$O(1)$
$f\{x\}$	$f : strong\_mmap(b, \sigma_1),\ x : b$	$set(\sigma_1)$	$O(1)$
$f[i]$	$f : \sigma_1 \times \ldots \times \sigma_k,\ i \leq k$	σ_i	$O(1)$
$s \cup t$	$s : set(b)$ or $strong\_set(b)$ $t : set(b)$ or $strong\_set(b)$	$set(b)$ or $strong\_set(b)$	$O(\#s + \#t)$
$s \cap t$	$s : set(b)$ or $strong\_set(b)$ $t : set(b)$ or $strong\_set(b)$	$set(b)$ or $strong\_set(b)$	$O(\#s + \#t)$
$s - t$	$s : set(b)$ or $strong\_set(b)$ $t : set(b)$ or $strong\_set(b)$	$set(b)$ or $strong\_set(b)$	$O(\#s + \#t)$
$s \times t$	$s : set(\sigma_1),\ t : set(\sigma_2)$	$set(\sigma_1 \times \sigma_2)$	$O(\#(s \times t))$
$s \times t$	$s : strong\_set(b_1),\ t : set(\sigma_2)$	$set(b_1 \times \sigma_2)$	$O(\#(s \times t))$
$s \times t$	$s : set(\sigma_1),\ t : strong\_set(b_2)$	$set(\sigma_1 \times b_2)$	$O(\#(s \times t))$
$s \times t$	$s : strong\_set(b_1),\ t : strong\_set(b_2)$	$set(b_1 \times b_2)$	$O(\#(s \times t))$
$\#s$	$s : set(\sigma)$ or $strong\_set(b)$	int	$O(\#s)$
$s \uplus t$	$s : set(\sigma),\ t : set(\sigma)$	$set(\sigma)$	$O(\#s + \#t)$
$s \uplus t$	$s : set(b)$ or $strong\_set(b)$ $t : set(b)$ or $strong\_set(b)$	$set(b)$ or $strong\_set(b)$	$O(\#s + \#t)$
$ToSet(f)$	$f : smap(\sigma_1, \sigma_2)$	$set(\sigma_1 \times \sigma_2)$	$O(\#f)$
$ToSet(f)$	$f : strong\_smap(b_1, \sigma_2)$	$set(b_1 \times \sigma_2)$	$O(\#f)$
$ToSet(f)$	$f : mmap(\sigma_1, \sigma_2)$	$set(\sigma_1 \times \sigma_2)$	$O(\#f)$
$ToSet(f)$	$f : strong\_mmap(b_1, \sigma_2)$	$set(b_1 \times \sigma_2)$	$O(\#f)$
$ToMap(s)$	$s : set(b_1 \times \sigma_2)$	$mmap(b_1, \sigma_2)$ or $strong\_mmap(b_1, \sigma_2)$	$O(\#s)$

Table 6.1: Simple Linear-Cost Expressions. K_1 and K_2 are $O(1)$ time computable (unless otherwise stated), boolean-valued, and well-typed under TE_1, TE_2. E_1 is $O(1)$-time computable (unless otherwise stated) and well-typed under TE'. Recall that μ ranges over the types that are comparable for equality in $O(1)$ time.

environment TE, and type τ, but omit set C which can be any admissible set of subtype constraints for the base types appearing in TE. We also include worst-case time complexities of expressions in Table 6.1 based on the naive Low SETL implementations of these expressions.

Theorem 6.3.1 *(Simple Linear-Cost Expressions) Let Expression E, Type Environment TE, and Type τ*

Expr. E	**Type Environment** TE	**Type** τ	**Cost**
$domain(f)$	$f : smap(\sigma_1, \sigma_2)$	$set(\sigma_1)$	$O(\#dom(f))$
$domain(f)$	$f : smap(b_1, \sigma_2)$ or $strong\_smap(b_1, \sigma_2)$	$set(b_1)$ or $strong\_set(b_1)$	$O(\#dom(f))$
$domain(f)$	$f : mmap(\sigma_1, \sigma_2)$	$set(\sigma_1)$	$O(\#dom(f))$
$domain(f)$	$f : mmap(b_1, \sigma_2)$ or $strong\_smap(b_1, \sigma_2)$	$set(b_1)$ or $strong\_set(b_1)$	$O(\#dom(f))$
$range(f)$	$f : smap(\sigma_1, b_2)$ or $strong\_smap(b_1, b_2)$	$set(b_2)$ or $strong\_set(b_2)$	$O(\#f)$
$range(f)$	$f : mmap(\sigma_1, b_2)$ or $strong\_mmap(b_1, b_2)$	$set(b_2)$ or $strong\_set(b_2)$	$O(\#f)$
$f \circ g$	$g : smap(\sigma_1, b_2)$, $f : smap(b_2, \sigma_3)$ or $strong\_smap(b_2, \sigma_3)$	$smap(\sigma_1, \sigma_3)$	$O(\#f + \#g)$
$f \circ g$	$g : smap(b_1, b_2)$ or $strong\_smap(b_1, b_2)$, $f : smap(b_2, \sigma_3)$ or $strong\_smap(b_2, \sigma_3)$	$smap(b_1, \sigma_3)$ or $strong\_smap(b_1, \sigma_3)$	$O(\#f + \#g)$
$f \circ g$	$g : mmap(\sigma_1, b_2)$, $f : smap(b_2, \sigma_3)$ or $strong\_smap(b_2, \sigma_3)$	$mmap(\sigma_1, \sigma_3)$	$O(\#f + \#g)$
$f \circ g$	$g : mmap(b_1, b_2)$ or $strong\_mmap(b_1, b_2)$, $f : smap(b_2, \sigma_3)$ or $strong\_smap(b_2, \sigma_3)$	$mmap(b_1, \sigma_3)$ or $strong\_mmap(b_1, \sigma_3)$	$O(\#f + \#g)$
$f \circ g$	$g : smap(\sigma_1, b_2)$, $f : mmap(b_2, \sigma_3)$ or $strong\_mmap(b_2, \sigma_3)$	$mmap(\sigma_1, \sigma_3)$	$O(\#f + \#g + \#(f \circ g))$
$f \circ g$	$g : smap(b_1, b_2)$ or $strong\_smap(b_1, b_2)$, $f : mmap(b_2, \sigma_3)$ or $strong\_mmap(b_2, \sigma_3)$	$mmap(b_1, \sigma_3)$ or $strong\_mmap(b_1, \sigma_3)$	$O(\#f + \#g + \#(f \circ g))$
$f\|_s$	$f : smap(b_1, \sigma_1)$ or $strong\_smap(b_1, \sigma_1)$ $s : set(b_1)$ or $strong\_set(b_1)$	$smap(b_1, \sigma_1)$ or $strong\_smap(b_1, \sigma_1)$	$O(\#f + \#s)$
$f\|_s$	$f : mmap(b_1, \sigma_1)$ or $strong\_mmap(b_1, \sigma_1)$ $s : set(b_1)$ or $strong\_set(b_1)$	$mmap(b_1, \sigma_1)$ or $strong\_mmap(b_1, \sigma_1)$	$O(\#f + \#s)$
$f[s]$	$f : smap(b_1, b_2)$ or $strong\_smap(b_1, b_2)$ $s : set(b_1)$ or $strong\_set(b_1)$	$set(b_2)$ or $strong\_set(b_2)$	$O(\#f + \#s)$
$f[s]$	$f : mmap(b_1, b_2)$ or $strong\_mmap(b_1, b_2)$ $s : set(b_1)$ or $strong\_set(b_1)$	$set(b_2)$ or $strong\_set(b_2)$	$O(\#f + \#s)$
f^{-1}	$f : smap(\sigma_1, b_2)$	$mmap(b_2, \sigma_1)$ or $strong\_mmap(b_2, \sigma_1)$	$O(\#f)$
f^{-1}	$f : smap(b_1, b_2)$ or $strong\_smap(b_1, b_2)$	$mmap(b_2, b_1)$ or $strong\_mmap(b_2, b_1)$	$O(\#f)$
f^{-1}	$f : mmap(\sigma_1, b_2)$	$mmap(b_2, \sigma_1)$ or $strong\_mmap(b_2, \sigma_1)$	$O(\#f)$
f^{-1}	$f : mmap(b_1, b_2)$ or $strong\_mmap(b_1, b_2)$	$mmap(b_2, b_1)$ or $strong\_mmap(b_2, b_1)$	$O(\#f)$
f/g	$f : smap(b_1, \sigma_2)$ or $strong\_smap(b_1, \sigma_2)$ $g : smap(b_1, \sigma_2)$ or $strong\_smap(b_1, \sigma_2)$	$smap(b_1, \sigma_2)$ or $strong\_smap(b_1, \sigma_2)$	$O(\#f + \#g)$
f/g	$f : mmap(b_1, \sigma_2)$ or $strong\_mmap(b_1, \sigma_2)$ $g : mmap(b_1, \sigma_2)$ or $strong\_mmap(b_1, \sigma_2)$	$mmap(b_1, \sigma_2)$ or $strong\_mmap(b_1, \sigma_2)$	$O(\#f + \#g)$

Table 6.1: Simple Linear-Cost Expressions continued.

be taken from any row of Table 6.1. Let $TE' \supseteq TE$, and let C' be any admissible set of subtype constraints containing constraints for all base types appearing in TE'. Then, $\langle TE', C', E, \tau \rangle$ is a linear-cost expression.

Proof: The proof follows by verifying that each of the expressions in Table 6.1 is well-typed, and by verifying that the stated complexities match the complexities of the Low SETL implementations. □

Expr. E	**Type Environment** TE	**Type** τ	**Cost**
$\{x \in s \mid K_1(x)\}$	$TE_1 \cup \{s : set(\sigma)\}$	$set(\sigma)$	$O(\#s)$
$\{x \in s \mid K_1(x)\}$	$TE_1 \cup \{s : strong\_set(b)\}$	$strong\_set(b)$	$O(\#s)$
$\{[x,y] : x \in s,$ $y \in e\{x\} \mid$ $K_1(x,y)\}$	$TE_1 \cup \{s : set(b_1)$ or $strong\_set(b_1)\}$ $\cup \{e : strong\_mmap(b_1,\sigma_2)$ or $mmap(b_1,\sigma_2)\}$	$set(b_1 \times \sigma_2)$ or $mmap(b_1,\sigma_2)$ or $strong\_mmap(b_1,\sigma_2)$	$O(\#s+$ $\#e)$
$\{x \in s \mid$ $K_1(x,e)\}$	$TE_1 \cup \{s : set(b_1)$ or $strong\_set(b_1)\}$ $\cup \{e : strong\_mmap(b_1,\sigma_2)$ or $mmap(b_1,\sigma_2)\}$ where cost of $K_1(x,e)$ is $O(\#e\{x\})$	$set(b_1)$ or $strong\_set(b_1)$	$O(\#s+$ $\#e)$
$\{E_1(x) : x \in s\}$	$TE' \cup \{s : set(\sigma_1)\}$ where $TE'[x \mapsto \sigma_1], C' \vdash_S E_1(x) : b_2$	$set(b_2)$ or $strong\_set(b_2)$	$O(\#s)$
$\{E_1(x) : x \in s\}$	$TE' \cup \{s : strong\_set(b_1)\}$ where $TE'[x \mapsto b_1], C' \vdash_S E_1(x) : b_2$	$set(b_2)$ or $strong\_set(b_2)$	$O(\#s)$
$\{[x, E_1(x)] : x \in s\}$	$TE' \cup \{s : set(\sigma_1)\}$ where σ_1 is not a base type, and $TE'[x \mapsto \sigma_1], C' \vdash_S E_1(x) : \sigma_2$	$set(\sigma_1 \times \sigma_2)$ or $smap(\sigma_1,\sigma_2)$	$O(\#s)$
$\{[x, E_1(x)] : x \in s\}$	$TE' \cup \{s : set(b_1)$ or $strong\_set(b_1)\}$ and $TE'[x \mapsto b_1], C' \vdash_S E_1(x) : \sigma_2$	$set(b_1 \times \sigma_2)$ or $smap(b_1,\sigma_2)$ or $strong\_smap(b_1,\sigma_2)$	$O(\#s)$
$\{[x, E_1(x,e)] :$ $x \in s\}$	$TE' \cup \{s : set(b_1)$ or $strong\_set(b_1)\}$ where $TE'(e) = strong\_mmap(b_1,\sigma_2)$ or $mmap(b_1,\sigma_2)$, $TE'[x \mapsto b_1], C \vdash_S E_1(x,e) : set(\sigma_3)$, and cost of $E_1(x,e)$ is $O(\#e\{x\})$.	$mmap(b_1,\sigma_3)$ or $strong\_mmap(b_1,\sigma_3)$	$O(\#s+$ $\#e)$
$\{[x, E_1(x,e)] :$ $x \in s\}$	$TE' \cup \{s : set(b_1)$ or $strong\_set(b_1)\}$ where $TE'(e) = strong\_mmap(b_1,\sigma_2)$ or $mmap(b_1,\sigma_2)$, $TE'[x \mapsto b_1], C \vdash_S E_1(x,e) : \sigma_3$, and cost of $E_1(x,e)$ is $O(\#e\{x\})$.	$smap(b_1,\sigma_3)$ or $strong\_smap(b_1,\sigma_3)$	$O(\#s+$ $\#e)$

Table 6.1: Simple Linear-Cost Expressions continued.

Example 1 Let type environment $TE = [s : set(\sigma_1), t : set(\sigma_2)]$, and $C = \{\}$. Then $\langle TE, C, s \times t, set(\sigma_1 \times \sigma_2)\rangle$ is a linear-cost expression. Another way of saying the same thing is that expression $E_1(s : set(\sigma_1), t : set(\sigma_2)) : set(\sigma_1 \times \sigma_2) = s \times t$ is a linear-cost expression.

Example 2 Let type environment $TE = [x : set(\sigma_1 \times \sigma_2)]$, and $C = \{\}$. Then $\langle TE, C, \#x, int\rangle$ is a linear-cost expression. Another way of saying the same thing is that expression $E_2(x : set(\sigma_1 \times \sigma_2)) : int = \#x$ is a linear-cost expression.

6.3.2 Composition of Linear-Cost Expressions

From Example 1 and Example 2 in Section 6.3.1, we see that expressions $E_2(x : set(\sigma_1 \times \sigma_2)) : int = \#x$, and $E_1(s : set(\sigma_1), t : set(\sigma_2)) : set(\sigma_1 \times \sigma_2) = s \times t$ are linear-cost expressions. What can we say about the composition of expressions E_1 and E_2, i.e. the expression

$$E_3(s : set(\sigma_1), t : set(\sigma_2)) : int \;=\; E_2(E_1(s,t)) \;=\; \text{let } x = s \times t \text{ in } \#x \text{ ?}$$

The time complexity of computing expression E_3 is $O(\#s \times \#t)$ which is not linear in the sum of the input sizes ($O(\#s + \#t)$) and the output size ($O(1)$). Thus, the composition of two arbitrary linear-cost expressions may not be a linear-cost expression. Note that by our definition, linear cost is a syntactic property. Thus, the expression $\#(s \times t)$ is not of linear cost, while $(\#s) \times (\#t)$ is of linear cost. We define some properties below that are useful in determining when the composition of two linear-cost expressions is a linear-cost expression.

Let us define the size of a data-item x by $\mathit{Size}(x)$ which is given by

$$\begin{array}{rcl} \mathit{Size}(x) & = & O(1), \text{ if } x \text{ is an integer or boolean} \\ & & O(1) + \Sigma_{y \in x} \mathit{Size}(y), \text{ if } x \text{ is a set} \\ & & O(1) + \Sigma_{[y,z] \in x} (\mathit{Size}(y) + \mathit{Size}(z)), \text{ if } x \text{ is a map} \\ & & O(1) + \Sigma_{i=1}^{k} \mathit{Size}(x[i]), \text{ if } x \text{ is a } k\text{-tuple} \end{array} \tag{6.1}$$

Definition 6.3.2 (k-absorbing, Output bounded, Partially k-absorbing in index set I)

k-absorbing: Expression $E(x_1, \ldots, x_n)$ is said to *absorb* its *k-th* argument x_k if the size of the output is an upper bound on the size of input parameter x_k to within a constant factor, i.e $\mathit{Size}(x_k) = O(\mathit{Size}(E))$.

Output Bounded: Expression E is *output bounded* if it absorbs all of its arguments, i.e. $\forall i = 1, \ldots, n$: $\mathit{Size}(x_k) = O(\mathit{Size}(E))$.

Partially k-absorbing in index set I: Expression E is *partially k-absorbing* in index set I if the *k-th* parameter x_k of Expression E is tuple-valued, and the output absorbs the components of x_k in index set I, i.e. $\forall i \in I : \mathit{Size}(x_k[i]) = O(\mathit{Size}(E))$.

Definition 6.3.3 (Input bounded, Partially input bounded with respect to index set I)

Input Bounded: Expression E is *input bounded* if the output size is bounded by the sum of the sizes of the inputs to within a constant factor, i.e $\mathit{Size}(E) = O(\Sigma_{i=1}^{n} \mathit{Size}(x_i))$.

Partially input bounded with respect to index set I: A tuple-valued expression E is said to be *partially input-bounded* with respect to index set I, if the output components of E in index set I are input bounded, i.e. $\forall i \in I : \mathit{Size}(E[i]) = O(\Sigma_{j=1}^{n} \mathit{Size}(x_j))$.

Example 3 Expression $E_1(s,t) = s \cup t$ is output bounded because it absorbs both its arguments s and t. Expression E_1 is also input bounded.

Example 4 Expression $E_2(s,t) = s \cap t$ is input bounded but not output bounded. In fact Expression E_2 does not absorb any of the parameters s and t.

Example 5 Expression $E_3(s,t) = s \times t$ is output bounded (if both s and t are non-empty) but not input bounded.

Example 6 Expression $E_4(f,g) = f \circ g$ is neither input bounded, nor output bounded.

The input and output boundedness properties are used in the following way. Consider the example where the composition of two linear-cost expressions results in an expression that is not of linear cost. Expressions $E_1(s,t) = s \times t$, and $E_2(x) = \#x$ are both linear-cost expressions and their composition $E_2(E_1(s,t))$ is not. This is because of the blowup in the size of the intermediate output $E_1(s,t)$, i.e. because the size of $E_1(s,t)$ is bounded neither by the sum of the sizes of the inputs, nor by the size of the final output. In other words, this is because Expression E_1 is not input bounded, and expression E_2 does not absorb its first parameter. It is easy to see that a sufficient condition for the composition $E_2(E_1)$ of linear-cost expressions E_2 and E_1 to be of linear cost is that E_1 should either be input bounded or E_2 should absorb its first parameter. This idea is formalized in the following proposition.

Proposition 6.3.4 *Expressions that are formed by the application of any of the following rules, are linear-cost expressions.*

1. **conditional:** *The condtional expression*

   $$E(x : bool, y : \tau, z : \tau) : \tau = \mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ z$$

   is an input bound linear-cost expression.

2. **parameter substitution:** *If* $E_1(x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n) : \tau$ *is an input bound linear-cost expression, and* $\tau_1 = \tau_2$*, then* $E_2(x : \tau_1, x_3 : \tau_3, \ldots, x_k : \tau_k) : \tau = E_1(x, x, x_3, \ldots, x_n)$ *is an input bound linear-cost expression.*

3. **tuple formation:** $E(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau_1 \times \ldots \times \tau_n = [x_1, \ldots, x_n]$ *is an input bound linear-cost expression.*

4. **projection function:** *The projection function* $\Pi^n_j(x : \tau_1 \times \ldots \times \tau_n) : \tau_j = x[j]$ *is a linear-cost function that is partially 1-absorbing in the index set* $I = \{j\}$*.*

5. **composition:** *If* $E_2(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$*, and* $E_1(y_1 : \tau'_1, \ldots, y_m : \tau'_m) : \tau_k$ *are linear-cost expressions, then the composition*

   $$\begin{aligned} & E_3(x_1 : \tau_1, \ldots, x_{k-1} : \tau_{k-1}, y_1 : \tau'_1, \ldots, y_m : \tau'_m, x_{k+1} : \tau_{k+1}, \ldots, x_n : \tau_n) : \tau \\ & \quad = E_2(x_1, \ldots, x_{k-1}, E_1(y_1, \ldots, y_m), x_{k+1}, \ldots, x_n) \end{aligned}$$

   is also a linear-cost expression if either E_1 *is input bound, or if* E_2 *absorbs its k-th parameter.*

   More generally, if parameter x_k*, and the output of function* E_1 *are tuple-valued, then Expression* E_3 *is of linear cost if Expression* E_2 *is partially k-absorbing in index set* I_{E_2}*, and Expression* E_1 *is partially input-bounded in index set* I_{E_1}*, and index set* $(I_{E_2} \cup I_{E_1})$ *covers all components of the tuple.*

A word of caution: When Proposition 6.3.4 is applied repeatedly to generate an expression $E(x_1, \ldots, x_n)$ by composition from, say j, basic linear-cost expressions $E_1, \ldots, E_j$, then j is not a factor in the analysis as long as it is unrelated to the input and output values. However, when it is related, then multiple occurrences of the same input parameter among $E_1, \ldots, E_j$ could prevent E from being of linear cost. Similarly, summing the intermediate outputs of the composed expressions $E_1, \ldots, E_j$ may contribute to asymptotic time costs and prevent E from being of linear cost.

In the previous section, we defined some simple expressions belonging to L_{IO}. All boolean-valued expressions are input bounded. In Table 6.2, we summarize the input and output boundedness properties of other simple expressions.

Example 7: Let $E_1(s : set(b), t : set(b)) : set(b) = s \cup t$, and $E_2(x : set(b)) : int = \#x$. Then, $E_3(s : set(b), t : set(b)) : int = E_2(E_1(s,t)) = \#(s \cup t)$ is a linear-cost expression because E_1 and E_2 are of linear cost, and E_1 is input bounded.

Example 8: Let $E_1(s : set(\sigma_1), t : set(\sigma_2)) : set(\sigma_1 \times \sigma_2) = s \times t$, and $E_2(u : set(\sigma_1 \times \sigma_2), v : set(\sigma_3)) : set((\sigma_1 \times \sigma_2) \times \sigma_3) = u \times v$. Then, $E_3(s : set(\sigma_1), t : set(\sigma_2), v : set(\sigma_3)) : set((\sigma_1 \times \sigma_2) \times \sigma_3) = E_2(E_1(s,t), v)$ is a linear-cost expression because E_1 and E_2 are of linear cost, and E_2 absorbs its 1^{st} parameter.

Expression	**Input Bounded**	**Output Bounded**
$\ni s$	$\times$	
$f(x)$	$\times$	
$f\{x\}$	$\times$	
$f[i]$	$\times$	
$domain(f)$	$\times$	
$range(f)$	$\times$	
$\#s$	$\times$	
$f[s]$	$\times$	
$f\|_s$	$\times$	
$\{x \in s \| K(x)\}$	$\times$	
$\{E(x) : x \in s\}$	$\times$	$\times$ if E is few-to-one
$s \cup t$	$\times$	$\times$
$s \cap t$	$\times$	
$s - t$	$\times$	
$s \times t$		$\times$
$s \uplus t$	$\times$	$\times$
$ToSet(f)$	$\times$	$\times$
$ToMap(s)$	$\times$	$\times$
f^{-1}	$\times$	$\times$
f/g	$\times$	
$f \circ g$		

Table 6.2: Input and Output boundedness of Simple Expressions

6.3.3 Dynamic Complexity and Linear-Cost Fixed Point Expressions

In this section we will extend the language L_{IO} to include some fixed point expressions that can be computed in linear time. The SQ$^+$ fixed point expressions are of the form $\mathrm{LFP}_{\leq,w}(E(x,y),x)$[3], or $\mathrm{GFP}_{\leq,w}(E(x,y),x)$, which are respectively the least fixed point greater than or equal to w, and the greatest fixed point less than or equal to w, of function $E(x,y)$ with respect to x. If $E(x,y)$ is monotone in x, inflationary at w (i.e. $w \leq E(w,y)$), and $range(E)$ is finite, then the fixed point expression $\mathrm{LFP}_{\leq,w}(E(x,y),x)$ can be evaluated by computing Tarski iteration sequences, in which we initialize p to w. and repeatedly assign $E(p,y)$ to p until $E(p,y)$ equals p. Alternately, a non-deterministic iteration schema called dominated convergence, may be used to compute such fixed point expressions. As defined in the previous chapter, if functions Δ and δ are feasible relative to each other, then the fixed point expression $\mathrm{LFP}_{\leq,w}(E(x,y),x)$ may be computed in the following way.

```
$p := w$;
while $\exists z \in \Delta(E(p,y),p)$ loop                (6.2)
    $p := \delta(p,z)$;
end loop
```

In this chapter, we consider the special case of expression $\mathrm{LFP}_{\leq,w}(E(x,y),x)$ where the partial order $\leq$ is the subset relation $\subseteq$ and the lattice under consideration is the subset lattice. In this case, it is easy to see that functions $\Delta(q,p) = q - p$ and $\delta(p,z) = p$ *with* z are feasible relative to each other, and that the least

[3]without loss of generality we assume that expression E has two input variables x and y; In general there could be many input variables $x, y_1, y_2, \ldots$

fixed point can be computed as follows.

```
$p := w$;
while $\exists z \in E(p,y) - p$ loop
    $p$ with $:= z$;
end loop
```
(6.3)

Unfortunately the cost of computing fixed point expressions using Program 6.3 is far too high. The requirement that expression $E(x,y)$ be computable in $O(1)$ time is sufficient to ensure that Program 6.3 is of linear cost, but is too restrictive. However, the condition that expression $E(x,y)$ be an input bound expression of linear cost does not suffice because the cost of Program 6.3 turns out to be quadratic in the sum of the sizes of the input and the output.

To get around this problem, we try to eliminate the redundancy in the repeated computation of expression $E(p,y)$ in each iteration of Program 6.3. In each iteration, the set p is modified by the addition of a single element z. It is, therefore, wasteful to recompute $E(p,y)$ from scratch in each iteration. Instead, we can use finite differencing to substitute the fresh evaluation of expression $E(x,y)$ by its incremental counterpart. More precisely, we do the following.

1. establish the invariant $t = E(p,y) - p$ on entry to the while loop of Program 6.3; the code to establish t is called the *pre-processing code.*

2. update t within the while loop when p is modified by the addition of z; the code to update t is called the *difference code.*

3. replace the expression $E(x,y) - x$ appearing in Program 6.3 by t.

The use of finite differencing together with dominated convergence for computing fixed points is reflected in the following code outline.

```
$p := w$;
$t := E(p,y) - p$;  -- Establish the invariant $t = E(p,y) - p$
while $\exists z \in t$ loop
    $p$ with $:= z$;
    ...         -- Re-establish the invariant $t = E(p,y) - p$
end loop
```
(6.4)

A similar dynamic computation can be used for the computation of greatest fixed points. The feasible functions Δ and δ used in the case of greatest fixed points are $\Delta(q,p) = p - q$, and $\delta(p,z) = p \textit{ less } z$ respectively.

The cost of establishing the invariant $t = E(p,y) - p$ is called the *pre-processing cost* and the cumulative cost of maintaining the invariant is called the *dynamic cost.* Expression $E_1(w,y) = \text{LFP}_{\leq,w}(E(x,y),x)$ is of linear cost if the pre-processing and dynamic costs are bounded by $O(\mathit{Size}(w) + \mathit{Size}(y) + \mathit{Size}(E_1))$. One possibility for the above conditions to be satisfied is if expression $E_2(w,y) = E(w,y) - w$ is of linear cost, and if the cost of re-establishing the invariant $t = E(p,y) - p$ with respect to the modification $p \textit{ with} := z$ is $O(1)$ time. This idea regarding when an invariant may be the re-established in $O(1)$ time with respect to an $O(1)$ time change in the values of one or more of the inputs, is formalized in the following definition.

Definition 6.3.5 Strong Continuity: Let $E(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$ be a well-typed SQ$^+$ expression. Let D be a set of ways in which the input variables $x_1, \ldots, x_n$ can be modified. For example, D could be the set of modifications $\{x_i \textit{ with} := z_i : i = 1, \ldots, n\}$, (assuming each x_i is a set). Let $\mathit{Cost}(m)$ denote the cost of performing a modification m in D, and let s be an arbitrary sequence of modifications drawn from D. We say that expression E is *strongly continuous* with respect to set D if for every modification m in an arbitrary sequence of modifications s, m's contribution to the dynamic cost of eagerly[4] maintaining the invariant $t = E(x_1, \ldots, x_n)$ with respect to the entire sequence of modifications s, is $O(\mathit{Cost}(m))$.

[4]by eagerly maintaining the invariant with respect to a sequence of modifications, we mean that the invariant is re-established after each modification

All $O(1)$ time computable expressions are strongly continuous with respect to set of modifications to any parameter. Similarly, the identity function is also strongly continuous with respect to the set of all modifications to the input parameter. The expression $E_1(x : set(b)) : int = \#x$ is strongly continuous with respect to the set of modifications $\{x\ with := z, x\ less := z\}$. The expressions $E_2(s : set(b), t : set(b)) : strong\_set(b) = s \cup t$, $E_3(s : set(b), t : set(b)) : strong\_set(b) = s \cap t$, $E_4(s : set(b), t : set(b)) : strong\_set(b) = s - t$, are all strongly continuous with respect to the set of modifications $\{s\ with := z, s\ less := z, t\ with := z, t\ less := z\}$. The expression $E_4(s : set(b_1), t : set(b_2)) : set(b_1 \times b_2) = s \times t$ is however not strongly continuous with respect to the set of modifications $\{s\ with := z\}$. This is because the addition of one element to s (i.e. an O(1) time modification) results in the addition of $\#t$ new pairs of values to the output, which can not be performed in $O(1)$ time.

Looking at Program 6.4 we see that if expression $E(x, y) - x$ is of linear cost, and is strongly continuous with respect to the set of modifications $\{x\ with := z\}$, then expression $\mathrm{LFP}_{\leq,w}(E(x, y), x)$ is also of linear cost. Thus, the use of incremental computation adds a new class of fixed point expressions to our linear-time language L_{IO}.

Unfortunately, even *strong continuity* can sometimes be too restrictive and result in the exclusion of an interesting class of fixed point expressions from language L_{IO}, as can be seen from the following example.

Example 9:(Graph Reachability) The problem is to find the set of vertices s reachable along paths in a directed graph G from an arbitrary set of vertices w. Let Graph G be represented by the set of vertices v and the finite set of edges e, where each edge is a pair of vertices. We regard e as a multi-valued map, so that for each vertex x, the term $e\{x\}$ represents the set of vertices adjacent to x, i.e. reachable from x along a single directed edge. Recall that $e[s] = \cup_{x \in s} e\{x\}$. Then, the graph reachability problem can be expressed as the following SQ$^+$ program

$$\mathrm{LFP}_{\subseteq,w}(s \cup e[s], s) \tag{6.5}$$

The following program outline uses finite differencing and dominated convergence to compute the set of reachable vertices

```
$p := w$;
$t := e[p] - p$;     -- Note that $(p \cup e[p]) - p = e[p] - p$
while $\exists z \in t$ loop
    $p\ with := z$;                                              (6.6)
    ...    -- Re-establish the invariant $t = e[p] - p$
end loop
```

Since expression $e[p] - p$ is not strongly continuous with respect to the set of modifications $\{p\ with := z\}$, the expression $\mathrm{LFP}_{\leq,w}(p \cup e[p], p)$ cannot be included in language L_{IO}. However, it is well known that the set of reachable vertices can be computed in linear time. Although the re-establishment of the invariant $t = e[p] - p$ with respect to the modification $p\ with := z$ can not be done in $O(1)$ time, it turns out that the cumulative cost of re-establishing the invariant with respect to the sequence of all modifications to p during the execution of the program is bounded (to within a constant factor) by the sum of the sizes of input e and the size of the final output (i.e. by the number of the reachable vertices). In other words, although the worst-case cost of re-establishing the invariant is not $O(1)$ time, the **amortized** cost of re-establishing the invariant is $O(1)$ time. Unfortunately, our definition of strong continuity does not capture amortized complexities, and therefore fixed point computations such as graph-reachability are not included in the language L_{IO}. In order to rectify this problem, we define the notion of *weak continuity.*

Definition 6.3.6 Weak Continuity: Let $E(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$ be a well-typed SQ$^+$ expression. Let D be a set of ways in which the input variables $x_1, \ldots, x_n$ can be modified. Let $Cost(s)$ denote the cost of performing an arbitrary sequence s of modifications selected from D. We say that expression E is *weakly continuous* with respect to set D if the the dynamic cost of eagerly maintaining the invariant $t = E(x_1, \ldots, x_n)$ with respect to an arbitrary sequence s of modifications drawn from D is

$$O(Cost(s) + Size(E_I) + Size(E_F) + \sum_{i=1}^{n} Size(x_i))$$

where $x_1, \dots, x_n$ are the initial values of the inputs, E_I is the initial value of expression E, and E_F is the final value of expression E.

From Definitions 6.3.5 and 6.3.6, it is obvious that if an expression E is strongly continuous with respect to a set of modifications D, then it is also weakly continuous with respect to D. The reverse, however, is not true. For example, recall that the expression $E_1(s : set(b_1), t : set(b_2)) : set(b_1 \times b_2) = s \times t$ is *not strongly continuous* with respect to the set of modifications $D_1 = \{s\ with := z\}$. However, it is not difficult to show that expression E_1 *is weakly continuous* with respect to the set of modifications D_1. If we consider the set of modifications $D_2 = \{s\ with := z, t\ with := z\}$, expression E_1 is still weakly continuous with respect to set D_2. Instead, if we consider the set of modifications $D_3 = \{s\ with := z, s\ less := z\}$, it turns out that expression E_1 is not weakly continuous with respect to set D_3. Similarly, expression $E_2(f : strong\_mmap(b, b), s : set(b)) : strong\_set(b) = f[s]$ is weakly continuous (but not strongly continuous) with respect to the set of modifications $\{s\ with := z, f\{x\}\ with := y\}$, but not weakly continuous with respect to the set of modifications $\{s\ with := z, s\ less := z\}$.

A list of simple L_{IO} expressions and some of their strong and weak continuity properties are given in Table 6.3. In addition, all $O(1)$ time computable expressions are strongly continuous (and therefore weakly continuous) with respect to the set of all modifications to their inputs. For proofs and more details about the strong and weak continuity properties of the simple expressions in Table 6.3, we refer the reader to [71, 68, 72, 13, 12]. In particular, [71], and [68] contain detailed descriptions of how to handle finite differencing of expressions like $\{x \in s\ |K(x)\}$ and $\{E(x) : x \in s\}$ with respect to modifications of the form $\delta K(x)$ and $\delta E(x)$ respectively.

A subclass of weakly continuous expressions can be built up starting from the simple expressions in Table 6.3 using the following two lemmas.

Lemma 6.3.7 *The continuity properties of some SQ^+ expressions are stated below.*

1. **conditional:** *The conditional expression*

$$E(x : bool, y : \tau, z : \tau) : \tau = if\ x\ then\ y\ else\ z$$

   is strongly continuous with respect to any set of modifications to input variables x, y, or z.

2. **parameter substitution:** *If $E(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau$ is strongly continuous with respect to the set of modifications $\{\delta_1 x_1, \delta_2 x_2, \dots, \delta_n x_n\}$, and if $\tau_1 = \tau_2$ and δ_1 and δ_2 are modifications of the same kind (e.g. both are set element additions), then expression $E'(x : \tau_1, x_3 : \tau_3, \dots, x_n : \tau_n) : \tau = E(x, x, x_3, \dots, x_n)$ is strongly continuous with respect to the set of modifications $\{\delta_1 x, \delta_3 x_3, \dots, \delta_n x_n\}$.*

3. **tuple formation:** *Expression $E(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau_1 \times \dots \times \tau_n = [x_1, \dots, x_n]$ is strongly continuous with respect to any set of modifications to input variables $x_1, \dots, x_n$.*

4. **projection function:** *The projection function $\Pi^n_j(x : \tau_1 \times \dots \times \tau_n) : \tau_j = x[j]$ is strongly continuous with respect to any set of modifications to input variable x.*

5. **fixed point:** *$E'(w : set(b), t : set(b)) : strong\_set(b) = LFP_{\subseteq, w}(E(s, t), s)$ is weakly continuous with respect to the set of modifications $\{w\ with := z, t\ with := z\}$ whenever expression $E(s, t)$ is monotone in both s and t, $w \subseteq E(w, t)$ for all t, either the range of expression E is finite or there are only finite ascending chains starting from w, and expression $E''(s : set(b), t : set(b)) : strong\_set(b) = E(s, t) - s$ is weakly continuous with respect to the set of modifications $\{s\ with := z, t\ with := z\}$. An analogous rule holds for greatest fixed points.*

The proofs of statements 1-4 of Lemma 6.3.7 are straightforward and are omitted. For the proof of statement 5, we refer the reader to [12].

Lemma 6.3.8 *The following are closure rules for weak and strong continuity.*

Exp.	**Input Types**	**Out. Type**	**Strong Cont.**	**Weak Cont.**
$\ni s$	$s : set(\sigma)$	σ	$\{\delta s, \delta^+ s, \delta^- s\}$	$\{\delta s, \delta^+ s, \delta^- s\}$
$f(x)$	$f : strong\_smap(b_1, \sigma_2)$, $x : b_1$	σ_2	$\{\delta f(x), \delta x\}$	$\{\delta f(x), \delta x\}$
$f\{x\}$	$f : strong\_mmap(b_1, \sigma_2)$, $x : b_1$	$set(\sigma_2)$	$\{\delta f\{x\}, \delta x, \delta^+ f\{x\}, \delta^- f\{x\}\}$	$\{\delta f\{x\}, \delta x, \delta^+ f\{x\}, \delta^- f\{x\}\}$
$f[i]$	$f : \sigma_1 \times \ldots \times \sigma_k$	σ_i	$\{\delta f, \delta f[i]\}$	$\{\delta f, \delta f[i]\}$
$domain f$	$f : strong\_smap(b_1, \sigma_2)$	$strong\_set(b_1)$	$\{\delta f(x)\}$	$\{\delta f(x)\}$
$domain f$	$f : strong\_mmap(b_1, \sigma_2)$	$strong\_set(b_1)$	$\{\delta f\{x\}, \delta^+ f\{x\}, \delta^- f\{x\}\}$	$\{\delta f\{x\}, \delta^+ f\{x\}, \delta^- f\{x\}\}$
$range f$	$f : strong\_smap(b_1, b_2)$	$strong\_set(b_2)$	$\{\delta f(x)\}$	$\{\delta f(x)\}$
$range f$	$f : strong\_mmap(b_1, b_2)$	$strong\_set(b_2)$	$\{\delta^+ f\{x\}, \delta^- f\{x\}\}$	$\{\delta^+ f\{x\}, \delta^- f\{x\}\}$
$\#s$	$s : set(\sigma)$	int	$\{\delta^+ s, \delta^- s\}$	$\{\delta^+ s, \delta^-\}$
$s \cup t$	$s : set(b)$ or $strong\_set(b)$ $t : set(b)$ or $strong\_set(b)$	$strong\_set(b)$	$\{\delta^+ s, \delta^- s, \delta^+ t, \delta^- t\}$	$\{\delta^+ s, \delta^- s, \delta^+ t, \delta^- t\}$
$s \cap t$	$s : set(b)$ or $strong\_set(b)$ $t : set(b)$ or $strong\_set(b)$	$strong\_set(b)$	$\{\delta^+ s, \delta^- s, \delta^+ t, \delta^- t\}$	$\{\delta^+ s, \delta^- s, \delta^+ t, \delta^- t\}$
$s - t$	$s : set(b)$ or $strong\_set(b)$ $t : set(b)$ or $strong\_set(b)$	$strong\_set(b)$	$\{\delta^+ s, \delta^- s, \delta^+ t, \delta^- t\}$	$\{\delta^+ s, \delta^- s, \delta^+ t, \delta^- t\}$
$s \times t$	$s : set(\sigma_1), t : set(\sigma_2)$	$set(\sigma_1 \times \sigma_2)$		$\{\delta^+ s, \delta^+ t\}$, $\{\delta^- s, \delta^- t\}$
$s \uplus t$	$s : set(\sigma), t : set(\sigma)$	$set(\sigma)$	$\{\delta^+ s, \delta^- s, \delta^+ t, \delta^- t\}$	$\{\delta^+ s, \delta^- s, \delta^+ t, \delta^- t\}$
$f[s]$	$f : strong\_mmap(b_1, b_2)$ $s : set(b_1)$ or $strong\_set(b_1)$	$strong\_set(b_2)$		$\{\delta^+ s, \delta^+ f\{x\}, \delta^- f\{x\}\}$, $\{\delta^- s, \delta^+ f\{x\}, \delta^- f\{x\}\}$,
$f\vert_s$	$f : strong\_mmap(b_1, b_2)$ $s : set(b_1)$ or $strong\_set(b_1)$	$strong\_mmap(b_1, b_2)$	$\{\delta^+ s, \delta^+ f\{x\}\}$, $\{\delta^- s, \delta^+ f\{x\}\}$	$\{\delta^+ s, \delta^+ f\{x\}\}$, $\{\delta^- s, \delta^+ f\{x\}\}$
f^{-1}	$f : strong\_mmap(b_1, b_2)$	$strong\_mmap(b_2, b_1)$	$\{\delta^+ f\{x\}\}$	$\{\delta^+ f\{x\}\}$
f/g	$f : strong\_smap(b_1, \sigma_2)$ $g : strong\_smap(b_1, \sigma_2)$	$strong\_smap(b_1, \sigma_2)$	$\{\delta f(x), \delta g(x)\}$	$\{\delta f(x), \delta g(x)\}$
f/g	$f : strong\_mmap(b_1, \sigma_2)$ $g : strong\_mmap(b_1, \sigma_2)$	$strong\_mmap(b_1, \sigma_2)$	$\{\delta f\{x\}, \delta g\{x\}, \delta^+ f\{x\}, \delta^+ g\{x\}\}$	$\{\delta f\{x\}, \delta g\{x\}, \delta^+ f\{x\}, \delta^+ g\{x\}\}$
$\{x \in s \mid K(x)\}$	$s : set(b)$ or $strong\_set(b)$	$set(b)$ or $strong\_set(b)$	$\{\delta^+ s, \delta^- s, \delta K(x)\}$	$\{\delta^+ s, \delta^- s, \delta K(x)\}$
$\{E(x) : x \in s\}$	$s : set(b)$ or $strong\_set(b)$	$set(b)$ or $strong\_set(b)$	$\{\delta^+ s, \delta^- s, \delta E(x)\}$	$\{\delta^+ s, \delta^- s, \delta E(x)\}$

Table 6.3: Strong and Weak Continuity properties of some simple L_{IO} expressions. We use δs, $\delta^+ s$ and $\delta^- s$ to abbreviate $s := z$, $s\ with := z$ and $s\ less := z$ respectively.

1. *If expression $E(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$ is strongly continuous relative to a set of modifications D, it is also weakly continuous relative to D.*

2. *If expression $E(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$ is strongly continuous (respectively weakly continuous) relative to a set of modifications D, it is also strongly continuous (respectively weakly continuous) with respect to any non-empty subset of D.*

3. *Let expression $E_1(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau$ be strongly continuous with respect to a set D_1 of modifications to the input variables $x_1, \dots, x_n$. Let D_2 be the set of modifications applied to variable y_k in the course of maintaining the invariant $y_k = E(x_1, \dots, x_n)$ with respect to modifications drawn from set D_1. Let $E_2(y_1 : \tau'_1, \dots, y_m : \tau'_m) : \tau'$ be an expression such that $\tau'_k = \tau$, and such that input variables $y_1, \dots, y_m$ are distinct from variables $x_1, \dots, x_n$. Let D_3 be a set of modifications to input variables $y_1, \dots, y_m$ except variable y_k. If expression E_2 is strongly continuous with respect to the set of modifications $D_2 \cup D_3$, then expression*

$$\begin{aligned}&E_3(y_1 : \tau'_1, \dots, y_{k-1} : \tau'_{k-1}, x_1 : \tau_1, \dots, x_n : \tau_n, y_{k+1} : \tau'_{k+1}, \dots, y_m : \tau'_m) : \tau' = \\ &\quad E_2(y_1, \dots, y_{k-1}, E_1(x_1, \dots, x_n), y_{k+1}, \dots, y_m)\end{aligned}$$

is strongly continuous with respect to the set of modifications $D_1 \cup D_3$.

4. *Let expression $E_1(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau$ be weakly continuous with respect to a set D_1 of modifications to the input variables $x_1, \dots, x_n$. Let D_2 be the set of modifications applied to variable y_k in the course of maintaining the invariant $y_k = E(x_1, \dots, x_n)$ with respect to modifications drawn from set D_1. Let $E_2(y_1 : \tau'_1, \dots, y_m : \tau'_m) : \tau'$ be an expression such that $\tau'_k = \tau$, and such that input variables $y_1, \dots, y_m$ are distinct from variables $x_1, \dots, x_n$. Let D_3 be a set of modifications to input variables $y_1, \dots, y_m$ except variable y_k. If expression E_2 is weakly continuous with respect to the set of modifications $D_2 \cup D_3$, then expression*

$$\begin{aligned}&E_3(y_1 : \tau'_1, \dots, y_{k-1} : \tau'_{k-1}, x_1 : \tau_1, \dots, x_n : \tau_n, y_{k+1} : \tau'_{k+1}, \dots, y_m : \tau'_m) : \tau' = \\ &\quad E_2(y_1, \dots, y_{k-1}, E_1(x_1, \dots, x_n), y_{k+1}, \dots, y_m)\end{aligned}$$

is weakly continuous with respect to the set of modifications $D_1 \cup D_3$, if either (1) expression E_2 absorbs its k-th parameter, or (2) if expression E_1 is input bound.

The proofs of statements 1-2 of Lemma 6.3.8 are straightforward and are omitted. For the proof of statements 3 and 4, we refer the reader to [12]

Using Lemmas 6.3.7 and 6.3.8, we can build up a class of expressions that are strongly or weakly continuous with respect to a set of modifications D. The weak and strong continuity properties of expressions help us extend our language L_{IO} of linear-cost expressions, as is summarized by Theorem 6.3.9.

Theorem 6.3.9 *L_{IO} contains the smallest set of expressions closed under the following rules.*

1. ***(basis)*** *The simple linear-cost expressions in Table 6.1 belong to L_{IO}.*

2. ***(fixed point)*** *Expression*

$$E_1(w : set(b), t : \tau) : strong\_set(b) = LFP_{\subseteq, w}(E(s,t), s)$$

belongs to L_{IO} when expression $E(s,t)$ is monotone in s, the range of expression E is finite, $w \subseteq E(w,t)$ for all t, expression $E(s : set(b), t : \tau) : set(b)$ is in L_{IO}, and expression $E_2(s : set(b), t : \tau) : strong\_set(b) = E(s,t) - s$ is weakly continuous with respect to $\{s \text{ with} := z\}$.

Similarly, expression $E_1(w : set(b), t : \tau) : strong\_set(b) = GFP_{\subseteq, w}(E(s,t), s)$ belongs to L_{IO} when expression $E(s,t)$ is monotone in s, the range of expression E is finite, $w \supseteq E(w,t)$ for all t, expression $E(s : set(b), t : \tau) : set(b)$ is in L_{IO}, and expression $E_2(s : set(b), t : \tau) : strong\_set(b) = s - E(s,t)$ is weakly continuous with respect to $\{s \text{ less} := z\}$.

3. ***(closure)*** *L_{IO} may further be augmented by application of Proposition 6.3.4.*

The proof that all expressions contained in language L_{IO} (as defined by Theorem 6.3.9) are of linear cost follows from the time complexities of the simple expressions in Table 6.1, the definition of weak continuity and the proof of Proposition 6.3.4.

6.4 Some Simple Examples

Two simple examples of problems expressible in L_{IO} are given below.

Example 10: (Graph Reachability Re-visited) The problem is to compute the set of vertices s reachable along directed paths from a set of vertices w in a directed graph G. It is expressible as the following SQ$^+$ program.

$$\mathrm{LFP}_{\subseteq,w}(s \cup e[s], s) \tag{6.7}$$

where $w : \mathit{set}(b)$, and $e : \mathit{strong\_mmap}(b,b)$. The proof that Expression 6.7 belongs to L_{IO} is outlined below.

1. From Table 6.1 we see that Expression

   $$E_1(s_1 : \mathit{set}(b), e : \mathit{strong\_mmap}(b,b)) : \mathit{set}(b)$$

   is a linear-cost expression. From Table 6.2 we see that E_1 is input bound. From Table 6.3, and Lemma 6.3.8(2) we see that E_1 is weakly continuous with respect to the set of modifications $\{s_1 \ \mathit{with} := z\}$.
2. From Table 6.1 we see that Expression $E_2(s_2 : \mathit{set}(b), t : \mathit{set}(b)) : \mathit{set}(b)$ is a linear-cost expression. From Table 6.3, and Lemma 6.3.8(2) we see that E_2 is weakly continuous with respect to the set of modifications $\{s_2 \ \mathit{with} := z, t \ \mathit{with} := z\}$.
3. The set of modifications applied to variable t in maintaining the invariant $t = e[s_1]$ with respect to the modification $s_1 \ \mathit{with} := z$, is $\{t \ \mathit{with} := z'\}$. Thus, it follows from Lemma 6.3.8(4), that expression $E_3(s_2 : \mathit{set}(b), s_1 : \mathit{set}(b), e : \mathit{strong\_mmap}(b,b)) : \mathit{set}(b) = E_2(s_2, E_1(s_1, e))$ is weakly continuous with respect to the set of modifications $\{s_1 \ \mathit{with} := z, s_2 \ \mathit{with} := z\}$. It also follows from Proposition 6.3.4 that expression E_3 is a linear-cost expression.
4. It follows from the parameter substitution case of Lemma 6.3.7 that expression $E_4(s : \mathit{set}(b), e : \mathit{strong\_mmap}(b,b)) : \mathit{set}(b) = E_3(s, s, e)$ is weakly continuous with respect to the set of modifications $\{s \ \mathit{with} := z\}$, and it follows from Proposition 6.3.4 that E_4 is a linear-cost expression. (Note that $E_4(s,e) = s \cup e[s]$).
5. It is easy to prove that $E_4(s,e)$ is monotone in s.
6. From Theorem 6.3.9(2), it follows that $\mathrm{LFP}_{\subseteq,w}(E_4(s,e), s)$ is a linear-cost expression, i.e

   $$\mathbf{LFP_{\subseteq,w}(s \cup e[s], s) \ is\ a\ linear\text{-}cost\ expression}.$$

Example 11: (Cycle Detection) Once again we assume that a directed graph is represented by a set of vertices v and multi-valued map e such that for any vertex x, $e\{x\}$ is the set of vertices adjacent to x. Graph G contains a cycle iff the largest set of vertices $s \subseteq v$ each containing a successor belonging to s is non-empty. This can be expressed as the following SQ$^+$ program

$$\neg \mathit{IsEmptySet}(\mathrm{GFP}_{\subseteq}((s - \{x \in s | \mathit{IsEmptySet}(e\{x\} \cap s)\}), s)), \tag{6.8}$$

or as the following more readable specification

$$\mathrm{GFP}_{\subseteq}((s - \{x \in s | e\{x\} \cap s = \{\}\ \}), s) \neq \{\}.$$

It is easy to prove that the expression

$$s - \{x \in s \mid e\{x\} \cap s = \{\}\}$$

is equivalent to

$$E(s,e) \quad = \quad \text{let} \quad f = \{[x, \#\{y \in e\{x\} \mid y \in s\}] : x \in s\} \quad \text{in} \quad s - \{x \in domain(f) \mid f(x) = 0\}. \tag{6.9}$$

Thus, the cycle detection specification can be re-written as

$$\neg IsEmptySet(\mathrm{GFP}_{\subseteq}(E(s,e),s)) \tag{6.10}$$

As before, we assume that the type of e is $strong\_mmap(b,b)$. The proof that Expression 6.10 belongs to L_{IO} is outlined below.

1. From Table 6.1 and Proposition 6.3.4, it is easy to infer that expression $E_1(s : strong\_set(b), e : strong\_mmap(b,b)) : strong\_smap(b,int) = \{[x, \#\{y \in e\{x\} \mid y \in s\}] : x \in s\}$ and expression $E_2(s : strong\_set(b), f : strong\_smap(b,int)) : strong\_set(b) = s - \{x \in domain(f) \mid f(x) = 0\}$ are linear-cost expressions. Since expression E_1 is input bound, it follows that expression $E(s,e)$ given by Equation 6.9 is a linear-cost expression. It is also easy to prove that expression $E(s,e)$ is monotone in s.
2. It is also easy to prove that expression $E_1(s,e) = \{[x, \#\{y \in e\{x\} \mid y \in s\}] : x \in s\}$ is weakly continuous with respect to the set of modifications $\{s\ less := z\}$. The difference code for maintaining the invariant $f = E_1(s,e)$ with respect to the modification $s\ less := z$ is

```
for $x \in e^{-1}\{z\}$ loop
    $f(x) - := 1$;
endloop.
```

   Note that we need to maintain an auxiliary expression e^{-1} in order to perform the difference code for re-establishing expression E_1 with respect to the change $s\ less := z$.

   Moreover, the expression $E_2(s,f)$ is also weakly continuous with respect to the set of modifications $\{s\ less := z, f(x) - := 1\}$. Since expression E_1 is input bound, it follows that expression $E(s,e)$ is weakly continuous with respect to the set of modifications $\{s\ less := z\}$.
3. It therefore follows that

$$\mathrm{GFP}_{\subseteq}(E(s,e),s) \tag{6.11}$$

   is a linear-cost expression. Moreover, the size of the output of Expression 6.11 is bound by the size of $domain(e)$, and hence Expression 6.11 is input bound. Thus, it follows from Proposition 6.3.4 that Expression 6.10 is a linear-cost expression.

Example 12: (Constant Propagation) The constant propagation problem described here is a variant of the one described in [17]. Constant Propagation is a technique used in many optimizing compilers. Given some form of intermediate representation of a program, the main task of constant propagation is to find out statements that define constant values. A statement defines a constant if it assigns the same constant value to its left hand side in all possible executions. For example, if all operands of an assignment are constant, then it defines a constant. Reif and Lewis [78] adopted the following condition to determine when a variable use[5] is constant.

- It represents a constant symbol.
- All of its reaching definitions which have been found to define constants define the same constant value, and none of its definitions has been proved to be a non-constant.

[5]An assignment `x := y + z` *defines* variable `x` and *uses* variables `y` and `z`.

A non-constant definition is either a read statement, or a definition one of whose operands is a non-constant. An operand is a non-constant if it has a non-constant definition, or two constant definitions defining different constant values. We take the following specification which are a modified form of but equivalent to the specifications in [12].

$$\begin{aligned}\text{LFP}_{\leq}([&E_1(\mathit{const}, \mathit{nonconst}, \mathit{prog}, \mathit{usetodef}), E_2(\mathit{const}, \mathit{nonconst}, \mathit{prog}, \mathit{usetodef})],\\ &[\mathit{const}, \mathit{nonconst}])\end{aligned} \tag{6.12}$$

where,

$$\begin{aligned}&E_1(\mathit{const}, \mathit{nonconst}, \mathit{prog}, \mathit{usetodef}) =\\ &\quad \text{let } Q = \{x \in \mathit{domain}(\mathit{usetodef}) \mid \mathit{usetodef}\{x\} \cap \mathit{nonconst} = \{\}\} \text{ in}\\ &\quad\ \{s \in \mathit{prog} \mid \forall t \in \mathit{operands}(s)\ (\ C(t) \lor\\ &\quad\ \ \#\{\mathit{compute}(y) : y \in \mathit{usetodef}\{[s,t]\} \mid [s,t] \in Q\ \land\ y \in \mathit{const}\} = 1)\}\end{aligned}$$

and

$$\begin{aligned}&E_2(\mathit{const}, \mathit{nonconst}, \mathit{prog}, \mathit{usetodef}) =\\ &\quad \text{let } Q = \{x \in \mathit{domain}(\mathit{usetodef}) \mid \mathit{usetodef}\{x\} \cap \mathit{nonconst} = \{\}\} \text{ in}\\ &\quad\ \{s \in \mathit{prog} \mid (s \in \mathit{read}) \lor \exists t \in \mathit{operands}(s)\ (\neg C(t)\ \land\ ([s,t] \notin Q \lor\\ &\quad\ \ \#\{\mathit{compute}(y) : y \in \mathit{usetodef}\{[s,t]\} \mid [s,t] \in Q\ \land\ y \in \mathit{const}\} > 1))\}\end{aligned}$$

As pointed out by Cai and Paige, Specification 6.12 can not be solved directly because expression E_1 is not monotonic in its parameter *const*. However, they pointed out that expression E_2 is monotone, and expression $E_1 \cup E_2$ is monotone. Thus, by introducing a new variable $\mathit{sum} = \mathit{const} \cup \mathit{nonconst}$, Specification 6.12 could be reformulated as

$$\begin{aligned}\text{LFP}_{\leq}([&E_1(\mathit{sum} - \mathit{nonconst}, \mathit{nonconst}, \mathit{prog}, \mathit{usetodef}),\\ &E_2(\mathit{sum} - \mathit{nonconst}, \mathit{nonconst}, \mathit{prog}, \mathit{usetodef})],\\ &[\mathit{sum}, \mathit{nonconst}])\end{aligned} \tag{6.13}$$

To see how Specification 6.13 can be computed in linear time, we just need to generalize our theory to compute fixed points on tuples of sets (instead of just sets, as given by Theorem 6.3.9). Consider the following specification

$$\begin{aligned}\text{LFP}_{\leq}([&E_1(x_1, \ldots, x_n, y),\\ &E_2(x_1, \ldots, x_n, y),\\ &\ldots\\ &E_n(x_1, \ldots, x_n, y)],\\ &[x_1, \ldots, x_n])\end{aligned} \tag{6.14}$$

where each expression E_i is monotone in all the parameters $x_1, \ldots, x_n$. It is easy to show that Specification 6.14 can be solved by

$$\begin{aligned}&x_1 = x_2 = \ldots = x_n = \{\};\\ &\text{while } \exists [j,t] \in \{[i,z] : i \in \{1, \ldots, n\}, z \in E_i(x_1, \ldots, x_n) - x_i\} \text{ loop}\\ &\quad x_j\ \mathit{with} := t;\\ &\text{endloop}\end{aligned} \tag{6.15}$$

Moreover, it is easy to prove that if each expression E_i is of linear cost and expression $E_i - x_i$ is weakly continuous with respect to the set of modifications $\{x_i\ \mathit{with} := z_i, i = 1, \ldots, n\}$, then the expression given by Specification 6.14 is of linear cost.

Theorem 6.4.1 *Specification 6.14 is of linear cost if*

- *each expression $E_i(x_1, \ldots, x_n, y)$ is of linear cost, and*
- *each expression $E_i(x_1, \ldots, x_n, y) - x_i$ is weakly continuous with respect to the set of modifications $\{x_i \ with := z_i, i = 1, \ldots, n\}$.*

Thus, in order to prove that Specification 6.13 is of linear cost, we just need to prove that

- Expression

  $$\begin{array}{l}(E_1(sum - nonconst, nonconst, prog, usetodef) \cup \\ E_2(sum - nonconst, nonconst, prog, usetodef)) - sum\end{array}$$

  is weakly continuous with respect to $\{sum\ with := z, nonconst\ with := z\}$.
- Expression $E_2(sum - nonconst, nonconst, prog, usetodef) - nonconst$ is weakly continuous with respect to $\{sum\ with := z, nonconst\ with := z\}$
- Expressions $E_1 \cup E_2$ and E_2 are of linear cost.

In order to show the above conditions, it is sufficient to show that expressions E_1 and E_2 are themselves weakly continuous with respect to $\{sum\ with := z, nonconst\ with := z\}$ and are of linear cost (using the fact that both E_1 and E_2 are input bound).

We show that expression E_1 is weakly continuous with respect to $\{sum\ with := z, nonconst\ with := z\}$ by decomposing it into simpler sub-expressions as below.

$$\begin{array}{lcl}
W_1(nonconst, usetodef) & = & \{x \in domain(usetodef) \mid \\
 & & \quad usetodef\{x\} \cap nonconst = \{\}\} \\
W_2(sum, nonconst, W_1) & = & \{[x,y] \in usetodef \mid x \in W_1 \ \wedge \ y \in sum\} \\
W_3(x, W_2) & = & \text{if } \#\{compute(y) : y \in W_2\{x\}\} = 1 \text{ then } 1 \\
 & & \text{else } 2 \\
W_4(W_2, W_3) & = & \{[x, W_3(x, W_2)] : x \in domain(W_2)\} \\
W_5(usetodef, W_4) & = & \{x \in domain(usetodef) \mid \\
 & & \quad x \notin domain(W_4) \vee W_4(x) = 2\} \\
W_6(usetodef, W_1) & = & domain(usetodef) - W_1 \\
W_7(W_5, W_6) & = & W_5 \cup W_6 \\
W_8(prog, W_7) & = & \{[s, \#\{t : [s,t] \in W_7\}] : s \in prog\} \\
W_9(prog, W_8) & = & \{s \in prog \mid W_8(s) = 0\}
\end{array}$$

It can be verified that expression W_9 is equivalent to expression E_1. Proving that E_1 is of linear cost is simple and is omitted. The proof that E_1 is weakly continuous is outlined below.

- Each of the expressions $W_1, \ldots, W_9$ is input bound.
- W_1 is weakly continuous with respect to $nonconst\ with := z$. The modification required to W_1 in the difference code is $W_1\ less := x$.
- W_2 is weakly continuous with respect to the set of modifications

  $$\{sum\ with := z, W_1\ less := x\}.$$

  The modifications to W_2 in the difference code are

  $$\{W_2\{x\} = \text{om}, W_2\{x\}\ with := z\}.$$

- W_3 is weakly continuous with respect to the set of modifications

  $$\{W_2\{x\} = \text{om}, W_2\{x\}\ with := z\}.$$

  The modification to W_3 in the difference code is $W_3(x) = 2$.

- W_4 is weakly continuous with respect to the set of modifications

$$\{W_2\{x\} = \text{om}, W_2\{x\}\ \textit{with} := z, W_3(x) = 2\}.$$

  The modifications to W_4 in the difference code are

$$\{W_4(x) = \text{om}, W_4(x) = 2\}.$$

- W_5 is weakly continuous with respect to the set of modifications

$$\{W_4(x) = \text{om}, W_4(x) = 2\}.$$

  The modifications to W_5 in the difference code are

$$\{W_5\ \textit{with} := x, W_5\ \textit{less} := x\}.$$

- W_6 is weakly continuous with respect to the modification $W_1\ \textit{less} := z$. The modification to W_6 in the difference code is $W_6\ \textit{with} := x$.
- W_7 is weakly continuous with respect to the set of modifications

$$\{W_5\ \textit{with} := x, W_5\ \textit{less} := x, W_6\ \textit{with} := x\}.$$

  The modifications to W_7 in the difference code are

$$\{W_7\ \textit{with} := x, W_7\ \textit{less} := x\}.$$

- W_8 is weakly continuous with respect to the set of modifications

$$\{W_7\ \textit{with} := x, W_7\ \textit{less} := x\}.$$

  The modifications to W_8 in the difference code are

$$\{W_8(s) += 1, W_8(s) -= 1\}.$$

- W_9 is weakly continuous with respect to the set of modifications

$$\{W_8(s) += 1, W_8(s) -= 1\}.$$

Similarly, it can be shown that Expression E_2 is also weakly continuous with respect to $\{\textit{sum with} := z, \textit{nonconst with} := z\}$, whereby we can prove that Specification 6.13 is of linear cost.

In [17] Cai and Paige showed that the constant propagation problem could be solved in worst-case linear time on a uniform cost sequential RAM. We have improved on the result by showing that the constant propagation problem can be solved in linear time on a pointer machine. To the best of our knowledge, this is the first linear-time pointer machine algorithm for constant propagation.

6.5 Conclusion

In this chapter we defined a linear-time subset L_{IO} of SQ$^+$. We used both static complexity (i.e. complexity of evaluation from scratch) and dynamic complexity (i.e. complexity of incremental evaluation) of expressions to add a rich class of expressions involving fixed point computations to L_{IO}. Three simple textbook problems, graph reachability, cycle detection, and constant propagation were shown to belong to L_{IO}. In the next two chapters, we will attempt to give further justification of the usefulness of our approach to algorithms. We will consider two non-trivial algorithmic problems and derive linear-time algorithms for both. In the first case, we see that our approach helps explain existing linear-time algorithm. In the second case, we see that our approach leads to the discovery of a new $O(N^3)$ algorithm (where N is the size of the input), significantly improving the best previously known $O(N^5)$ time algorithm.

Chapter 7

A Linear Time Algorithm To Solve Fixed-Point Equations On Transition Systems

7.1 Introduction

We look at the problem of computing the least fixed point of a system of equations over a transition system. The problem has been the subject of considerable interest because of applications to model-checking [58, 40], i.e. determining whether or not a given system satisfies a formula of the modal mu-calculus. Arnold and Crubille [5] presented the first linear-time algorithm for this problem, improving the best known algorithms [35, 40] that were all quadratic. Arnold and Crubille's algorithm is linear in the size of the transition system but is quadratic in the size of the formula. The currently best known algorithm [24, 109, 4] is linear in both the size of the transition system and the size of the formula. More general algorithms that can perform model-checking for the entire modal mu-calculus have also been presented in [23, 4, 91].

In this chapter, we specify the original problem as an SQ^+ specification, and show that this specification belongs to L_{IO}, the linear-time subset of SQ^+ presented in Chapter 6. This algorithm has been included in this thesis to illustrate the applicability of the three-step transformational program design methodology to problems that are a lot more complex than simple textbook problems like graph reachability and cycle testing. The problem tackled in this chapter serves as an excellent example for illustrating the usefulness of our algorithm design methodology because although the problem is sufficiently non-trivial, it is easy to state and understand.

The final algorithm derived in this chapter is similar to the algorithms described in [24, 109, 4], and is also linear in both the size of the transition system and the size of the formula. The original linear-time algorithm by Arnold and Crubille is difficult to understand because it works by associating program fragments with each distinct equation in the system of equations, and involves a complicated mechanism of putting these program fragments together into a program that computes the required fixed point in linear time. The proof of correctness and the time complexity analysis are complex, and give little intuition to the reader about the insights that lead to the discovery of this algorithm. The work of Vergauwen and Lewi [109] does a better job of explaining the algorithm, since they start with an inefficient but intuitively understandable algorithm, and use finite differencing to derive the final linear-time algorithm. However, their time complexity analysis is still quite complex. In contrast, we show that our methodology helps integrate the concerns of correctness and efficiency, and allows the algorithm design process to be guided by complexity considerations. We believe that the approach that we present here significantly simplifies the algorithm and its analysis of time complexity.

In Section 7.2 we present a formal statement of the problem as originally formulated by Arnold and Crubille [5]. In Section 7.3 we prove that the SQ^+ specification of this problem belongs to L_{IO}, and derive a linear-time algorithm. In Section 7.4 we give a brief description of the original algorithm by Arnold and

Crubille, and in Section 7.5, we describe how the two algorithms compare in terms of simplicity and efficiency.

7.2 Systems of Equations on Transition Systems

A (finite) transition system is a tuple $G = \langle S, T, \alpha, \beta \rangle$, where S is a finite set of states, T is a finite set of transitions (from a source state to a target state), and $\alpha : T \longrightarrow S$ and $\beta : T \longrightarrow S$ are respectively the source and the target mappings.

Consider the following sorted signature D with two sorts, σ_s for states, and σ_t for transitions, having the following operators:

- Constants 0_{σ_s} and 1_{σ_s} of sort σ_s.
- Constants 0_{σ_t} and 1_{σ_t} of sort σ_t.
- Binary operators $\cup_{\sigma_s}$ and $\cap_{\sigma_s}$ of sort $\sigma_s \times \sigma_s \longrightarrow \sigma_s$
- Binary operators $\cup_{\sigma_t}$ and $\cap_{\sigma_t}$ of sort $\sigma_t \times \sigma_t \longrightarrow \sigma_t$
- Unary operators A, A^*, B, B^* of sort $\sigma_t \longrightarrow \sigma_s$.
- Unary operators A', B' of sort $\sigma_s \longrightarrow \sigma_t$.

Given a transition system G, an operator ω of D is interpreted by $(\omega)_G$ in the following way.

$$\begin{array}{ll}
 & (0_{\sigma_s})_G = (0_{\sigma_t})_G = \{\} \\
 & (1_{\sigma_s})_G = S, (1_{\sigma_t})_G = T \\
(\cup_{\sigma_s})_G : 2^S \times 2^S \longrightarrow 2^S, & (\cup_{\sigma_s})_G(X_1, X_2) = X_1 \cup X_2 \\
(\cup_{\sigma_t})_G : 2^T \times 2^T \longrightarrow 2^T, & (\cup_{\sigma_t})_G(X_1, X_2) = X_1 \cup X_2 \\
(\cap_{\sigma_s})_G : 2^S \times 2^S \longrightarrow 2^S, & (\cap_{\sigma_s})_G(X_1, X_2) = X_1 \cap X_2 \\
(\cap_{\sigma_t})_G : 2^T \times 2^T \longrightarrow 2^T, & (\cap_{\sigma_t})_G(X_1, X_2) = X_1 \cap X_2 \\
(A)_G : 2^T \longrightarrow 2^S, & (A)_G(Y) = \{\alpha(t) : t \in Y\} \\
(B)_G : 2^T \longrightarrow 2^S, & (B)_G(Y) = \{\beta(t) : t \in Y\} \\
(A^*)_G : 2^T \longrightarrow 2^S, & (A^*)_G(Y) = \{s \in S \mid \forall t \in T \ (s = \alpha(t) \implies t \in Y)\} \\
(B^*)_G : 2^T \longrightarrow 2^S, & (B^*)_G(Y) = \{s \in S \mid \forall t \in T \ (s = \beta(t) \implies t \in Y)\} \\
(A')_G : 2^S \longrightarrow 2^T, & (A')_G(X) = \{t \in T \mid \alpha(t) \in X\} \\
(B')_G : 2^S \longrightarrow 2^T, & (B')_G(X) = \{t \in T \mid \beta(t) \in X\}
\end{array}$$

Let $\mathcal{L} = \{z_1, \ldots, z_p\}$ and $\mathcal{L}' = \{z'_1, \ldots, z'_{p'}\}$ be two sets of parameters associated with a transition system G such that each parameter z_i (respectively z'_i) is associated with a subset $(z_i)_G$ of S (respectively, a subset $(z'_i)_G$ of T). Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ and $\mathcal{Y} = \{y_1, \ldots, y_m\}$ be two sets of variables of sort σ_s and σ_t respectively. Then, all well-formed terms W and W' of sorts σ_s and σ_t respectively, over $D \cup \mathcal{L} \cup \mathcal{L}' \cup \mathcal{X} \cup \mathcal{Y}$ can be associated with mappings

$$(W)_G : (2^S)^n \times (2^T)^m \longrightarrow 2^S, \text{ and } (W')_G : (2^S)^n \times (2^T)^m \longrightarrow 2^T.$$

A system of equations Σ over $\mathcal{L}, \mathcal{L}', \mathcal{X}, \mathcal{Y}$ is a pair $(\Sigma_{\sigma_s}, \Sigma_{\sigma_t})$ of sets

$$\Sigma_{\sigma_s} = \{x_i = W_i : x_i \in \mathcal{X}\}, \text{ and } \Sigma_{\sigma_t} = \{y_i = W'_i : y_i \in \mathcal{Y}\},$$

where $W_1, \ldots, W_n$ are well-formed terms of sort σ_s and $W'_1, \ldots, W'_m$ are well-formed terms of sort σ_t, over $D \cup \mathcal{L} \cup \mathcal{L}' \cup \mathcal{X} \cup \mathcal{Y}$. Thus,

$$(\Sigma)_G = \langle (W_1)_G, \ldots, (W_n)_G, (W'_1)_G, \ldots, (W'_m)_G \rangle$$

is a mapping from $(2^S)^n \times (2^T)$ to itself. **The problem is to compute the least fixed point of the system of equations Σ.**

A system of equations Σ is said to be *simple* if each equation has one of the following forms:

1. $x = k$, where k is either a constant or a parameter of sort σ_s
2. $x = x' \cup_{\sigma_s} x''$
3. $x = x' \cap_{\sigma_s} x''$
4. $x = A(y)$
5. $x = B(y)$
6. $x = A^*(y)$
7. $x = B^*(y)$
8. $y = k'$, where k' is either a constant or a parameter of sort σ_t
9. $y = y' \cup_{\sigma_t} y''$
10. $y = y' \cap_{\sigma_t} y''$
11. $y = A'(x)$
12. $y = B'(x)$

Every system of equations Σ can be transformed into a simple system of equations Σ' with only a constant factor increase in the size of the system, where the size of system Σ is defined as the number of occurrences of operators and parameters in Σ. So, we shall look at the problem of computing the least fixed point of a simple system of equations Σ on a transition system G with parameters $\mathcal{L}$ and $\mathcal{L}'$.

7.3 Linear-Time Algorithm

We assume that we are given the input set of states S, set of transitions T, parameters $z_1, \ldots, z_p$ which are subsets of S, parameters $z'_1, \ldots, z'_p$ which are subsets of T, and maps α and β that map transitions to their source and target states respectively. We assume that the inputs have been pre-processed so that each state is an element of base type b_s and each transition is an element of base type b_t, where base types b_s and b_t satisfy the subtype constraints $b_s < \sigma_s$ and $b_t < \sigma_t$ respectively.

Let TE be a a type environment containing the following types for the input parameters and variables.

- $S : \mathit{set}(b_s)$
- $T : \mathit{set}(b_t)$
- $z_i : \mathit{set}(b_s)$ for $i = 1, \ldots, p$
- $z'_i : \mathit{set}(b_t)$ for $i = 1, \ldots, p'$
- $\alpha : \mathit{strong\_smap}(b_t, b_s)$
- $\beta : \mathit{strong\_smap}(b_t, b_s)$
- $x_i : \mathit{strong\_set}(b_s)$ for $i = 1, \ldots, n$
- $y_i : \mathit{strong\_set}(b_t)$ for $i = 1, \ldots, m$

```
$x[1] = x[2] = \ldots x[n] = \{\}$;
$y[1] = y[2] = \ldots y[m] = \{\}$;
$x' = x$;
$y' = y$;
$x = [W_1(x', y'), \ldots, W_n(x', y')]$;
$y = [W'_1(x', y'), \ldots, W_m(x', y')]$;
while $(x \neq x') \vee (y \neq y')$ loop
    $x' = x$;
    $y' = y$;
    $x = [W_1(x', y'), \ldots, W_n(x', y')]$;
    $y = [W'_1(x', y'), \ldots, W_m(x', y')]$;
end loop
```

Figure 7.1: Naive algorithm based on Tarski Iteration

Let C be the set of subtype constraints $\{b_s < \sigma_s, b_t < \sigma_t\}$. Let Σ be the simple system of equations $\{x_i = W_i, i = 1, \ldots, n\} \cup \{y_i = W'_i, i = 1, \ldots, m\}$. It is easy to verify that each W_i is a well-typed SQ$^+$ expression of type *strong_set*(b_s) and each W'_i is a well typed SQ$^+$ expression of type *strong_set*(b_t), i.e.

$$TE, C \vdash_S W_i : \mathit{strong\_set}(b_s) \text{ and } TE, C \vdash_S W'_i : \mathit{strong\_set}(b_t).$$

Thus, the least fixed point of the system of equations Σ may be specified as the following well-typed SQ$^+$ program.

$$\text{LFP}_{\leq}([W_1, \ldots, W_n, W'_1, \ldots, W'_m], [x_1, \ldots, x_n, y_1, \ldots, y_m]) \tag{7.1}$$

where the partial order $\leq$ is the point-wise $\subseteq$ relation.

It is easily verified that each expression W_i and W'_i is monotone in $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$. Moreover, the range of each expression W_i and W'_i is bounded by the finite sets S and T. It follows that Expression 7.1 can be computed by the algorithm in Figure 7.1 using Tarski iteration. The time complexity of this naive implementation is $O((n+m)^2 \times (\#S + \#T)^2)$ assuming that there are n equations of sort σ_s and m equations of sort σ_t.

The algorithm in Figure 7.1 is expensive because of the wasteful re-computation of all expressions W_i and W'_i in each iteration. A more efficient algorithm may be obtained by using dominated convergence and finite differencing. In fact, as seen from Theorem 6.4.1 presented in Chapter 6, Specification 7.1 can be implemented in linear time if the following conditions are satisfied.

1. Each expression W_i and W'_i is monotone in $x_1, \ldots, x_n, y_1, \ldots, y_m$.

2. Each expression W_i and W'_i is a linear-cost expression.

3. Each expression $W_i - x_i$ and $W'_i - y_i$ is weakly continuous with respect to the set of modifications x_i *with* $:= s$ for $i = 1, \ldots, n$ and y_i *with* $:= t$ for $i = 1, \ldots, m$.

If the above conditions are satisfied, then the implementation of Specification 7.1 given in Figure 7.2 runs in linear time.

The algorithm in Figure 7.2 is based on the following idea.

1. Initialize each $x[i]$ and $y[i]$ to the empty set $\{\}$.

2. Establish the invariants $dx[i] = W_i - x_i$ and $dy[i] = W'_i - y_i$ for all i.

3. While there exists a $dx[i]$ or $dy[i]$ that is non-empty, extract an element from it, and add it x_i or y_i respectively, and re-establish the invariants $dx[i] = W_i - x_i$ and $dy[i] = W'_i - y_i$ for all i.

```
$x[1] = x[2] = \dots x[n] = \{\}$;
$y[1] = y[2] = \dots y[m] = \{\}$;
for $i = 1, \dots, n$ loop
    $dx[i] = W_i(x, y)$
end loop
for $i = 1, \dots, m$ loop
    $dy[i] = W_i'(x, y)$
end loop
$I_1 = \{i \in 1, \dots, n : dx[i] \neq \{\}\}$;
$I_2 = \{i \in 1, \dots, m : dy[i] \neq \{\}\}$;
while $(I_1 \cup I_2 \neq \{\})$ loop
    if $I_1 \neq \{\}$ then
        $i =\ni I_1$;
        $z =\ni dx[i]$;
        for $j = 1, \dots, n$ loop
            ...   −− Code to re-establish the invariant $dx[j] = W_j(x, y) - x_j$
            ...   −− with respect to the modification $x[i]$ $with := z$
            if $dx[j] \neq \{\}$ then $I_1$ $with := j$ endif
        end loop
        for $j = 1, \dots, m$ loop
            ...   −− Code to re-establish the invariant $dy[j] = W_j'(x, y) - y_j$
            ...   −− with respect to the modification $x[i]$ $with := z$
            if $dy[j] \neq \{\}$ then $I_2$ $with := j$ endif
        end loop
    else
        $i =\ni I_2$;
        $z' =\ni dy[i]$;
        for $j = 1, \dots, n$ loop
            ...   −− Code to re-establish the invariant $dx[j] = W_j(x, y) - x_j$
            ...   −− with respect to the modification $y[i]$ $with := z'$
            if $dx[j] \neq \{\}$ then $I_1$ $with := j$ endif
        end loop
        for $j = 1, \dots, m$ loop
            ...   −− Code to re-establish the invariant $dy[j] = W_j'(x, y) - y_j$
            ...   −− with respect to the modification $y[i]$ $with := z'$
            if $dy[j] \neq \{\}$ then $I_2$ $with := j$ endif
        end loop
    endif
end loop
```

Figure 7.2: Linear-time implementation using dominated convergence and finite differencing

The weak continuity of each of the expressions $W_i - x_i$ and $W_i' = y_i$ ensures that the cumulative cost of re-establishing the invariant $dx[i] = W_i - x_i$ or $dy[i] = W_i' - y_i$ is bounded by $O(\#S + \#T)$, and therefore the complexity of the algorithm is linear in the size of the transition system, i.e. $O(\#S + \#T)$.

Let us prove that each of the expressions W_i and W_i' is of linear cost, and that each of the expressions $W_i - x_i$ and $W_i' - y_i$ is weakly continuous with respect to the set of modifications x_i $with := s$ for $i = 1, \dots, n$ and y_i $with := t$ for $i = 1, \dots, m$. Note that each expression W_i is of one of the forms (1)-(7), and W_i' of the forms (8)-(12) listed in Section 7.2. The monotonicity of each of these expressions is easily verified. With the possible exception of expressions of the form (6) and (7) (i.e. A^* and B^*), it is easily verified that all other expressions are simple linear-cost expressions. Consider the expression of form (6), i.e.

$$A^*(y) = \{s \in S \mid \forall t \in T\ (s = \alpha(t) \implies t \in y)\}. \tag{7.2}$$

Expression 7.2 can be re-written as the equivalent expression

$$A^*(y) = \{s \in S \mid \forall t \in \alpha^{-1}\{s\}\ t \in y\} \tag{7.3}$$

Recall that expression $E_1(f : \mathit{strong\_smap}(b_1, b_2)) : \mathit{strong\_mmap}(b_2, b_1) = f^{-1}$ is an input-bound simple linear-cost expression, and that expression $E_2(s : \mathit{set}(b_2), f : \mathit{strong\_mmap}(b_2, b_1), y : \mathit{strong\_set}(b_1)) : \mathit{strong\_set}(b_2) = \{x \in s \mid \forall t \in f\{s\}\ t \in y\}$ is a simple linear-cost expression. It follows from the composition rule for linear-cost expressions (in Chapter 6) that expression $E_2(s, E_1(f), y)$ is also a linear-cost expression. Hence, Expression 7.3 is a linear-cost expression. Similarly, expression $B^*(y)$ can be re-written as

$$B^*(y) = \{s \in S \mid \forall t \in \beta^{-1}\{s\}\ t \in y\}, \tag{7.4}$$

and is also a linear-cost expression. Now, it only remains to show that each expression $W_i - x_i$ and $W_i' - x_i'$ is weakly continuous with respect to the set of modifications $x_i\ \mathit{with} := s$ for $i = 1, \ldots, n$ and $y_i\ \mathit{with} := t$ for $i = 1, \ldots, m$.

If W_i is of the form (1)-(3), or W_i' is of the form (8)-(10), it is trivial to show the weak continuity of $W_i - x_i$ and $W_i' - y_i$. We consider the rest of the cases below.

Case 4 W_i is the expression $A(y_j) = \{\alpha(t) : t \in y_j\}$ where $y_j \in \{y_1, \ldots, y_m\}$.

Consider the problem of maintaining the invariant $x_i = A(y_j)$ with respect to the set of modifications $x_k\ \mathit{with} := s$ for $k = 1, \ldots, n$ and $y_k\ \mathit{with} := t$ for $k = 1, \ldots, m$. Clearly the value of $A(y_j)$ does not change with respect to the modifications $\{x_k\ \mathit{with} := s, k = 1, \ldots, n\}$ and $\{y_k\ \mathit{with} := t, k = 1, \ldots, j-1, j+1, \ldots, m\}$. Now consider the modification $y_j\ \mathit{with} := t$. The update code required to re-establish the invariant $x_i = A(y_j)$ is $x_i\ \mathit{with} := \alpha(t)$. Since x_i is a strongly based set of type $\mathit{strong\_set}(b_s)$ and α is a strongly based single-valued map of type $\mathit{strong\_smap}(b_t, b_s)$, the cost of performing the update is $O(1)$. From a simple application of the composition rules, it follows that expression $W_i - x_i$ is strongly continuous (and therefore also weakly continuous) with respect to the set of modifications $x_k\ \mathit{with} := s$ for $k = 1, \ldots, n$ and $y_k\ \mathit{with} := t$ for $k = 1, \ldots, m$.

Case 5 W_i is the expression $B(y_j) = \{\beta(t) : t \in y_j\}$ where $y_j \in \{y_1, \ldots, y_m\}$.

This case is similar to Case 4 and is omitted.

Case 6 W_i is the expression $A^*(y_j) = \{s \in S \mid \forall t \in T\ (s = \alpha(t) \implies t \in y_j)\}$ where $y_j \in \{y_1, \ldots, y_m\}$.

As mentioned earlier, expression $A^*(y_j)$ can be re-written as the equivalent form

$$\{s \in S \mid \forall t \in \alpha^{-1}\{s\}\ t \in y_j\},$$

which is further equivalent to

$$\text{let } f = \{[s, \#\{t \in \alpha^{-1}\{s\} \mid t \notin y_j\}] : s \in S\} \text{ in } \{s \in S \mid f(s) = 0\}.$$

Consider the problem of maintaining the invariant $x_i = A^*(y_j)$ with respect to the set of modifications $x_k\ \mathit{with} := s$ for $k = 1, \ldots, n$ and $y_k\ \mathit{with} := t$ for $k = 1, \ldots, m$. Clearly the value of $A^*(y_j)$ does not change with respect to the modifications $\{x_k\ \mathit{with} := s, k = 1, \ldots, n\}$ and $\{y_k\ \mathit{with} := t, k = 1, \ldots, j-1, j+1, \ldots, m\}$. Now consider the modification $y_j\ \mathit{with} := t$. The update code required to re-establish the invariant $f = \{[s, \#\{t \in \alpha^{-1}\{s\} \mid t \notin y_j\}] : s \in S\}$ is $f(\alpha(t)) - := 1$ which can be done in $O(1)$ time. Moreover, the expression $\{s \in S \mid f(s) = 0\}$ is also strongly continuous with respect to the modification $f(s) - := 1$. Since expression f is input bound, it follows from two applications of the composition rule that expression $A(y_j) - x_i$ is strongly continuous (and therefore also weakly continuous) with respect to the set of modifications $x_k\ \mathit{with} := s$ for $k = 1, \ldots, n$ and $y_k\ \mathit{with} := t$ for $k = 1, \ldots, m$.

Case 7 W_i is the expression $B^*(y_j) = \{s \in S \mid \forall t \in T \ (s = \beta(t) \implies t \in y_j)\}$ where $y_j \in \{y_1, \ldots, y_m\}$.

This case is similar to Case 6 and is omitted.

Case 11 W_i' is $A'(x_j) = \{t \in T \mid \alpha(t) \in x_j\}$ where $x_j \in \{x_1, \ldots, x_n\}$.

Consider the problem of maintaining the invariant $y_i = A'(x_j)$ with respect to the set of modifications $x_k \ with := s$ for $k = 1, \ldots, n$ and $y_k \ with := t$ for $k = 1, \ldots, m$. Clearly the value of $A'(x_j)$ does not change with respect to the modifications $\{x_k \ with := s, k = 1, \ldots, j-1, j+1, \ldots, n\}$ and $\{y_k \ with := t, k = 1, \ldots, m\}$. Now consider the modification $x_j \ with := s$. The update code required to re-establish the invariant $y_i = A'(x_j)$ is

```
for $t \in \alpha^{-1}\{s\}$ loop
    $y_i$ with := $t$
end loop
```

which takes time $O(\#\alpha^{-1}\{s\})$. The cumulative cost of the update code with respect to any sequence of additions of elements $s_1, s_2, \ldots, s_p$ to set x_j is $\Sigma_{l=1}^{p} O(\#\alpha^{-1}\{s_p\})$ which is bounded by $\Sigma_{s \in S} O(\#\alpha^{-1}\{s\}) = O(\#T)$. Thus, we see that expression $A'(x_j)$ is weakly continuous with respect to the set of modifications $x_k \ with := s$ for $k = 1, \ldots, n$ and $y_k \ with := t$ for $k = 1, \ldots, m$. From a simple application of the composition rule, it follows that expression $A'(x_j) - y_i$ is also weakly continuous with respect to the set of modifications $x_k \ with := s$ for $k = 1, \ldots, n$ and $y_k \ with := t$ for $k = 1, \ldots, m$.

Case 12 W_i' is $B'(x_j) = \{t \in T \mid \beta(t) \in x_j\}$ where $x_j \in \{x_1, \ldots, x_n\}$.

This case is similar to Case 11 and is omitted.

This concludes the proof that Specification 7.1 is of linear cost, and may be implemented to run in linear time using the algorithm in Figure 7.2.

7.4 A Brief Description of Arnold and Crubille's Algorithm

In the original algorithm by Arnold and Crubille [5], each state s in set S is associated with a boolean valued attribute $s.x$ for each variable x of sort σ_s. Similarly, each transition t in set T is associated with a boolean valued attribute $t.y$ for each variable y of sort σ_t. The attribute $s.x$ (respectively $t.y$) is *true* if state s belongs to set x (respectively transition t belongs to set y). The main idea behind the algorithm may be understood by answering the following questions.

1. How can the addition of state s to one or more sets $x_{i_1}, x_{i_2}, \ldots$ affect the values of other sets ?
2. How can the addition of transition t to one or more sets $y_{i_1}, y_{i_2}, \ldots$ affect the values of other sets ?

Consider the first question. Equations of the form $x_{i_1} = x_{i_2} \cup_{\sigma_s} x_{i_3}$ or $x_{i_1} = x_{i_2} \cap_{\sigma_s} x_{i_3}$ may result in the addition of state s to other sets of states. Equations of the form $y_{i_1} = A'(x_{i_2})$ or $y_{i_1} = B'(x_{i_2})$ may result in the addition of a transition t (such that $\alpha(t) = s$ or $\beta(t) = s$) to other sets of transitions. Next, consider the second question. Equations of the form $y_{i_1} = y_{i_2} \cup_{\sigma_t} y_{i_3}$ or $y_{i_1} = y_{i_2} \cap_{\sigma_t} y_{i_3}$ may result in the addition of transition t to other sets of transitions. Equations of the form $x_{i_1} = A(y_{i_2})$, $x_{i_1} = B(y_{i_2})$, $x_{i_1} = A^*(y_{i_2})$, or $x_{i_1} = B^*(y_{i_2})$ may result in the addition of state $\alpha(t)$ or $\beta(t)$ to other sets of states.

The key idea behind the original algorithm is to propagate the effects of addition of states and transitions until no more states and transitions need to be added. Thus, whenever a state s is newly added to one or more sets $x_{i_1}, x_{i_2}, \ldots$, the algorithm ensures that all equations of the form $x_{i_1} = x_{i_2} \cup_{\sigma_s} x_{i_3}$, $x_{i_1} = x_{i_2} \cap_{\sigma_s} x_{i_3}$, $y_{i_1} = A'(x_{i_2})$, and $y_{i_1} = B'(x_{i_2})$ are re-visited to see if the state s, or a transition t (such that $\alpha(t) = s$ or $\beta(t) = s$) needs to be added to any other sets. Similarly, whenever a transition t is added to one or more sets $y_{i_1}, y_{i_2}, \ldots$, the algorithm ensures that all equations of the form $y_{i_1} = y_{i_2} \cup_{\sigma_t} y_{i_3}$, $y_{i_1} = y_{i_2} \cap_{\sigma_t} y_{i_3}$, $x_{i_1} = A(y_{i_2})$, $x_{i_1} = B(y_{i_2})$, $x_{i_1} = A^*(y_{i_2})$, and $x_{i_1} = B^*(y_{i_2})$ are re-visited to see if transition t or states $\alpha(t)$ and $\beta(t)$ need to be added to other sets. The algorithm has a sophisticated control flow mechanism that uses a combination of recursion and iteration, and is cleverly designed to guarantee that whenever a state or transition is newly added to a set, the equations of the appropriate form are re-visited to propagate the effects of the addition. The efficiency of the algorithm rests on the fact that whenever an equation is re-visited, it is re-evaluated incrementally rather than from scratch.

7.5 Comparison Between the Two Algorithms

Let us see how our algorithm compares with that of Arnold and Crubille. A close look at the two algorithms reveals that both deal mainly with the following three issues.

1. Incremental re-evaluation of each expression W_i and W_i' with respect to the addition of a state s to set x_i or transition t to set y_i.

2. Ensure that whenever a state or transition is added to a set, the appropriate equations are re-visited to propagate the effect of the addition of a state or transition, to other sets. Make sure that the algorithm terminates only when a fixed-point has been reached.

3. Appropriate data structures that allow the addition of a state s (respectively a transition t) to a set of states (respectively transitions) in $O(1)$ time.

We derive our linear-time algorithm by following a process of top-down step-wise refinement. We start with Specification 7.1. The monotonicity of all expressions W_i and W_i', and the fact that the ranges of these expressions are finitely bounded by sets S and T respectively guarantees that the desired least fixed point exists and can be naively computed by the algorithm in Figure 7.1. The next step is to use dominated convergence and finite differencing to eliminate the wasteful re-computation of each expression W_i and W_i' in each iteration. This leads to the development of the algorithm in Figure 7.2. The pieces left unspecified in the pseudo-code in Figure 7.2 are the parts dealing with the incremental re-evaluation of expressions W_i and W_i' with respect to addition of a states and transitions to sets x_i or y_i, and can be dealt with independently. Thus, there is a clear separation of first two issues mentioned above. The third issue which deals with the problem of data-structure selection (to ensure $O(1)$ time cost of element additions) is taken care of by the type-system. The well-typedness of the SQ$^+$ specification, guarantees that the final High SETL implementation is also well-typed, which further guarantees that all associative access operations including set element additions are implementable in $O(1)$ time. Once again, there is a clear separation between the issue of data-structure selection and the other two issues. The three-step algorithm design methodology based on dominated convergence, finite differencing and data-structure selection homes in on the linear-time solution in exactly the same way as it did for textbook problems like graph reachability, cycle testing, and constant propagation.

The algorithm by Arnold and Crubille, in contrast, is a lot more difficult to understand because there is no clear separation between the three issues. First, the authors have to rely on a low-level description of data-structures that allow $O(1)$ time additions of states and transitions to sets. As a result, the reader is needlessly burdened with low-level implementation details from the very beginning. The algorithm contains two key procedures `update_state` and `update_transition` which ensure that the appropriate set of equations are re-evaluated whenever a state or transition is newly added to a set. The correctness of Arnold and Crubille's algorithm rests on the fact that procedure `update_state` is always called with argument s subsequent to the addition of state s to some set x_i, and that procedure `update_transition` is always called with argument t subsequent to the addition of transition t to some set y_i. The above is ensured by the use of a complicated mixture of recursion and iteration that is so difficult to understand that it is not even immediately obvious that the algorithm even terminates. First, a proof of partial-correctness is given, in which it is shown that if the algorithm terminates, then it computes the desired least fixed-point. It is left to the readers to convince themselves that the total number of times each procedure `update_state` and `update_transition` can get called is bounded by $n \times \#S$ and $m \times \#T$ respectively, and this fact is subsequently used to derive a linear upper bound on the running time of the algorithm.

7.5.1 A Closer Look at the Running Times of the Two Algorithms

Both algorithms are linear in the size of the transition system, i.e. $O(\#S + \#T)$. In the analysis of both algorithms, it is assumed that the number of equations on the transition system, i.e. $n + m$ is a constant. Let us take a look at how the running times of the two algorithms vary with the number of equations in the transition system. The running time of the algorithm by Arnold and Crubille is $O((n+m)^2 \times (\#S+\#T))$ on an array-based model of computation. It is easy to check that the running time of our algorithm (outlined in the program in Figure 7.2) is $O((n+m)^3 \times (\#S+\#T))$ on a pointer-machine model of computation, and $O((n+m)^2 \times (\#S+\#T))$ on an array-based model of computation. Thus, the constant factors related to the number of equations in the transition system are the same in both algorithms.

However, a closer look at the algorithm in Figure 7.2 reveals a possibility for further improvement. Consider the effect of the addition of state z to set x_i. All equations $x_j = W_j$ for $j = 1, \ldots, n$ and $y_j = W'_j$ are re-visited to re-establish the invariants $dx[j] = W_j - x_j$ and $dy[j] = W'_j - y_j$. This is wasteful because only those equations that have set x_i on their right hand side need to be re-visited. In the algorithm in Figure 7.2, an equation may be considered for re-evaluation as many as $(n + m) \times (\#S + \#T)$ times. As a result, even though the cumulative cost of re-establishing the invariants for all $n + m$ equations is bounded by the size of the total output (i.e. $n \times \#S + m \times \#T$), the cost of the algorithm is $O((n+m)^2 \times (\#S+\#T))$.

The algorithm in Figure 7.2 may be improved as follows. With each variable x_i (respectively y_i) associate an index set I_{x_i} (respectively I_{y_i}) containing indices of equations that have x_i (respectively y_i) on their right hand side. When state z is added to set x_i, only those equations whose indices lie in index set I_{x_i} are re-visited. Thus, an equation gets re-evaluated only if the value of at least one of the variables on its right hand side changes. Since each equation has at most two variables on its right hand side, and the number of times each x_i or y_i can change is bounded by $\#S$ and $\#T$ respectively, it follows that each equation can be re-visited at most $2 \times max(\#S, \#T)$ times. Once again, the cumulative cost of re-establishing the invariants for all $n + m$ equations is bounded by the size of the total output (i.e. $n \times \#S + m \times \#T$). Thus, the total running time of the algorithm is $O((n+m) \times (\#S+\#T))$ which is linear in the size of the transition system and the number of equations (or the size of the mu-calculus formula).

7.6 Conclusion

In this chapter, we looked at the problem of computing the fixed point of a system of equations on a transition system. Earlier, we had seen that the use of dominated convergence, finite differencing, and the use of the type system for data-structure selection helped transform abstract SQ$^+$ specifications of problems such as graph reachability, cycle testing, and constant propagation into highly efficient linear-time implementations. In this chapter, we showed that the same methodology can be used to get a linear-time algorithm for the problem of computing the fixed-point of a system of equations on a transition system. We compared our linear-time solution with the existing linear-time solution due to Arnold and Crubille, and although the essential idea behind the two algorithms is very similar, we believe that our approach significantly simplifies the original algorithm. Moreover the complexity of our algorithm matches the complexity of the best known algorithms for the problem.

Chapter 8

An Improved Intra-Procedural May-Alias Analysis Algorithm

8.1 Introduction

In this chapter, we look at the problem of computing intra-procedural may-alias information for programs in conventional imperative languages like C. The problem of computing alias information is of considerable pragmatic interest with applications to optimizing compilers, program environments, and program understanding tools. Existing work on alias analysis can be broadly classified into two categories: *flow sensitive* [10, 18, 32, 20, 33], and *flow insensitive* [6, 28, 47, 92, 100]. Flow sensitive algorithms are in general more precise, and more expensive than flow insensitive algorithms. Some recent work investigating the relative merits of the two approaches can be found in [50, 101, 116]. In this chapter, we look at a flow-sensitive intra-procedural may-alias computation. We look at the problem specification as formulated by Hind et al. [51], and use dominated convergence, finite differencing, and data structure selection to discover an improved algorithm for computing intra-procedural may-alias information.

Hind et al. ([51]) use a standard data flow framework [80, 104] to formulate an intra-procedural may-alias computation. They compute the intra-procedural aliasing information by applying well-known iterative techniques to a sparse version of the program Control Flow Graph (CFG) called the Sparse Evaluation Graph (SEG) ([21]). A transfer function (relating the data flow information flowing into and out of a node) is defined for each node that could potentially cause a pointer assignment. The input is the set of aliases holding at the entry node of the SEG. The computation applies the transfer function at each node to the set of aliases holding at the entry to the node, in order to compute the set of aliases holding on exit from the node. This set of aliases is propagated to all the successor nodes. The computation proceeds iteratively and in each iteration all the SEG nodes are re-visited to compute the new set of aliases holding on exit from the node, and update the set of aliases holding on entry to the successor nodes. The computation terminates when a complete iteration leads to no change in the sets of aliases holding on entry to any node in the SEG. It is assumed that precomputed information in the form of summary functions is available for all function-call sites in the procedure being analyzed. The time complexity of the intra-procedural may-alias computation for the algorithm presented by Hind et al. ([51]) is $O(N^6)$ in the worst case (where N is the size of the SEG).

In this chapter we present a new algorithm for intra-procedural may-alias computation with a worst case time complexity of $O(N^3)$. We formulate the intra-procedural may-alias computation as an SQ$^+$ specification, and show that it may be computed in linear time (linear in the sum of the sizes of the input and output). Since the worst case size of the output alias-relation is bounded by $O(N^3)$, (at most N^2 aliases can hold at any SEG node and there are at most N SEG nodes) it follows that our algorithm runs in worst case $O(N^3)$ time. Thus, our approach leads to the discovery of a much improved new algorithm.

Unfortunately, a pointer machine model turns out to be inadequate for an efficient implementation of our algorithm. For the remainder of this chapter we consider an extended type system additionally containing arbitrary length (but finite) tuples. If t is a tuple (implemented as a linked list), then the evaluation of expression $t[i]$ takes time proportional to i on a pointer machine. Upto now we have assumed that tuples are of fixed, finite lengths that are known at compile time. Thus, the time to evaluate expression $t[i]$ for any tuple t is bounded by the size of the tuple, which is a constant for the program. For this application, the length of tuples depends on the input, and cannot be assumed to be a constant. We conservatively assume that once a tuple of an arbitrary length is created, its length is not modified for the remainder of the execution of the program. Since linked lists are not appropriate data structures for implementing such tuples, we choose arrays as the alternative.

In Section 8.2, we informally describe the effect of the addition of arbitrary length tuples to the type system. In Chapter 6 we considered the problem of computing the least fixed point of a system of equations under the assumption that the number of equations is a constant. For the current problem, the number of equations will, however, depend on the input. In Section 8.3, we re-visit the problem of computing the least fixed-point of a system of equations under the setting of the new type system, and relax the assumption that the number of equations is a constant. We state sufficient conditions under which the fixed-point for such systems of equations may be computed in linear time. In Section 8.4, we provide a brief tutorial on *aliasing*. In Section 8.5 we formulate the may-alias analysis problem as a problem of computing the least fixed-point of a system of equations. In Section 8.6 we present two algorithms, the first by Hind et al. [51] that runs in $O(N^6)$ time, and the second which improves the first by using a worklist strategy and runs in $O(N^5)$ time. Finally in Section 8.7 we present our improved algorithm that runs in worst-case $O(N^3)$ time.

8.2 Extension of the Type System

The type system defined thus far contains arbitrary sized finite sets and maps but only fixed length tuples (similar to records). We use linked lists to implement such fixed length tuples, and the time taken to access the i^{th} element $t[i]$ of a tuple t is proportional to i. Under the assumption that the length of the tuple is bounded by a constant (independent of the program input), the time complexity of all such accesses is $O(1)$ time. Now, we would like to extend the type system with arbitrary length tuples (similar to arrays). In order to access element $t[i]$ efficiently, we will assume that the tuple is implemented as an array. Thus, for the remainder of this chapter, we will assume that the underlying model of computation is the Random Access Memory model (RAM) rather than the weaker pointer-machine model. Since the RAM model is strictly more powerful than a pointer-machine model, all the previous results about the time complexities of implementation of primitive operations will still be valid. The new set of types *Type* is given by

$$\begin{array}{lll} \tau & ::= & \mathit{bool} \mid \sigma \mid \mathit{strong\_set}(b) \mid \mathit{strong\_smap}(b,\sigma) \mid \mathit{strong\_mmap}(b,\sigma) \mid \mathit{tuple}(\tau) \\ \sigma & ::= & \mathit{int} \mid b \mid \mathit{set}(\sigma_1) \mid \mathit{smap}(\sigma_1,\sigma_2) \mid \mathit{mmap}(\sigma_1,\sigma_2) \mid \mathit{tuple}(\sigma) \end{array} \tag{8.1}$$

The above extension to the type system is quite non-trivial. For example, we now allow types of the form *tuple*(*strong_set*(b)), which is a tuple containing arbitrarily many strongly based sets. Up to now, the type system ensured that the total number of strongly based sets and maps in a program was a constant independent of the program input. This fact is crucial to the proof that operations such as set membership test on strongly-based sets can be performed in $O(1)$ time. Recall that an element x of base type b (satisfying the subtype constraint $b < \sigma$) is implemented as a record containing the corresponding value of type σ, and a pointer to a linked list of nodes. The linked list contains one node for every strongly-based set containing element x and one node for every strongly-based map (*smap* or *mmap*) containing x in its domain. In order to test the membership of element x in a strongly-based set s ($x \in s$), the linked list needs to be traversed to see if there is a node corresponding to set s. Under the assumption that the total number of strongly-based sets and maps in the program is a constant, the cost of traversing the list can also bounded by a constant, and is therefore $O(1)$ time. Unfortunately, we can no longer assume that the number of strongly-based sets and maps in the program is bounded by a constant, in the new type system. Although, types such as *set*(*strong_set*(b)) are still disallowed, types such as *tuple*(*strong_set*(b)) can lead to arbitrarily many strongly-based sets.

We solve this problem by modifying the implementation of base types in the following way. Let t be of type *tuple*(*strong_set*(b)) where base type b satisfies the subtype constraint $b < \sigma$. As before, we implement an element x of type b as a pointer to a record containing the corresponding value of type σ and a pointer to a linked list. The linked list contains only one node for corresponding to variable t, which further contains a pointer to an array of records that contains one record for each element of tuple t. Similarly, if t were of type $tuple(tuple(strong\_set(b)))$, the linked list would still contain only one node for t which would contain a pointer to an array of array of records. Such an implementation ensures that the number of nodes in the linked list is always bounded by the number of variables in the program. Thus, for any strongly-based set s or $t[i]$, the operations $x \in s$ and $x \in t[i]$ can both be performed in $O(1)$ time. Our implementation relies on a RAM model of computation to perform array accesses in $O(1)$ time.

The new type system incurs an additional cost in terms of space complexity. In the earlier type system, the use of based data structures increased the space utilization by no more than a constant factor. In the new type system however, the penalty in terms of space requirements could be substantially higher. Consider a tuple t of type $tuple(strong\_set(b))$. Let n denote the length of t, and m denote the number of distinct values of type b that exist during the life of the program. The use of based data structures could incur a space penalty of $O(nm)$. If we make the assumption that each array (of size n) is initialized in $O(1)$ time[1], then the time complexity of initialization is still linear, i.e. $O(m)$.

In the next section, we re-visit the problem of computing the least fixed point of a system of equations in the setting of the new type system, and relax the assumption about the number of equations in the system being a constant.

8.3 Computing the Fixed-Point of a System of Equations

Consider the problem of computing the least fixed-point of the following system of equations

$$\begin{aligned} x_1 &= F_1(x_1, x_2, \ldots, x_n, y) \\ x_2 &= F_2(x_1, x_2, \ldots, x_n, y) \\ &\vdots \\ x_n &= F_n(x_1, x_2, \ldots, x_n, y), \end{aligned} \tag{8.2}$$

Assume that y is an external input[2], each of the variables x_i is set-valued, the number of equations n is a constant, and that each expression $F_i(x_1, \ldots, x_n, y)$ is a well-typed SQ$^+$ expression having the same type as x_i. It was shown in Chapter 6(Theorem 6.4.1) that the problem of computing the least fixed point of the system of equations given by 8.2 could be formulated as the following SQ$^+$ specification.

$$\begin{aligned} \text{LFP}_{\leq}([&F_1(x_1, \ldots, x_n, y), \\ &F_2(x_1, \ldots, x_n, y), \\ &\ldots \\ &F_n(x_1, \ldots, x_n, y)], \\ &[x_1, \ldots, x_n]) \end{aligned} \tag{8.3}$$

Moreover, it was shown that Specification 8.3 could be computed in linear time (linear in the sum of the sizes of the input and the output) if the following conditions hold:

1. Each expression $F_i(x_1, \ldots, x_n, y)$ is monotonic in the arguments $x_1, \ldots, x_n$.

2. Each expression $F_i(x_1, \ldots, x_n, y)$ is of linear cost.

3. Each expression $F_i(x_1, \ldots, x_n, y) - x_i$ is weakly continuous with respect to the set of modifications $\{x_j \ with := z_i, j = 1, \ldots, n\}$.

Unfortunately, this result is no longer true if we relax the assumption that the number of equations n is a constant independent of the program input. Let us take a closer look at the time complexity of

[1]However, if the assumption that the initialization of an array of size n takes $O(1)$ time is not acceptable, then the time complexity of initialization is also $O(nm)$

[2]in general, there may be more than one external inputs

```
$x[1] = x[2] = \ldots x[n] = \{\}$;
for $i = 1, \ldots, n$ loop
    $dx[i] = F_i(x, y)$;        -- $F_i(x,y) \equiv F_i(x[1], x[2], \ldots, x[n], y)$
end loop
$I = \{i \in 1, \ldots, n : dx[i] \neq \{\}\}$;
while $(I \neq \{\})$ loop
    $i = \ni I$;
    $z = \ni dx[i]$;
    for $j = 1, \ldots, n$ loop
        ...   -- Code to re-establish the invariant $dx[j] = F_j(x,y) - x_j$
        ...   -- with respect to the modification $x[i]\ with := z$
        if $dx[j] \neq \{\}$ then $I\ with := j$ endif
    end loop
end loop
```

Figure 8.1: Implementation of Specification 8.3 using dominated convergence and finite differencing

implementation of Specification 8.3 shown in Figure 8.1, where x is a tuple containing the values $x_1, \ldots, x_n$ (i.e. $x[i] = x_i$).

If the above conditions of monotonicity and weak continuity hold, then a time complexity analysis[3] reveals that the cost incurred for each expression F_i in the algorithm given in Figure 8.1 is $O(\#y + \Sigma_{i=1}^{n}\#x[i])$. Thus, the total cost of the algorithm in Figure 8.1 is $O(n \times (\#y + \Sigma_{i=1}^{n}\#x[i]))$, i.e. n times the sum of the sizes of the input and the output. If n can no longer be assumed to be a constant, then the time complexity of our implementation is no longer linear in the sum of the input and output sizes.

Now, consider a special case of the system of equations given by 8.2 in which there is no input y, and each expression F_i depends only on a few of the parameters $x_1, \ldots, x_n$. More precisely, we consider the case in which the number of expressions $F_{i_1}, F_{i_2}, \ldots$ that depend on each parameter x_i is bounded by some constant c. Such a system of equations is shown below.

$$\begin{array}{rcl} x_1 & = & F_1(x_{11}, x_{12}, \ldots, x_{1k_1}) \\ x_2 & = & F_2(x_{21}, x_{22}, \ldots, x_{2k_2}) \\ & \vdots & \\ x_n & = & F_n(x_{n1}, x_{n2}, \ldots, x_{nk_n}), \end{array} \tag{8.4}$$

Assume that each variable x_{ij} belongs to the set $\{x_1, \ldots, x_n\}$, and that the number of occurrences of any variable x_i in the multi-set

$$\{x_{11}, x_{12}, \ldots, x_{1k_1}, x_{21}, x_{22}, \ldots, x_{2k_2}, \ \ldots, \ x_{n1}, x_{n2}, \ldots, x_{nk_n}\}$$

is bounded by a constant c. In this case, the input can be pre-processed (in $O(n)$ time) to compute a map

$$\mathit{affected} = \{[i,j] : \text{variable } x_i \text{ affects expression } F_j\},$$

and implementation given in Figure 8.1 can be modified by replacing the for-loop "for $j = 1, \ldots, n$ loop" with "for $j \in \mathit{affected}\{i\}$ loop". The cost incurred for each expression F_i in the resulting implementation is

$$O(\Sigma_{j \in \mathit{affected}\{i\}}\#x[j]).$$

Since the number of expressions affected by each variable x_i is bounded by a constant c, the total cost of the modified implementation is $O(c \times \Sigma_{i=1}^{n}\#x[i])$, which is linear in the size of the output. Thus, we have the following theorem.

[3]we assume a RAM model, and that tuple access $x[i]$ takes $O(1)$ time

Theorem 8.3.1 *The system of equations given by 8.4 can be solved in time linear in the size of the output if the following conditions hold.*

1. *Each expression $F_i(x_{i1}, \dots, x_{ik_i})$ is monotonic in the arguments $x_{i1}, \dots, x_{ik_i}$.*
2. *Each expression $F_i(x_{i1}, \dots, x_{ik_i})$ is of linear cost.*
3. *Each expression $F_i(x_{i1}, \dots, x_{ik_i}) - x_i$ is weakly continuous with respect to the set of modifications $\{x_j \text{ with} := z_i, x_j \in \{x_{i1}, \dots, x_{ik_i}\}\ \}$.*
4. *The number of occurrences of any variable x_i in the multi-set*

$$\{x_{11}, x_{12}, \dots, x_{1k_1}, x_{21}, x_{22}, \dots, x_{2k_2},\ \dots,\ x_{n1}, x_{n2}, \dots, x_{nk_n}\}$$

is bounded by a constant.

Theorem 8.3.1 will be used to get a linear-time algorithm for computing may-alias information.

8.4 Preliminaries

Consider the C assignment statement

$$a = \&b; \tag{8.5}$$

The effect of the assignment is to assign the address of b to a. Therefore, after execution of Statement (8.5), expressions $*a$, and b refer to the same storage location. Consequently, any modification to the value stored at the location corresponding to $*a$ will result in the modification of the value stored at the location corresponding to b, and vice-versa. In this case, $*a$ and b are said to be *aliased.* We call expressions such as $*a$, and b, *access-paths* ([60]). More precisely, an access-path is the l-value of an expression constructed from variables, pointer dereferences, and structure field selection operators. The aliasing of $*a$ and b is represented by the alias-pair $\langle *a, b\rangle$.

An *alias relation* R at statement S is a set of alias-pairs $\langle x, y\rangle$. Such a relation R is the *must-alias* relation at statement S, if, an alias-pair $\langle x, y\rangle$ belongs to R, iff access-paths x and y refer to the same storage location in all execution instances of statement S. The absence of an alias-pair $\langle u, v\rangle$ from R implies that access-paths u and v refer to different locations in at least one execution instance of statement S. An alias-relation R is the *may-alias* relation at statement S, if, an alias-pair $\langle x, y\rangle$ belongs to R, iff access-paths x and y refer to the same storage location in *some* execution instance of statement S. Thus, the absence of an alias pair $\langle u, v\rangle$ from R implies that u and v must refer to different storage locations in all execution instances of statement S.

The computation of the actual must-alias and may-alias relations at all program points is undecidable ([59, 76]). Therefore, we must be satisfied with computing safe approximations to these relations. In order to understand what it means for an alias relation to be a *safe approximation* to the must-alias relation (respectively, the may-alias relation), we must understand how the information contained in the must-alias relation (respectively the may-alias relation) is used. For each alias pair $\langle x, y\rangle$ in the computed must-alias relation, we want to be sure that x and y refer to the same storage location in all execution instances. Therefore, any under-approximation (subset) of the actual must-alias relation is a safe approximation *e.g.* the empty set $\{\}$. For each alias pair $\langle u, v\rangle$ *not* in the computed may-alias relation, we want to be sure that u and v refer to different storage locations in all execution instances. Therefore, any over-approximation (*i.e.* superset) of the actual may-alias relation is a safe approximation *e.g.* the complete relation. It is fairly obvious that the actual must-alias relation is reflexive, symmetric, and transitive, and that the actual may-alias relation is reflexive, and symmetric (though not necessarily transitive). It is not difficult to prove that if relation R is a safe approximation to the actual must-alias relation, then the reflexive, symmetric and transitive closure of R is also safe. Correspondingly, it is not difficult to see that if relation R is a safe approximation to the actual may-alias relation, then the largest reflexive and symmetric subset of R is also safe.

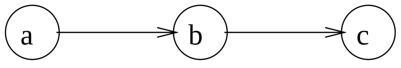


Figure 8.2: An example alias graph

In this chapter, we will be concerned with the problem of computing a safe approximation to the may-alias relation at every program point, and from now on whenever we refer to an alias relation, we will mean a may-alias relation. May-alias relations, in addition to being reflexive and symmetric, also satisfy a congruence property, *viz.* the existence of a may-alias pair $\langle x, y \rangle \in R$ implies the existence of another may-alias pair $\langle *x, *y \rangle$ in R, if x and y are non-null, pointer-valued access paths. It is again not difficult to see that if R is a safe approximation to the actual may-alias relation, then the largest subset of R that satisfies the congruence property is also safe.

Since the may-alias analysis requires the computation of a may-alias relation at every program point, it is beneficial to have a compact representation for may-alias relations. The goal is to minimize space usage for alias information by not explicitly storing alias pairs that are inferable from the reflexivity, symmetry, or congruence properties of may-alias relations. The next sub-section gives a brief description of the compact representation used in [51].

8.4.1 Compact Representation of Alias Information

Hind et al. ([51]) use a compact representation of alias information in which the memory locations are associated with names, and are referred to as *named objects* ([53, 19]). The names are either user variable names or new names created by the analysis. The compact representation of an alias relation requires that for each alias-pair:

1. at least one access path component is a named object *i.e.*, does not contain a dereference, and
2. the other access path component has no more than one level of dereferencing.

Thus, all alias-pairs are of the form $\langle *a, b \rangle$ (respectively $\langle b, *a \rangle$), or $\langle a, b \rangle$, where a and b are named objects. It is important to note that alias pairs of the form $\langle a, b \rangle$, where both a and b are named objects can arise only because of *structural* aliasing, *i.e.* because of program constructs such as C's union or FORTRAN's EQUIVALENCE statement. These may-alias pairs are in fact must-alias pairs at all program points. Such pairs of named objects are therefore closed under transitivity[4], *i.e.* if $\langle a, b \rangle \in R$, and $\langle b, c \rangle \in R$, then $\langle a, c \rangle \in R$. However, other alias pairs of the form $\langle *a, b \rangle$ are not closed under transitivity.

The compact representation of the alias relation, can be mapped to and from a directed-graph-based representation. Each node in the directed graph corresponds to a distinct location. Thus, each equivalence class of named objects is mapped to a node in the graph. Furthermore, every alias-pair of the form $\langle *a, b \rangle$ contributes a directed edge from the node corresponding to a to the node corresponding to b. Such a graph is called an *alias-graph.* For example, look at the alias-graph in Figure 8.2 (taken from [51]). The corresponding compact representation for this alias graph is $\{\langle *a, b \rangle, \langle *b, c \rangle\}$. The explicit representation of the alias information may be extracted from the compact representation (or the alias graph) by using Definition 8.6. If relation R is a compact representation of an alias relation, we define the corresponding explicit aliases of a_i, where a is a named object and i is the number of dereferences ($e.g.$ $a_2 \overset{def}{=} **a$) by

$$ExplicitAliases(a, i) = \begin{cases} \{a\} \text{ if } i = 0 \\ \{r : \langle *s, r \rangle \in R | s \in ExplicitAliases(a, i-1)\} \text{ if } i > 0 \end{cases} \tag{8.6}$$

In terms of the alias graph, we can think of $ExplicitAliases(a, i)$ as returning the set of vertices reachable by paths of length i from vertex a. It is easy to verify that the explicit alias information of the alias graph in Figure 8.2 includes the alias-pairs $\langle *a, b \rangle, \langle *b, c \rangle, \langle **a, c \rangle, \langle a, a \rangle, \langle b, b \rangle, \langle c, c \rangle$, and all other alias-pairs that can *further* be inferred by reflexivity and congruence. Note that in this example, none of the named objects are aliased to each other, and hence each equivalence class of named-objects is a singleton set.

[4]remember that must alias relations are transitively closed.

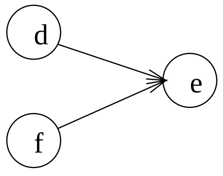


Figure 8.3: Another alias graph

It is however, important to realize that the compact representation comes at the cost of a possible loss in precision. For example, consider the explicit alias relation

$$R = \{\langle *a, b\rangle, \langle *b, c\rangle, \langle **a, *b\rangle, \langle a, a\rangle, \langle b, b\rangle, \langle c, c\rangle\}. \tag{8.7}$$

Note, that relation R contains the alias pairs $\langle *a, b\rangle$, and $\langle *b, c\rangle$, but does not contain the alias pair $\langle **a, c\rangle$. This implies that although it is possible for $*a$ to be aliased to b, and $*b$ to be aliased to c, it is never possible for $**a$ to be aliased to c. This information, unfortunately, cannot be captured by the compact representation. The best compact representation of relation R will still be the alias-graph in Figure 8.2, and it is easily verified that $ExplicitAliases(a, 2) = \{c\}$, *i.e.* the explicit alias representation constructed from relation R contains the alias pair $\langle **a, c\rangle$. This is a consequence of the fact that a certain amount of transitivity is built into the way we extract explicit alias information from the compact representation using Definition 8.6. However, extraction of explicit alias pairs from the compact representation does not involve taking a full transitive closure. For example, looking at the alias graph in Figure 8.3, we see that the presence of alias pairs $\langle *d, e\rangle$ and $\langle *f, e\rangle (\equiv \langle e, *f\rangle)$ in the compact representation does not imply the existence of the pair $\langle *d, *f\rangle$ in the corresponding explicit representation. An interesting discussion on the relative merits and precision of both the compact and explicit representations may be found in [61] and the appendix of [51].

Definition 8.6 is a modified form of the definition given in [51], where an explicit recursive procedure $ComputeAliases(a, i)$ is given to compute the explicit aliases of named object a with i levels of dereference. The set of explicit aliases computed by Definition 8.6 is the same as those computed by the procedure $ComputeAliases$ in [51]. In the rest of the paper, we always use the compact representation of may-alias relations, and use Definition 8.6 to extract explicit alias information from the compact representations.

8.5 May-Alias Analysis

The intra-procedural may-alias analysis is performed on the *sparse evaluation graph* (SEG) ([21]). The SEG is a sparse representation of the *control flow graph* (CFG) comprising of *gen* nodes, *i.e.* nodes that may potentially modify the alias information, and *join* nodes, *i.e.* nodes where alias information is merged at join points. The SEG also contains a special entry node N_{Entry}, and an exit node N_{Exit}, such that every node is on some path from N_{Entry} to N_{Exit}. The computation of the may-alias information at each program point (*i.e.* on entry to, and, on exit from each SEG node n) is done by an iterative dataflow analysis ([80, 104]). Equations 8.8 and 8.9 (taken from the definitions in [51]) define the relationship between the alias information flowing into and out of an SEG node n.

Let In_n and Out_n be the alias relations holding immediately before and immediately after SEG node n. For a node n, In_n is the union of the Out sets of its predecessor nodes:

$$In_n = \bigcup_{p \in Preds(n)} Out_p. \tag{8.8}$$

The intuition behind Equation 8.8 is as follows. If an alias pair $\langle x, y\rangle \in Out_p$ for some predecessor (in the SEG) node p of node n, then, it may be the case that x and y are aliased in some execution instances in which control reaches node n from node p. Therefore, all such alias pairs $\langle x, y\rangle$ must be in the alias relation In_n holding at entry to node n.

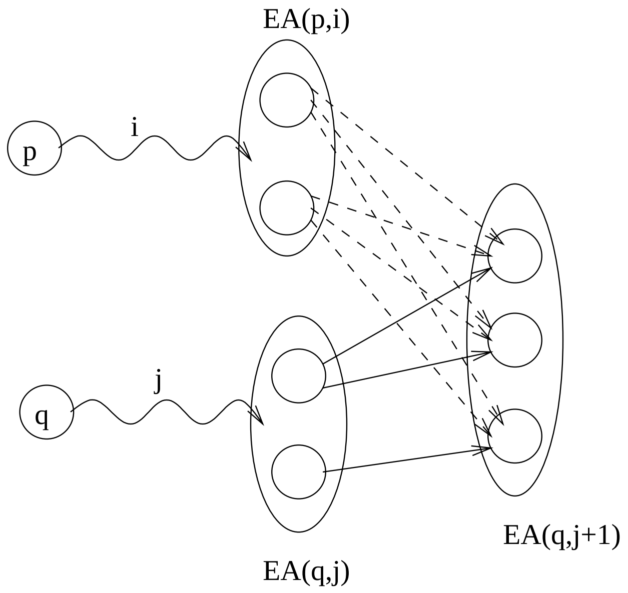


Figure 8.4: Dashed edges correspond to the alias pairs generated by the assignment $p_i = q_j$

The alias relation Out_n holding on exit from node n is defined as a function f_n of the alias relation In_n holding on entry to node n by Equation 8.9. Let the SEG node n correspond to the assignment $p_i = q_j$, where p_i is a pointer expression with i levels of indirection from variable p, and q_j is a pointer expression with j levels of indirection from variable q.[5] Out_n is defined as

$$Out_n = f_n(In_n) \overset{def}{=} In_n - Must(EA(p,i)) \ \cup (\bigcup_{a \in EA(p,i), b \in EA(q,j+1)} \{\langle *a, b\rangle\}), \tag{8.9}$$

where $EA(p,i)$ stands for $ExplicitAliases(p,i)$. The functions EA and $Must$ are defined below by Equations 8.10 and 8.11 respectively.

$$EA(p,i) = \begin{cases} \{p\} & \text{if } i = 0 \\ \{r : \langle *s, r\rangle \in In_n | s \in EA(p, i-1)\} & \text{otherwise} \end{cases} \tag{8.10}$$

$$Must(S) = \begin{cases} In_n & \text{if } S = \{\} \\ \{\langle *p, q\rangle \in In_n | p \in S\} & \text{if } |S| = 1 \text{ and } S = \{p\} \text{ for some } p \\ \{\} & \text{if } |S| > 1 \end{cases} \tag{8.11}$$

Notice that the alias set In_n is implicitly used in the computation of both $EA(p,i)$ and $Must(S)$ (for any set of named-objects S). It would in fact be better to use $EA_{In_n}(p,i)$ and $Must_{In_n}(S)$ instead of $EA(p,i)$ and $Must(S)$ respectively. However, we will continue to use $EA(p,i)$ and $Must(S)$ whenever In_n is clear from the context. In terms of the alias graph representation, we may think of In_n as a multi-valued map, where $In_n\{p\}$ (i.e. the image of p under map In_n) represents the set of objects pointed to by object p. Then, it is easily verified that $EA_{In_n}(p,i)$ is simply equal to $In_n^i[\{p\}]$ (recall that $f[s] = \cup_{x \in s} f\{x\}$ and $f^i[s] = f^{i-1}[f[s]]$).

Equation 8.9 may be intuitively understood as follows. In a typical dataflow analysis framework ([3]), Out_n is defined by an equation of the form

$$Out_n = (In_n - Kill_n) \cup Gen_n,$$

[5] $\&q$ is treated as q_{-1}.

where $Kill_n$ and Gen_n denote the dataflow information killed and generated at node n respectively. Let us look at the Gen set for the assignment $p_i = q_j$ given by the expression

$$\bigcup_{a \in EA(p,i), b \in EA(q,j+1)} \{\langle *a, b \rangle\}. \tag{8.12}$$

The objects in $EA(p,i)$[6] and $EA(q,j)$ are respectively the possible left hand sides and the possible right hand sides of the assignment. The effect of the assignment $p_i = q_j$ is to assign the value of some member of $EA(q,j)$ to some member of $EA(p,i)$. Thus, the corresponding effect of the assignment on the may-alias graph must be to add edges from every object in $EA(p,i)$ to every object that can be pointed to by a member of $EA(q,j)$, *i.e.* to every object in $EA(q,j+1)$. In Figure 8.4, the situation is described pictorially, and the dashed edges correspond to the Gen set of the assignment $p_i = q_j$.

The $Kill$ set is given by the expression $Must(EA(p,i))$ which corresponds to those alias pairs that must necessarily be killed by the assignment. If the set $EA(p,i)$ is a singleton set, say $\{a\}$, then a must necessarily get modified by the assignment, and hence all alias pairs of the form $\langle *a, x \rangle$ must be killed. If, however, $EA(p,i)$ is a set with more than one object, say $\{a, b, \dots\}$, then at most one of these objects will get modified by the assignment, and the others will remain unchanged. Since it is not possible to know which object gets modified, therefore, it is not possible to determine which alias-pairs get killed. Hence, $Must(S)$ is conservatively defined to be the empty set $\{\}$, whenever $|S| > 1$. An interesting problem arises when $EA(p,i)$ itself evaluates to the empty set. In this case, we define the $Kill$ set to be the input alias set In_n, so that Out_n evaluates to the empty set. This stops the flow of alias information through node n, until some more alias information reaches node n, such that $EA(p,i)$ becomes non-empty. If, on termination of the iterative analysis, there are one or more SEG nodes for which the corresponding sets $EA(p,i)$ are empty, then it can be proven that if an execution path ever reaches any of these nodes, then a null pointer dereference error will occur.

In the next section, we describe the original algorithm by Hind et al. [51] and a small improvement to the algorithm by using the worklist based strategy. The termination of each of these algorithms relies on the fact that function f_n (given by Equation 8.9) is monotonic with respect to the parameter In_n. It is easily verified that function $Must$, as defined by Equation 8.11, is anti-monotonic (decreasing), and function $ExplicitAliases$, as defined by Equation 8.10, is monotonic with respect to In_n. Then, it is easily verified from Equation 8.9 that function f_n is monotonic with respect to In_n.

8.6 Computing May-Alias Information

This section describes two algorithms to compute the may-alias information at each SEG node n. First we describe a very simple iterative algorithm used by [51] that computes a safe approximation to the may-alias relation at entry to every SEG node n, which runs in $O(N^6)$ time where N is the size of the SEG. Next we explain how the simple iterative algorithm can be improved by using a worklist based strategy to run in $O(N^5)$ time. Finally, in Section 8.7 we shall see how our language-theoretic approach leads to the discovery of a much more efficient algorithm based on the ideas of *dominated convergence* and *finite differencing* ([68]), that runs in worst-case $O(N^3)$ time.

8.6.1 Simple Iterative Algorithm

From Section 8.5, it is clear that the may-alias relations to be computed at the entry and exit to each SEG node n, *i.e.* In_n, and Out_n must satisfy Equations 8.8 and 8.9. In fact, Equations 8.8 and 8.9 may be combined into the single equation

$$In_n = \bigcup_{p \in Pred(n)} f_p(In_p), \tag{8.13}$$

[6]Recall that one way to think of $EA(p,i)$ is as the set of nodes reachable by a path of length i from node p.

Input: $In(0)$
Output: $In(i)$ for all $i = 0, \ldots, N$

```
Init :  $\forall i = 1, \ldots N$  $In(i) = \{\}$
        $\forall i = 0, \ldots N$  $Out(i) = f_i(In(i))$
Loop :  repeat
        $\forall i = 1, \ldots, N$ loop
            $In(i) = \cup_{p \in Pred(i)} Out(p)$
            $Out(i) = f_i(In(i))$
        end$\forall$
        until all the $Out(i)$'s converge.
```

Figure 8.5: Simple iterative algorithm for computing may-aliases

for each SEG node n, where f_p is the transfer function for node p. In other words, the computed may-alias information must be a fixed-point of the following system of equations:

$$\begin{array}{rcl} In_1 & = & \bigcup_{p \in Pred(1)} f_p(In_p) \\ In_2 & = & \bigcup_{p \in Pred(2)} f_p(In_p) \\ & \cdots & \\ In_N & = & \bigcup_{p \in Pred(N)} f_p(In_p). \end{array} \tag{8.14}$$

It is not difficult to see that the may-alias information that we seek to compute, is in fact, the least fixed point of the system of equations 8.14. All other fixed points of the system of equations 8.14 are also safe approximations of the may-alias relations at every node n, although they are not as accurate (precise) as the least fixed point.

Let the nodes of the SEG be numbered from $0, \ldots, N$, where node 0 denotes the entry node N_{Entry} of the SEG. Then, if each of the transfer functions f_p is monotonic, the least fixed point of the system of equations 8.14 can be computed by Kildall's iterative algorithm ([56]) shown in Figure 8.5.

Before we are ready to calculate the run-time complexity of the algorithm in Figure 8.5, we need to understand a few details about the low level implementation. Since an alias relation is implemented as a set of alias pairs, the efficient computation of Out_n from In_n using Equation 8.9 requires that the primitive operations such as set membership test and insertion into a set, be performed efficiently. The implementation of Hind et al. ([51]) relies on hashing to perform the set-based operations efficiently. Thus, the complexity of their algorithm is average-case $O(N^6)$ time, under the assumption that hashing takes $O(1)$ time on the average. However, it is possible to use a more sophisticated data-structure for implementing sets that allows primitive operations such as set membership test and insertion into a set to be implemented in worst-case $O(1)$ time without the use of hashing. We do not present the details of these data structures in this paper but refer the interested reader to [65, 43] which describe data-structures that can be used to perform all set membership tests and set insertions in our algorithm in worst-case $O(1)$ time.

Let us make the simplifying assumption that the SEG has been pre-processed so that the maximum number of predecessors of any node is no more than two. This can easily be done by adding a series of dummy join nodes to the SEG. The resulting number of nodes in the new SEG is still linear in the number of edges in the old SEG. We will let N denote the number of vertices in the processed SEG. Note, however, that both the number of vertices and the number of edges in the original SEG are linear in the size of the procedure. Let A_{max} denote the maximum number of aliases holding at any program point, and let L be a bound on i and j, *i.e.* the maximum number of dereferences to any of the named objects appearing in the assignment $p_i = q_j$.

Lemma 8.6.1 *$EA(p,i)$ can be computed in $O(i \times A_{max})$ time.*

Proof: We compute $EA(p,i)$ by successively computing $EA(p,0), EA(p,1), \ldots$. The computation of $EA(p,k+$ $1)$ from $EA(p,k)$ can be done by just computing the neighbors of all nodes in $EA(p,k)$, and can take at most A_{max} time. Thus, the successive computations of $EA(p,0), \ldots, EA(p,i)$ take $O(i \times A_{max})$ time. □

Lemma 8.6.2 *The function $f_n(In_n)$ given by Equation 8.9 can be computed in $O(L \times A_{max})$ time.*

Proof: The computation of $f_n(In_n)$ is done by first computing $EA(p,i)$ and $EA(q,j+1)$. Since L is an upper bound on i and j, we see from Lemma 8.6.1 that $EA(p,i)$ and $EA(q,j+1)$ can be computed in $O(L \times A_{max})$ time. Next, it is easy to see that the computation of $Must(EA(p,i))$ takes at most $O(A_{max})$ time. Finally, since the size of Expression (8.12) is bounded by A_{max}, we see that the entire computation of $f_n(In_n)$ takes at most $O(L \times A_{max})$ time. $\square$

Corollary 8.6.3 *Each iteration of the algorithm in Figure 8.5 takes $O(N \times L \times A_{max})$ time.*

Lemma 8.6.4 *The number of iterations in the algorithm in Figure 8.5 is bounded by $(N \times A_{max}) + 1$.*

Proof: Each iteration of the algorithm, except for the last one, results in the addition of at least one new edge to the *Out* set of some node i. The total number of such edges for each node is bounded by A_{max}. Thus, the total number of iterations is bounded by $(N \times A_{max}) + 1$. $\square$

Corollary 8.6.5 *The time complexity of the algorithm in Figure 8.5 is $O(L \times A_{max}^2 \times N^2)$ time.*

It is quite reasonable to assume that L is actually a small constant. Also, in the worst case, A_{max} may be as large as N^2. Thus, a coarser estimate of the time complexity of the simple algorithm is $O(N^6)$. Next, we show how a simple worklist based strategy can improve the worst-case time complexity of the algorithm to $O(N^5)$. Finally, we present a much more efficient algorithm for computing the same may-alias information in worst-case $O(N^3)$ time.

8.6.2 A Worklist Based Strategy

One of the main sources of inefficiency in the simple iterative algorithm is the repetitive computation of $f_k(In(k))$ for all SEG nodes k in each iteration, including even all those nodes for which there is no change in the value of their *In* sets in the previous iteration. A simple optimization that eliminates needless repeated computations of $Out(k) = f_k(In(k))$, is the use of a worklist based strategy. A worklist of nodes, for which the *In* sets changed in the previous iteration is maintained. In each iteration, the computation of *Out* is performed only for the nodes in the worklist, and another worklist comprising of those nodes for which the *In* sets change in this iteration, is created for use in the next iteration. A careful implementation of the worklist strategy can improve the worst-case time complexity of the may-alias computation to $O(L \times A_{max}^2 \times N)$ time. This is because, for each SEG node k, the re-computation of $Out(k)$ is done only whenever at least one new alias pair is added to $In(k)$. Using the fact that the size of $In(k)$ is bounded by A_{max}, and that the computation of $Out(k)$ takes $O(L \times A_{max})$ time, we get the worst-case time complexity $O(L \times A_{max}^2 \times N)$ for the worklist strategy based algorithm. Again, assuming that L is a small constant, and that $A_{max} = O(N^2)$, we see that the complexity of the worklist based algorithm is $O(N^5)$ time.

8.7 Going Beyond the Worklist Strategy

For every SEG node i, let $i1, i2, \ldots, ik_i$ denote the predecessors of node i in the SEG. Let expression $F_i(In_{i1}, In_{i2}, \ldots, In_{ik_i})$ be defined by

$$F_i(In_{i1}, In_{i2}, \ldots, In_{ik_i}) = f_{i1}(In_{i1}) \cup f_{i2}(In_{i2}) \cup \ldots \cup f_{ik_i}(In_{ik_i}) \tag{8.15}$$

Then, the may-alias relation that we wish to compute is simply the least fixed point of the following system of equations.

$$\begin{array}{lcl} In_1 & = & F_1(In_{11}, In_{12}, \ldots, In_{1k_1}) \\ In_2 & = & F_2(In_{21}, In_{22}, \ldots, In_{2k_2}) \\ & \vdots & \\ In_N & = & F_N(In_{N1}, In_{N2}, \ldots, In_{Nk_N}), \end{array} \tag{8.16}$$

According to Theorem 8.3.1, the least fixed point of the System of Equations 8.16 can be computed in linear time if the following conditions hold.

1. Each expression $F_i(In_{i1}, \ldots, In_{ik_i})$ is monotonic in its arguments.
2. Each expression $F_i(In_{i1}, \ldots, In_{ik_i})$ is of linear cost.
3. Each expression $F_i(In_{i1}, \ldots, In_{ik_i}) - In_i$ is weakly continuous with respect to the set of modifications $\{In_j \; with := z_i$ where $In_j \in \{In_{i1}, \ldots, In_{ik_i}\}\ \}$.
4. The number of occurrences of any variable In_i in the multi-set

$$\{In_{11}, In_{12}, \ldots, In_{1k_1}, In_{21}, In_{22}, \ldots, In_{2k_2},\ \ldots,\ In_{N1}, In_{N2}, \ldots, In_{Nk_N}\}$$

   is bounded by a constant.

We already know that the first condition is true. If we assume that every node has at most two predecessors and two successors, then the last condition is also satisfied as a variable In_i may appear in the arguments of at most two expressions. Thus, inorder to get a linear-time algorithm for may-alias analysis, we just need to prove conditions 2 and 3.

First, let us try to prove that each expression $F_i(In_{i1}, \ldots, In_{ik_i})$ is of linear cost. Since each node i can have at most two predecessors, and the operator $\cup$ absorbs both its parameters, it suffices to prove that expression $f_n(In_n)$ is of linear cost for an arbitrary node n.

If we think of In_n in terms of the alias graph representation, it may be implemented as a multi-valued map where $In_n\{p\}$ denotes the set of objects pointed to by p. Under this representation, expression $EA(p, i)$ is equivalent to expression $In_n^i[\{p\}]$. Similarly, the expression

$$\cup_{a \in EA(p,i), b \in EA(q,j+1)} \{\langle *a, b\rangle\}$$

can be expressed as the cross product $In_n^i[\{p\}] \times In_n^{j+1}[\{q\}]$. Expanding out the definition of $Must$ given by Equation 8.11, expression $f_n(In_n)$ (as defined by Equation 8.9) can be expressed as

```
f_n(In_n) = let
                $S_i = In_n^i[\{p\}]$
            in
            let
                $T_{j+1} = In_n^{j+1}[\{q\}]$
            in
                if $S_i = \{\}$ then                                  (8.17)
                    $\{\}$
                else
                    if $\#S_i = 1$ then
                        $(In_n)|_{domain(In_n) - S_i} \ \cup\ (S_i \times T_{j+1})$
                    else
                        $In_n \ \cup\ (S_i \times T_{j+1})$
```

The outline of the proof that Expression $f_n(In_n)$ is of linear cost is given below.

- Expression $In_n[s]$ is a simple, input-bound, linear-cost expression.
- Using the composition rule, it follows that Expression $In_n^i[s]$ is an input-bound linear-cost expression for any constant i.
- Thus, expressions S_i and T_{j+1} (as defined in Equation 8.17) are input-bound and of linear cost.
- Since S_i is input bound, the boolean expressions $S_i = \{\}$ and $\#S_i = 1$ are also of linear cost.

- Expression $domain(In_n) - S_i$ is input-bound and of linear cost. Therefore, expression

$$(In_n)|_{domain(In_n)-S_i}$$

is also input-bound and of linear cost.

- Expression $S_i \times T_{j+1}$ is of linear cost since S_i and T_{j+1} are both input-bound and of linear cost.

- Since operator $\cup$ absorbs both of its parameters, it follows that expressions

$$(In_n)|_{domain(In_n)-S_i} \ \cup \ (S_i \times T_{j+1})$$

and

$$In_n \ \cup \ (S_i \times T_{j+1})$$

are both of linear cost. Thus, it follows that Expression $f_n(In_n)$ given by Equation 8.17 is of linear cost.

Thus, Condition 2, i.e. that each expression $F_i(In_{i1}, \ldots, In_{ik_i})$ should be of linear cost, is also true. Now, let us take a look at Condition 3 which requires that each expression $F_i(In_{i1}, \ldots, In_{ik_i}) - In_i$ be weakly continuous with respect to the set of modifications $\{In_j \ with := z_i$ where $In_j \in \{In_{i1}, \ldots, In_{ik_i}\} \ \}$. Once again, it suffices to prove that each expression $f_n(In_n)$ is weakly continuous with respect to $In_n \ with := z$, since we know that operator $\cup$ absorbs both of its arguments, and that operators $\cup$ and $-$ (i.e the set difference operator) are strongly continuous with respect to addition of elements to both parameters.

The outline of the proof of weak continuity of $f_n(In_n)$ with respect to the modification $In_n \ with := z$ is given below.

- Expression $In_n[s]$ is a simple, input-bound, linear-cost expression which is weakly continuous with respect to $\{In_n \ with := z, s \ with := z'\}$.

- Using the composition rule, it follows that Expression $In_n^i[s]$ (for any constant i) is an input-bound linear-cost expression which is weakly continuous with respect to $In_n \ with := z$.

- Thus, expressions S_i and T_{j+1} (as defined in Equation 8.17) are input-bound, of linear cost and weakly continuous with respect to $In_n \ with := z$.

- Since S_i is input bound, the boolean expressions $S_i = \{\}$ and $\#S_i = 1$ are input-bound and weakly continuous with respect to $In_n \ with := z$.

- Expression $domain(In_n) - S_i$ and expression $(In_n)|_{domain(In_n)-S_i}$ can also be seen to be input-bound and weakly continuous with respect to $In_n \ with := z$.

- Next, it is easy to see that expression $S_i \times T_{j+1}$ is also weakly continuous with respect to $In_n \ with := z$.

- Since operator $\cup$ absorbs both of its parameters, it follows that expressions

$$(In_n)|_{domain(In_n)-S_i} \ \cup \ (S_i \times T_{j+1})$$

and

$$In_n \ \cup \ (S_i \times T_{j+1})$$

are both weakly continuous with respect to $In_n \ with := z$. Thus, it follows that Expression $f_n(In_n)$ given by Equation 8.17 is weakly continuous with respect to $In_n \ with := z$.

This concludes the proof that the System of Equations 8.16 can be solved in linear time. Since, the worst-case size of the output is bounded by $O(N^3)$, we get a new $O(N^3)$ time algorithm for computing the may-alias relation.

Chapter 9

Conclusion

9.1 Future Work

The two main goals of this thesis were 1) a formal development of the data structure selection transformation to be used along with the finite differencing and dominated convergence transformations as a part of a transformational program design methodology, and 2) demonstration of the usefulness of this methodology for a better understanding of algorithms. This research was pioneered in the late 1970's by Bob Paige with the goal of developing a transformational program development methodology that could significantly improve the productivity of designing and maintaining highly reliable algorithmic software. The current thesis has been concerned with the development of the theoretical basis for such a methodology. We have demonstrated how this methodology can be applied by an algorithmician using a pencil and paper, and can help gain a better understanding of algorithms, or even help discover new algorithms. What remains, however, is the development of a practical program development system that can use the ideas presented in this thesis to automaticaly or semi-automatically transform abstract program specifications into high performance implementations.

A prototype of such a program transformation system was developed and used by Cai and Paige to conduct experiments for testing the viability of the transformational approach for high performance algorithm implementation. The results of these experiments were presented in [16]. A simple but conservative model of productivity was used, and a five-fold improvement in productivity of high performance algorithm implementation in C was demonstrated. The experiments were conducted only on small scale examples, but the results were encouraging. The authors hypothesized that their results would scale up even for large scale examples. In this section, we list some possibilities for future work that could help in the development of such a system.

1. For most of this thesis, the data structures have been developed for a pointer machine model of computation. In some of the later chapters, we discussed how a RAM implementation could improve the run-time performance by significant constant factors. Unfortunately, the RAM implementations may have a greater space complexity. For example, the space required to implement both weakly and strongly based sets on a pointer machine is linear in the sizes of these sets. However, an alternate RAM implementation of a strongly based set (say, of type $strong\_set(b)$), which can improve the time complexity of associative access operations by a significant constant factor, requires space proportional to the total number of elements of base type b. These pragmatic issues are discussed in greater detail in [65]. This space-time trade-off requires some further investigation for a more effective use of based data structures.

2. The algorithms considered in this thesis are batch-input algorithms, i.e. algorithms that read in their entire input at the beginning of the program. The linear-time read method of [73], which is used to read-in the input and set up the appropriate data structures, requires that the entire input be available at the beginning of the program. Therefore, our typed set-theoretic languages are limited to batch-input programs.

Moreover, our definition of Low SETL does not allow creation of values of a base type at run time. This further restricts the expressiveness of these languages. The expressiveness of these languages can be easily enhanced by allowing the creation of *new* values of base types, i.e. under the restriction that each created value of a base type be distinct from all existing values of the same base type. Interestingly, if Low SETL is extended in this manner, the complex reading algorithm of Paige and Yang ([73]), which is based on an algorithmic tool called multiset discrimination ([15]), can itself be expressed as a well-typed Low SETL program. We conjecture that this generalization is sufficient to make Low SETL *algorithmically expressive*[1] relative to a pointer machine. However, this generalization may not be sufficient to make Low SETL algorithmically expressive relative to more permissive models of computation such as a RAM. Relaxing the batch-input requirement would be one way of adding to the expressiveness of the language. This would require the development of a more general read method that could be called multiple times during the execution of a program.

3. The time complexity analysis of the data structures in this thesis does not take constant factors into account although such constant factors are important in practice. Constant factor differences in space utilization can especially make a significant difference in practice. Space usage can be reduced by sharing the space for two or more strongly based sets if their life-spans during the execution of the program do not intercross with each other. The above optimization would be especially effective for Low SETL programs that are obtained by translation from High SETL because the translation of High SETL to Low SETL requires the creation of many strongly based sets that have short life-spans.

4. In this thesis, we have defined a type system for each of Low SETL, High SETL, and SQ^+. The Low SETL type system is undecidable because of the presence of certain side-conditions in some type rules that require proofs of properties that are in general undecidable (e.g. proving that an expression e evaluates to a value that is not contained in set s in any possible execution of the program). However, the High SETL and SQ^+ type systems do not suffer from this shortcoming, and given a type environment TE, and a set of subtype constraints C, it is relatively straightforward to check if a High SETL or SQ^+ program P is well-typed. In addition, we have defined translations which are always guaranteed to transform well-typed High SETL and SQ^+ programs into well-typed Low SETL programs. Thus, a Low SETL program that is obtained as a result of translation from a well-typed High SETL or SQ^+ program does not need to type-checked again. One of the issues that we have not addressed in this thesis is that of type inference. We have not given algorithms that can automatically determine whether a High SETL or SQ^+ program is well-typed. For the examples presented in this thesis, the well-typedness of programs has been proven by hand. Development of type inference algorithms would significantly simplify the task of data structure selection.

5. In Chapter 6 we defined a worst-case linear-time subset L_{IO} of SQ^+. Our definition extends the linear-time language originally proposed by Cai and Paige in [13]. Their complexity arguments were based on a set-theoretic complexity measure i.e. on the assumption that all primitive set-theoretic operations could be done in $O(1)$ time, although they did not show how this could be done. Cai and Paige also did some preliminary work [14] on the development of a hierarchical family of functional problem specification languages $L_1 : i = 1, 2, \ldots,$ where each language L_i was restricted to express only those queries that could be compiled into implementations with worst-case run-time performance bounded by an i^{th} degree polynomial in the space required to store its input and output. It would be interesting to apply our work on data structure selection to these languages.

6. The biggest unresolved problem with the definition of High SETL and SQ^+ is the incorporation of finite differencing directly into the semantics of the languages. The cost of computing expressions in High SETL and SQ^+ may be determined in two ways. One way is to determine the cost of evaluating the expression from scratch. The other way is to use finite differencing. The cost of these differential calculations are determined by associating precise amortized complexities with an eager

[1]Recall that a programming language PL is algorithmically expressive relative to an implementation language IL, if for an PL computable function f, there exists a PL implementation of f that matches the performance of the best-known IL implementation of f.

strategy for maintaining equality invariants $v = E(x_1, \ldots, x_n)$ with respect to worst-case sequences of modifications to variables $x_1, \ldots, x_n$. We defined the second complexity measure in the context of L_{IO}, the linear-time subset of SQ$^+$. However, it would be interesting to see if such a complexity measure could be defined for High SETL and SQ$^+$ in totality.

9.2 Summary

The area of algorithms has had a profound impact on every area of computer science. The area of programming languages and their compilers too has benefitted greatly from advances in algorithms. We have tried to show that the area of programming languages can in return benefit the area of algorithms be helping in the development of a systematic approach for algorithm design and exposition.

This thesis defines the syntax and semantics for three increasingly more abstract, set-theoretic algorithm specification languages Low SETL, High SETL, and SQ$^+$. The dynamically typed versions of these languages suffer from the same weaknesses as the languages SETL and SETL2, i.e. that these languages lack computational transparency because of the hash-based implementation of sets and maps. We rectify this weakness by imposing a novel static type system (in which types are associated with data structures) on these languages, and prove that the well-typedness of a program in these languages guarantees that the program can be transformed into a hash-free implementation that is amenable to algorithmic analysis.

We have also presented three algorithms that serve as examples illustrating the usefulness of our languages for algorithm explanation and discovery.

1. Our first example is an abstruse algorithm for database query optimization originally proposed by Willard [111]. Willard's algorithm uses hashing extensively to show that a large subset of the relational calculus (RCS) can be transformed into linear-time implementations. Willard's time complexity analysis relies on the assumption that each hash operation takes $O(1)$ time on the average. By expressing RCS queries as High SETL specifications, we show how semantics-preserving program transformations can be used to transform such abstract queries into highly efficient implementations. In the process, we not only make Willard's algorithm easier to understand, but also improve its time complexity from expected to worst-case linear time.

2. Our second example deals with the problem of computing the least fixed point of a system of equations on a transition system. This problem has been widely studied because of its applications to model checking. The first linear-time algorithm was presented by Arnold and Crubille [5]. Later the algorithm was simplified and improved independently by Cleaveland and Steffen [25], Anderson [4], and Vergauwen and Lewi [109]. Our transformational methodology leads to the derivation of an algorithm that matches the best known algorithm for the problem. More importantly, we feel that our derivation sheds some light on the key insights that led to the discovery of the original linear-time algorithm.

3. Our third and final example deals with the problem of computing an intra-procedural may-alias analysis. We show that the use of our algorithm design methodology leads to the discovery of a new $O(N^3)$ time algorithm (where N is the size of the program being analyzed) which is a vast improvement over the best previously known algorithm ([51]) that runs in time $O(N^5)$.

The popular view held by the algorithms community is that the most important steps of an algorithm are discovered through sheer inspiration. They believe that it is not possible to develop a small set of rules that would apply to a wide class of problems and allow a systematic approach towards algorithm design. This view has had a detrimental effect on the way that algorithms are explained, because the lack of a suitable language or notation for expressing the intuition behind an algorithm makes the task of algorithm exposition very difficult. Quite often, the algorithms are presented in their final form, and the intuition behind the algorithm remains a mystery to the reader.

Contrary to this view, we believe that just as abstraction can significantly simplify the task of algorithm implementation, in the same way, the use of abstraction can lead to a better understanding of algorithms. In our experience, the use of abstraction has not only enabled a deeper understanding of complex algorithms, but in some cases, has also led to the discovery of improved algorithms. Our set-theoretic languages and the transformational program design methodology based on finite differencing, dominated convergence, and data structure selection, are effective mathematical tools that are applicable to a large and important class of problems. We believe that this approach brings us a step closer to the goal of having a simple, systematic, widely-applicable methodology for algorithm design.

Bibliography

[1] A. Aho, J. Hopcroft, and J. Ullman. *Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[2] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, Massachusetts, 1983.

[3] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1988.

[4] Henrik Reif Anderson. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, April 1994.

[5] A. Arnold and P. Crubille. A linear algorithm to solve fixed point equations on transition systems. In *Information Processing Letters*, volume 29, pages 57–66, 1988.

[6] J. Banning. An efficient way to find the side-effects of procedure calls and the aliases of variables. In *6th ACM Symposium on the Principles of Programming Languages*, pages 29–41, 1979.

[7] Amir M. Ben-Amram. What is a "pointer machine"? *SIGACT News*, 26(2):88–95, June 1995.

[8] G. Birkhoff. *Lattice Theory.* American Mathematical Society, Providence, 1966.

[9] B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Science of Computer Programming*, 24(3):189–220, 1995.

[10] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[11] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, Jan 1977.

[12] J. Cai and R. Paige. Binding performance at language design time. In *Proc. Fourteenth ACM Symp. on Principles of Programming Languages*, pages 85 – 97, Jan. 1987.

[13] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11:197–261, 1988/89.

[14] J. Cai and R. Paige. Languages polynomial in the input plus output. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology*, Workshops in Computing, pages 287–302. Springer-Verlag, 1992. Conference Record of the Second AMAST.

[15] J. Cai and R. Paige. Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science*, 145(1-2):189–228, July 1995. URL http://cs.nyu.edu/cs/faculty/paige/papers/hash.ps.

[16] J. Cai and R. Paige. Towards increased productivity of algorithm implementation. In *Proc. ACM SIGSOFT*, pages 71–78, Dec. 1993.

[17] Jiazhen Cai. Fixed point computation and transformational programming. Technical Report DCS-TR-217, The State University of New Jersey, Rutgers, 1987. PhD. Thesis.

[18] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 296–310, 1990.

[19] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Programming Language Design and Implementation*, pages 296–310, 1990.

[20] J. D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In *20th SIGACT-SIGPLAN ACM Symposium on the Principles of Programming Languages*, pages 232–245, 1993.

[21] Jong Deok Choi, Michael Burke, and Paul Carini. Automatic construction of sparse data flow evaluation graphs. In *18th annual ACM symposium on Principles of Programming Languages*, pages 55–66, 1991.

[22] R. Cleavaland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal-calculus. Technical report, Computer Science Dept., North Carolina University, 1992.

[23] R. Cleaveland and M. Klein. Faster model checking for the modal μ-calulus. In *Computer-Aided Verification (CAV '92)*, volume 663 of *LNCS*, pages 410–422, 1992.

[24] Rance Cleaveland and Bernhard Steffen. Computing behavioral relations, logically. In Leach Albert, B. Monien, and M. Rodriguez Artalejo, editors, *Automata, Languages and Programming (ICAPL'91)*, volume 510 of *LNCS*, 1991.

[25] Rance Cleaveland and Bernhard Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2:121–147, 1993.

[26] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, 1986.

[27] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, Jun. 1970.

[28] K. D. Cooper and K. Kennedy. Inter-procedural side-effect analysis in linear time. In *SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 487–506, 1988. SIGPLAN Notices, 23(7).

[29] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific J. Math.*, 82(1):43–57, 1979.

[30] H. Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23:83–92, 1969.

[31] Martin D. Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, Complexity, and Languages.* Academic Press, second edition, 1994.

[32] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *International Conference on Computer Languages*, pages 2–13. IEEE, 1992.

[33] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 230–241, 1994.

[34] Robert Dewar, Arthur Grand, Ssu cheng Liu, and Jacob Schwartz. Programming by refinement, as exemplified by the SETL representation sublanguage. *TOPLAS*, 1(1):27–49, July 1979.

[35] A. Dicky. An algebraic and algorithmic method of analyzing transition systems. *Theoretical Computer Science*, 46:285–303, 1986.

[36] Edsger W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*, chapter Why Naive Program Transformation Systems are Unlikely to Work ? Springer-Verlag, 1982.

[37] E.W Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[38] W. Dowling and J. Gallier. Linear time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[39] J. Earley. High level iterators and a method for automatically designing data structure representation. *J. of Computer Languages*, 1(4):321–342, 1976.

[40] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Proc. 1st IEEE Symposium on Logic in Computer Science*, pages 267–278, 1986.

[41] S. Freudenberger, J. Schwartz, and M. Sharir. Experience with the SETL optimizer. *ACM TOPLAS*, 5(1):26–45, 1983.

[42] Deepak Goyal. An improved intra-procedural may-alias analysis algorithm. Technical Report 777, New York University, 1999.

[43] Deepak Goyal and Robert Paige. The formal reconstruction and speedup of the linear time fragment of Willard's relational calculus subset. In Bird and Meertens, editors, *Algorithmic Languages and Calculi*, pages 382–414. Chapman and Hall, 1997.

[44] Deepak Goyal and Robert Paige. A new solution to the hidden copy problem. In Giorgio Levi, editor, *Proc. 5th International Static Analysis Symposium*, number 1503 in LNCS, pages 327–348. Springer, September 1998.

[45] G. Gratzer. *General Lattice Theory*. Birkhauser, 1978.

[46] Carl A. Gunter. *Semantics of Programming Langauges Structures and Techniques*. The MIT Press, 1992.

[47] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 97–105, 1998. SIGPLAN Notices, 33(5).

[48] M. S. Hecht. *Flow Analysis of Computer Programs,*. Elsevier, Amsterdam, 1977.

[49] M Hennessy. *The Semantics of Programming Languages: An elementary introduction using structural operational semantics*. Wiley, 1991.

[50] M. Hind and A. Pioli. Assesing the effects of flow-sensitivity on pointer alias analyses. In *5th International Static Analysis Symposium*, number 1503 in LNCS, 1998.

[51] Michael Hind, Michael Burke, Paul Carini, and Jong Deok Choi. Interprocedural pointer alias analysis. Accepted for publication in TOPLAS in 1999. An earlier version appeared as IBM Technical report #21055, December 1997.

[52] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, Dec. 1969.

[53] Susan Horwitz, P. Pfeiffer, and Tom Reps. Dependence analysis for pointer variables. In *Programming Language Design and Implementation*, pages 28–40, 1989.

[54] Harry B. Hunt-III, Thomas G. Szymanski, and Jefferey D. Ullman. Operations on sparse relations. *CACM*, 20(3):127–132, March 1977.

[55] J.P. Keller and R. Paige. Program derivation with verified transformations – a case study. *CPAM*, 48(9-10), 1995.

[56] Gary A Kildall. A unified approach to global program optimization. In *ACM Symp. on Principles of Prog. Lang.*, pages 194–206, 1973.

[57] D. E. Knuth. *The Art of Computer Programming, Vol 1: Fundamental Algorithms.* Addison-Wesley, 1973.

[58] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[59] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[60] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Programming Language Design and Implementation*, pages 21–34, 1988.

[61] T. Marlowe, W. Landi, B. Ryder, J. Choi, and P. Carini. Pointer induced aliasing: A clarification. *SIGPLAN Notices*, 28(9):67–70, September 1993.

[62] Kurt Melhorn. *Data Structures and Algorithms 1:Sorting and Searching.* Springer-Verlag, 1984.

[63] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT press, 1990.

[64] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications, A formal introduction.* Wiley, 1992.

[65] R. Paige. Real-time simulation of a set machine on a RAM. In N. Janicki and W. Koczkodaj, editors, *Computing and Information*, volume II, pages 69–73. Canadian Scholars' Press, Toronto, May 1989.

[66] R. Paige. Efficient translation of external input in a dynamically typed language. In B. Pehrson and I. Simon, editors, *Technology and Foundations - Information Processing 94*, volume 1 of *IFIP Transactions A-51*, pages 603–608. North-Holland, Amsterdam, Sept. 1994. Conference Record of IFIP Congress 94.

[67] R. Paige and F. Henglein. Mechanical translation of set theoretic problem specifications into efficient RAM code - a case study. *Journal of Symbolic Computation*, 4(2):207–232, Aug. 1987.

[68] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. on Programming Languages and Systems*, 4(3):401–454, 1982.

[69] R. Paige, R. Tarjan, and R. Bonic. A linear time solution to the single function coarsest partition problem. *Theoretical Computer Science*, 40(1):67–84, Sep. 1985.

[70] Robert Paige. *Formal Differentiation: A Program Synthesis Technique.* UMI Research Press, 1981. Revision of Ph.D. thesis, NYU, Jun 1979.

[71] Robert Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In H. Gallaire, J. Minker, and J. M. Nicolas, editors, *Advances in Database Theory*, volume 2, pages 171–209. Plenum Press., New York, 1984.

[72] Robert Paige. Programming with invariants. *IEEE Software*, 3(1):56–69, Jan 1986.

[73] Robert Paige and Zhe Yang. High level reading and data structure compilation. In *Proc. 24th ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang.*, pages 456–469, Paris, France, 15–17 January 1997.

[74] P. Pepper. A simple calculus for program transformation. *Science of Computer Programming*, 9:221–262, 1987.

[75] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, Denmark, 1981.

[76] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(6):1467–1471, November 1994.

[77] J. H. Reif and H. R. Lewis. *Symbolic evaluation and the global value graph.* Harvard University, Aiken Computation Laboratory, 1982.

[78] John H. Reif and Harry R. Lewis. Symbolic evaluation and the global value graph. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 104–118, Los Angeles, California, January 17–19, 1977. ACM SIGACT-SIGPLAN.

[79] John C. Reynolds. *The Craft of Programming.* Prentice-hall International, 1981.

[80] Barry K Rosen. Data flow analysis for procedural languages. *Journal of the ACM*, 26(2):322–344, April 1979.

[81] D. Sands. Total correctness by local improvement in program transformation. In *Proc. 22nd ACM Symp. on Principles of Programming Languages*, pages 221–232, Jan. 1995.

[82] E. Schonberg, J. Schwartz, and M. Sharir. An automatic technique for selection of data representations in SETL programs. *ACM TOPLAS*, 3(2):126–143, Apr. 1981.

[83] A. Schönhage. Storage modification machines. *Siam Journal of Computing*, 9(3):490–508, August 1980.

[84] J. Schwartz. *On Programming: An Interim Report on the SETL Project, Installments I and II.* New York University, New York, 1974.

[85] J. Schwartz. Automatic data structure choice in a language of very high level. *CACM*, 18(12):722–728, Dec. 1975.

[86] J. Schwartz. Optimization of very high level languages, parts I, II. *J. of Computer Languages*, 1(2-3):161–218, 1975.

[87] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, New York, 1986.

[88] Jack Schwartz. More on the copy optimization of setl programs. Setl Newsletter 131, New York University, June 1974.

[89] Jack Schwartz. A coarser, but simpler and considerably more efficient copy optimization technique. Setl Newsletter 176, New York University, August 1976.

[90] Jack Schwartz. "copy on assignment" optimization in setl. Setl Newsletter 164A, New York University, April 1976.

[91] H. Seidl. Fast and simple nested fixed points. *Information Processing Letters*, 59:303–308, 1996.

[92] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *24th SIGACT-SIGPLAN ACM Symposium on the Principles of Programming Languages*, pages 1–14, 1997.

[93] Micha Sharir. A few cautionary notes on the convergence of iterative data-flow analysis algorithms. Setl Newsletter 208, New York University, April 1978.

[94] Micha Sharir. Algorithm derivation by transformations. Technical Report 21, Computer Science Department, NYU, October 1979.

[95] Micha Sharir. Some observations concerning formal differentiation of set-theoretic expressions. Technical Report 16, Computer Science Department, NYU, October 1979.

[96] M. SIntzoff. Calculating properties of programs by valuations on specific models. In *ACM SIGPLAN Notices*, volume 7(1), pages 203–207, 1972.

[97] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A laboratory based approach.* Addison-Wesley, 1995.

[98] D. Smith. Kids - a knowledge-based software development system. In *Proc. Workshop on Automating Software Design, AAAI-88*, pages 129–136, Sept 1988.

[99] K. Snyder. The SETL2 programming language. Technical Report 490, Courant Insititute, New York University, 1990.

[100] B. Steensgaard. Points-to analysis in almost linear time. In *23rd SIGACT-SIGPLAN ACM Symposium on the Principles of Programming Languages*, pages 32–41, 1996.

[101] P. Stocks, B. Ryder, W. Landi, and S. Zhang. Comparing flow and context sensitivity on the modifications-side-effects problem. In *International Symposium on Software Testing and Analysis*, pages 21–31, 1998.

[102] P. Suppes. *Axiomatic Set Theory.* Dover, 1972.

[103] A. Tanenbaum. *Type Determination for Very High Level Languages.* PhD thesis, New York University, Department of Computer Science, Oct 1974. Appears in Courant Computer Science Report 3.

[104] R. Tarjan. Fast algorithms for solving path problems. *Journal of the ACm*, 28(3):594–614, 1981.

[105] Robert Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of computer and System Sciences*, 18:110–127, 1979.

[106] Robert Endre Tarjan. *Data Structures and Network Algorithms*, chapter Foundations, pages 2–3. SIAM, 1983.

[107] A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific J. of Mathematics*, 5:285–309, 1955.

[108] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, Edinburgh University, Department of Computer Science, Edinburgh University, Mayfield Rd., EH9 3JZ Edinburgh, May 1988. Available as Technical Report CST-52-88.

[109] B. Vergauwen and J. Lewi. A linear algorithm for solving fixed-point equations on transition systems. In J.C. Raoult, editor, *CAAP'92*, volume 581 of *LNCS*, 1992.

[110] Dan E. Willard. *Predicate Oriented Database Search Algorithms.* PhD thesis, Harvard, May 1978.

[111] Dan E. Willard. Predicate retrieval theory. Technical Report 83-3, SUNY Albany, 1983. PhD. Thesis.

[112] Dan E. Willard. Quasi-linear algorithms for processing relational data base expressions. In *Proceedings of the* 9^{th} *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 243–257, 1990.

[113] Dan E. Willard. Applications of range query theory to relational data base join and selection operations. *J. Computer and System Sci.*, 52:157–169, 1996.

[114] Glynn Winskell. *The Formal Semantics of Programming Languages.* Foundations of Computing. MIT Press, 1994.

[115] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.

[116] S. Zhang, G. Ryder, and W. Landi. Experiments with combined analysis for pointer aliasing. In *Workshop on program analysis for software tools and engineering*, pages 11–18, 1998.