LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TH-400028

A Language for Specifying Compiler Optimizations for Generic Software

J. J. Willcock

December 20, 2007

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

A LANGUAGE FOR SPECIFYING COMPILER OPTIMIZATIONS FOR GENERIC SOFTWARE

Jeremiah J. Willcock

Submitted to the faculty of the University Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Computer Science

Indiana University

December 2007

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Andrew Lumsdaine, Ph.D.

Arun Chauhan, Ph.D.

David R. Musser, Ph.D.

Daniel J. Quinlan, Ph.D.

Amr Sabry, Ph.D.

September 14, 2007

UCRL-TH-400028

To my parents, James and Bonnie

Acknowledgements

I wish to thank my advisor, Andrew Lumsdaine, for much help throughout my career. He has provided a great deal of support and guidance during my time in graduate school. Without him, I would not have been exposed to either generic programming or programming languages to the level that I have. He has also shown me in detail how to perform research and write high-quality papers, and I am grateful to be able to benefit from his experience in these areas.

I am also grateful to my mentor at Lawrence Livermore National Laboratory, Daniel Quinlan, for aid in creating the thesis and support in both software development and writing the thesis. The ROSE framework, of which he is the main developer, was invaluable in applying the work in this thesis to commonly used programming languages.

I also thank my committee, Arun Chauhan, Andrew Lumsdaine, David Musser, Daniel Quinlan, and Amr Sabry for their discussions and comments in formulating the topic of my thesis, and for their comments on the thesis itself. It has been greatly improved by their influences.

I have benefited greatly from discussions with the other members of the Open Systems Laboratory at Indiana University. I am particularly thankful to Jeremy Siek for his interest in and knowledge of generic programming. Discussions with him and Jaakko Järvi helped me greatly to understand the subtleties of generic programming, and of all of the design and implementation options possible for it. I am also grateful to Jaakko for his collaborations on many papers. Ronald Garcia and the rest of the OSL's current and former members have also provided valuable feedback.

I have also benefited greatly from the knowledge of Todd Veldhuizen. His tutorial on fixpoint-based program analysis was my first introduction to the subject, and piqued my interest in it. His other tutorials have been extremely interesting as well, and have taught me parts of computer science that I would not have been exposed to otherwise.

Several of my professors have excited my interest in programming languages and their design and implementation. Foremost among these is Daniel Friedman, whose books and teaching have inspired me greatly. I am also thankful to Kent Dybvig for his classes on compilers; the AST-based model of programs, and the use of many small passes to compile them, have been influential on my own views on the subject. Amr Sabry's semantics and type theory classes have interested me in the more theoretical aspects of programming languages.

I am grateful to my family for support throughout my entire graduate career. My parents, James and Bonnie Willcock, and my sister Ashley Okalski, have been helpful through my entire career, and I have relied heavily on them; they have always been available for comfort when working on my thesis has been difficult.

I would also like to thank Laura Hopkins and Michael Morrone for their tutorials on writing and presentation skills, and Laura in particular for reviewing several of my papers and presentations.

The Indiana University Computer Science thesis template by Douglas Eck, Kyle Wagner, and Ryan Scherle was important to greatly simplify properly formatting the dissertation.

This work was supported by a US Department of Energy High Performance Computer Science Fellowship. This work was also supported in part by a grant from the Lilly Endowment and by National Science Foundation grants CCF-0541335 and CNS-0221387. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory in part under Contract W-7405-Eng-48 and in part under Contract DE-AC52-07NA27344. The project 07-ERD-057 was funded by the Laboratory Directed Research and Development Program at LLNL. This project was also partially funded by the Air Force Research Laboratory.

Disclaimer. This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any

Abstract

Compiler optimization is important to software performance, and modern processor architectures make optimization even more critical. However, many modern software applications use libraries providing high levels of abstraction. Such libraries often hinder effective optimization — the libraries are difficult to analyze using current compiler technology. For example, high-level libraries often use dynamic memory allocation and indirectly expressed control structures, such as iterator-based loops. Programs using these libraries often cannot achieve an optimal level of performance. On the other hand, software libraries have also been recognized as potentially aiding in program optimization. One proposed implementation of library-based optimization is to allow the library author, or a library user, to define custom analyses and optimizations. Only limited systems have been created to take advantage of this potential, however. One problem in creating a framework for defining new optimizations and analyses is how users are to specify them: implementing them by hand inside a compiler is difficult and prone to errors. Thus, a domain-specific language for library-based compiler optimizations would be beneficial. Many optimization specification languages have appeared in the literature, but they tend to be either limited in power or unnecessarily difficult to use. Therefore, I have designed, implemented, and evaluated the **Pavilion** language for specifying program analyses and optimizations, designed for library authors and users. These analyses and optimizations can be based on the implementation of a particular library, its use in a specific program, or on the properties of a broad range of types, expressed through concepts. The new system is intended to provide a high level of expressiveness, even though the intended users are unlikely to be compiler experts.

Contents

CHAPTER 1

Introduction

Modern software, including scientific software, is becoming much larger and more complex than in the past. These increases in size and complexity cause problems in both efficiently writing, and in maintaining, large software packages [79]. One way in which software grows is to be written using greater levels of code reuse, often by the use of pre-written libraries. As complexity increases, it becomes impractical to rewrite common functionality from scratch for each new program. Libraries almost always define various forms of *abstraction*: constructs (either explicitly or implicitly represented in the library) that allow programmers to use a library without fully understanding the details of its implementation. Abstractions are useful because they allow programmers to think at a higher level: a component (such as a binary tree) can be thought of as a single entity (a sorted sequence of elements), rather than as a particular implementation (for example, an AVL tree). Given an abstract interface to a component, its implementation can also be swapped for another (such as a B-tree) without needing to change the code using the abstraction.

Several forms of abstraction are commonly used. The most basic is the function; functions have been used since the early days of computing to convert a particular section of code into an entity usable via procedure calls. Libraries such as the Basic Linear Algebra Subprograms (BLAS) [95] use functions as their main form of abstraction, with data structures provided by the user but required to satisfy a particular memory layout. BLAS, and libraries built atop it such as LAPACK [5], are commonly used in scientific software. Another common form of abstraction is the object, and related ideas from object-oriented programming. An object merges data values and operations on them, and allows run-time subtyping with dispatching based on which subclass is in use.

The C++ language is commonly used for large applications, and this thesis applies program analyses and optimizations to programs written in that language. The form of abstraction used most often in the C++ Standard Library [73], and which is considered in this thesis, is generic programming. The Standard Library, portions of which are based on the previous Standard Template

Library [135, 139], provides a collection of templated data structures, as well as generic algorithms that operate on both library-defined and user-defined data structures. In fact, the algorithms only require particular properties from their inputs, and work on any data structure that satisfies the required properties; generic programming techniques are used to allow this flexibility. The C++ Standard Library data structures and algorithms are used often in applications. They are also used to build other libraries with their own abstractions, such as the Boost Graph Library [131, 132, 134].

In order to achieve a meaningful level of reuse, abstractions must have well-defined semantics; Marschner et al [103] discuss this issue in the context of reusable hardware components. These semantics are rarely defined formally, but they still exist. Any programmer using the abstraction must understand the semantics in order to know what functionality the component provides, and what restrictions exist on its use. Without known semantics, the only documentation on a component's behavior is its implementation, in which case having a reusable component does not provide nearly as much benefit. Component documentation often gives information about its semantics, and this level of information is usually enough for users of the component. More formal semantics are sometimes required, however, and even when they are not required they provide more exact documentation of the component's behavior and limitations.

1.1. Generic programming

One common form of abstraction in modern software is that of *generic programming* [7, 76, 135, 139]. The core idea of generic programming is to define functions that are parameterized over the types of their inputs, and that have as few restrictions on the inputs as possible, making them more broadly applicable. Generic programming is explained in detail in Section 5.4.1 on page 70; only the basic idea is required to understand this introduction. Generic programming is most commonly used in C++, even though that language does not provide explicit support for it. The terminology and definitions used are primarily from the Standard Template Library [7, 135], one of the first major libraries to use generic programming in C++.

In generic programming, functions are defined to have the fewest requirements possible on their input types. Those requirements are grouped into *concepts*; the grouping is done in such a way that related requirements are in the same concept. The relation of *refinement* is used to include

the requirements from one concept into another, establishing a hierarchy. A concept is simply such a grouping of constraints, applicable to one or more types. The Forward Iterator concept from the STL [135], for example, defines the requirements for an iterator that supports forward movement and element access within a container. The requirements in a concept can take several forms: they can require particular functions or operators to be defined (such as the increment operator in Forward Iterator), they can require certain semantic properties to be true of the functions and operators on the type (such as that incrementing two copies of the same iterator produces equal results), or they can require particular related types to exist and have certain properties (such as the value type of the iterator). The semantic properties, in particular, are interesting for the purposes of this thesis because they provide information about the behavior of particular objects that can be useful for software verification or optimization [57, 126].

Concepts are a good place to specify semantics, as a concept defines common properties of many types and their corresponding functions; thus, a set of semantics, or optimizations, may be specified once to apply to all of these types [127]. An optimization or program analysis could then automatically apply to all types satisfying the requirements of the concept without needing to be rewritten for each particular type in use.

1.2. Compiler use of abstractions

One problem with the use of abstractions written by third parties is that they may be used incorrectly or inefficiently [11, 60, 64, 65, 115]. Many such errors could be found automatically by a compiler-based check; invocations of library abstractions could also be optimized by a compiler, forming an *Active Library* [151]. The user could do error checking manually, but that is tedious and it is easy to make mistakes; also, if the user produced the errors originally, they likely do not fully understand the behavior of the abstractions in use. In the area of optimization, the user cannot do certain transformations at all without breaking the abstraction interface provided by the library: an optimization may require information about both uses of the library and the normally-hidden details of its implementation [124]. One example of this type of optimization is sorting the elements of each bucket of an open hash table to allow binary searches; this transformation relies both on the

hash table being built of buckets which are searched during certain operations, and also requires that the elements stored in the hash table are totally ordered.

Compilers are incapable of these verification and optimization tasks, however, because they do not understand the semantics of the abstractions used by the programmer. The compiler's only information about the abstractions is based on their implementations, which must necessarily be at a lower level of abstraction than their interfaces. In fact, the compiler does not even know where the interfaces are: it cannot reliably differentiate between a call to a library function and a call to a user function, and concepts are not explicitly represented at all in current C++. The compiler can get some information from implementations, such as types, but it often cannot understand their run-time behavior; the iterator example in [127, § 2.1] shows this limitation using an iterator operation that is semantically treated as addition and subtraction, but internally is much more complicated. The implementations of abstractions, such as data structures, have become increasingly complicated; compilers (especially production compilers) have difficulty with even simple data structures such as linked lists. Moreover, abstractions may be built upon other abstractions: for example, the graph classes in the Boost Graph Library are implemented using abstract data structures defined in the C++ Standard Library [131]. All of these features make information about the behavior and uses of abstractions difficult for compilers to analyze.

The main reason for these difficulties is that compilers use program analysis techniques to determine information about the program. Program analysis provides ways to obtain information about the control flow and data flow of a program, but it requires certain information as input. For example, tracing data through a program requires knowing at least an approximation of the program's control flow. Virtual functions and dynamic loading of libraries prevent a reasonable approximation of the control flow from being created without significant effort, and possibly not at all. On the other hand, simple procedural programs have a good approximation of their control flow in the text of the programs themselves. Similarly, programs with only simple local and global variables require only simple analyses to track their data flow. Pointers and aliasing (the fact that two pointers may point to the same location in memory) make information about data flow much more difficult to obtain. Sophisticated data structures and object-oriented programming tend to use all of these problematic language features. Although compiler research has produced some

partial solutions to these problems, those solutions are imperfect, and usually involve inefficient, non-scalable algorithms for analysis. Thus, using only the implementations of abstractions to analyze them is insufficient [64, 65]. One example of this is a union-find data structure; for this data structure, queries which do not modify the data structure semantically do modify its implementation (to collapse paths within the structure). The fact that such queries can be removed is difficult for a compiler to determine.

However, if the compiler were able to understand the high-level semantics of the abstractions used by a given program, it would be better able to check and optimize uses of the abstractions [127]. Thus, as it is impractical for the compiler to determine these semantics itself, there must be a way for the authors of the abstractions to provide this information to the compiler. Some form of specification must be provided, either as an extension to the compiler, or as information to assist the compiler's previous analyses [117]. Each of these categories allows many possible implementation choices.

One possible way to specify information about user-defined data types and algorithms to a compiler is to define custom optimizations by hand, as well as to define new analyses and extend existing analyses. For example, a user could create a new compiler that knows (in its alias analysis implementation) that distinct instances of certain classes will not share data, a property of the std :: vector class in the C++ standard library. Also, a user could indicate that a certain kind of loop over a given data structure is safe to parallelize, which has been done as part of the ROSE project [113]. The approach of using a custom compiler, however, has the disadvantage of being unnecessarily complex and fragile [117]. Compilers tend to be sophisticated software packages, and a good-quality compiler also uses several programming techniques to make it compile programs faster. Creating even relatively simple, yet robust, optimizations is difficult, even if these advanced techniques are not used. A library to simplify the creation of optimizations within the compiler is unlikely to simplify the process. A possible alternative is to add modules to the compiler [50], or to use source-to-source transformation to implement optimizations [122]; these approaches have many of the same issues, however. Thus, if such a level of effort was required for each particular domain-specific abstraction, the effort would never be made, and the custom analyses and optimizations would never be written. Therefore, as suggested by [117], a domain-specific language

for writing analyses and optimizations would be useful to aid users in writing optimizations, and to ensure that they would be defined when needed.

One alternative to user-written optimizations would be for the implementors of the system compiler to add the optimizations into it. However, as Robison argues, there are many optimizations that would be useful, but each does not have a large enough base of users to justify adding it to a commercial compiler [117]. Therefore, he states, users should be able to define their own optimizations; he discusses adding such optimizations as directly written plugins to the compiler itself, but suggests two forms of declarative optimization specification (typestates and user-defined data flow analyses) as better alternatives.

The authors and implementors of particular abstractions have attempted to provide optimizations specifically for their abstractions, using functionality within some modern programming languages. Within C++, the expression template technique allows a class to have a limited opportunity to examine and alter expressions using objects of its type [148]. This capability has been used, for example, in the Blitz++ array library; that library implements loop fusion and blocking using expression templates [150]. Fast expression templates allow more information about aliasing behavior and operand reuse to be given to the expression template implementation, allowing more opportunities for certain optimizations [66].

A better solution to the problem of defining abstraction semantics in a way usable for a compiler is for users to write their own analyses and optimizations (collectively referred to as *program operations*) in a domain-specific language. The ability for users to define their own optimizations, rather than just providing information for predefined optimizations, gives more flexibility; using a domain-specific language for defining those optimizations adds more usability than requiring the users to write optimizations by hand. In particular, extended regular expressions (regular expressions with intersection and complement) with pattern variables, combined with the ability to nest queries over possible program paths and to call native functions from within a regular expression, are a good way to write such optimizations. I have designed, implemented, and evaluated the **Pavilion** system, including its domain-specific language for program analyses and optimizations, to allow library authors to easily write custom analyses and optimizations for their generic libraries.

Allowing users to write their own analyses and optimizations based on their particular applications and concepts will allow many more optimizations to be developed. The **Pavilion** language is designed to be simple enough to be usable by the level of programmer who is writing generic libraries; relative to writing or modifying a compiler, it is much easier. The **Pavilion** system hides the details of how analyses and transformations are performed, and thus requires much less background understanding of implementation details by the user. Therefore, from a pragmatic perspective, having the **Pavilion** system available to use would encourage library authors to specify the custom semantics of their abstractions. Program traces match how programmers think about their program runs, and regular expressions are already familiar to them from other tools [46]; past work on regular-expression-based program analysis has also argued that they are a natural specification form [101]. The syntactic forms provided by the **Pavilion** language allow relatively sophisticated analyses and transformations to be written while further hiding the details of the internal program representation. In particular, generalized matching of expression trees provides much more power than is possible using literal tree pattern matching (such as that used in Simplicissimus [125]) or simple finite-state analysis of data flow (such as that used in Broadway [64, 65]). Therefore, adding extra functionality to the **Pavilion** language beyond those previous systems will enhance the ability of users to easily write sophisticated optimizations for their abstractions.

1.3. Contributions

This work provides the following contributions to the state of research in compiler analysis and optimization specification and implementation, and in user-defined optimization:

- Defines a powerful, declarative language (the **Pavilion** language) for generically specifying program analyses and transformations. This language is based on regular expressions, evaluated over program traces; these expressions may be both generated by and include calls to Scheme programs.
- Combines and implements a number of extensions to the basic regular expression framework. These extensions are analyzed as to their implementability and utility to program analysis (Chapter 3), and a set of extensions which form a particular point in the design space is chosen, justified, and implemented (Chapter 5). Some of these extensions have

not appeared in past systems for program analysis; others, while having appeared individually, have not previously been combined within a single system.

- Defines an approach to implementing the chosen set of regular expression extensions (Chapters 6 and 4).
- Shows how code generation, when added to extended regular expressions, allows the direct expression of generic analyses and optimizations. In particular, the thesis shows how simpler analyses and optimizations can be composed into more complicated ones in a systematic way; thus, optimizations for particular instances of a generic component can be defined by applying an optimization generator for the component to analysis and optimization fragments for its parameters.
- Evaluates the designed **Pavilion** system on a set of analysis and optimization examples (Chapter 7).

CHAPTER 2

Related work

Several areas of work relate to that presented here. Many methods of adding extra information to compiler analyses and optimizations, as well as adding completely new analyses and optimizations, have been explored. In the area of abstraction-based analysis and transformation of programs, three varieties of abstractions have been used to give extra information to the compiler: concepts, types, and functions. Some of these systems are purely designed to analyze programs for correctness, while others are designed to optimize their performance. Also, systems for user-defined optimizations and analyses have been implemented, most of which operate on abstractions. Among these systems, different formulations are used for the analyses and optimizations, each with a different level of power and ease of use. Other systems, also using a variety of formalisms, are designed to ease the task of compiler writers in defining analyses and optimizations, without being directly intended for users.

Work related to the language and implementation approach chosen for the **Pavilion** system falls into several categories. Various extensions to the basic regular expression framework have been proposed; some of those have been used for program analysis, while others have been used for analyzing dynamic traces and graphs. Program traces, especially dynamic traces, are used for many purposes; monitoring program correctness is one use in which they are mixed with extended regular expressions and declarative specification languages.

This chapter discusses related work in compiler analysis, regular expressions, and program traces. It first describes previous systems for program analysis and optimization based on abstractions, especially those intended for use by program and library authors. Systems for user-defined optimization in general are also discussed here, as most of them support at least type-based or function-based analysis. It then summarizes work on easier ways for compiler writers to specify analyses and optimizations, particularly focused on declarative specification frameworks, but does

not include systems for compiler writers that only provide simple libraries to assist in writing analyses manually. The chapter then finishes with information about previous uses of regular expressions and program traces in the context of program analysis and optimization. Related work to specific language features, and a more detailed comparison of the features of the **Pavilion** language to past work, is in Chapter 3. Sources related only to regular expression models and other implementation approaches are mentioned in Chapters 4 and 6.

2.1. Concept-based analysis and optimization

A few systems analyze or optimize programs based on concepts, or other constructs (categories of types that share common properties) that are similar to concepts. The main work in concept-based optimization is Simplicissimus, which provides optimizations in the form of expression rewrites guarded by concept constraints on the involved types [125]. Simplicissimus is intended for algebraic simplification of generic programs, and so the restrictions on allowed optimizations are less of a problem in that domain. The XL programming language also supports expression rewriting using a form of abstraction similar to a concept [41]. Gregor and Schupp, the authors of Simplicissimus, later designed the STLlint system for checking software using semantic information in concepts [60, 61]. STLlint does not provide optimizations, but supports a much broader range of analyses than Simplicissimus; users can also plug in information about their own concepts and generic algorithms.

The ROSE framework for user-defined program transformations has been used to implement optimizations based on semantic properties of types, where groups of types with similar properties (basically equivalent to concepts) can be handled by a common transformer [113]. One concept-based optimization implemented in this framework is automatically parallelizing loops on user-defined array-like types using OpenMP [112, 113]. ROSE requires analyses to be written explicitly as manipulations on an abstract syntax tree. This is the most powerful mechanism possible but can be difficult for non-experts to use. Transformations can either manipulate the AST directly or replace part of the AST with source code given in text form.

Some of the proposals for concepts in C++ also included the ability for each concept to specify a set of axioms, each of which is a (possibly conditional) equivalence between two expressions [62, §concept.axiom] When the condition is known to be satisfied, the specification allows

the equivalence to be used as a rewrite rule in either direction. However, this system does not allow any other form of optimization, and does not include a method to determine whether a particular rewrite would be profitable. No analyses or optimizations based on the axiom syntax have been implemented.[1]

Thus, several authors have explored concept-based optimization, but not in combination with a powerful and easy-to-use declarative analysis and optimization specification language. However, many such languages could be extended with predicates on the types involved in rewrite patterns (or the concepts the types model); the advantage of the **Pavilion** system is that optimizations can be composed from simpler components, rather than just having particular optimizations conditioned on types or concepts.

2.2. Type-based analysis and optimization

Some systems for analysis and optimization allow properties to be specified for single types, without allowing common properties among groups of types to be abstracted using concepts. For example, the Glasgow Haskell Compiler includes a type-based expression rewrite feature, but not concept-based rewrites [143, § 7.10].

The CodeBoost system by Bagge et al uses the Stratego language for term rewriting to transform programs written in a subset of C++ [8–10]. It supports type-based analyses, those based on specific function names, and generic rewrite rules based on properties of particular types and operations (similar to, but more restrictive than, concepts). Transformations are expressed as rewrite rules, with a powerful language to create and combine rules. Rules can also be conditional, based on program analysis results. Although CodeBoost contains a typestate analysis, which allows properties of the run-time values of an expression to be abstracted into a finite number of categories and analyzed statically, user rules can define other simple analyses. In CodeBoost, transformations can be written within an input program, rather than in a separate file. The underlying Stratego language has been used to implement several program analyses, including flow-sensitive constant propagation on structured programs [109], and compiler components such as instruction selection [20]. CodeBoost (and Stratego, which can be used to transform programs in other languages) are a user-friendly way to define some kinds of analyses and optimizations. The constant propagation analysis

[1] Jaakko Järvi, private communication.

implemented in CodeBoost, however, does not do arbitrary iteration (i.e., it requires structured programs); also, the user must still write definitions of what are effectively flow equations by hand. Work on using the Program Query Language to find security holes in programs also uses analysis based on particular types and functions [104]. A much earlier paper by Sharp and Otto suggests type-specific optimizations, with an implementation using explicit program analysis, type tests, and AST manipulation [130].

2.3. Function-based and user-defined analysis and optimization

The MOPS system checks software for security flaws using a finite-state model of the behavior of particular functions [26]. For example, MOPS can express an analysis that requires that privileges be dropped using a *seteuid(getuid())* system call before running another program using *execve()*. MOPS, however, relies on particular functions and AST patterns, and on hand-specified finite automata. Simple patterns can also be used, but they are limited to direct matching on the program AST. The FLAVERS system uses a similar error detection procedure based on finite-state automata, but can analyze concurrent programs [47].

The TAMPR system also uses rewriting for program transformation, allowing numerical software to be automatically derived from high-level specifications [19]. This system is designed for transforming pure functional programs, however, and does not include program analysis.

The goal of the telescoping languages project is for users to write their programs in a scripting language, using domain-specific libraries, and then have them compiled into efficient lower-level programs [83, 84]. Efficiency in the generated code depends on extensive preprocessing of the libraries, allowing high levels of optimization without needing expensive optimizations at script compilation time. This work also uses script and library annotations to aid in optimization [25].

A number of papers allow libraries to specify verification conditions, but not optimizations. Ramalingam et al allow libraries to define custom preconditions, and generate program analyses to test them automatically [115]. Engler et al provide similar functionality for testing software for errors [49].

Several authors have explored ways for users to extend a compiler with extra analyses and optimizations, as well as other functionality, without needing the compiler's source code or understanding the compiler's internal details. ROSE is one system that exposes an abstract syntax tree for a C++ program, allows extensions to modify the tree, and then unparses the resulting AST so that the system compiler can compile it into object code [122]. Extensions must be written as C++ code, using explicit tree manipulation, with limited support for pattern matching on the AST and writing replacement code as C++ source code rather than AST fragments. A later paper by Schordan and Quinlan adds better pattern matching on the AST using an attribute grammar, but still requires transformations to be written by hand [123]. As ROSE provides concept-based optimization as well, it is discussed further in Section 2.1 on page 10.

Several systems for user-defined and function-based program analysis and optimization use typestates as their basic framework. A typestate is the combination of the type of some object and an abstraction of its run-time state, usually from a finite list of possible states [141]; for example, a number may be positive, negative, or zero. Later authors have extended this framework with more flexibility, but keeping the same general idea [94]. The Broadway system, by Guyer and Lin, uses typestates in a system for defining library-specific optimizations [64, 65]. Broadway provides an annotation language to define library behavior and optimizations, including typestates, read/write sets, and aliasing properties. The designers of Broadway claim, echoing the motivation of this thesis, that fully general optimizer specification systems are too difficult to use [65]. Type qualifiers, especially flow-sensitive type qualifiers, are another framework based on typestates used to validate program correctness [55]. Type qualifier work requires user annotations of qualifiers, with some type inference supported. The annotations are used to check whether desired properties, such as function preconditions, are satisfied by a program. Work by Chin et al on "semantic type qualifiers" extends this model by automatically proving that the user-defined type qualifier propagation rules model the desired program properties correctly [28].

2.4. User annotations

Many standard compilers allow users to specify simple annotations that affect the compiler's optimization of their programs. These annotations usually only affect the single site where they are

used. As Robison explains in [117], adding more of these annotations into a system compiler is not practical; however, they do form one way of allowing a programmer to specify semantics of their code to the compiler.

One annotation provided in the C99 standard is *restrict* [74, §6.7.3.1]. This keyword is attached to particular pointer types and states that two pointers with the *restrict* property can never be aliased and used in ways that might overlap; Koes et al later generalized such annotations to express a wider range of aliasing properties [87]. This functionality, however, is limited to concrete locations within a program, and only supports one type of annotation. The IBM compilers for C, C++, and Fortran provide a much wider range of annotations, including statements that the iterations of a particular loop are independent [72]. Other compilers, as well as tools such as Lint [77], provide similar annotation mechanisms. However, all of these mechanisms are limited in scope, as they only modify existing optimizations within the compiler, and new annotations cannot be added by the user.

2.5. Declarative specification languages

A number of declarative specification languages are intended for compiler writers; a number of these are described here. Several of them, including some for user-defined optimization, use regular expressions and/or model checking as their basis. A few kinds of analysis specifications are used for declarative specification, ordered from most to least general:

- Explicit specification of flow equations.
- Temporal logic and model checking.
- Pattern matching or logic programming over the control flow graph.
- Various limited or special-purpose forms of analysis.

2.5.1. Flow equations. Many powerful declarative specification languages use explicitly written flow equations to specify analyses. In these systems, every kind of control flow graph node (or program statement) must be paired with a set of equations that specify its effect on the analysis variables. An example of this format for analyses is the constant propagation and constant folding analysis from [85]; the presentation of this analysis will follow that paper. The goal of this analysis is to determine, at each program point (node in the control flow graph) and for each variable,

whether the variable always has a constant value. In this case, each program point has an associated analysis variable. Assume a simple programming language with assignments and control flow statements, with a control flow graph in which each node is labeled with a statement (a more traditional notation which is dual to that used for model checking). In this case, the variable v is constant at the beginning of a node n if it is constant at the end of every node m which has an edge to n in the control flow graph, and those definitions of v all share the same constant value. The variable v is a constant at the end of n if it was a constant at the beginning of n and was not modified within n, or if it was assigned a value in n which is known to be a constant. The analysis variables then contain maps from each program variable to either a particular constant value or an indicator that the variable does not have a constant value. This text form of the analysis can easily be translated into a set of equations — a *transfer function* — for each type of control flow node; Kildall instead defines the function from particular nodes and input flow values to output flow values. For this analysis, the update function for each node in the control flow graph is a function that iterates through all variables in the input map, updating each using the rules given above. The analysis also requires a function to combine the values from multiple input edges to a program point; this function also iterates through all of the variables, combining their input values together.

One modern example of a system using flow equations to specify analyses is Rhodium [98]. In Rhodium, in addition to flow equations, the user must also give a semantic definition to each flow variable in terms of the state of an executing program. The contents of that state can be extended by users as well, for example to record parts of the execution history of the program. Rhodium then automatically generates verification conditions that, if proven using an automated theorem prover, are sufficient to show that the flow equations correctly reflect the semantics of the analysis. Program transformations, expressed as rewrite rules, can also be given; Rhodium can then produce conditions for the transformation to preserve program behavior. This system also has many other usability and flexibility features: user-defined widening operators can be provided, and analyses can be composed and converted to be flow-insensitive or interprocedural automatically. Rhodium also supports infinite analysis domains (lattices), as long as the domains do not have infinite ascending chains (which would cause analysis not to terminate). An earlier system allowing analyses to be

proven correct is by Knoop, Rüthing, and Steffen; this system also generates proof obligations for analysis correctness [86].

Another example of a system using explicit specification of flow equations is Sharlit [144]. As opposed to Rhodium, Sharlit's emphasis is on generating fast implementations of optimizations, rather than on proving them correct. Unlike Rhodium, which is implemented as an interpreter for optimization specifications, Sharlit is a compiler that converts flow equations into an efficient implementation in C++. Flow equations in Sharlit can be written for either statements or basic blocks, and various combination operations can be defined to allow interval analysis. The code generator also implements other optimizations on the program analyses. Thus, Sharlit is a powerful system for specifying optimizations, and includes software to produce efficient implementations of them. The Program Analyzer Generator (PAG) is another system for defining analyses declaratively using flow functions expressed in a custom functional programming language [4]. It can also generate interprocedural analyses from user specifications.

A few recent papers define algorithms for generating flow equations from another kind of specification. These flow equations are then used to implement a program analysis. The most recent is an extension to the Rhodium system by Scherpelz et al [116, 120]. This system requires the user to define all of the flow variables that will be used in the analysis, and how they correspond to program semantics (as the Rhodium system itself requires). Flow equations are then generated by deriving the weakest precondition for an output flow variable to be true, and then finding the weakest (least restrictive) set of input flow variable values that satisfy that precondition; this process generates the most optimistic correct analysis for the given property and set of variables.

A paper by Ramalingam et al defines another approach to flow equation generation: converting a predicate program that runs in parallel with the input program (as is commonly done in model checking) into a set of flow equations for use in program analysis [115]. First-order properties containing quantifiers over variables in the program can also be automatically translated. This system also works by abstracting the program into another using only Boolean variables, but in this case the predicates are based on specifications of library function preconditions.

2.5.2. Program semantics and abstract interpretation. A number of systems for defining program analyses directly use the framework of abstract interpretation [39]. For example, Yi and

Harrison define a functional programming language for writing abstract interpreters, with built-in data types for common lattices [159]. That system is able to dynamically tune analysis precision by adding approximation functions into the analysis; these functions reduce the precision of the program state information where they are inserted, in order to speed analysis. The TVLA system also uses abstracted interpreters for program analysis, but represents the program state by structures in a three-valued logic [99]. Program properties to test are given as logical formulas in a first-order logic with a transitive closure operation. Library-specific analyses have also been implemented in a translator that produces input for TVLA [115]; that is discussed in Section 2.5.1 on page 16. The SPARE system by Venkatesh and Fischer is similar, but uses denotational semantics rather than operational semantics [152]. This system is designed to incrementally update analysis information as the program is changed in an editor.

Debois performs program optimizations using interpreters that keep statically-known portions of the program's execution history; partial evaluation and sophisticated duplicate code removal are then used to optimize a target program by running it using the interpreter [44]. This technique is known to fail for many common optimizations, however; Debois gives constant propagation as an example of such an optimization.

2.5.3. Temporal logic and model checking. Temporal logic has been used to specify program analyses, and, in combination with rewrite rules, to specify optimizations and other program transformations. Examples of temporal logics used for hardware and software verification include linear temporal logic (LTL), computational tree logic (CTL), and full computational tree logic (CTL*). Model checking is then used to test whether a logical formula is true over all possible executions of the program, possibly using a conservative approximation to the program for efficiency. Steffen initiated this line of work by showing that data flow analyses can be thought of as model checking problems, and that a program can be abstracted into a slightly modified version of its control flow graph [138]. Later work using this analogy formed two major branches: using temporal logic to express patterns over the program control flow graph, and using model checking techniques to test a logical formula over all possible executions of a more powerful approximation to the program.

The first of these branches is used more often for compiler optimization. In this model, each event that changes the execution state is the execution of a program statement, and the control flow

graph of a program is used as an approximation to its possible executions. This model is easy to implement, and matches common program analyses well, as was originally noted by Steffen [138]. The Cobalt system of Lerner, Millstein, and Chambers is an example of such a system. It uses a subset of temporal logic to express program analyses as patterns over the CFG, with optimizations specified as rewrites on individual program statements [97]. Cobalt can then either interpret the optimization or produce proof obligations for it to be correct (semantics-preserving). The formalism and techniques used for this were previously used to prove optimizations correct manually by Lacey et al [91]. For correctness conditions to be generated, each analysis must specify the semantic property that it represents for each program point. Cobalt then generates proof obligations stating that the rewrite is safe when the predicate for the optimization is satisfied, and that the analysis tests the correct predicate. These statements are then proved by the Simplify theorem prover [45]. Earlier work by Lacey and de Moor also uses temporal logic (interpreted over the CFG) to specify program analyses [90]. Rewrites in that system are expressed on the control flow graph, not the abstract syntax tree as most other optimization specification languages do. Rus and Van Wyk previously used this form of model checking, but including data flow as well as control flow in the program model, for parallelization of scientific software [118].

Temporal logic is a collective name for several modal logics that can express the properties of a system over time. For example, linear temporal logic (LTL) has operators to represent "φ is true from now on" ($\Box\varphi$), "φ will be true at some time in the future" ($\Diamond\varphi$), and "φ will be true until ψ is true, and ψ will eventually be true" (φ U ψ). An example of an LTL formula is $(\neg p \text{ U } (q \wedge r)) \vee (\Box\Diamond s)$, which represents the property that either p is false until some point in time in which both q and r are true, or s will be true infinitely often in the future, or both.

The second branch (model checking) is more often used for program verification, and often (for example, in the Bandera system [36]) requires user assistance in defining an accurate finite-state model of the program for the model checker to use. On the other hand, model checking can test arbitrary program behavior, and formulas can reference values of program variables and similar semantic features beyond just static information on control flow graph nodes. Model checking can also be much more accurate, as a good model of the program can exclude control flow paths that cannot actually occur during concrete execution, even if they are possible in the control flow graph.

For model checking in general, usually the software or hardware being studied must be abstracted to have only a finite number of states. There is some work on model checking infinite-state systems, but most common implementations require finite-state systems. The problem of creating a reasonable finite-state model is referred to as model extraction. Steffen's original work, as stated earlier, used a slightly modified version of the program's control flow graph as its model, which is inaccurate but is easy to compute. The AX system for program verification uses a variety of techniques to extract a model from a C program [69]. In particular, it can abstract the values of program variables in a user-defined way, and can use program slicing to remove non-essential parts of the program. The Bandera system uses similar techniques, and is able to generate proof obligations that show whether the abstractions applied to program values are correct [36]. Bandera requires the user to define abstract domains for variables and their values in the program manually, however.

Ball et al translate a program and a set of predicates into a conservative approximation of the program whose variables are the truth values of the predicates [12]. This new approximation, a Boolean program (a program with only Boolean-valued variables) can then be verified by a model checker. The variables in this approximation roughly correspond to flow variables used to analyze the original program; each new variable is a predicate on the state of the input program. The Bebop engine previously used Boolean programs, along with binary decision diagrams, for program analysis [13]. The Moped system uses push-down Boolean programs — programs which allow recursive function calls but in which all variables are Boolean-valued — for model checking [52].

In the systems using full model checking over a more accurate program model than the control flow graph, the logical formulas used for verification can include references to program variables and other forms of program state. Because the model checker is interpreting an abstracted version of the program, this information is available when the formula is being tested. The languages used for expressing program queries in these systems are usually forms of temporal logic. The AX system uses linear temporal logic, which it passes directly to the SPIN model checker [69]. The Bandera system uses a simplified language with higher-level constructs, which it then translates to LTL or CTL for existing model checkers [37].

2.5.4. Pattern matching and logic programming. Pattern matching and logic programming have also been used to define compiler optimizations. In particular, regular expressions over possible traces of program execution have been used for this purpose, such as by de Moor et al [43]. In their framework, regular expressions, extended with variables bound as part of the matching process, are used to search for possible program control flow paths (lists of control flow graph nodes). Rewrite rules on the program are then used to specify optimizations. Side conditions beyond regular expressions cannot be defined directly in their optimization specification language, but can be expressed by escaping to an analysis implementation written in Prolog. Liu et al discuss and analyze algorithms for solving both existential and universal path queries with variables [101]. Both of these papers use finite automata internally to implement patterns, and the pattern matching process in both is guaranteed to terminate.

Other systems use extended logic programming languages for greater power and flexibility in specifying analyses and optimizations. One extension used for this purpose is path logic programming, which adds regular expression-based queries (using regular expressions with variables) over possible program paths [46]. The system described in that work transforms the .NET bytecode format into an abstract syntax tree. Logic programs are evaluated on the AST, and then return rewrites on the AST as their results, similarly to what is done in the Whirlwind [96] system. The authors of the .NET transformation system later added the ability to update analysis results based on rewrites, and defined program paths used in queries by context-free grammars [136]. This work was later extended to the context of aspect-oriented programming, allowing advice (modifications to the program execution) to be applied only if the program path until that point matches a given pattern (also a form of function-based analysis and transformation) [3]. This basic framework was also used for program refactoring in the JunGL system [153]; the JunGL language combines path queries, logic programs in Datalog, and a functional programming language to specify complex program transformations.

Prolog itself, without extensions, has also been used to express program analyses. Early work in that direction, by Hildum and Cohen, uses logic programs to transform either an abstract syntax tree or a linear internal representation [68]. Dawson et al also show that high-quality implementations of Prolog can be used to efficiently implement program analyses [40]. That work also argues

that logic programs are a good way to express analyses. This framework for optimizations has some limitations, however: the Dawson et al paper mentions that their system is not able to compute analyses that require widening to handle infinite-height lattices. Saha and Ramakrishnan later showed how program transformations could translate analyses written as logic programs to enable incremental and demand-driven evaluation [119]. Their system is stated to not be able to handle analyses requiring negation, and is likely to share the limitation on widening with the Dawson et al system. Several of the other optimization systems, especially those based on automata, may have similar or more severe limitations (such as only allowing finite lattices). Systems based on logic programming typically use tabled execution [142, 155] to prevent infinite loops.

The Datalog language [22] (a restriction of Prolog used for database queries) has also been used for program analysis in the `bddbddb` system by Lam et al [93]. For this purpose, the structure of a program is encoded in a database (in `bddbddb`, the database is represented by binary decision diagrams), and analyses are expressed as queries over the database. This system provides the Program Query Language to express analyses as queries over possible program traces; these queries are then translated into Datalog. The `bddbddb` system has also been used to find security holes in programs [104]. The Optimix system expresses analyses using relational structures and queries over them [6], as does the APTS system for program derivation [110]. The Blast query language also uses this formalism, extending it with reachability of program paths tested using a model checker [16].

2.5.5. Attribute grammars. A few authors have defined program analyses in terms of attribute grammars. An attribute grammar computes a property on each node of a tree, usually based on the parent or the children of the node. Multiple attributes can be defined, but usually their definitions cannot be circular. Farrow extended this by allowing circular definitions, using a fixpoint computation algorithm to solve the circular equations [53]. The examples in that paper are of using circular definitions for program analysis. Circular attribute grammars have also been used for a similar purpose in the context of intentional programming [146].

2.5.6. Restricted analysis classes. Instead of a limited set of forms of analysis, a system could also limit the particular analyses performed to a predefined list. The work by Whitfield and Soffa on the GOSpel language uses this strategy: only a few kinds of analyses are supported, but arbitrary

rewrites can be used to modify the program code [156]. The GENesis system can then generate implementations of optimizations defined in this way. Karuri defines a similar, but more powerful, system for dependence-based transformations [82].

CHAPTER 3

Feature analysis

Many choices are possible for implementing a language for program analysis and optimization. The fundamental question is whether to use an approach based on explicit propagation of data flow facts, or to use an approach based on regular-expression-based analysis of possible program traces. This chapter justifies that the regular expression approach is a reasonable design choice, especially when ease of use for non-expert users is considered. Given that the regular expression approach has been chosen, many extensions to the basic model are possible; this chapter explains these extensions. Although some of the feature discussion is concerned with implementation issues, most of the implementation discussion is left until Chapter 4 on page 44. From these features, a particular set has been chosen for the **Pavilion** system; these design decisions are explained in Chapter 5 and justified and evaluated in Chapter 7.

3.1. Trace-based model of programs

The **Pavilion** language defines program analyses and transformations as operations on push-down automata representing possible traces of program execution. Push-down automata are used to allow context-sensitive analysis of programs with recursive procedures; extended regular expressions are used to specify analyses and transformations because they do not require non-tail recursion. The control flow graph of the program is used to approximate its possible sequences of execution, as in [138]. In particular, the symbols in the automaton are basic actions taken by the program, such as operations on primitive data types, as well as markers for conditional branches and function calls. This trace model of programs has been used in several past systems for program analysis, such as [43, 101, 145]. Also, outside the context of program analysis, push-down automata are a well-understood subject in computer science; algorithms for manipulating them in various

ways are standard. In particular, most computer scientists have been introduced to push-down automata and operations on them, and they are standard in other work in automaton-based program analysis.

An an example, a simplified pseudocode version of the trace for *a = b + c;* is:

(1) $t_1 = b$

(2) $t_2 = c$

(3) $t_3 = t_1 + t_2$

(4) $a = t_3$

and the trace for the true branch of the statement ***if*** *(a == 0) b = 5;* is:

(1) $t_1 = a$

(2) $t_2 = 0$

(3) $t_3 = (t_1 == t_2)$

(4) Condition: t_3 is true

(5) $t_4 = 5$

(6) $b = t_4$

A model based on program traces fits well with the way that programmers view their programs [46]. In particular, programs end up as sequences of simple instructions; dynamically-determined traces are used for some analysis tasks such as cache modeling. Interpreters of programs similarly perform sequences of low-level actions on behalf of a higher-level program; small-step operational program semantics are also defined by breaking a program's execution into simple operations. The trace-based approach, especially when nested path quantifiers (Section 3.10) are included, is capable of defining program states based on the sequences of actions performed by the program; this feature allows the emulation of some analyses normally expressed using abstract interpretation.

Another advantage of using program traces as the model for analysis and transformation is that a mechanism is already known for modifying traces: finite-state transducers. Finite-state transducers can be easily applied to push-down automata, and the algorithm for that can be extended to remove the restriction that the transducer has only a finite number of states. Modification of program traces using transducers can be used to insert markers during a program analysis to indicate "interesting"

portions of the trace, such as locations of possible security flaws. The transducer model provides a clean mechanism for this task, and transducers can be composed to apply several transformations in a row, each using information from the previous transducers. The **Pavilion** language includes a simple mechanism for transducers to insert or remove elements from program traces; this feature is used to annotate traces of possible program errors and to mark within a program which AST modifications should be made to implement a program transformation.

3.2. Regular expressions

The approach chosen for representing program analyses in the **Pavilion** system is regular expressions with pattern variables, plus a number of extensions described in this chapter. This basic model has been used previously for program analysis [43, 101]; those papers justify the usability of that model. In particular, regular expressions are familiar to users; regular expressions are commonly taught in undergraduate and graduate computer science curricula, and are used in many programming languages and in tools such as `grep`. In this way they are similar to context-free grammars and push-down automata: users can understand them easily because they are a core part of computer science. Also, implementation of regular expressions is well understood. They can be extended to modify strings through a notation for transducers [81] as well.

Standard regular expressions on strings include the empty string ε, single symbols (such as a), concatenation (represented by juxtaposition), disjunction (the $|$ operator), and Kleene star (the * operator, representing zero or more occurrences of a nested regular expression). Some simple examples are:

- ab^*a (all strings of at least two symbols, starting and ending with a, and containing only b except at the start and end)
- $(1(01^*0)^*1 \mid 0)^*$ (all multiples of 3, represented in binary)
- $(\varepsilon \mid a)ba^*$ (strings which start with either b or ab, and contain only a after that)

One possible alternative to regular expressions for program analysis is for analyses and transformations to be written as explicitly specified automata. With such a model, the user would define a set of states, along with the data carried by each. The MOPS system for program security analysis uses such a model, requiring the user to specify deterministic finite automata (and intersections of

them) explicitly [26, 27]. Arbitrary data could belong to each state (as long as the data is comparable for equality), and arbitrary computations could be used to compute successor states. However, the user may not easily be able to define what states are required. Also, the user would be required to describe the automaton's transitions, along with how the state variables change for each action in the program. When a transducer is required, automata could also include output actions associated with their transitions. Several automaton models are possible: deterministic, nondeterministic, alternating, and alternating with synchronization; epsilon transitions may or may not be allowed. More flexibility makes writing automata easier on the user, but complicates the underlying implementation. Writing explicit equations for state transitions, however, is required for program analysis specification systems based on flow equations; some relatively new techniques are able to generate these equations from a higher-level specification [116, 120].

A similar approach to defining automata is for the user to define a program that operates over a single program trace, and which is then converted by the program analysis generator to operate over infinite sets of traces rather than single traces. The generator would internally convert the user's program, possibly on the fly as it is executed, into an automaton. This approach would have many of the same advantages as defining explicit automata: any type of information may be stored in the program's variables, and arbitrary computations can be used to update it; see Section 3.5 on page 30 for more about these advantages. However, as explained in Section 4.2 on page 46, automata used for analyzing and transforming programs rather than single strings must have states that are computably comparable for equality. In the context of a program for analyzing traces, this requirement becomes that the continuation of the program after each access to the input trace is computable, and that the same continuations must eventually reappear if given an input string that repeats a sequence of symbols enough times. In some programming languages, such as Haskell, functions cannot be compared for equality at all; in others, the comparison is not accurate enough to cause a cycle to occur for many programs (for example, comparison of closures in standard Scheme is by their addresses). As in the case of automata, different levels of power might be provided for programs; nondeterminism and alternation might be required to ease some analyses. For example, intersection of regular expressions is useful to indicate that several actions must occur in the trace, but that their order does not matter. Random access to the input trace would be too difficult to

implement and have semantic problems: it is impossible to determine which elements of the trace will be accessed, and it is easy to make a program which has an infinite number of possible states. Thus, it would be difficult to use a standard programming language in combination with random access to program traces.

Relative to approaches based on explicitly specified data flow equations, an approach based on regular expressions on traces has both advantages and disadvantages. A trace-based approach can automatically distinguish different traces over the same region of the program which produce different analysis results; a disjunctive lattice is required to achieve the same precision using flow equations. Using traces also allows actions to be examined explicitly, rather than just as their effects on the program state. On the other hand, state-based analyses have been extended to allow some of this extra information to be added [32, 44].

However, an approach based on analysis of program traces using regular expressions suffers from some disadvantages relative to explicit flow equations. Although regular expressions with pattern variables have been used in past work for program analysis, they are still not the standard way such analyses are expressed. The advantage of trace-based approaches in separating different traces over the same section of the program can also be a disadvantage: this approach does not allow the traces to be combined, leading to less efficiency in some cases than traditional data flow analysis unless extra operators are added. Flow equations support more generality in the results of program analyses: any lattice (or semilattice in some cases) can be used as the result of an analysis, and widening can be used to allow infinite-height lattices [38]. Many program analysis languages based on lattices, such as PAG [4], allow the construction of new lattices; such tasks are more difficult using regular expressions.

Because of the framework of abstract interpretation [39], flow values can be made to directly match run-time program states. In particular, typestates are a simple abstraction of the values of program variables [141], and are easy to express using standard data flow analysis techniques; they form a simple model for users to define their own optimizations [64, 65]. Typestate analyses can be expressed using regular expressions, but they are more difficult; some of the other abstract interpreters defined in the literature cannot be usefully expressed by standard regular expressions at all. Regular expressions, especially with the extensions described below, can also be difficult to

implement efficiently; users may have difficulty understanding the performance of their analyses and transformations. Implementation issues are described in more detail in Chapters 4 and 6.

3.3. Extensions to the basic regular expression model

In addition to the basic regular expression operations and pattern variables as used in [43, 101, 102], several other language features would be desirable. Some are beneficial for implementing concept-based optimizations, while others are designed to improve the orthogonality and completeness of the language. This section discusses these possible features and their utility; Chapter 4 on page 44 shows how the combination of features affects the options for implementing the **Pavilion** language.

Each section in the rest of this chapter is about one possible language feature or set of features. The first section explains the addition of intersection and complement operators to regular expressions to produce extended regular expressions, why that is useful, and how it affects the language implementation. The second section then describes the implementation option of embedding the **Pavilion** language within an existing programming language, and how that would change the language features. Next, the ability to define functions with an analysis or transformation specification is described. Another important feature, the ability to have analyses return results beyond simple Boolean values, is in the fourth section. Related to that (for implementation reasons) is the issue of quantification of variables, in the fifth section. The ability to call functions in the underlying programming language used for program operations is next, followed by the ability to nest path quantifiers within more complicated expressions. After that is the issue of where specifications should be placed relative to the source code. The final section discusses how transformations on the input program are specified, and what types of transformations are supported.

3.4. Extended regular expression operators

One extension to traditional regular expressions that is useful for several applications is to add intersection and complementation operators to produce *extended regular expressions* [2, p. 410]. Intersection allows, for example, a direct expression of the requirement that two different values are computed (in any order) in a certain part of the program trace. For example, the extended regular expression $((a|b|c)^*a(a|b|c)^*) \wedge ((a|b|c)^*a(a|b|c)^*)$ represents a string that contains at least

one a and one b, in any order. Intersections are produced by the expansion of the expression patterns shown in Section 5.3 on page 66. Complementation is less directly useful, but it can express, for example, the property that some incorrect action by a program is not negated by a later corrective action; it also allows optimizations to be enabled by requirements such as that a variable is not modified before its previous value is used. Complementation also allows universal quantification over paths to be implemented by existential quantification (using the equivalence $\forall x.pat \equiv \neg\exists x.\neg pat$). Previous systems for program analysis specification often did not include these extra operators, or limited their use.

Normal regular expressions can be directly translated into nondeterministic finite automata [70, p. 30]; the operators used in the regular expressions translate directly into automaton features, and the translation produces an automata (with ε-transitions) that is linear in size relative to the size of the regular expression. The obvious way to implement the extension of regular expressions to extended regular expressions is to change the implementing automaton structure from nondeterministic finite automata to alternating finite automata. Alternating finite automata generalize nondeterministic automata by allowing both disjunctions and conjunctions on states; in addition to states with the property "at least one successor state leads to an accepting path," the property "all successor states must lead to accepting paths" is also allowed [24]. However, a direct translation into alternating finite automata is incorrect for extended regular expressions [89, 158].

The explanation for this difference is given by Kupferman and Zuhovitsky [89], and the explanation here follows theirs. The reason for the difference in power is that $(A \wedge B) \cdot C$ is not always equal to $(A \cdot C) \wedge (B \cdot C)$ when A, B, and C are arbitrary regular expressions. One example of this inequivalence is $(\varepsilon \wedge a) \cdot a^*$, which is the empty language (as a is a terminal symbol, and ε is the empty string), while $(\varepsilon \cdot a^*) \wedge (a \cdot a^*)$ is equivalent to $a \cdot a^*$. Note that the equivalence does hold universally if $\wedge$ is replaced by $\vee$; this equation is used to convert non-extended regular expressions to nondeterministic automata. In particular, when matching a string w, $(A \wedge B) \cdot C$ requires that the prefixes of w matching A and B are identical. On the other hand, $(A \cdot C) \wedge (B \cdot C)$ allows different prefixes of w to be used, as long as the corresponding suffixes both match C. Extensions to the alternating finite automaton model can directly express this synchronization, such as partially input-synchronized alternating finite automata [158] and alternating automata with synchronized

universality and negation [89]. We use a model that is similar to partially input-synchronized alternating finite automata to implement extended regular expressions in the **Pavilion** system; this model is explained in Section 4.4.4.

3.5. Embedding in an existing language

One possible implementation option that would greatly increase the elegance and power of the language is to define it as a domain-specific embedded language (DSEL) within an existing programming language. This approach would solve many of the other problems in this chapter: user-defined functions would be directly supported, whether they were written as regular expressions, path quantifications, or as native functions in the underlying language. Analysis results could belong to a wide variety of data types, and a large standard library would make those objects easy to manipulate in powerful ways. Variables in analyses could be directly supported, although quantification on them might be more difficult. The language syntax would also be familiar, and the language definition would be pre-written: the **Pavilion** language would just be a set of library types and functions added to it.

Unfortunately, existing programming languages do not have the features required for direct use for expressing regular expressions and using them for program analysis; Section 4.2 on page 46 explains this issue in more detail. The general problem is that standard programming languages do not support extensional equality comparison of functions or continuations, even for those simple cases where such a comparison would be easily computable. A mixed approach is also possible, in which some parts of a specification are explicitly defined data structures while others are embedded code. This hybrid approach is used for some parts of specifications in the **Pavilion** system, and is explained in more detail in Section 5.1 on page 60. The advantage of a hybrid approach is that some of the data structures can be generated by native-language code, and calls to native functions can be expressed directly; these two features greatly ease implementation and increase language expressiveness.

3.6. User-defined functions

Allowing the definition of functions within an analysis would be beneficial for many reasons. Standard programming languages have functions in order to reuse and organize code in different

parts of an application, and then form functions into libraries for cross-application reuse. Similarly, allowing program analyses and transformations to be reused will make them easier to write, and enable knowledge from one developer to be passed to others. Also, more complicated analyses and transformations can be developed from simpler ones; analyses using the semantic information from one concept can be used to give information to those for other concepts, and the combination of information from several analyses, with each using information from the others to enhance its precision (superanalysis [31]) may also be implementable using this mechanism. Some past work has allowed functions to be defined and called during an analysis. Ben-David et al allow properties to be written as semi-extended regular expressions with automata labeling edges [14]. The PSL/-Sugar specification language for hardware also allows extended regular expressions to be defined into variables and then reused across several property specifications [1]. The PQL system includes even recursive calls to subqueries defined as separate functions [104]. Systems based on embedding into general-purpose programming languages, such as [46], also allow function definitions and calls through those languages, although not always within regular expressions. The Bandera system [36] allows predicates to be defined based on properties in the program state, but those properties cannot use temporal quantifiers or represent program path fragments.

A distinction must be made between functions that express properties of program points and those that express properties of paths ("state formulas" and "path formulas," to use CTL* terminology [48]). Functions on program points are useful for tasks such as checking whether an expression is computed on all paths leading into a particular point (Section 3.10 on page 38). Functions on paths are less useful when arbitrary nesting of path quantifiers is allowed, but they allow analyses to be more precisely applied to single traces rather than to the collection of traces through a given program point.

One example of an optimization that uses user-defined functions (expressed using the native function mechanism) is the constant propagation analysis (including value tracking through variable assignments) shown in Section 7.4 on page 95. In this optimization, the *find−constant* function (a property of nodes) is used to find all of the variables that have constant values. The function is necessary because a loop might contain a sequence of variable assignments which passes a constant value through several assignments, leaving it in its original variable at the end of the loop body.

Such a loop would preserve the constant through all iterations. For the optimization to obtain that information, a recursive set of path quantifiers must be used: at each program point that contains a variable copy, the incoming constants must be found. The function is used to provide the necessary recursion to handle arbitrary programs.

Definition and reuse of functions over single program points or states is straightforward: all that is required is the capability to call functions at particular points in the analysis, which is useful for many other purposes as shown in Section 3.9 on page 36. Models that explicitly build an automaton for each subterm of an extended regular expression and then compose them are able to directly process non-recursive inclusion of separately defined paths. For example, the automaton for each path can be stored in a variable, and then incorporated just as if it had been written as a regular expression. On the other hand, regular expressions that invoke themselves recursively through tail recursion are more difficult for such models; the entire automaton cannot be built until it already exists, leading to an infinite recursion. Approaches based on building deterministic automata on the fly from the input regular expressions also have problems with this feature; methods for concatenating new automata onto the ends of existing ones cannot distinguish tail from non-tail recursion.

On the other hand, models that build automata incrementally can often "hand off" the flow of control within the analysis to a separate automaton. For example, a model that builds a nondeterministic automaton from each regular expression can just, when an automaton in a separate function is invoked in a tail call, add appropriate edges to the start state of the new automaton; no knowledge of the automaton other than the name (and possibly arguments) of its start state is necessary. Using a predictable pattern (for example, based on the variable name) for start state names is enough to allow connections to be made, even for automata which have not yet been generated. Models that create deterministic automaton states from a nondeterministic automaton as they are needed can also use this approach, as can alternating automaton models with added synchronization.

Paths defined as separate, reusable entities may cause changes in the semantics of regular expressions: arbitrarily recursive path expressions allow context-free grammars or even Boolean grammars [107] to be specified. However, allowing only tail recursion in path expressions is safe,

and non-recursive inclusion of path expressions is safe anywhere. A static check could be provided to ensure that only tail recursion is used.

3.7. Non-Boolean analysis results

In past systems using regular expressions for program analysis, the results of an analysis are limited to a Boolean value indicating whether the property was or was not satisfied, as well as possibly a set of variable bindings and a program trace or set of program traces. On the other hand, systems that use flow equations to compute analysis results can process a wide variety of data types, and often include syntax for analysis writers to define new data types, such as in [4]. Parsing techniques used for strings, such as LALR(1) parser generators [78] and parser combinators [71], also support arbitrary data attached to nonterminals and computed by parser actions. Thus, it would be useful for the **Pavilion** language to support arbitrary data types as well. In addition to simulating flow equation-based program analysis, arbitrary data types are beneficial in increasing the generality and elegance of the **Pavilion** language. They are also able (through storing environments mapping from program expressions to properties) to replace pattern variables; they are used for this purpose in standard dataflow analyses (such as in analysis code generator PAG [4]).

Many analyses in the **Pavilion** language use analysis results that are functions from environments to Boolean values. The function $find-constant$ in the constant propagation analysis shown in Section 7.4 on page 95 is an example of such a result type. It returns a mapping from program variables to their constant values (for those variables that have them). A single Boolean value would not be appropriate for this function to return because the variables and constant values returned from it are not known on entry; thus, the function is not just a predicate on existing values, but must return variable bindings. It might also return several variable bindings, so a single environment is not enough to return. It must return a set of environments; a mapping from environments to Boolean values (a characteristic function for the set) provides a more general form for its result.

One issue in allowing non-Boolean result types for extended regular expressions is that the operators used in forming them are particularly tied to Boolean values. The $\wedge$, $\vee$, and $\neg$ operators obviously have this problem, but even concatenation and Kleene star are defined in terms of $\wedge$ and $\vee$, such as in [129]. The logical operators can be generalized to arbitrary functions easily enough,

but concatenation and Kleene star remain as issues. Those two operators have been dualized (with the $\wedge$ and $\vee$ operations interchanged in their definitions) [108]; this change provides a template for generalizing them to use arbitrary functions instead of $\wedge$ and $\vee$ by showing what must be modified in the definitions of concatenation and Kleene star to obtain definitions consistent with new $\wedge$ and $\vee$ operators. Other generalizations of regular expression return types have also been defined, such as to values in arbitrary lattices [100], but this work limits the operations used to the standard regular expression operations; the class of automata into which their regular expressions translate is not closed under complementation. Also, only the meet and join operators of the lattice can be used, not arbitrary combine functions.

The main issue with arbitrary data types is the termination of analyses using them. Boolean expressions have only two possible values, and there are simple rewrite rules to simplify those expressions. Such a set of rewrite rules is used in [129] to define a proof procedure for equality comparison of extended regular expressions. A similar set of rewrite rules or an alternative representation for states would be necessary to support other data types. These issues are explained in more detail in Section 4.2 on page 46. For arbitrary data types to be truly useful, a standard library for manipulating the more sophisticated data types must be provided; allowing calls to code in a general-purpose programming language (described in Section 3.9 on page 36) with an existing standard library is one useful solution for that.

3.8. Variable quantification

One issue in the design of the **Pavilion** language is whether, and how, variables can be quantified. Two basic forms of quantification, existential and universal, are defined in standard logics. In an existential quantification, the requirement is that some value of the variable satisfies a given property. For universal quantification, the requirement is that all possible values satisfy the property. In the **Pavilion** language design, the main issue is which of those quantifications are allowed, and how they are allowed to be nested, both relative to each other and relative to path quantifiers.

Existential quantification is required for program analysis, and is provided by almost all past program analysis frameworks based on regular expressions [43, 101]. This form of quantification

allows queries such "there was some expression in the program that was assigned to variable x." Because the push-down automaton of the program contains arbitrarily assigned values that represent AST nodes, many parts of the program must be matched by variables, and those are almost always existentially quantified; similar uses motivated their inclusion in past work as well. An interesting possibility for implementing variables that fits well with some underlying implementation strategies is to require variables to be explicitly declared, assigned, and used. In particular, approaches based on nondeterministic automata keep variables explicitly as members of the automaton state, and so manipulating them directly is possible. This strategy makes implementation simpler, and allows variables to be reassigned after they have already been bound. However, that implementation restricts language design in many other ways; with this implementation, variables must be explicitly represented by single values in the automaton state, preventing disequality constraints on values such as are required for fully general complementation of regular expressions.

Universal quantification, on the other hand, is useful for completeness but is less directly motivated by practical requirements. It can, however, be used for implementing the complement of an existentially quantified clause; the complement might be implicitly created by translating a universal path quantification as explained in Section 3.4 on 28. The **Pavilion** system implements complementation directly, but must keep the same information for existential quantifiers inside complement operators as would be required for a universal quantifier, because of the duality between existential and universal quantifiers expressed by the identity $\neg\exists x.\varphi \equiv \forall x.\neg\varphi$.

A final issue in variable quantification is how quantifiers can be nested, relative to each other and to path quantifiers. If patterns such as $\forall x.\exists y.pat$ are allowed, the resulting set of variable bindings is a function, rather than a value or tuple of values. This issue does not prevent an effective implementation, but makes the task more difficult. Also, Liu and Stoller [102] give the regular expression $(\exists x.pat)^*$ as an example of a class of regular expressions that are more difficult to implement: every iteration of the loop defined by the * operator can have a different value for x. Limiting variable quantifiers to just inside path quantifiers prevents such cases, but leads to an unnecessary loss in flexibility. Variable quantifiers can also be limited to be just outside path quantifiers; Liu et al show an implementation strategy for that case [101]. Fully general variable quantification is implementable, however, such as by the approach discussed in Section 4.4.4 on page 53.

Variable quantifiers are frequently used in practical **Pavilion** analyses to "hide" variables from enclosing operations. In particular, many analyses use intermediate variables to represent internal information about the traces they match. A concrete example of this feature is in the parity analysis shown in Section 11 on page 100. In this analysis, a function *parity−of−vars* is used to find the parities of all variables at a given program point. This analysis works by finding the last assignment to each variable, and determining the parity of the right-hand side of that assignment. A temporary pattern variable **$e** is used to represent the last-assigned expression for each variable. However, this variable name is internal to the *parity−of−vars* function, and in particular is reused in the *parity−of−exprs* function. As each variable can only be assigned once in any particular context, a local scope must be used to prevent interference between the two functions, and between recursive calls of the same function.

3.9. Native functions

A desirable feature for increasing the power of a domain-specific language is for analyses and transformations in the **Pavilion** language to be able to call native functions defined in a general-purpose programming language. As the **Pavilion** language is interpreted by a program written in Scheme, native functions must be written in Scheme (or languages that it can call using a foreign function interface). Several nice features can be easily implemented using native functions. Tests for whether a particular type models a particular concept, for example, are not possible to implement directly using regular expressions; a native call to Scheme code is required to access this information. This particular functionality could be implemented as a special case in the **Pavilion** implementation, but there is no fundamental need for that. The ability to unparse a portion of an abstract syntax tree into human-readable form is also useful, particularly for debugging and when implementing pure program analyses whose results are intended for user consumption. Certain information does not need to be provided in the program trace if it can be obtained through separate function calls; expression types and file positions are examples of this information. Mutation of the program can also be provided through this mechanism, although special-purpose functionality is better for that.

Native functions are heavily used in **Pavilion** analyses and optimizations. Scheme functions are used to represent the user-defined functions shown in Section 3.6, and the ability to embed arbitrary code is used for other computations that cannot be expressed otherwise. Two examples of such computations are in the integer parity analysis shown in Section 11 on page 100. When an integer constant is found in the program, its parity must be determined; a built-in mechanism for this computation does not exist in the **Pavilion** system, as it is specific to this particular analysis. Thus, an embedded Scheme function performs the arithmetic operation required. Similarly, in the addition case of the analysis, the parity of the sum of two values is computed using Scheme code.

In the interest of defining a powerful language, the **Pavilion** language allows arbitrary native function calls and thus is Turing-complete. This property differs from certain previous systems, such as [106], that are able to guarantee that all analyses terminate with a fixed maximum complexity. With a Turing-complete language, of course, neither of those properties can be assured. However, many analyses can be expressed within those more restricted languages, and that subset should terminate even when written in the more powerful **Pavilion** language. An implementation goal of the **Pavilion** system is that any analysis or transformation that appears to "obviously" terminate (for example, has only a finite number of states when expressed as an explicit automaton) should also terminate when written using the **Pavilion** language.

3.10. Nested path quantification

The ability to nest path quantifiers is useful in an analysis specification language. Nested path quantifiers allow a query on a particular path to depend on other paths to the program points on the original path. For example, it is possible to express queries such as "at every program point where *f()* is called, was there some incoming path where *g()* has been called?" This capability allows the specification of properties such as whether a variable has been initialized, whether all previous assignments to a variable used the same value, and whether some outgoing path from the current program point triggers a certain bug. The role of nested path quantification in testing for AST patterns and generalized expression patterns is shown in Section 5.3 on page 66.

Path quantifiers come in two standard forms: existential and universal. The quantifiers can also be evaluated forward and backward in time (i.e., referring to paths until a certain program point or

those after it). Path quantification is necessary for the correctness of most program optimizations. For example, as shown using the regular-expression-based implementation in [42], a certain variable is constant at a program point if all paths leading into that program point have the variable defined as that constant; this query uses a universal, backward path quantification.

Nested path quantifications are not difficult to implement: only some small changes to the implementation allow path quantification expressions to be functions in the underlying programming language, and then the ability to call natively-defined functions allows nested path quantifiers as well. The only potential problems are ensuring that the native function calls have access to the current program point, passing variables between the layers of nesting, and ensuring that whatever caches are used for results of the inner quantifier expressions persist across different calls. Also, allowing path quantifications to be nested recursively causes more difficulty for the implementation.

One important use for nested path quantifications is in determining whether a particular expression is computed at a given program point. An expression pattern can be expanded in three different ways: one that only evaluates the current path, one that evaluates all paths leading to the current program point, and one that evaluates some path (using an existential quantifier) leading to the current program point. In the second case, the tree pattern is expanded into a single terminal pattern for the root of the expression tree, and then a universal, backward path query to determine if the operands of the root node were computed using the correct expressions; the third case is analogous using an existential quantifier. Some simplifications can be used in this process for efficiency, but in general a path quantification occurs in the translation. More details of the translation, including its use of path quantifiers, are shown in Section 5.3 on page 66. It would be possible to do without path quantification for this purpose, but it would be much more difficult. The problem then becomes that there is a certain point in a path (specified by part of a regular expression), and there must be a check that all of the string seen so far matches a pattern (i.e., instead of checking all paths into a certain program point, only the current path is checked). This refinement gives a more precise analysis, but would require one of two changes: either a two-way automaton must be used for path checking or the previous part of the regular expression must be modified. Two-way automata are equivalent in power (without variables) to one-way automata, but the transformation is complicated. The second option has the problem that the regular expression needing to be modified may be deeply nested in

a more complicated expression, and the transformation required is not compositional (because of concatenation and Kleene star, in particular). The current **Pavilion** system requires combinations of regular expressions on the same trace to be created manually; the difficulties in performing the combinations are explained in the context of the example in Section 7.1 on page 84. Nested path quantification is less precise, but much simpler to implement and more compositional.

There is a subtle, but important, difference between path quantifications that are based on program states versus those based on program points in the source code. A program analysis system associates an analysis result with each location in the program, which may or may not contain more information than just a program point. In context-insensitive systems for program analysis, the analysis result at a given program point is the same no matter how that point was reached. In this case, locations used for analysis are just program points. In context-sensitive systems, on the other hand, a location in the program includes both a program point and the call stack by which it was reached; the stacks may be abstracted, but this definition is true abstractly. In the **Pavilion** system, the locations at which path quantifications start are exactly the same as the program points; extra information such as the current procedure stack is not included. This behavior differs from that of some previous algorithms for context-sensitive program analysis and model checking, such as [51], in which path quantifications start from context-sensitive locations which do include call stacks.

Adding information about calling contexts or extra information about the values of program expressions or variables (as in a path-sensitive analysis) into the definition of a location produces a benefit in the quality of analysis that can be performed. The full-state semantics also have the advantage of matching the semantics of logics such as CTL* that only consider states (to whatever level of detail is used in the analysis) and paths between them as objects of interest; the particular way in which the model was generated is not considered. On the other hand, using only program points as the units of analysis within an otherwise context-sensitive model is somewhat less elegant, and also less precise. However, it matches how programs are transformed: a transformation on a program does not apply to only a single calling context, unless parts of the program are cloned [35]. If the greater precision of a location-based approach is needed, the path quantifier can be removed by modifying its enclosing regular expression; see Section 3.10 on page 38 for more information

on this transformation. That transformation adds even greater precision than a location-based approach, but is not compositional: extensive changes to the enclosing regular expression might be required to remove a path quantifier. Both approaches are feasible, but the program-point-based approach is more likely to match how users view their programs, and matches how program optimizations and other transformations are typically applied.

A few previous systems provide nesting of quantifications over program paths. In model checking, the logic CTL* [48] and the modal μ-calculus [88] provide the ability to quantify over all paths from a given state, and allowing quantification over all paths leading to a given state is a standard extension to CTL*. A first-order version of μ-calculus supports quantification over both program paths and pattern variables. An algorithm for fully context-sensitive analysis of pushdown automata using such logics is given in [17]. As many program optimizations require conditions such as "all paths up to this program point have property *P*," non-nested universal path quantifications are supported by previous optimization specification languages based on regular expressions [43, 101]. Error checking and verification systems, such as FLAVERS [47] and MOPS [26], use only single existential path queries.

3.11. Placement of analysis and optimization specifications

A particular usability issue with the **Pavilion** language is where analyses and optimizations should be placed relative to the programs being transformed. The basic issue is whether program operation specifications should be within the source code (in some form), or in a separate file; if they are in the source code, how closely should they be tied to the program? This issue is also affected by the implementation language and status of the language: if the **Pavilion** language is widely used, and at the point where it would be considered for standardization as a part of C++, integrating it closely would be much easier to accomplish than if it is a prototype system with only a few users. If the language were widely accepted, compiler vendors would be more likely to add it to their compiler implementations; direct integration into a compiler would allow more direct integration into the language. Also, the language used for the implementation and how program operations are compiled affects this issue: using some of the closer levels of integration require (or

strongly suggest) an implementation in C++ to directly take advantage of the ROSE infrastructure or another similar compiler infrastructure.

The closest form of integration is to have analysis and optimization specifications as an integral part of the C++ language. This option is the best for ease of use, as program operations would be directly in the source code, and have direct access to C++ features such as template instantiation. Concepts and related constructs of generic programming are supported in a language extension (ConceptC++ [58]) proposed for addition to the next version of the C++ language [62]. For concept-based analyses and optimizations, two locations might be provided for specifications: directly within a concept definition, and as a separate declaration. Placing the specifications within a concept definition is more convenient when they are directly part of a given concept. Allowing them to be separate is also required, however, as a program operation might be added later by a user who did not write the concept in use; this requirement is similar to the need for retroactive modeling of concepts (declaring that a type models a concept separately from the definition of the type) [56]. Similar concerns apply to type-based operations and those for specific library functions. The proposal for extending C++ with concept support includes the ability to specify axioms associated with each concept [62, §concept.axiom]. These axioms are explicitly allowed to be used as rewrite rules, applied in either direction; they can also be defined to be conditional based on the values of other expressions in the program.

This approach, however, is not feasible using current ROSE technology. First, ConceptC++ is not supported by the current version of EDG front end for parsing and type checking C++, preventing its use by ROSE, which uses that front end. Adding this support to the front end is difficult. There is currently no easily extensible parser for the full C++ language; the only implementation available of ConceptC++ is in the GNU `g++` compiler [58, 59], which is not easy to use in a source-to-source transformation framework.

A more limited approach would be to provide analysis and optimization specifications as annotations within a ConceptC++ program, with a separate parser for them. Thus, the language-supported concepts would be used to trigger program operations, but the implementation of those operations would be separate. However, these annotations would not be well-integrated into the language, and would not be subject to the language's syntax and type checking. Similarly to the

previous approach, the lack of ConceptC++ support in ROSE prevents any use of ConceptC++, even without trying to extend the language with analysis and optimization specifications.

A similar approach that can be implemented using the current C++ language is to provide traits classes that indicate which types model each concept [75, 105, 125]. Such an implementation simulates concepts using C++, and provides methods to use template metaprogramming [149] to determine whether a type models a particular concept. Traits classes have been used, including within implementations of the C++ Standard Library [73], to dispatch algorithms to specialized implementations based on the concepts modeled by their argument types; the algorithm *std::advance* with three separate implementations, chosen based on the type of iterator used as an argument, is an example. Allowing the **Pavilion** system to use this approach would involve associating annotations to traits classes, and then instantiating those traits classes with particular types to determine which program operations apply to those types. ROSE does not currently support instantiating templates with new types during a transformation, however; some workarounds are possible, but they are not elegant. An alternative would be for the **Pavilion** implementation to instantiate the templates itself, but template instantiation is a complicated process and is difficult to get fully correct; an approximation to the process would not match the behavior of normal C++ traits classes. Using traits classes also shares the problem with all annotation-based approaches that C++ language mechanisms for correctness checking do not apply.

Instead of using C++ templates to represent concepts, a separate representation could also be provided using annotations in the source code. Several previous systems for program analysis use this approach for other types of annotation: for example, the STLlint system uses it to specify the behavior of generic algorithms [61], and the Broadway system uses it to specify properties of library functions [65]. In the **Pavilion** system, annotations would often need to duplicate the information provided by traits classes, as several standard C++ concepts (such as those for iterators [73, §24.3.3]) already include traits classes that must be specialized for models. Also, this strategy requires replicating the template instantiation process (or something similar) based on the concepts defined using annotations; however, these concepts do not pretend to be normal C++ templates and thus differences in behavior are more acceptable. This approach shares the previously stated disadvantages of

annotation-based systems. It is, however, implementable in the current ROSE framework, and does not require any extensions to the C++ language.

A final location for specifications, the least integrated into the program's source code, is a separate file. This file can be parsed using any syntax desired, but the parser cannot reuse any components from a C++ front end. This approach has all of the difficulties of annotations: concept relationships must be specified and checked manually; and type checking of the specification file would be completely separate from that of the C++ code, allowing possible mismatches in specifications. However, specifications in separate files are easy to implement, and do not tie the implementation to any particular language. The previous implementation strategies require integration into a C++ front end, and that tends to imply using C or C++ rather than a higher-level language. Traditional systems for analysis and optimization specification, intended for use in building compilers, use separate files; they are not meant to integrate with program source, and that decision is appropriate in that context.

The **Pavilion** system currently uses a separate file for specifications, largely for simplicity. Also, the **Pavilion** implementation is in Scheme, which is not able to directly integrate with the ROSE framework to allow direct processing of C++ code. A translation process is implemented to pass information between Scheme and the ROSE framework, but this process does not provide the level of integration necessary for the more direct inclusion of program operation specifications in source code. It would be possible to implement such a translation process, however. The **Pavilion** implementation uses the embedding of Scheme code into regular expressions to elegantly express parameterized optimizations (including those based on concepts); see Section 5.4.2 on page 72 for more details.

3.12. Conclusion

Overall, a language approach based on extended regular expressions, interpreted over program traces, allows a large amount of flexibility in adding new features. Many possibilities exist, although some cause implementation issues; the tradeoffs between the features and implementation ensures that deciding on a particular feature set to use is a nontrivial task. The next chapter explains these implementation choices.

CHAPTER 4

Regular expression implementation options

When designing the **Pavilion** language, implementability was a major concern. Many features are desirable to include, especially to have a fully general and orthogonal language, but implementation concerns may preclude some of them. Seemingly easy solutions exist in the literature for implementing these features, but those solutions are limited to the context of parsing single strings rather than computing intersection of two languages. Work on analyzing (or model checking) languages as opposed to single strings uses languages that are much more limited in functionality. The goal of this work is to attempt to analyze infinite languages using specifications with almost as much power as those used on single strings.

Particular desired language features tend to suggest particular implementation approaches; however, those often conflict with other language features. This work attempts to find a reasonable combination that is both implementable and easy to use. As the goals of the **Pavilion** language are both ease of use and power, elegance and orthogonality of language features were also major concerns. These goals motivate the desire to include language features for completeness, even when they are not motivated by particular examples. Compositionality of the language is also a major design constraint, as that is required for combining independently written analyses and optimizations, and for using results from one analysis (for example, on types modeling a particular concept) to give information to some other analysis. Another, more minor, concern was the performance and scalability of the implementation; however, language power was considered more important. Some performance issues have been considered, however, and future implementations may use more sophisticated techniques to implement the **Pavilion** language more efficiently.

Overall, the main design goals of the **Pavilion** language are ease of use, expressiveness, support for abstraction-based optimization, and the possibility of an efficient implementation. The subgoals to achieve ease of use are to have an orthogonal feature set, generality of the language constructs supported, and similarity to already known languages and structures. Expressiveness is intended to

be achieved by allowing varied value types, functions, compositionality, and the ability to call into a lower-level programming language when necessary.

This chapter explains the implementation concerns in designing the regular expression portions of the **Pavilion** language. In particular, it surveys a number of automaton models useful for implementing regular-expression-based program analysis and optimization, and analyzes their effect on possible language features.

4.1. Generalizing regular expressions for program analysis

One issue in designing the **Pavilion** language is to determine which generalizations of standard regular expressions should be included, considering the goals of power, ease of use, orthogonality, and implementability. Past work in this area has also included pattern variables as part of regular expressions [46, 101], and shown that they are useful for program analysis. Many other possible and desirable features exist; implementability becomes a major concern in determining which can be included.

In order to use regular expressions for program analysis, they must be translated into some type of automaton. Regular expressions (including extended regular expressions) can be directly interpreted on strings, for example by treating them as Boolean grammars and using a variant of the CYK algorithm [107]. However, as shown by [42], this approach does not work for intersecting a context-free language with a regular expression. Thus, an automaton representation is necessary, even if the states in the automaton correspond fairly closely to regular expressions (as in [42, 129]).

Because of pattern variables and arbitrary computations within regular expressions, the set of states used for the automaton created from any particular regular expression is not known until the automaton is applied to a particular program. Some past work, such as [42], solved this problem by first applying a version of the regular expression in which all pattern variables are ignored to the program, and then filling in the variable values later once particular matching program paths are known. The disadvantage of this approach is that it restricts flexibility and efficiency: having variables filled in during the first pass over the program allows more non-matching paths to be skipped immediately, and allows computations and arbitrary variable quantifications (both existential and universal) within the regular expression. The disadvantage of a dynamically created state

set is that it complicates the implementation: see Section 6.2 on page 77 for details. It also means that techniques that rely on having the entire state set in advance, such as explicit construction and operations on finite-state automata, cannot be applied without modifications.

Unlike program analysis using sophisticated extensions to regular expressions, parsing strings using regular expressions or more complicated grammars is a well-understood problem. In fact, standard tools exist for parsing strings using context-free grammars, such as Yacc [78]. One particularly simple approach to string parsing is parser combinators [71], in which parsers are represented as functions, and complicated parsers are built from simpler ones using a set of combinators. This approach is designed for parsing context-free languages, but allows arbitrary computations to be embedded into the parser. Also, a data structure can be built incrementally based on parts of the input string, and arbitrary values can be passed between portions of the parser. Thus, the parser combinator approach gives both flexibility and a simple interface for building sophisticated parsers, but it only applies to single strings.

Any parsing algorithm that is able to parse a string online (reading one character at a time in sequence) can easily be converted into an automaton. The transformation is just to turn the algorithm into a coroutine; every time the algorithm is about to read a character from the input string, it should save its state and then suspend itself. Equivalently, the function to read from the input string should accept a continuation function that is called with the newly-read character. This function then contains all of the current state of the parser. These transformations can be done independently of what kind or amount of state the parser stores internally.

4.2. The state equality problem

The problem with applying standard approaches to string parsing to program analysis is that, unlike string parsing algorithms, language intersection algorithms require a computable mechanism to determine if two automaton states are equal. Equality of automaton states is not needed for string parsing because the string itself is finite, and this finiteness means that (in the absence of epsilon transitions, which can be handled using a graph reachability algorithm) there cannot be any infinite loops in the parsing process. On the other hand, for language intersection, the program trace set and the language produced by the analysis are almost always infinite, and so some method of detecting

loops is required for the analysis to terminate. Loops in both the program's push-down automaton and the analysis automaton must be detected, but the push-down automaton (when the Steffen model of the program is used) is directly based on the (finite) control flow graph, so loop checking for that automaton is easy.

Checking for loops in the analysis or transformation automaton would be easy if extensional equality (equality based on input-output mappings, rather than syntactic structure) were supported for arbitrary functions. If extensional equality were computable, any approach that is capable of parsing (or transducing) single strings could be used for program analysis (or transformation) using the conversion algorithm given above. The lack of extensional function equality requires that the **Pavilion** language be restricted to produce a state set that, at least on any given input program, is finite. Because of the goal of flexibility in the language, the restrictions imposed do not prevent all infinite-state systems (to preserve Turing-completeness), but regular expressions that appear to be finite-state (and that can be processed by other regular-expression-based program analysis approaches) must produce finite state sets.

Thus, several approaches allow the implementation of regular expressions as automata in a way that preserves easy comparison of states. The rest of this chapter surveys the possible implementations, giving their effects on the regular expression extensions they allow. Most are based on deterministic automata, computed on the fly from extended regular expressions, but nondeterministic automata are useful for trace transduction. The **Pavilion** system, as explained in Chapter 6 on page 75, uses a mix of deterministic and nondeterministic automata in an attempt to get the best of both types of automata.

The automaton model chosen has significant impacts on which language features are easily supportable. Table 1 shows the possible features associated with each automaton model, as well as those supported in each layer of the mixed implementation chosen for the **Pavilion** system.

4.3. Nondeterministic finite automata

One form of automaton that can be used for implementing regular expressions is that of nondeterministic automata. These are particularly good for existential path queries using standard regular

Feature	NFA	DFA	Deriv.	ASUN[a]	Conc. tree[b]	NFA/CT[c]
Intersection	Yes[e]	Yes	Yes	Yes	Yes	Yes[g]
Union	Yes	Yes	Yes	Yes	Yes	Yes
Complementation	No	Yes	Yes	Yes	Yes	Yes[g]
Concatenation and Kleene star	Yes	Yes	Yes	Yes	Yes	Yes
Symbol patterns	Yes[d]	Yes[e]	Yes	Yes	Yes	Yes
Existential var. quantification	Yes	No	Yes	Yes	Yes	Yes
Universal var. quantification	No	No	Yes	Yes	Yes	Yes[g]
Node property queries	Yes	Yes	Yes	Yes	Yes	Yes
Non-Boolean path query results	No	Yes	Yes[d]	No	Yes[d]	Yes[d]
Tail recursive path queries	Yes	No	Yes	Yes	Yes	Yes[f]
Symbol transduction	Yes	No	No	No	No	Yes[f]

[a] Section 4.4.3 on page 52.
[b] Section 4.4.4 on page 53.
[c] Section 5.1 on page 60.
[d] With some limitations explained in the text.
[e] But impractical, especially in combination with other supported features.
[f] In the outer layer only.
[g] In the inner layer only.

TABLE 1. Directly supported features for each implementation approach.

expressions, and are used for this purpose, for example, in [101]. However, they have difficulty implementing negation and universal quantification.

Nondeterministic automata can be built through several procedures. They can be built directly, using textbook methods [70, pp. 30–33]. In particular, union, concatenation, and Kleene star are easy and direct in this model, and intersection is possible with only a multiplicative increase in automaton size. Existentially quantified variables can also be directly included: an automaton can just branch for each possible value of the variable. Nondeterministic automata can also be directly chained: the accepting states of one automaton can just contain epsilon transitions to the start state of another, and the second automaton does not need to have any particular form (or be previously defined) for these transitions to be added. Transduction can also be directly supported, including the ability to decide how to modify a particular area of the program using information later in the trace. Thus, a nondeterministic automaton can be built on the fly from a regular expression, and just directly interpreted over the input push-down automaton.

The main advantage of an approach based on nondeterministic automata is that existential constructs and disjunction are easy and efficient. The automaton can just split into two branches for a

disjunction; concatenation of two regular expressions (where the two parts of an input string can be split at any of a number of locations) and the Kleene star operator can also be directly handled in this way. Also, existential quantifications over possible variable bindings are easy; different branches of the automaton's execution can independently choose which value of a variable to bind, and whichever branch succeeds provides the final binding for the variable. Existential quantifications over paths are also direct: push-down automaton already include nondeterminism, and intersecting one with a nondeterministic automaton just uses that capability. Using nondeterminism in combination with existential path quantification also provides efficiency benefits: computations and results for the same branch of the automaton over the same nonterminal in the grammar can be combined, regardless of which other branches of the automaton are also being processed for that nonterminal.

The major disadvantage of nondeterministic automata is that complementation is expensive and complicated. The standard algorithm for complementing automata is to produce a deterministic automaton and then interchange the accepting and non-accepting states [70, p. 59]. When an automaton has a large or infinite number of states, this method is not practical. A nondeterministic automaton can be determinized on the fly as it is executed, but the number of states created may be exponentially large. The states of the complemented automaton would then be subsets of the original state sets; these states might end up being combined into an even larger automaton that could become nondeterministic again (perhaps through disjunction or concatenation). Thus, negation is difficult if a nondeterministic automaton approach is chosen. This state explosion, in the context of building automata explicitly, is explained on page 75 of [70].

A similar issue affects using nondeterministic automata for universal quantification over variables and paths. For these cases, branches in the automaton's execution tree cannot be handled separately; for example, some possible bindings of a universally quantified variable might be accepted by one branch of the automaton and the rest by another branch. Thus, the automaton must be determinized, just as is required for negation. For universal quantification over paths, the same issue applies: some paths could be accepted by one branch of the automaton and the rest by another. If quantification is instead implemented by finding all possible resulting states of the automaton for all paths in the program flow grammar, care must be taken that paths where the automaton aborts are also included; this restriction effectively requires determinization as well. Thus, most language

features require deterministic automata; luckily, transducers are not allowed in most places where determinism is required. The problem with determinization, however, is performance and the explosion in possible states.

4.4. Deterministic finite automata

The other approach to converting extended regular expressions into automata is to build deterministic automata, either directly or by interpreting automata in some other model in a deterministic manner. Deterministic automata have several advantages for program analysis: they can be easily complemented, and universal quantifications over program paths are directly supported. Several distinct approaches can be used to produce deterministic automata or other models that can be treated as deterministic automata, with varying tradeoffs between efficiency and feature support.

4.4.1. Regular expression derivatives. An efficient procedure to convert extended regular expressions directly into deterministic automata is given by Sen and Roşu in [129]. Their approach uses the concept of the *derivative* of an extended regular expression with respect to a particular symbol; the result of this operation is a new extended regular expression that represents the language of all suffixes of strings in the original language beginning with the given symbol. A similar construction is given in work on *language factors* [42]. Sen and Roşu define a proof procedure, based on the derivative function and coinduction, to determine whether two extended regular expressions produce identical languages. By exploring from a given regular expression with all possible symbols, and removing redundant states using the equality comparison algorithm, they are able to directly produce minimal deterministic automata from an extended regular expression without variables. This method has several advantages: it can handle arbitrary extended regular expressions, produces deterministic automata directly, and can support named regular expressions that are called in tail position from other regular expressions. The disadvantage is that a set of simplification rules on extended regular expressions is required as part of the proof procedure, and the rules given in [129] depend on the fact that the regular expressions are either matched or not matched; some other types could be supported with this set of rules, but they do not work for all possible types. Arbitrary values could be used as results if a new set of simplification rules (and proof rules) is found for each new value type. The derivative-based approach is useful, and extending the set of simplification

rules would allow a wider variety of result types. It has the disadvantage, however, of relying on a set of rules for each result type (or family of result types); omissions in the rule set are likely to lead to infinite loops in the program analyzer.

4.4.2. Explicit operations on automata. A direct way to implement regular expressions is as deterministic finite automata produced through explicit automaton operations. The algorithms used for these operations are all standard algorithms from textbooks (such as [70]), but with some modifications required to handle variables. Also, to preserve determinism, algorithms that produce nondeterministic automata (such as union and concatenation) must be immediately followed by conversion to deterministic automata; this conversion may produce an exponential increase in the number of states in the automaton for automata without variables or an arbitrary increase when variables are included. A further disadvantage of this approach is that the entire automaton for each regular expression must be present at once for operations, preventing automata from being built by chaining tail calls to separately defined (and possibly recursive) functions. Also, the structure of a state in this model when variables are used might be complicated; there might be an arbitrary tree of nested sets of constraints on variable bindings, each corresponding to a particular automaton used in the final construction. Despite these limitations, automata constructed in this manner are a reasonable implementation choice when variables are not present, but they become impractical when variables are required.

4.4.3. Synchronized automaton models. Several authors have defined models of alternating finite automata with synchronization, in an attempt to directly model the semantics of extended and semi-extended regular expressions within an automaton-based model. These often extend the previous alternating finite automaton model [24]. An alternating finite automaton is a generalization of a nondeterministic finite automaton, in the sense that an alternating automaton can have conjunction and negation operators in addition to disjunction operators. In particular, a normal nondeterministic automaton accepts a string aw from a state q_1 whenever there is some outgoing edge from q_1 that is labeled with the symbol a and leads to a state q_2 from which the string w is accepted (ignoring ε-transitions). In an alternating automaton, an arbitrary Boolean function (determined by the input symbol) must be satisfied to accept a string from a given state, when computed on arguments that are determined by whether each other state in the automaton leads to an accepting computation

on the remainder of the string. However, the direct alternating finite automaton model is inadequate for implementing semi-extended regular expressions, as explained in Section 3.4 on page 28. Therefore, several authors have developed extensions of the alternating finite automaton model with semantics that better match those of semi-extended and extended regular expressions.

One such extension, used for semi-extended regular expressions, is the partially input-synchronized alternating finite automaton (PISAFA) model [158]. This form of automaton is similar to an alternating finite automaton, but includes a tree of *synchronizing states* that are required to be reached in matching positions within a particular subtree of the automaton's computation tree. Also, states are labeled as existential and universal based on which of their children are required to accept for that state to accept, rather than allowing an arbitrary Boolean combination. Each node of the computation tree of a PISAFA corresponds to an automaton state and a position in the input string, as long as that state is activated at that position in the string. The edges in the graph represent successor relationships in the automaton itself, with existential states leading to one child (the chosen one in that particular execution) and universal states leading to branches in the tree.

Kupferman and Zuhovitsky later simplified the PISAFA model to produce alternating finite automata with synchronized universality (ASU, for semi-extended regular expressions) and alternating finite automata with synchronized universality and negation (ASUN, for extended regular expressions) [89]. In the ASUN model, each universal or negation state is paired with a corresponding synchronization state, effectively forming a tree of synchronizations within the automaton. This restriction is not explicit in their definition of an ASUN, but is implied by the definition of the ϕ function on page 454 of [89]. Thus, the net effect is that the automaton is actually a tree of nondeterministic sub-automata, with each state in the sub-automaton possibly being a call to a conjunction or negation of nested sub-automata. Kupferman and Zuhovitsky do not define an explicit state for their automaton model, as would be required to convert an ASUN into a deterministic automaton, as their goal of string recognition does not require it. They do, however, define a computation graph to represent (without synchronization) information about the runs of an automaton on a particular string; this graph could be used as a state in the deterministic form of an ASUN.

All of these models are shown to successfully implement semi-extended or extended regular expression semantics, and they all lead to efficient algorithms for recognizing whether a string

matches a particular regular expression. Neither model explicitly defines an automaton state in the manner that would be required to produce a deterministic automaton, but they have concepts that could be used for this purpose. Both of them, however, are limited to Boolean-valued regular expressions, and none of this work uses regular expressions with pattern variables; they produce automata with statically known, finite numbers of states.

4.4.4. Concatenation trees. An approach that combines the features of a syntax-based model for regular expression parsing (such as that using derivatives of regular expressions) and an automaton-based model (such as the PISAFA and ASUN models) allows a direct implementation of extended regular expressions, including pattern variables, and a much simpler generalization to a wider range of result value types. The data structure used is referred to as a *concatenation tree*, and so this approach is named the concatenation tree approach. It mixes the best features of the syntactic and automaton approaches to processing extended regular expressions by keeping the basic regular expression syntax but replacing the Boolean operations by arbitrary functions (in a function space for which equality is decidable and which is closed under certain operations).[1]

The type used for functions in the concatenation tree approach is the binary decision diagram (BDD) [21], although any function space closed under certain operations and whose elements are comparable for equality can be used. The basic definition of the concatenation tree data type is:

```
data CT = Epsilon
        | Sym SymPattern
        | Concat CT CT
        | Star CT
        | Combine BDD [CT]
```

In this Haskell listing, the type *[CT]* refers to a list of *CT* objects. The only difference from a standard extended regular expression is the use of *Combine* and an arbitrary BDD for joining separate trees in parallel. Whenever concatenation trees are built, a set of simplification rules is used on *Combine* nodes: any of the arguments which are themselves *Combine* nodes are folded into the current node, and duplicated and unused arguments are removed. Also, the identity element *Epsilon*

[1] Some of the rewrite rules shown here rely on zero elements of the regular expression output values; this restriction may limit the applicability of this data structure to some value types.

for the *Concat* operation is simplified out, and *Concat* operations that have an argument which is false are simplified to false. The constant false is represented using a *Combine* operation with no arguments and the appropriate BDD. These optimizations are necessary to reduce the sizes of the trees generated from the derivative operation shown below.

To use the concatenation tree data structure in the derivative-based framework from [129], two operations must be defined: one determines whether a concatenation tree accepts the empty string, and the second produces the derivative of a concatenation tree with respect to a particular symbol. The Haskell code for the two operations is:

```
empty (Epsilon) = True
empty (Sym _) = False
empty (Concat a b) = (empty a) && (empty b)
empty (Star _) = True
empty (Combine b args) = applyBdd b (map empty args)

deriv e (Epsilon) = falseConcatenationTree
deriv e (Sym s) = if (matchPattern s e) then Epsilon else falseConcatenationTree
deriv e (Concat a b) = if (empty a)
                       then Combine orBdd [deriv e b, Concat (deriv e a) b]
                       else Concat (deriv e a) b
deriv e (Star a) = Concat (deriv e a) (Star a)
deriv e (Combine b args) = Combine b (map (deriv e) args)
```

As can be expected from the data structure definition, almost all of the rules are the same as those for extended regular expressions given in [129]. The only difference is in the rules for *Combine*: in the definition of each operation on a *Combine* node, the operation is applied recursively to the children of the *Combine* node, and then those separate results are combined.

The actual **Pavilion** system has several extra features beyond this basic definition. The first is that the results of the *empty* and *matchPattern* operations are not just Boolean values; they are mappings from sets of variable bindings (environments) to Boolean values. This change allows pattern

variables to be included in symbol patterns, and requires that the variables are bound consistently through the evaluation of the entire regular expression. As part of this, the result of the *deriv* operation is also a map from environments to concatenation trees; the operations are applied separately to each possible result of such functions, and then recombined in a way that ensures consistent variable bindings. The use of functions, rather than just returning a single environment from each match operation, allows for the arbitrary complementation and disjunction of concatenation trees containing pattern variables. The functions are represented using decision trees similar to those in [137], which are similar to BDDs but allow an arbitrary number of possible values to be tested for each variable.

Along with pattern variables, concatenation trees can also be extended to support quantification over pattern variables. For this extension, a new node type *Quantify* is added which gives the variable v over which to quantify, the reduction operation to use (including its neutral element), and a map from possible values of v to the concatenation trees that are inside the quantification. Applying *empty* to this node type is straightforward: *empty* is applied to the outputs of the inner map of concatenation trees, and the resulting function (from sets of variable bindings to Boolean values) is reduced using the given reduction operation. The definition of *deriv* on a *Quantify* node is somewhat more complicated. As usual, *deriv* is first applied to the nested concatenation tree within the *Quantify* node. This operation produces a function from environments to concatenation trees; this function is then separated (uncurried) into a function from environments not containing v to functions from values of v to concatenation trees. This separation splits out the bindings of v into a separate, inner layer of the function; the inner function can then be wrapped into a newly created *Quantify* node (with the same variable v and reduction function as the previous node). After the separation and reintroduction of *Quantify* operations, the result is a function from environments not containing v to concatenation trees; this type is exactly what *deriv* should return.

Another feature that can be added to the concatenation tree framework is the ability to call Scheme functions during the operations. In the **Pavilion** system, native functions are only evaluated during the *empty* operation; thus, they are only applied at single program points. A function call may include concatenation trees as its arguments, and the result of calling *empty* on each of them is passed to the function; information about the program point being analyzed can also be given to the

native functions. These functions can then recursively call back into the program analysis engine with new regular expressions; the function can use the results of these nested analyses (which can be recursive) in computing its result.

The definition of concatenation trees given so far is limited to Boolean-valued results and results that are functions that return Boolean values. These result types can use simplification rules that are almost the same as those for the Boolean case. Generalizing the result type to allow other types would be beneficial, but might lose some of the simplification rules (shown above) that are required for some analyses to terminate. A replacement for BDDs, such as multi-valued decision diagrams [137], would also need to be used for the functions in *Combine* operations.

This hybrid approach is similar to syntactic models (those based on rewrite transformations and proof rules on the syntax of the regular expressions) in that the basic structure of a state is as a tree, with nodes whose semantics are similar to those of the regular expression operators. The tree starts out almost identical to the input extended regular expression, and the definitions of the empty and derivative operators correspond directly to those for regular expressions. The approach also uses a set of simplification rules similar to those in [129], except that the functions used to combine regular expressions are represented as a data structure (or normal form) that can be easily computed. The difference in combination operators, however, is the main advantage of the hybrid approach, and the largest way in which it differs from a more purely syntactic approach: the set of simplification rules is largely independent of the particular value type involved.

The concatenation tree approach is also similar to the use of alternating finite automaton models with added synchronization in that the tree of concatenations represents the set of synchronized regions in the automaton that have been entered and not yet left, and the functions from the values of successor regular expressions to a combined result are similar to the state definitions in an alternating finite automaton [24], with the difference that no input is consumed by a combining function. In particular, an ASUN is a nested structure of nondeterministic finite automata, connected by intersections and complementations; a concatenation tree, on the other hand, is a tree of combination functions (including intersections and complementations), connected by concatenation and star operators. Another difference between a concatenation tree and an ASUN is that a concatenation tree, like other syntactic approaches to extended regular expressions, modifies the expression tree itself

when operating on a string; no other information is used. However, the ASUN model, like other automaton models, uses both an automaton (which is fixed) and a state (which changes based on the string being processed). It is likely to be possible to create a version of a concatenation tree that is more similar to an automaton, which might lead to efficiency advantages. Another major difference is that an ASUN is structured much more like a standard automaton, while a concatenation tree has almost the same syntax and semantics as extended regular expressions. In the mixed nondeterministic automaton and concatenation tree approach shown later (required for transduction, but not otherwise necessary), there is one level of nondeterministic automaton on the outside of the structure, but a completely different kind of language representation internally; the ASUN model has the advantage of having the same model at all levels, and could probably allow transduction in its outermost level as well.

A concatenation tree is also similar to a Boolean grammar [107] in that they are both composed of intersection, union, complement, and concatenation nodes. A Boolean grammar allows cycles in its graph, in order to be a superset of a context-free grammar. A concatenation tree, on the other hand, is a tree, with a special node type for the Kleene star operator. A logical operation in a concatenation tree is also expressed as a BDD or some other representation for functions, rather than as a tree of AND, OR, and NOT operations. A concatenation tree could likely be extended into a graph (allowing only tail recursion), and this change might allow the special case for the Kleene star operator to be removed.

Overall, a concatenation tree is a generalization of an extended regular expression to allow arbitrary return value types while keeping the original regular expression structure and the derivative operation. Unlike automaton states, however, concatenation trees are expensive to work with — they are still trees, not scalar values. Also, to do a full comparison (as opposed to a syntactic comparison) of the equality of two trees, a coinductive procedure similar to that defined for extended regular expressions in [129] would need to be created; the **Pavilion** system just uses syntactic equality. The goal of using explicitly represented functions, as opposed to trees of simple operations, is to allow syntactic comparison of the trees to proceed more easily, without requiring result-type-specific simplification rules or a coinductive proof system. As the use of concatenation trees in the **Pavilion** system is not in a context where the entire automaton is being created explicitly,

and the set of symbols that would need to be tested is infinite or extremely large, more expensive proof procedures are not practical. Thus, concatenation trees provide a good, simplified approach to representing extended regular expressions and their derivatives as automaton states.

4.4.5. Deterministic automata and transformation. Unfortunately, although deterministic automata are a reasonable alternative for program analysis, they are not adequate for transformation. For program transformation, finite-state transducers are required; not all transducers are deterministic, and not all relations expressible using finite-state transducers can be implemented by deterministic transducers [63, § 2.2]. One cause of this problem is when a transducer modifies an early part of the program trace (for example, by annotating the first part of an erroneous execution path) based on events that occur later in the trace (for example, when a later part of the error occurs); nondeterminism is required for the flow of information backward in time. However, all regular expressions that only recognize trace properties rather then modifying them can be expressed deterministically. Also, unnecessary nondeterminism in transducers (for example, due to disjunction of regular expressions) can be removed. The **Pavilion** solution to this problem is to mix deterministic and nondeterministic automaton in the manner shown in Section 5.1 on page 60.

4.5. Summary

Finding an appropriate set of extensions to traditional regular expressions that is both useful for program analysis and efficiently implementable is a challenging task. In particular, unlike string parsing, more fundamental concerns of computability are involved. Even for standard regular expressions that only have finite sets of states, the implementation must be carefully designed to preserve this property. The concatenation tree approach allows the extra features to be added in a deterministic framework, but does not allow transformation. Nondeterministic automata allow transformation through the use of trace transducers, but are limited in the regular expression features they can support. A combination of the two approaches, shown in Section 5.1 on page 60, attains the best of both approaches.

CHAPTER 5

Analysis and optimization specification language

In order to specify analyses and optimizations, users and library authors benefit from a domain-specific language. This chapter shows the design of that language, justifying the design decisions made. In particular, a set of choices was made from the possible features described in Chapter 3; this chapter describes and justifies the choices made. Please note that the language design shown here is more general than that of the current implementation; however, all features shown here are believed to be implementable. Details on the implementation and its limitations are in Section 6 on page 75.

The **Pavilion** framework models programs by the language of possible traces over their control flow graphs, as done by Steffen [138]. The current implementation uses a context-free language (expressed as a push-down automaton) for the possible traces in order to achieve context-sensitivity. Section 6 on page 75 on implementation shows how more information about the program, whether from user-written analyses or not, can be integrated into this model. Note that the language of traces for a program is different from a control flow graph in that the program actions are associated with graph edges rather than nodes, producing an automaton; Steffen views analyses using this model for programs as a limited form of model checking [138]. The model has also been used in other work on program analysis, such as [26, 42, 43, 101]. Unlike traditional model checking, only finite portions of program traces are considered; program optimization does not require analysis of liveness or fairness properties that would require infinite traces. The limitation to finite traces is shared with previous systems using regular-expression-based techniques for program optimization.

Given this way of modeling programs, the next design issue is how to express properties of these traces. Note that, unlike some previous work (but like GraphQL [102]), properties can be expressed using information about the program model itself rather than only the possible traces; in other words, properties are not limited to only those that fit into the formal model of languages over program traces, but can access some information about the program beyond its push-down

automaton as well. This feature is necessary for some forms of program analysis, such as reliably matching AST and expression patterns in the program.

The **Pavilion** language itself consists of a core language, providing basic features for program analysis and transformation, and routines to generate these low-level analyses for some special cases, including expression tree matching and abstraction-based optimizations.

The **Pavilion** language is partially embedded into Scheme, in the sense that a **Pavilion** specification is a Scheme program that generates a lower-level data structure (in the form of an S-expression) representing the analysis or optimization, and this data structure can contain embedded Scheme functions which are called by the **Pavilion** interpreter (an application of the 3-D code technique[1]). Thus, regular expressions are almost always generated, either partially or completely, by uses of the Scheme quasiquotation mechanism, which allows a data structure to be assembled easily from both constant and non-constant parts. The partial embedding approach has several benefits; the main one is code reuse. Analyses and optimizations should be packaged into libraries, and having them generated (and stored in variables) makes this much easier. Also, specifications can be generated by combining other specifications, allowing abstraction-based operations (see Section 5.4 on page 70). Code generation also allows tedious analyses to be derived automatically, and in a manner that hides some details of the trace representation; in particular, expressions in the program are split into several entries in the generated traces, and recognizing AST patterns thus requires fairly long regular expressions.

5.1. Language layers

The **Pavilion** language, both in semantics and implementation, is divided into two layers. The outer layer provides disjunctive features and transduction; the inner layer does not allow transduction but does allow arbitrary extended regular expression features. Both semantic and implementation reasons account for this separation: transduction only makes sense near the top of a regular expression, and certainly not inside a complement operator or nested path quantification. The implementation benefit is that the outer layer can be implemented as a standard nondeterministic transducer with pattern variables, while the inner layer requires more sophisticated techniques (shown in Section 4.4.4 on page 53) to provide extra analysis functionality in a deterministic framework.

[1] Daniel Friedman, personal communication.

The outer language layer consists of the standard regular expression operators, plus existential quantification of pattern variables and symbol transduction operations. The language features provided are comparable to those in GraphQL [102], with the addition of transduction. In particular, the supported operators are **sym** (with some limitations), **concat**, $*$, **delete**, **insert**, **exists**, plus other analysis features that cause a transition to the inner language layer. The inner layer contains all language functionality with the exception of the transduction operators **delete** and **insert**.

The implementation automatically switches from the outer language layer to the inner one whenever it sees a feature not supported by the outer layer: intersection (although this could be supported, with a restriction on transduction, by the outer layer), complementation, universal quantification on variables, path quantification, and some forms of native function call. For implementation reasons, symbol patterns are also handled by the inner layer, but this modification does not affect the language semantics. Bindings of variables are automatically passed between the layers as necessary, with the exception that a variable that has only negative constraints when leaving the inner layer (or a symbol pattern) causes an error when passed to the outer layer. Variables that may have several possible values are converted nondeterministically when returning to the outer layer. Both layers have the same syntax; they are just different subsets of the overall language syntax. All features of standard regular expressions, such as were provided in previous analysis frameworks, are provided in both layers. The use of two layers is mostly transparent; the inner layer allows all features from the outer layer except for transduction, which does not make sense to use within a conjunction or negation.

5.1.1. Extended regular expression operators. Extended regular expressions are built from patterns for single symbols in the program trace using a set of operators. This section first describes the language for patterns on symbols, followed by the operators for combining extended regular expressions into larger ones. All of the operators are standard operators for extended regular expressions on strings (see [2, p. 410] for a description in the string context).

5.1.1.1. *Symbol patterns.* A single symbol in a program trace is matched using a pattern expression with the **sym** operator. For example, the regular expression *(***sym** *A)* can only match a single element of a program trace, and that element is required to match the pattern *A*.

A pattern expression matches a tree of functors and arguments, similar to a term in Prolog or Stratego:

- *(***and** *p1 p2 ... pn)*, which matches a term that matches all of *p1*, *p2*, etc.
- *(***not** *p)*, which matches a term that does not match the pattern *p*
- *(arg1 arg2 ... argn)* (where *arg1* is not **and** or **not**), which matches a term that is a list and whose elements match the patterns *arg1*, *arg2*, etc.
- **$_**, which is a wildcard matching any term.
- **$***var*, which matches any term, binding it to the variable **$***var*. If that variable is already bound, the new value must equal the previous value.
- Strings, symbols, numbers, and other Scheme objects, which match themselves.

Variables and patterns similar to those given here (and often including intersection and complementation of patterns) are standard in regular-expression-based program analysis and optimization [43, 101, 102]. The single-assignment property and implicit matching through patterns (rather than requiring an explicit assignment operation) are also standard.

5.1.1.2. *Concatenation and empty string.* The simplest way to form larger extended regular expressions is through concatenation. In the **Pavilion** language, concatenation is expressed through the **concat** operator. The empty string is represented by *(***concat***)*. Any number of EREs can be concatenated, such as in *(***concat** *a (***concat***) (***concat** *b c) d)*.

5.1.1.3. *Kleene star.* EREs can be repeated using the unary Kleene star operator, which represents the union of all possible numbers of repetitions (zero or more) of a given ERE. For example, *(∗ (***sym** *A))* represents any number of copies of a symbol matching the pattern *A*.

5.1.1.4. *Logical operations.* In addition to the union operator supported by traditional regular expressions, extended regular expressions support the intersection and complement operators as well. In the **Pavilion** language, the **or** operator is used for union, **and** for intersection, and **not** for complementation. The **and** and **or** operators have arbitrary arity, including zero. The Boolean values *#t* and *#f* match everything and nothing, respectively.

5.1.2. Variable quantification. In the **Pavilion** language, users can quantify over the possible values of pattern variables. The language supports two kinds of quantification: universal and

existential. A universal quantification requires that a particular regular expression be matched, regardless of which value is substituted in for a particular variable. An existential quantification only requires that some value produce a match. Variable quantifications can be nested arbitrarily with the normal semantics. For example, it is possible to require that for each value of an outer variable, a regular expression must match for some value of an inner variable.

Although Section 3.8 on page 34 provides more justification for the presence of quantification over variables, the basic idea is that symbols in the program trace can contain several elements that range over arbitrary values. For example, the particular names used for variables and temporaries in a particular program are generally not known when an analysis is written; also, the values of program constants are also likely to be unknown. Thus, pattern variables must be provided to match these elements of the program. Allowing quantification of these variables makes more explicit what is required of their values; it also allows more flexibility in areas such as how pattern variables and paths are related. A distinction can be made, for example, between "there is some value of x for which all paths out of node n match regular expression R" and "for each path out of node n, there is some value of x that allows R to match that path." Making variable quantification implicit in the language, as many past systems have done, would only allow one of those two queries to be expressed. Variable quantifiers can also be nested arbitrarily inside regular expressions. For example, the expression *(***exists** *(a) (∗ re))* (from [102]) represents an arbitrary number of repetitions of *re*, each of which can have a different value for *a*. Allowing such nesting gives more flexibility and orthogonality to the **Pavilion** language.

The syntax of variable quantifications is straightforward. The **exists** operator begins an existential quantification; it is followed by a list of variables and a body expression. For example, {*(***exists** *(a) (***sym** *(Let* **$a $**_*)))*} is a regular expression that existentially quantifies *a* in the nested expression *(***sym** *(Let* **$a $**_*))*. Variables are untyped, and can match any value that is in the correct position within a symbol. As in the Stratego language, but unlike Prolog, variables must either be completely bound or completely unbound: a variable may not be bound to a term that contains other variables which have not been bound. A variable can remain unbound through the entire length of a quantifier, however; a variable declared by a quantifier does not need to be mentioned at all.

Previous work on regular expressions with pattern variables for program analysis has allowed existential variable quantification, but positioned in different ways in relation to path quantifiers and the regular expression as a whole. For example, the work presented in [43] allows existential variable quantifications immediately outside the top-level universal path quantification, but they are not allowed in other places. GraphQL [102], on the other hand, allows only existential variable quantifiers, but allows them in arbitrary positions within the regular expression. Logic programming, also, usually has existential queries for variable values; normally, all variables in a rule are implicitly quantified immediately outside the definition of that rule.

5.1.3. Path quantification. In addition to quantifying over the possible values of variables, programs in the **Pavilion** language can also quantify over possible paths through the input program. These path quantifiers can be arbitrarily nested, both within other path quantifiers, within variable quantifiers, and within regular expressions. A path quantifier expresses the requirement that some (or all) paths from a program point must match a given regular expression; the paths can be either forward or backward in the program execution sequence. Although the paths considered are context-sensitive (require matching procedure calls and returns), paths start at program points, ignoring stack contents (see Section 3.10 on page 39 for more details). Paths are required to span from the current program point to either the beginning or end of the program's execution, depending on whether they are forward or backward path quantifiers.

The syntax for path quantifiers in the **Pavilion** language is simple: a path quantifier is a node property, so it is wrapped in the *(***node** *...)* form. Inside that, the node property is either *(***some−path** *dir re)* or *(***all−paths** *dir re)*, where *dir* is either **forward** or **backwards**, and *re* is a regular expression.

Most previous languages for program analysis and optimization using regular expressions do not include explicit path quantification. The languages for optimization, such as the one in [43], include universal quantifications of the form "at each program point, do all paths from the beginning of the program to that point have property *p*?" Error checking software, such as FLAVERS [47], uses existential queries instead: "is there some path through the program having property *p*?" The system described in [46] for program transformation uses a logic program as the top level of the specification, with multiple regular-expression-based path queries, both existential and universal,

nested inside. In the model checking community, on the other hand, the logic CTL* includes arbitrarily nested path quantifiers, both universal and existential; this logic is often extended to allow quantification backwards in time.

5.1.4. Function definition. In order to promote reuse of analysis and transformation specifications, as well as to allow flow equations to be specified, recursive procedures can be defined for specifications of path properties, control flow model node properties, and path transformations. Recursion is allowed for all of these, but path property and transformation procedures can only be tail recursive (so that context-free specifications cannot be written directly). Path properties here refer to extended regular expressions and other forms of specifying languages or properties over traces, and node properties are those properties that are true or false at particular nodes in the program model; CTL makes a similar distinction between state formulas and path formulas [29].

One language feature that arises from the combination of path quantifiers and procedures is that program analyses can be specified using flow equations as well as regular expressions, or any combination of the two, as explained in Section 7.4 on page 95. The features described above only allow analyses that return Boolean values (possibly associated with particular pattern variables), but extending the return values of procedures to include arbitrary data types allows fully general flow equations (see Section 3.7 on page 33 for an example of flow equations expressed using the **Pavilion** language). As some program analyses are more naturally expressed using flow equations, allowing them as a possible expression of properties increases the expressiveness of the specification language.

Functions that produce regular expressions are just Scheme functions, embedded using the native function call mechanism. If a simple automaton is being defined, the Scheme function might only return a constant regular expression (which might itself refer to a function returning that same regular expression again). A macro is defined to produce such constant regular expression generators.

5.1.5. Native language access. The ability to use Scheme's **letrec** syntax and the 3-D code technique by Daniel Friedman allow automata to be composed easily. Access to Scheme functions is provided in several places in the **Pavilion** language. In the top-level nondeterministic automaton layer of regular expression specification, a **call** operator is provided. This operation, written in the

form *(***call** *f args ...)*, defines a new state in the automaton; when this state is reached, the Scheme function *f* is called with the current symbol, current environment of variable bindings, and the values of *args*. The function is also given a continuation that it calls for each successor state in the automaton. At the inner layer of automata based on concatenation trees, the **call** operator works somewhat differently: the syntax *(***call** *f res ...)* is used to indicate that the results of the regular expressions *res* (those variable bindings at which the regular expressions accept) are passed to *f*, which then returns a new result in the same form.

5.2. Transduction operators

Two operators are provided at the top level of the **Pavilion** regular expression syntax for trace transduction. The **delete** operator is exactly like **sym**, except that it deletes the matched symbol from the output trace. It is not useful in modifying the program abstract syntax tree, but is more useful when the trace itself is the output of the program analysis; an example of this case is when a program is being searched for erroneous possible execution traces, and any bad traces found are printed out (or annotated in a printout of the source code). The **insert** operator accepts a symbol pattern, except that the variables used in the pattern must already have been bound by previous symbol matching operations. For example, the regular expression *(***insert** *(Hello World* **$var***))* inserts a symbol whose content is the list containing *Hello*, *World*, and the current contents of the variable **$var**.

Special symbols whose first element is **mutate** are used for AST manipulation as the result of a program analysis (similar to the approach used in [96]). The rest of the elements in such a symbol must form a program in a simple, stack-based language. The details of this language are specific to ROSE AST operations, and are not presented here for brevity.

5.3. Code generation and AST pattern matching

Having the **Pavilion** language partially embedded in Scheme has allowed several features to be implemented using code generators: Scheme programs that produce regular expressions as their output. In particular, patterns for expressions in the program abstract syntax tree are translated into traces by one code generator. Also, each analysis or optimization based on an abstraction is produced by a code generator that instantiates the specification on a particular data type; the

contents of models are also code generators that, given variable names as parameters, produce specifications for sub-parts of analyses and optimizations. The pattern matching code generator for ASTs is described here; the use of code generators in the optimization of concepts and their models is described in Section 5.4 on page 70.

Patterns over the program's abstract syntax tree are useful for some program analysis and optimization tasks, in particular when examining expressions in the program for algebraic rewriting; they have been used in past systems such as Simplicissimus [125] and MOPS [26]. Thus, having the ability to express AST patterns is useful; however, they are not the native model used by the **Pavilion** system.

One problem with expressing program properties as languages over traces is that a single expression in a program's abstract syntax tree (AST) can map to several elements in a trace. Although patterns over traces are a powerful formalism for program analysis and transformation, many analyses and optimizations require processing of expressions. For example, a simple distributivity optimization might replace $a*c+b*c$ with $(a+b)*c$ to save a multiplication. Because repeated variables in expression patterns are more difficult to implement, the simplified pattern $a*c+b*d$ is used in the rest of this example. This pattern, expressed directly as a low-level trace pattern, is:

*(***concat** *(***sym** *(Let* **$t1** *(Multiply* **$a** **$c**)))
*(***sym** *(Let* **$t2** *(Multiply* **$b** **$d**)))
*(***sym** *(Let* **$t3** *(Add* **$t1** **$t2**))))

Even this three-element subtrace is a simplification; even assuming that the temporary variables **$t1** and **$t2** cannot be overwritten, it is possible that the expressions **$a**, **$b**, **$c**, or **$d** are complicated to compute. Thus, each of them may expand into several elements in the program trace. Also, relying on $a*c$ being computed before $b*d$ is an assumption about the translation of the program AST into a set of traces. The trace pattern for any string ending with a computation of $a*c+b*d$, when generalized to account for all of these details but still simplified from a practical implementation, becomes:

*(***concat** *(***and** *(***concat** *#t (***sym** *(Let* **$t1** *(Multiply* **$a $c***))) #t)*

*(***concat** *#t (***sym** *(Let* **$t2** *(Multiply* **$b $d***))) #t))*

*(***sym** *(Let* **$t3** *(Add* **$t1 $t2***))))*

For more complicated expressions, the trace patterns are even more verbose.

In order to solve this problem, a code generator has been defined for creating patterns that match the trace versions of AST patterns. Repeated variables are not supported in the code generator, as they require a more complicated analysis, outside the framework of program traces, to determine the equality of different expression trees. The example given above can be written as the following call to the code generator:

*(***match−ast−pattern**

'(Let **$t** *(Add (Multiply* **$a $c***) (Multiply* **$b $d***))))*

Several possible translations from AST patterns to trace patterns are possible, depending on the levels of precision and complexity desired in the translation. All approaches are similar, and use the single-trace translation as a subroutine.

To translate an AST pattern into a pattern on a single trace, a generalization of the example above is used. In detail, the expansion of an expression pattern *(Let r (F a b c ...))* is a conjunction of computations of *a*, *b*, *c*, etc. into newly defined (and existentially quantified) temporary result variables *ta*, *tb*, *tc*, etc., followed by a symbol matching *(Let r (F ta tb tc ...))*. In the full implementation, there is also a test that the scopes of the temporaries have not been exited before the final computation. If *a*, for example, is a complicated expression, a recursive call to the code generator is used to test that the subexpression is computed (replacing the *#t* in the first branch of the expanded pattern given above). For example, the pattern *(Let* **$r** *(Add* **$a** *(Multiply* **$b $c***)))* is translated into (with some simplifications applied for presentation):

```
(exists ($t1)
 (concat (and (concat
                #t
                (sym (Let $t1 (Multiply $b $c)))
                (* (sym (not (ForgetTemporary $t1))))))
  (sym (Let $r (Add $a $t1)))))
```

The use of the *ForgetTemporary* symbol here is to ensure that the scope of the temporary variable bound to **$t1** does not end before the use of the variable in the *Add* operation.

The advantage of this translation of AST patterns is that it is straightforward and precise. It only applies to one trace, and does not consider any other branches in the program's execution that only compute parts of the expression. In the literal AST matching case, this feature is not particularly important, as most temporary variables are only defined at one point in the program; however, it becomes important for generalized pattern matching (described below) and when short-circuiting and conditional operators are included in the program. Also, this translation matches any part of the trace whose suffix computes the desired expression, making analyses more difficult to formulate (for example, in the privilege dropping example in Section 7.1 on page 7.1). Two-way automata or another technique may be able to solve this problem, as explained in Section 3.10 on page 38.

Because program analyses using AST matching actually require that *all* computations through a given program point compute the target expression, however, another translation of AST patterns is required. This translation also makes AST pattern matching easier to integrate into other analyses, as the generated trace pattern matches only one symbol (with a large path quantification on the program point just before the symbol). In detail, the all-paths translation handles the top-level computation of an expression tree separately: the translation of *(Let r (F a b))* is:

```
(concat (node (all-paths backwards
                (and computation-of-a-into-$t1
                     computation-of-b-into-$t2)))
        (sym (Let r (F $t1 $t2))))
```

In this translation, the trace patterns used for matching *a* and *b* allow arbitrary strings of symbols both before and after the desired computation, as long as the temporaries **$t1** and **$t2** do not go out of scope. The single-trace AST pattern matching code generator is used to produce the sub-patterns of *a* and *b*; the change in the translation is only at the top level. A similar translation can be defined using the **some−path** operator instead of **all−paths**; this change is used for error checking, in which the requirement is often that some incoming path computes an undesirable expression, rather than all of them as in the case of optimizations. Also, optimizations can require that some expression (for example, mutating a variable) not occur during a particular part of the trace; existential quantification is used for these queries as well.

Once a shorthand syntax for expression patterns is defined, the translation can also be generalized to allow an expression pattern to match other ways of computing a single expression than just a direct subtree in the AST. For example, the pattern for $a * (b + c)$ could also be made to match $int\ x;\ x = b + c;\ a * x$ with some simple additions to the translator; see Section 7.3 on page 94 for more details on this change. In this more complicated form of expression pattern, the differences between the single-path, universal, and existential formulations of AST pattern matching become important. It is possible to have the same variable assigned different values on different program paths leading into the same point, unlike the direct translation of ASTs to traces in which most variables are temporaries that are only defined at one location in the program.

5.4. Optimization using abstractions

One major benefit of making analyses and optimizations simpler to write is that library writers can specify the semantics of their abstractions through custom analyses and optimizations. However, several new issues arise: What kinds of abstractions should have analyses or optimizations defined, and how does the compiler know that a particular implementation corresponds to one of the abstractions? How should an analysis or optimization specify the abstractions to which it applies? Where should the analyses and optimizations be placed relative to the abstractions? How are analyses and optimizations specified for abstractions whose source code is either unavailable or cannot be modified?

5.4.1. Generic programming. The framework of generic programming provides a good answer to the first question, in that it defines a particular form of abstraction that maps well to current programming practice, and that allows analyses and optimizations to be reused across a broad range of different implementations of the same basic abstraction. Generic programming is also commonly used in current C++, including in the Standard Library [73], and is likely to have more direct language support in the future [58, 62].

Generic programming is a technique for increasing the reusability of software libraries by abstracting the details of data types from the algorithms that use them [76]. The C++ tradition of generic programming began with the Standard Template Library [135, 140], which later became part of the C++ Standard Library [73]. Type parameterization is used to define algorithms that can be applied to any data structure whose type matches a certain set of constraints. These constraints are usually grouped into sets referred to as *concepts*. A type (or a list of types) is said to *model* a concept when it satisfies the concept's requirements; the word *model* is also used as a noun to refer to the details (for example, function definitions) of how a type models a concept. Although these constraints cannot be expressed fully in current C++, proposals have been written to add them to future versions of the language [62].

A concept can contain several kinds of requirements. The main kind is the function requirement, or *associated function*, that defines an operation that each model of the concept must provide. For example, in the Forward Iterator concept from the STL [135], one requirement is that every object whose type models the concept has a dereference ($*$) operator. A concept may also contain associated types and refinements of other concepts. An *associated type* is a type related to the main types in a concept; for an iterator, one associated type is the type of the values the iterator produces when dereferenced. A concept B can also include the requirements of another concept A; this is referred to as a *refinement* relationship (in this case, B refines A).

When a type satisfies the requirements of a concept, it is said to *model* the concept, or to be a *model* of the concept. In some implementations of generic programming, this is defined implicitly: all types (or sets of types) that provide the required functions, operators, and associated types of a concept are automatically considered to be models of that concept. The semantic properties are not usually checked when determining models relationships. In other implementations of generic

programming, a type that satisfies the requirements of a concept must be explicitly declared to be a model of the concept, through a *model declaration* or *concept map* [58]. This requirement allows a check (by hand) that the semantic properties are satisfied, in addition to the compiler's check of the other requirements. The differences between these two methods of establishing models relationships are explained in more detail in [56].

One other aspect defined in a concept is its associated semantics. These define what the required functions are required to do, and how they relate to one another. An example of semantic properties is given in the Input Iterator concept of the STL. The semantics of input iterators define the idea of a range defined by two iterators: when incrementing the first iterator in a range, it must eventually reach (and compare equal to) the second iterator [135]. The STL concepts frequently also define algorithm complexity requirements on iterator operations, which are preconditions of the complexity guarantees of the algorithms using those iterators; this thesis does not consider complexity requirements further. Semantic requirements have also been defined for concepts outside the STL. For example, Schupp et al described a collection of concepts from abstract algebra, giving their required identities as semantic requirements [128].

Generic programming in C++ is typically implemented using templates, which are a language feature for parameterizing a function or type at compile time. The advantage of this is that the template system is flexible, and no performance is inherently lost by the compile-time parameterization. Generic programming is not tied to compile-time parameterization, however: a system for translating generic libraries (and software using them) to have all parameterization occur at run-time is shown in [133]. The **Pavilion** system uses only compile-time instantiation of generic algorithms, as that matches common C++ practice.

5.4.2. Integrating concept support into the Pavilion system. The ability to manipulate **Pavilion** analyses and optimizations as Scheme data structures, and in particular to accept them as arguments to and return them from functions allows the integration of concept support into the **Pavilion** system to be direct and elegant. In general, a generic analysis or optimization is analogous to a generic algorithm: as a generic algorithm calls functions and uses associated types from its type parameters, a generic optimization is built from components (analysis and optimization fragments) from its type parameters. Models of concepts for particular types are extended with *associated*

analysis fragments and *associated transformation fragments*. A generic optimization then accepts a set of type parameters that model specified concepts, looks up the necessary fragments from the appropriate models, and combines them into a single optimization customized (instantiated) for those particular types. This approach is elegant, and simplifies the writing of generic analyses and optimizations, as well as the task of writing models of concepts that are used in those analyses and optimizations. The main disadvantage is that a generic optimization is instantiated on particular types, and thus must often be instantiated on all relevant types in the program; a program may have many types that model a given concept. An analogy between generic algorithms and generic proofs of correctness — which, when instantiated with partial proofs based on a type parameter, produce a concrete proof of an algorithm's correctness for that type — has been found by Vargun [147, p. 62].

The constant propagation optimization of Section 7.2 (on page 90) provides an example of a generic optimization. The optimization applies to all types that model the Constant Propagable concept, which is a refinement of the Assignable concept. The Assignable concept has two associated analysis fragments, and the Constant Propagable concept has an analysis fragment and an optimization fragment. These are composed to form the overall constant propagation optimization for a given type. To apply this optimization in all possible ways to a whole program, the optimization is instantiated for each type that is a model of Constant Propagable, and all of these are combined together using nondeterministic choice (the **or** operator). This combination has the effect of applying the optimization for all types simultaneously.

Concepts and models in the **Pavilion** language are not provided in full detail, as they are only used for analysis and optimization, not program type checking. Therefore, a **Pavilion** concept only has a list of type parameters and a list of refinements, and a model has an arbitrary collection (that may not match other models of the same concept) of members. Like C++ traits classes, errors in model definitions are only found when a generic optimization is instantiated. A concept is defined using syntax (a Scheme macro invocation) such as *(***concept** *(Name param1 ...) (Refinement1 arg1 ...) ...)*. This macro defines a Scheme function named *Name* that can be applied to a list of types (as in [157]). The function returns a model, instantiated for the type parameters, if there is one; the no-op *#f* is returned if there is no corresponding model. The **model** macro defines models; the syntax for it is

*(***model** *(Name type1 ...) ('key1 value1) ...).* In this call, *Name* is the concept name, and the *type*$*$ elements are quasipatterns for the `match.ss` library in PLT Scheme. The keys in the *('key value)* clauses are also patterns for the same library. The **model** macro updates the concept function with a new model for types matching the given patterns; older models take precedence over newer ones. The model function accepts a key as argument, and matches it against the keys in the call to the **model** macro; it then returns the corresponding value. The values tend to be functions that return analyses or optimizations, but they can be any Scheme object. Examples of **Pavilion** concepts and models are given in Chapter 7 on page 84.

A more sophisticated implementation of generic optimizations would keep the same basic idea of producing optimizations by combining parts of models, but would work at a higher level to achieve greater performance of the optimization engine. In particular, it is better to instantiate an optimization for each model (which might apply to a broad range of types) rather than to each type individually. This approach would require a technique for determining each different combination of models that might form an optimization for some type, and would compose the optimization as generically as possible from those models. It would also be useful as a future improvement to combine the same optimization applied to different types in a more sophisticated way that could reuse common parts of the optimization on different types.

CHAPTER 6

Implementation

In the implementation of a domain-specific language for program analysis and transformation, many tradeoffs exist between language features and possible implementations. This chapter explains the design decisions that were made for all features except the regular expression implementation, which is described in Chapter 4 on page 44. It also explains the final implementation approach chosen for those features. The main goals of the language and its implementation were to enable a powerful specification language along with the possibility of an efficient implementation. Generality, completeness, and orthogonality of language features were also important goals.

In brief, the **Pavilion** system represents input and output programs as push-down automata, and the analyses and transformations applied to them as transducers. The transducers are treated as finite-state, but their possible state sets are determined at run time (when a particular program is being analyzed). Thus, a fixpoint-based algorithm is used to enable new states to be created and processed as needed. Scheme is used for the implementation because of its flexibility and higher-order features.

6.1. Modeling the input and output programs

The **Pavilion** system represents input and output programs as push-down automata, as is standard for context-sensitive, automaton-based program analysis; one previous system using this model is MOPS [26]. The push-down automaton for a program represents its possible traces of execution (sequences of program statements and expressions evaluated), approximated by considering any reachable path in the program's interprocedural control flow graph to be a valid trace. The symbols in the alphabet of the automaton represent actions performed by the program, such as expression evaluations or control flow decisions. "Administrative" symbols are also inserted, indicating information such as the beginnings and ends of variable and temporary lifetimes, and the AST node being processed. Push-down automata are advantageous for several reasons. For

example, Lal and Reps show that push-down automata are directly able to express the semantics of the *setjmp*/*longjmp* construct in C [92].

The **Pavilion** system transforms programs written in C++ into output programs in C++ using the ROSE framework for source-to-source transformation [122]. The other languages supported by ROSE may be added to future versions of the **Pavilion** implementation. ROSE represents programs by abstract syntax trees; the **Pavilion** system converts that representation into a push-down automaton. The **Pavilion** specification is used to transduce the automaton, producing a new automaton annotated with AST modifications to apply. The annotations, expressed in a simple stack-based language, are then applied to the original AST. After applying the modifications to the AST, it is unparsed into C++ code by ROSE, which then compiles the modified program using an underlying system compiler.

The advantage of this approach is that push-down automata, and the closely related and equivalent model of context-free grammars, are a well-understood construct in computer science. Additionally, various operations, such as checking for language emptiness, are decidable and have polynomial complexity in this model [70, p. 137]. A finite-state transducer can also be easily applied to a push-down automaton, producing a new automaton or set of accepted traces as a result.

The other competing approach, used in past versions of the **Pavilion** system, is to represent sets of program traces by *flow grammars* [145]. This model represents the same traces as the push-down automaton model, but uses a different data structure with different efficiencies for operations. Context-free grammars are easier to intersect with finite-state automata when the set of automaton states is not known in advance; the corresponding algorithm for a push-down automaton effectively treats it as a grammar. On the other hand, forward and backward path quantifications are simpler to implement with push-down automata, as an automaton explicitly separates out the current program point from the calling context. Also because of the explicit notion of a program point, it is easier to implement other node properties as well. Intersection of a push-down automaton and a finite-state automaton is also just a graph search algorithm which is simple to implement. Fixpoint computation is only required when the set of possible output finite-state automaton states is required for a region of the push-down automaton (which might contain cycles), rather than for a single edge.

6.2. Fixpoint-based computation of analysis results

The results of nested program analyses are computed using a fixpoint-based algorithm similar to that used in flow-equation-based program analysis, but modified to account for the use of regular expressions and deterministic automata. The approach is basically that of FLAVERS [47], and is briefly explained here. Assume that the analysis will be conducted to determine whether all paths ending at a particular program point satisfy a given regular expression. To determine whether this property is satisfied, it is necessary to find all of the resulting regular expressions (using the derivative algorithm on concatenation trees shown in Section 4.4.4 on page 53) obtained from running the regular expression backwards on all paths to the beginning of the program. Assume for simplicity that a context-insensitive program analysis is being used (the actual **Pavilion** system uses a more complicated context-sensitive analysis).

Given a particular program point p and concatenation tree c, the set of trees reached from program paths which end at p is the union, for each edge e entering p, of the concatenation trees produced from evaluating $deriv(c,e)$ at program point $source(e)$. These input unions can be evaluated independently. A fixpoint computation is required because there could be cycles in the program's control flow graph, and because the computation of the derivative of a concatenation tree might require a nested path quantification which is computed using the same algorithm. It is possible to optimize this algorithm for the case of a path quantification, as in this case the union can be replaced by the combine operation (normally either *and* or *or*) of the quantifier being used. Also, the resulting concatenation tree for each path is not important in this case, but only whether it accepts the empty string at the beginning of the program (and for which variable bindings it does that). The context-sensitive analysis cannot always use this optimization; it sometimes needs the actual concatenation trees for regions of the program.

This form of computation provides many of the advantages of tabled execution [142,155] which was previously used by the **Pavilion** system, but allows the computations of the derivatives to themselves be updated based on path quantifications. It is possible to implement that feature using tabled execution as well, but many values are memoized and then never accessed. A more direct fixpoint computation stores fewer intermediate values which are never used, saving memory.

6.3. Other implementation features

Other features of the implementation are interesting but not essential to the overall language design or functionality. For completeness, these design decisions are explained, along with motivation for the particular decisions that were made.

6.3.1. Assembling regular expressions. As the **Pavilion** language is partially embedded in Scheme, the normal Scheme language parser is used to parse specifications. In particular, the input to the **Pavilion** system is a Scheme program, which then calls functions to perform program operations; these functions are given specifications as their inputs. This approach allows analysis and optimization specifications to be generated and/or assembled programmatically, which greatly simplifies the implementation and the process of adding new features to the **Pavilion** language. Also, raw Scheme code can be embedded directly into the specification data structures, allowing easy use of native functions. More information on the use of automatic generation of regular expressions is given in Section 5.3 on page 66.

6.3.2. Converting abstract syntax trees to push-down automata. In the implementation of the translation from ROSE abstract syntax trees to push-down automata, a set of terminals and nonterminals is created, along with definitions for the nonterminals. Each nonterminal corresponds roughly to a node in the interprocedural control flow graph of the program. Thus, most AST nodes correspond to multiple nonterminals. Each terminal corresponds to an action taken by the program; every subexpression evaluation counts as a separate action in order to simplify analysis. A form similar to three-address form is used (implicitly) for expressions, but links are kept to the original AST nodes for both terminals and nonterminals.

Symbols in the automaton are represented by simple trees (equivalent to Prolog or Stratego terms), as in some implementations of program analysis using logic programming [15]. In particular, three-address form is used for expressions to simplify analyses, just as in the ROSE low-level control flow graph implementation. Terminals are not arbitrary terms, however; only a limited set of term patterns is used, with particular parameters that are filled in from the ROSE AST. Thus, terminals can be represented as a disjoint union of possible patterns and their arguments, rather

than as arbitrary terms. Patterns used for matching against terminals can include arbitrary patterns (either more or less general than the patterns used for the terminals themselves).

The actual generation of the automaton for a particular program is done in two phases: a C++ program converts a ROSE AST into a flow grammar [145], and a Scheme program converts that into a push-down automaton. The C++ program is generated by a template that is written in a domain-specific language (represented as Scheme data structures with C++ expression trees mixed in) that is then compiled to pure C++ code. This approach has the advantage that some of the bookkeeping associated with creating new nonterminals and adding grammar rules for them is handled automatically. Additionally, the terminals in the code template are represented in the same way as they are in analysis specifications, and are thus separated from the particular pattern-plus-arguments representation that is used in the generated grammar. The grammar is then dumped as a Scheme data structure, for use by the **Pavilion** interpreter. A preprocessing pass in the **Pavilion** interpreter converts this grammar into a push-down automaton. Combining these passes in the future would simplify the implementation.

6.3.3. Representing concepts. In the **Pavilion** system, a concept is represented as a Scheme function that maps from the concept's type parameters to either *#f* (if no model exists for those type parameters) or a model instantiated for a particular set of concrete types; this view of concepts and models is based on the formalization in [157]. An instantiated model is then a function that accepts a tag symbol as its sole argument, and that returns the desired information as its result. In the present implementation, the only information that can be stored in models is regular expressions used as parts of analyses and optimizations. There is no checking that any particular members exist in models: as in C++ templates, missing members result in an error when they are accessed. There is also no specialization ordering on models of a concept; the first model that matches a particular set of argument types is chosen. There are Scheme macros **concept** and **model** that define the respective constructs; examples of their use are given in Section 5.4 on page 70. Other possible implementations for concepts and models, which would be more suitable for a production-grade implementation, are explored in Section 3.11 on page 40.

6.3.4. Implementation language. One interesting design decision when implementing the **Pavilion** system was whether to implement a compiler or an interpreter, and which programming

language to use. Various tradeoffs were available between what features were possible in the analysis language and how they mapped to the underlying language implementation. As one design goal was to produce efficient analysis and transformation programs from specifications, compiling program operations into a lower-level language would be useful; there are several impedance mismatches between a language such as C++ and a high-level language for specifying program operations, however. In the end, implementation as an interpreter in Scheme ended up being the most productive approach. Implementation as a compiler producing C++ code was used for some limited prototype versions of the **Pavilion** system, however, and the full system should also be implementable in this manner.

The ideal way to implement the **Pavilion** language would be as an domain-specific embedded language within a general-purpose programming language. This embedding would allow all general-purpose language features to be used, and would provide access to existing libraries. It would also ease implementation, because language features not specific to program analysis and transformation would not need to be implemented, and a compiler or interpreter for program operations would be unnecessary. Two basic choices exist for embedding regular expressions in an existing language: directly interpreting the regular expressions as normal program code, and providing a library to build a data structure for each regular expression.

The problem with a direct interpretation approach is that the semantics of regular expressions, particularly extended regular expressions, do not fit well with existing languages. Extended regular expressions include intersection operations that would require two paths of the program to run in parallel over the input and complete after reading the same number of symbols [158]. Regular expressions also include nondeterminism. Thus, a threaded execution model would be required to implement extended regular expressions within a normal programming language, and there would still be a requirement that only a finite number of threads exist at any given point in the analysis, and that multiple "equivalent" strings produce the same sets of threads. Thus, it is difficult to use a normal programming language directly to interpret regular expressions.

An alternative way to embed regular expressions into an existing language is to create a library that creates regular expressions in expression tree (or automaton) form, either at compile time or run time. The problem with this approach is that the regular expressions must still be processed

(compiled and/or interpreted) separately from the host language, and so they are not well integrated into the host language. Many restrictions on regular expression operations would also be required to ensure that program states will eventually repeat. This approach would be as difficult or almost as difficult as directly compiling regular expressions without embedding into a language, and would not have the benefits of a fully embedded implementation.

Given that an embedded domain-specific language was not an ideal approach, the decision was made to explicitly define a separate language for analyses and transformations, and then interpret it. The next choices, then, are what language to use to implement the interpreter. The final decision was that PLT Scheme should be used for implementing the **Pavilion** system.

Several languages were examined for the overall implementation. C++ was not a practical option: the system would frequently need to manipulate trees, and memory management, creation, and pattern matching of trees would have been inconvenient in C++. The other languages considered were all functional. Stratego [154] was used for some early versions of **Pavilion**. It has the advantage of nice parsing and pretty-printing tools for user-defined languages, and direct support for rewrite rules is beneficial for some parts of the implementation. However, the ability for every rule in Stratego to fail caused some issues with debugging, and avoiding these issues required using a programming style that made the code look more like Scheme than traditional Stratego code. Scheme, and in particular the PLT dialect [54], was used in the final implementation because of each access to tree structures (through the included pattern matching library). Also, PLT Scheme has a large standard library. Dynamic typing and an interpreter made prototyping and development much faster, although Stratego also has these features. Thus, PLT Scheme was chosen, and turned out to be an effective language for implementing the **Pavilion** system.

6.3.5. Compilation issues. In the future, it would be useful to write a compiler from the **Pavilion** language to C++. That would allow a more efficient implementation, allowing analyses to be run on larger input programs. It would also allow tighter integration with the ROSE framework, which is written in C++. C++ has weaknesses that make writing such a compiler more difficult, however, and certain parts of the **Pavilion** language definition are currently tied to a Scheme implementation.

C++ has some strong advantages as a target language. Not all information about the program must be conveyed using the generated push-down automaton; ROSE methods could be used to

access omitted information as needed. The C++ Standard Library also provides a number of efficient data structures, and C++ is a language designed for high performance. However, the features provided by C++ do not match what is required for many operations used in program analysis. C++ does not natively support discriminated unions, which are useful for representing automaton states, for example. The C++ **union** construct is limited, especially when user-defined data types are involved, and using a library such as Boost.Variant would add an extra dependency and is not completely equivalent to an algebraic data type. Objects can be used to represent discriminated unions, but then memory management becomes a problem. Also, in C++, first-class functions and continuations must be explicitly represented as closures; functional languages, on the other hand, have direct support for storing functions (including functions referring to variables in outer scopes) as data. The compiler must create all of these objects, and produce code to manage the memory for them.

The **Pavilion** language also currently has some features that tie it to Scheme. One is that Scheme programs can be used to generate analyses and optimizations in the **Pavilion** language; this feature is useful to promote code reuse and make tedious analyses (such as expression tree matching) easier to write. Without some metaprogramming capability, these analyses would need to be either written by hand or built into the language. However, code generation in Scheme does not require an implementation in that language: it would be enough to write the compiler in Scheme, regardless of what language it generates. Using such a front end would also handle the expression of analyses and optimizations as S-expressions. The language is more intertwined with Scheme, however: Scheme functions can currently be included directly into analyses and optimizations. This feature greatly increases flexibility and expressiveness, as any functionality desired can be encoded using Scheme code. C++ code (or code in the target language of the compiler) would need to be embedded instead; some parts of the **Pavilion** system that generate C++ code indeed do include syntax trees for C++ code to include in the output. Thus, a compiler from a modified form of the **Pavilion** language to C++ would be difficult, but possible, to implement.

6.4. Summary

Overall, the **Pavilion** language design was impacted by implementation concerns, although most of the language changes stemmed more from theoretical limitations applicable to all implementations such as extensional equality on functions. Choosing the correct language and implementation approach was important for a practical implementation of **Pavilion**, and a large amount of experimentation was required to make those decisions. The one design decision, other than the language design itself, that had a large impact was the use of a fixpoint-based algorithm as an implementation model; it gave much more flexibility and a common model to handle many kinds of program analysis. This technique also gives room to add more features in the future, such as adding direct flow equation support using arbitrary lattices and widening operators.

CHAPTER 7

Examples and evaluation

Given the design and implementation of the **Pavilion** system, it is important to ensure that its features match those required for the specification of analyses and optimizations, especially those based on concepts and other abstractions. This chapter shows a set of examples of **Pavilion** specifications, evaluating the language's ability to express each sample optimization or analysis. All of the analyses and optimizations shown are simplified, because the full versions are often repetitive in ways that are not essential for understanding. Almost all of the examples could be expanded into full versions, however, with notes given for those that could not be directly expanded. The chapter concludes with an overall evaluation of the language design and its current implementation.

7.1. Executing programs without dropping privileges

One simple example of a control flow analysis that can be expressed in the **Pavilion** system is the privilege dropping example from [26, property 3]. The assumption of this verifier is that the program being verified runs under a Unix-like operating system, and will run with elevated privileges through the setuid mechanism. The program will also be spawning new programs to run with its real user ID (without any elevated privileges), and the requirement being verified is that it does, in fact, drop its privileges using *seteuid()* before running a new program with *execve()*. Therefore, in more detail, the property is that a call to *seteuid()* preceeds each call to *execve()*, and the last call to *seteuid()* has as its argument the current real user ID (obtained from *getuid()*). Because this example is intended to be simple, it will be assumed that the only acceptable call to *seteuid()* is the literal tree *seteuid(getuid())*, as in the original implementation in MOPS. As an example, the program shown in Figure 1 satisfies the property while the program shown in Figure 2 does not.

This analysis uses only a simple form of abstraction, namely the knowledge of the effects of various system calls on the current effective user ID [26]. Thus, the **Pavilion** specification of it is

```
int main(int, char**) {
  seteuid(getuid());
  return execve("/bin/ls", ...);
}
```

FIGURE 1. A program that correctly drops privileges using *seteuid()*.

```
int main(int, char**) {
  if (rand() % 2) seteuid(getuid());
  if (rand() % 2) seteuid(0);
  // The last call to seteuid might have had 0 as its argument,
  // or seteuid may not have been called at all.
  return execve("/bin/ls", ...);
}
```

FIGURE 2. A program that does not drop its privileges properly.

not generic. It does not perform any transformation on the input program; it does, however, use transduction to insert annotations into the push-down automaton indicating the positions of parts of the error (for example, where the last call to *seteuid()* is when it is invalid). Although this version of the analysis annotates the AST of the input program, the **Pavilion** program analyzer is also capable of determining the shortest trace through the program that triggers an error. This feature is common to past program verification systems, and is useful for tracking down how a bug might be triggered at runtime.

The *seteuid()* analysis is presented here in two forms: once using explicit trace operations, and again with AST pattern matching used for brevity. The pattern matching used in both versions is unsophisticated, only matching the literal trace equivalents of patterns in the AST; the same level of matching is provided by MOPS itself (using direct tree pattern matching).

The analysis and annotation specification, written using explicit trace operations, is shown in Figure 3. The analysis has two branches, connected by the first **or** operator: one handles the case where the program runs *execve()* without running *seteuid()* at all, and the other handles the case where the last call to *seteuid()* before the program calls *execve()* does not have *getuid()* as its argument. In the first branch of the analysis, the arguments to the functions being called are ignored, allowing each function call to correspond to only one element in the program's trace. Thus, the analysis directly expresses the sequence of actions required for the error to occur: a sequence of

code that does not call *seteuid()* (or *execve()*, which would terminate the program), then a call to *execve()*; if this sequence is found, an annotation is inserted into the program to mark the offending call to *execve()*.

In the second branch, the argument used in the call to *seteuid()* becomes important, making the analysis more complicated. As before, the program trace is required to start with a leading fragment, but in this case calls to *seteuid()* are allowed in the fragment (as they will be undone by a later call to *seteuid()*). After this, the variable **$not−getuid−result** is matched to any temporary variable in the program trace that is not the result of a call to *getuid()*, and the variable **$not−getuid−exprlist** is matched to any one-argument expression list whose element is not from *getuid()*. When the argument list to the last call to *seteuid()* in the program before the call to *execve()* is such an expression list, the error is annotated in the program. As the arguments to most function calls are ignored (except for that used in the last call to *seteuid()* in the trace), direct use of single-symbol trace matching is effective for this example.

The specification can also be implemented using AST pattern matching; this version is shown in Figure 4. AST pattern matching automatically implements checking of the arguments to operators and functions, but this analysis often does not use that functionality. The only place in the analysis that uses it is the check for calls to *seteuid()* that do not have *getuid()* as their argument. This call to **match−ast−pattern** abstracts the search for calls to *seteuid()*, and requires that any such calls matched have something that is not *getuid()* as their argument. Otherwise, AST pattern matching does not benefit this example, and in fact makes it much more difficult to write. The reason for the complexity is that, because of arguments, the regular expression produced by AST pattern matching code will match traces of arbitrary length; in particular, the code generator adds a wildcard to the beginning of the regular expressions it produces. This property means that any AST pattern can match any number of calls to *execve()*, as long as the matched trace substring ends with the desired AST pattern. The privilege dropping analysis, however, should match only the first call to *execve()*. Thus, the trace fragment found in the first branch of the analysis should have both the “no calls to *execve()*” property and the “ends with a call to *execve()*” property. The obvious problem with a direct intersection of these two properties is that it can never be satisfied: the analysis needs to except

```
(define seteuid−func "seteuid(__uid_t)")
(define execve−func "execve(char const*, char *const [], char *const [])")

`(concat
 (sym (StartOfTrace))
 (or
  (concat ; A sequence of elements which are not calls to seteuid() or execve()
   (* (sym (and (not (Let $_ (NamedFunctionCall ,seteuid−func $_)))
                (not (Let $_ (NamedFunctionCall ,execve−func $_))))))
   (sym (Let $result (NamedFunctionCall ,execve−func $_)))
   (insert (mutate $result ; Add the marker into the output trace
                   "execve() could be called without dropping privileges"
                   generateError))
   #t)
  (concat
   (and
    ; Prevent calls to execve(), which terminates the program
    (* (sym (not (Let $_ (NamedFunctionCall ,execve−func $_)))))
    (concat
     #t
     ; Bind $not-getuid-result to any expression which is not getuid()
     (sym (and (Let $not−getuid−result $_)
               (not (Let $not−getuid−result (NamedFunctionCall "getuid()" $_)))))
     ; Ensure that the expression is still in scope
     (* (sym (not (ForgetTemporary $not−getuid−result))))
     ; Bind to any call to seteuid() with the "wrong" argument
     (sym (Let $not−getuid−exprlist (ExprList ($not−getuid−result))))
     (* (sym (not (ForgetTemporary $not−getuid−exprlist))))
     (sym (Let $seteuid−result
               (NamedFunctionCall ,seteuid−func $not−getuid−exprlist)))))
   ; Prevent later calls to seteuid() from overriding the earlier one
   (* (sym (and (not (Let $_ (NamedFunctionCall ,seteuid−func $_)))
                (not (Let $_ (NamedFunctionCall ,execve−func $_))))))
   ; Require that the program end with execve()
   (sym (Let $execve−result (NamedFunctionCall ,execve−func $_)))
   (insert (mutate $execve−result ; Add the markers into the output trace
                   "execve() could be called with incorrect privileges"
                   generateError))
   (insert (mutate $seteuid−result
                   "This is the location of a possible most recent seteuid() call"
                   generateError))
   #t)))
```

FIGURE 3. The *seteuid()* verification, expressed using explicit **Pavilion** trace operators.

the last call to *execve()*, that is intended to be matched by the wrong-argument portion of the analysis, from the no-calls check. However, just concatenating the properties does not work in general, as the pattern match to find the *execve()* call can match previous *execve()* calls. Therefore, what is required is the regular expression *(***and** *(***concat** *,no−calls−to−execve (***sym $_***)) ,call−to−execve)* which prevents calls to *execve()* except for the one that ends at the end of the trace fragment; this exception is implemented by the extra wildcard trace element *(***sym $_***)* in the first branch of the **and** operator. This particular idiomatic syntax is difficult to discover, however, and requires internal knowledge of how AST pattern matching works at the level of program traces. Thus, this example does not benefit, and is in fact greatly complicated, by the use of single-trace AST pattern matching. However, AST pattern matching can also be implemented in a more approximate way using nested path quantification. Because the privilege dropping analysis is an error check, looking for only one error trace, existential queries are used. The version of the analysis rewritten to use this functionality (the **match−ast−pattern−existential** construct) is given in Figure 5. This version of the analysis and transformation much more directly expresses the specification. The extra unary **and** construct around most of the second part of the analysis is used to force the **Pavilion** implementation into the inner regular expression mode, allowing more flexibility in variable binding.

Overall, the **Pavilion** system is able to express the privilege dropping verification problem from MOPS. The best way to implement it is using backwards existential path queries; if a single-trace analysis is required, direct access to program traces is the best approach, but it requires much more information about the exact layout of **Pavilion** program traces. Using AST pattern matching on a single trace does not require as much information about program trace details, but requires the use of an idiomatic technique that comes from implementation details of the AST pattern matching code generator. The AST pattern matcher using backwards path queries is much better in that it hides most of the details of how AST patterns are transformed into trace patterns, and does not require explicit combination of overlapping components of the regular expression. Note that this verification example is different from optimization examples in that the goal is to find some path that violates the required program property, and so existential queries are required. In optimizations, the usual case is that all paths leading into a particular program point have a property.

```
(define (does-not-contain . ,re*) `(not (concat #t (or ,@re*) #t)))

`(concat
 (sym (StartOfTrace))
 (or
  (concat
   (and
    (concat
     ; Prevent calls to either seteuid() or execve()
     ,(does-not-contain (match-ast-pattern `(Let $_ (NamedFunctionCall ,seteuid-func $_)))
                        (match-ast-pattern `(Let $_ (NamedFunctionCall ,execve-func $_))))
     (sym $_))
    ; Require a call to execve()
    ,(match-ast-pattern `(Let $result (NamedFunctionCall ,execve-func $_))))
   (insert (mutate $result "execve() could be called without dropping privileges" generateError))
   #t)
  (concat
   (and
    ; Prevent calls to execve(), which terminates the program
    ,(does-not-contain (match-ast-pattern `(Let $_ (NamedFunctionCall ,execve-func $_))))
    (concat
     #t
     ,(match-ast-pattern ; Check for a call to seteuid() with the "wrong" argument
       `(Let $seteuid-result
             (NamedFunctionCall ,seteuid-func
                                (ExprList ((not (NamedFunctionCall "getuid()" $_)))))))))
   (and ; Prevent any later calls to seteuid() from overriding the previous one
    (concat
     ,(does-not-contain (match-ast-pattern `(Let $_ (NamedFunctionCall ,seteuid-func $_)))
                        (match-ast-pattern `(Let $_ (NamedFunctionCall ,execve-func $_))))
     (sym $_))
    ; Require a call to execve()
    ,(match-ast-pattern `(Let $execve-result (NamedFunctionCall ,execve-func $_))))
   (insert ; Add the error markers into the output trace
    (mutate $execve-result "execve() could be called with incorrect privileges" generateError))
   (insert
    (mutate $seteuid-result "This is the location of a possible most recent seteuid() call"
            generateError))
   #t)))
```

FIGURE 4. The *seteuid()* verification, expressed with the help of the single-trace AST pattern matching code generator.

```
‘(concat
 (sym (StartOfTrace))
 (or
  (concat
   ,(does-not-contain ; Prevent calls to seteuid() or execve()
     (match-ast-pattern-existential ‘(Let $_ (NamedFunctionCall ,seteuid-func $_)))
     (match-ast-pattern-existential ‘(Let $_ (NamedFunctionCall ,execve-func $_))))
   ,(match-ast-pattern-existential ‘(Let $result (NamedFunctionCall ,execve-func $_)))
   (insert (mutate $result "execve() could be called without dropping privileges" generateError))
   #t)
  (concat
   (and
    (concat
     ,(does-not-contain ; Prevent calls to execve(), which terminates the program
       (match-ast-pattern-existential ‘(Let $_ (NamedFunctionCall ,execve-func $_))))
     ,(match-ast-pattern-existential ; Match a call to seteuid() with the "wrong" argument
       ‘(Let $seteuid-result
             (NamedFunctionCall ,seteuid-func
                                (ExprList ((not (NamedFunctionCall "getuid()" $_)))))))
     ,(does-not-contain ; Prevent later calls to seteuid() from overriding the earlier one
       (match-ast-pattern-existential ‘(Let $_ (NamedFunctionCall ,seteuid-func $_)))
       (match-ast-pattern-existential ‘(Let $_ (NamedFunctionCall ,execve-func $_))))
     ,(match-ast-pattern-existential
       ‘(Let $execve-result (NamedFunctionCall ,execve-func $_)))))
   (insert ; Insert the error markers into the output trace
    (mutate $execve-result "execve() could be called with incorrect privileges" generateError))
   (insert
    (mutate $seteuid-result "This is the location of a possible most recent seteuid() call"
            generateError))
   #t)))
```

FIGURE 5. The *seteuid()* verification, expressed with the help of the existential path query AST pattern matching code generator.

7.2. Simple generic constant propagation

A basic optimization using semantic properties attached to concepts is to propagate constants from definitions of variables to uses. The generic version of this optimization is a generalization of the standard version; in particular, the generic version allows a model of a concept to specify exactly what a constant is, how they are assigned to variables, how they are used, how a variable could be mutated, and how to change an AST to replace a use of a variable with a constant. In the

simple form of constant propagation used here, all partial traces leading to the program point being optimized must have the following form (as shown in regular expression form in [42]):

(1) The start of the trace.

(2) Any actions by the program.

(3) An assignment of a constant c to a variable v.

(4) A region in which no modifications occur to v.

(5) A use of the variable v; this use is changed to c by the optimization.

Thus, to create a concept-based optimization, this basic pattern is filled in using information from a particular model. Two concepts are used for the optimization: an Assignable concept, roughly based on the one in [135], containing information about variables of a given type; and a more specialized Constant Propagable concept containing information about constants. The simple constant propagation analysis uses the following members from Assignable:

- *modification−of−within*: a function accepting a pattern variable containing the name of a program variable and returning a regular expression that matches a change of the variable.
- *variable−use*: a function accepting a pattern variable v containing the name of a program variable and a pattern variable e that will be bound to an expression, and returning a regular expression that matches a use of variable v as an operand of the expression e.

and the following members from Constant Propagable:

- *assign−constant*: a function accepting pattern variable names v for the name of a program variable and c for a constant value, and returning a regular expression matching an assignment of that constant to that variable.
- *change−variable−use−to−constant*: a function accepting pattern variable names e for a variable use to change, v for the name of a program variable containing a constant, and c for the value of the constant; and returning a regular expression that modifies the program AST to replace the variable v with the constant c in the use e.

Given the definitions of these two concepts and appropriate models, the constant propagation optimization applied to a type t is shown in Figure 6. In this code, syntax such as *((assignable t) 'variable−use)* is used to access a member of a model (in this case, the model of Assignable for the type t). There is a test, not shown here, to ensure that this model exists before the

constant propagation optimization is instantiated for the type *t*. The function **wrap−partial−trace** is used to mark that this optimization is not required to match an entire program trace, but only a part of the trace.

```
(let ((variable−use ((assignable t) 'variable−use))
      (modification−of−within ((assignable t) 'modification−of−within))
      (assign−constant ((constant−propagable t) 'assign−constant))
      (change−variable−use ((constant−propagable t) 'change−variable−use−to−constant)))
  (wrap−partial−trace
   `(concat
    (node
     (all−paths backwards
      (concat (sym (StartOfTrace))
              #t
              ,(assign−constant '$constant−var '$constant−value)
              ,(does−not−contain ,(modification−of−within '$constant−var))
              ,(variable−use '$constant−var '$result−var))))
     ,(change−variable−use '$result−var '$constant−var '$constant−value))))
```

FIGURE 6. Simple, generic constant propagation optimization, applied to a type *t*.

This specification of constant propagation matches the specification in [42] fairly closely, even using a query which matches any program point that has all incoming paths matching an inner regular expression. The use of single-element trace patterns for AST matching (using nested path quantifications) makes the optimization much more direct to express; a previous version using single-trace pattern matching was much more complicated.

The other part of the constant propagation analysis is the concept and model definitions. These are straightforward, and are shown in Figure 7. The models make heavy use of AST pattern matching; the generalized constant propagation optimization uses **all−paths** queries so that the extra prefixes included in AST matching regular expressions are not a problem. Also note the use of **match−ast−pattern−existential** in the definition of *modification−of−within*: for this part of the analysis, it is necessary to prevent all possible modifications of the given variable. The AST modification code is omitted from the example as it is long; it is a sequence of AST operations, represented in a Forth-like language, to replace the call to *use()* in the input program with a call to *useInteger()*.

The **Pavilion** language is capable of directly expressing the generic version of the simple constant propagation optimization. Breaking the optimization into two parts (the generic optimization

```
(concept (assignable t))

(model (assignable (class "A"))
       ('modification-of-within
        (λ (variable-var)
         '(or ,(match-ast-pattern-existential
                '(Let $_ (NamedFunctionCall "assign(A&, int)"
                                            (ExprList ((V ,variable-var) $_)))))
              ,(match-ast-pattern-existential
                '(Let $_ (NamedFunctionCall "copy(A&, A const&)"
                                            (ExprList ((V ,variable-var) $_))))))))
       ('variable-use
        (λ (variable-var result-var)
         (match-ast-pattern-universal
          '(Let ,result-var (NamedFunctionCall "use(A const&, char const*)"
                                               (ExprList ((V ,variable-var) $_))))))))

(concept (constant-propagable t) (assignable t))

(model (constant-propagable (class "A"))
       ('assign-constant
        (λ (variable-var constant-var)
         (match-ast-pattern-universal
          '(Let $_ (NamedFunctionCall "assign(A&, int)"
                                      (ExprList ((V ,variable-var) (Int ,constant-var))))))))
       ('change-variable-use-to-constant
        (λ (result-var constant-var constant-value)
         '(insert (mutate ...)))))
```

FIGURE 7. Concept and model definitions for simple constant propagation on the type *A*. The AST modification is omitted as it is long and based on ROSE details that are not important to the optimization.

and the models for particular types) simplified it greatly, and the models can take advantage of tree pattern matching to condense their code. The nested path query mechanism is important to the correctness of this optimization, and the ability to implement optimizations as transducers allows a restriction on when the path queries are applied (in this case, only immediately after a use of the variable). Being able to have optimizations that are composed from separately generated fragments, and to have Scheme functions that return regular expressions, are important to the modularity and

compositionality of the constant propagation optimization. Thus, the **Pavilion** system is effective at representing this optimization.

7.3. Algebraic rewriting

A useful optimization for many types of program is to remove unnecessary computations using algebraic laws. For example, some operations, such as integer addition, have neutral elements; that is, they have an element e which satisfies the equations $e \odot x = x$ and $x \odot e = x$ for all elements x. Thus, the program can be optimized by removing operations involving the neutral element. Similar optimizations have been done in the context of generic programming by the Simplicissimus system [125], which allows concepts to provide rewrite rules that are applied to the program AST.

The **Pavilion** system can also express algebraic simplification rules, both on particular types and generically, with some limitations. The AST pattern matching regular expression generator and nested path quantifications can directly express these simplifications. An example is the generic optimization for removing neutral elements on the left side of an operation shown in Figure 8.

```
(define (optimize-neutral model-tag)
 (if (models? (monoid model-tag))
    (let ((op ((monoid model-tag) 'op))
          (neutral ((monoid model-tag) 'neutral)))
      (wrap-partial-trace
       `(concat
        (node
         (all-paths backwards ; Require that all definitions of $result-var be of the given form
                    ,(match-ast-pattern `(Let $result-var (,op ,neutral $b)))))
        ; Apply the optimizing transformation
        (insert (mutate $result-var $b replaceExpr)))))
    #f))
```

FIGURE 8. A generic optimization using the neutral element of a monoid.

The **match-ast-pattern** function must be generalized to support matching generic operations rather than just concrete operator or function invocations. The only change that is required is that the pattern matching code generator, rather than allowing only concrete operations as the body of a *Let* block, must also allow Scheme functions. These functions, such as the *op* and *neutral* functions shown in the example, are given the pattern variable names for the result and operands of their local

part of the expression tree; the functions then generate corresponding analyses, often using calls to the concrete expression tree matching mechanism. As an example of this, the concept and model declarations for Monoid on integers are shown in Figure 9.

```
(concept (monoid t))

(model (monoid int-plus)
  ('op (λ (result-var a-var b-var)
         (match-ast-pattern-universal '(Let ,result-var (Add ,a-var ,b-var)))))
  ('neutral (λ (result-var)
              (match-ast-pattern-universal '(Let ,result-var (Int 0))))))
```

FIGURE 9. The concept declaration for the Monoid concept, and a model of that concept for integers.

In this example, rather than using types as models of the concept, separate model tags are used. This modification allows a single type to model the same concept in several ways; for example, integers are the element type of several monoids, including those for addition, multiplication, as well as bitwise AND, OR, and XOR. An explicit list of those models desired must be created as part of applying the optimization; the generic optimization is then instantiated for each of those models.

The basic implementation of expression tree rewriting has several limitations. The ability to track values of expressions through assignments to different variables, rather than just matching the explicit syntax of expression trees, requires a more sophisticated data flow analysis which is explained in Section 7.4. Tree equality, which is required for idempotency properties such as $x \,\&\, x \rightarrow x$, is difficult to support because of the splitting of expression trees to produce traces. The implementation of tree equality would also use recursive path quantifiers and would walk through corresponding parts of the two expression trees, ensuring that the variables used have not been modified between the two trees.

7.4. Tracking values through copies

The simple constant propagation optimization in the previous section is useful, but the functionality of the optimization leaves much to be desired. In particular, code such as *a = 5; b = a; c = b + 1;* cannot be optimized using its limited level of analysis: a constant can only

be found when it is used in the same variable in which it was originally stored. A better analysis would remove this restriction, allowing the tracking of a constant through several variables. The analysis is still generic; extra concepts (or added associated program analyses in existing concepts) are required to define which operations on a particular type implement the generic operations of copying a value or assigning it to a variable.

The difficulty of representing this optimization is due to the possibility of an unbounded number of copies of the constant being made; normally, a regular expression in the **Pavilion** language only contains a fixed number of variables, and each variable can only be bound once on any given program path. Several implementations of this analysis could have been expressed using various extensions to the core **Pavilion** language, but none could be directly expressed without either language modifications or extensive use of native function calls. The approach of recursive path quantification was chosen, despite it requiring major changes to the **Pavilion** implementation, because it allowed much greater flexibility in the language: this change enables the analysis of constants propagated through copied variables, but also a large subset of the many other analyses that can be expressed in a flow-equation-based program analysis framework.

The core idea of the language extension required is to allow properties of program points, rather than just properties of single paths, to be recursive. Several mechanisms exist to allow regular expressions to represent unbounded paths, such as the Kleene star and the directly tail recursive regular expressions allowed at the outer layer of the **Pavilion** language. However, these mechanisms only allow single paths to have unbounded length, and in particular cannot express arbitrary numbers of distinct values for the same variable in partially overlapping parts of a path (variable quantifiers only apply to well-defined, properly nested portions of a regular expression). Other analyses, such as constant folding, cannot be expressed on single paths at all: a simple loop such as **for** (**int** *i = 0; f(); ++i);* has an unbounded number of distinct paths, each with a different (but constant) value of *i* reached at the end of the loop.

Flow equations can easily express this analysis using a small loss of analysis precision relative to working on each path separately; they fold together all possible constant values at each program point, rather than each path. Because the constant propagation (and folding) lattice is not distributive, this approach has an inaccuracy, but it can handle unbounded loops such as the one

shown above without any problems. The difference between the two approaches is that the constant folding analysis can distinguish the paths through the loop (because of their different values for *i*, whose value is tracked by the analysis), but they all traverse the same program points (with different values for *i*, and different numbers of iterations, but these are not used in the flow-equation-based analysis).

The generic optimization is shown in Figure 10. The concept and model definitions are not shown for this example; they are the same as for the previous version of constant propagation, with an extra member to locate the assignment of the value of one variable to another. The use of recursive regular expressions complicates the expression of this optimization, but it is still relatively straightforward. The **letrec** construct is used to allow recursion in the definition of *find−constant*. Within this syntax, a helper regular expression *re* stores the actual path query; this query represents (using the **or** operator) either assigning a constant to the variable **$constant−var** or calling the query recursively and then copying the variable assigned there into **$constant−var**. Note that the definition of *re* is outside the λ expression; thus, it is only computed once. The λ expression defines a function which is called on each program point; it runs the query given in *re* at that point, passing it an empty environment. The change in environment in the recursive call, as well as the use of the *rename−var−in−result* helper function, allows the pattern variable **$constant−var** to have different values at different program points. Finally, the main regular expression uses the *find−constant* function to locate a constant assignment to a variable, followed by a use of the variable; it then replaces the variable with the constant value, completing the optimization.

The main source of complexity in this regular expression is the need to define a recursive regular expression; this necessitates the use of **letrec**, as well as wrapping a function around the inner regular expression. Also, because a single variable is used to represent the program variable containing the constant, the environment must be cleared in the nested regular expression call, and the result of that call must have one of its output variables renamed. In all, recursive path quantifications require more sophisticated use of Scheme in the regular expressions than non-recursive quantifications. A user could still learn to create such regular expressions, however. It might be possible to define Scheme macros to simplify these operations.

```
(let* ((variable-use ((assignable t) 'variable-use))
       (modification-of-within ((assignable t) 'modification-of-within))
       (copy-variable ((assignable t) 'copy-variable))
       (assign-constant ((constant-propagable t) 'assign-constant))
       (change-variable-use ((constant-propagable t) 'change-variable-use-to-constant)))
 (letrec ((find-constant ; A helper to find the values of all constant program variables
           (let ((re
                  `(all-paths backwards
                    (concat
                     (sym StartOfTrace)
                     #t
                     (or
                      ; Direct assignment of a constant
                      ,(assign-constant '$constant-var '$constant-value)
                      ; A copy of a previously defined constant
                      (exists ($constant-var-2)
                       (concat
                        (node (call ,(λ (npda-src env)
                                        ; Rename this variable to avoid a name conflict
                                       (rename-var-in-result
                                        ; Recursively call this analysis before the variable is assigned
                                        (find-constant npda-src)
                                        '$constant-var '$constant-var-2))))
                        ; Find a copy of the previous variable into the current one
                        ,(copy-variable '$constant-var-2 '$constant-var))))
                     ; Ensure that the variable is not reassigned
                     ,(does-not-contain (modification-of-within '$constant-var))))))
             (λ (npda-src)
               ; Call the defined regular expression, using an empty environment of pattern variables
               (interp-expr-step re npda-src '())))))
  (wrap-partial-trace
   `(concat
     (node (call ,(λ (npda-src env) (find-constant npda-src))))
     ; Find and change a use of the constant variable
     ,(variable-use '$constant-var '$result-var)
     ,(change-variable-use '$result-var '$constant-var '$constant-value)))))
```

FIGURE 10. Generic constant propagation optimization, including value tracking through variable assignments, applied to a type *t*.

The implementation of expression tree matching in the **Pavilion** system can also be extended to track expressions through copies, using a similar mechanism. The approach is exactly the same

as in the original version, except that, instead of requiring that the result of one computation be used as the argument to another, any value which can be found to be a copy of the original result can be used. For example, in *x = a + b; y = x; c = y + x;*, the three expressions *a + b*, *x*, and *y* are all considered to be computations of *a + b*. The procedure used for this extension is similar to the procedure for constant propagation through copies. A concept is defined of "the same expression;" for this analysis, two expressions a and b are considered to be the same if a is assigned to a variable v, b is a read of v, and v is not reassigned until the definition of b. With this definition, the expression a computes $b \oplus c$ when either a is an application of $\oplus$ to expressions b' and c' which compute b and c, respectively; or when a is the same as some expression a' which itself computes $b \oplus c$ (recursively). The implementation of this extended definition of computation is more complicated than the previous definition, because of the added recursion; the final definition uses recursively nested path quantifiers and native functions extensively.

7.5. Integer parity analysis

An example of a program analysis which essentially requires a fixpoint-based algorithm is the analysis of the parity of integers (determining whether they are even or odd) in a program. In the simplified version of the analysis shown here, each integer constant is given a parity, and the parities of sums of integers are computed. The parity of each integer variable is also kept, and uses of the variable preserve the assigned parity. The analysis is sophisticated enough to correctly analyze loops which have induction variables which keep a single parity throughout all loop iterations. To create a simple example, the transformation just replaces the left side of every operation *x & 1* with the parity of x when that is known.

The analysis used is shown in Figure 11. It is divided into two parts, each a function which takes a program point as its argument. The first function, *parity−of−exprs*, returns a map from expressions (the **$e** variable) to possible parities (the **$parity** variable). The map is represented as a relation, so an expression could theoretically have multiple parity values; however, this cannot occur in practice because the analysis is optimistic and uses an all-incoming-paths query. A variable can only have both parities coming out of a computation when it had them entering the computation, and each constant can only have one parity. The results from separate paths into a program

```
(define do-assignment (match-ast-pattern-universal '(Let $_ (Assign (V $var) $e))))
(define any-assignment (match-ast-pattern-existential '(Let $_ (Assign (V $var) $_))))
(letrec ((parity-of-vars ; Find the parity of each variable for which it is uniquely defined
          (λ (npda-src env)
            (interp-expr-step ; Evaluate the given path query, using an empty environment
             '(all-paths backwards
                         (concat
                          (sym (StartOfTrace))
                          #t
                          (exists ($e) ; Forget the value of $e after the assignment is found
                            (concat (node (call ,parity-of-exprs)) ; Get parity of each expression
                                    ,do-assignment))
                          ; Ensure that the variable is not reassigned
                          ,(does-not-contain any-assignment)))
             npda-src '()))))
        ; A helper function to add parity information for integer constants (not shown)
        (add-parity-for-constant ...)
        (parity-of-exprs-add (λ (npda-src env)
          (let* ((pe (parity-of-exprs npda-src env))
                 (pe1 (rename-var-in-result
                       (rename-var-in-result pe '$parity '$parity1) '$e '$e1))
                 (pe2 (rename-var-in-result
                       (rename-var-in-result pe '$parity '$parity2) '$e '$e2)))
            ; Combine the input parities using the exclusive OR operation (not shown)
            ...)))
        (parity-of-exprs (λ (npda-src env)
          (interp-expr-step
           '(all-paths backwards
                       (concat
                        (sym (StartOfTrace))
                        #t
                        (or (exists ($val)
                              (call ,add-parity-for-constant (sym (Let $e (Int $val)))))
                            (exists ($var) (concat (node (call ,parity-of-vars))
                                                   (sym (Let $e (V $var)))))
                            (exists ($e1 $e2 $parity1 $parity2)
                              (concat (node (call ,parity-of-exprs-add))
                                      (sym (Let $e (Add $e1 $e2))))))
                        (* (sym (not (ForgetTemporary $e))))))
           npda-src '()))))
```

FIGURE 11. Essential functions in an analysis of the parity of integer values in a program.

point are intersected, and so multiple parities for one expression can never be produced. The second function, *parity−of−vars*, returns a map from variable names (the **$var**) variable) to possible parities (the **$parity** variable). There is a mutual recursion between the functions: *parity−of−exprs* calls both itself (for additions) and *parity−of−vars* (for variable references), and *parity−of−vars* calls *parity−of−exprs* to determine the parity of the right-hand side of each variable assignment.

The main difficulty in writing the parity analysis, and the main complexity in it, is in creating the native functions that rename variables from recursive calls, and the functions that combine the results of the recursions together. They use details about how pattern variables are represented internally. Also, in order to ensure the comparability of the resulting states, helper functions for combining input parity values to an operation must be defined outside the bodies of the *parity−of−exprs* and *parity−of−vars* functions. Other than these details, the parity analysis code directly represents the required abstract operations (parity of constants, preservation through variable assignments, and the computation of output parity from additions). The difficulties stem from the details of the implementation, and language limitations in the placement of native function calls.

7.6. Evaluation

These examples show that the **Pavilion** system can represent a wide range of program analyses and optimizations, both generic and non-generic. An analysis for software verification, as well as several optimizations, were presented. The examples shown here are for simplified versions of the analysis problems, and specifications for the full problem versions — for example, parity analysis with the full set of integer operators in C++, including compound operators such as `+=` — would be much more complicated. However, these extra cases are just additions to the existing list, and are not fundamentally more complex than the simplified specifications. The use of regular expressions, when combined with the extensions shown in this thesis, is beneficial in making specifications match the user's normal view of how their programs operate; flow equations, however, can also be used when they are more natural.

One interesting lesson from designing and implementing the **Pavilion** system is that regular expressions are not enough to express many program analyses directly. Several past systems used only basic regular expressions with variables, with a single level of path quantification; this is adequate

for simple optimizations such as constant propagation, but makes expression tree matching (for example) much more difficult to express. Compositionality is limited in this approach as well, as regular expressions must be combined in complicated ways to allow expression trees to be matched based on matching their components; having the operators in the expression tree also be arbitrary regular expressions (as is required for generic analyses) causes even more problems.

The examples show that the extensions to regular expressions provided by the **Pavilion** language are used in practical analyses. Intersection is used to express the concept of a number of actions happening in an arbitrary order; complementation is used to express an event not happening in a particular part of the trace. Variable quantification is used to limit the scopes of pattern variables to where they are used. Path quantification, both universal and existential, is used to match expression tree patterns; recursive path quantifications allow more sophisticated pattern matching. Generating regular expressions from Scheme code, and calling Scheme code within regular expressions, greatly aids the reusability of specifications, and allows recursive regular expressions to be created. The only features which have not been used are forward path quantification, which is used in other program analyses than are featured here, and universal variable quantification, which is included for completeness as it is dual to existential quantification. Non-Boolean analysis results (other than pattern variable bindings) are also not used, but they are a reasonable generalization of Boolean analysis results.

Analyses such as tracking variable values through several copies would not be possible to implement in systems that use only basic regular expressions plus pattern variables. Flow equations, on the other hand, can express these more complicated analyses, and can express every analysis possible with regular expressions; however, they cannot express these analyses easily. Also, users are unlikely to understand how to write efficient flow equations for a given analysis; one example of this problem is shown in [120], which discusses a standard analysis being strengthened by adding more precise, yet correct, rules; these rules are not part of standard specifications of the analysis. The **Pavilion** system can use recursive path quantification, in combination with regular expressions, to express many kinds of flow equations in a more direct manner. Added features, such as arbitrary lattices and widening to allow infinite-height lattices, are necessary to enable the full range of standard program analyses and optimizations to be specified.

The use of Scheme to express analyses and optimizations in the **Pavilion** system has both advantages and disadvantages. The main advantage is that arbitrary code can be embedded into regular expressions, and arbitrary code can be used to generate them. This feature allowed generic optimizations to be composed from sub-optimizations for the individual types and concepts involved. It also allowed a subroutine to be defined for generating regular expressions to match expression trees. A disadvantage of using Scheme S-expressions for program analyses is that they are fairly verbose. A more customized syntax would reduce this problem, but would also reduce the flexibility of being able to easily embed arbitrary Scheme code into regular expressions.

One design decision which has greatly affected the implementation of the **Pavilion** language, and its expressiveness, is the model used for variables. The language includes single-assignment variables, as is used in the Stratego language [154]; each variable is either completely unbound or completely bound (ground). Logic programming languages, on the other hand, typically allow variables to be bound to terms which themselves contain variables; such terms are not fully defined, but later unifications can attach ground terms to the variables. In either of these models, variables are not explicitly assigned; each use of a variable v in the program can either bind v to a value or require that a value is equal to v's current binding. This model has major effects on implementation. The implementation splits execution paths based on the bindings of variables, confident that the variable will never change. Also, the values of variables are stored as constraints — in the current implementation, just equality and disequality with constants — which effectively map variable bindings to nested regular expressions. A model with explicit reassignment of variables, on the other hand, would require that the values be explicitly stored, such as in a nondeterministic finite automaton model (Section 4.3 on page 47). Explicit variable assignment and reassignment would be useful to implement some forms of value tracking: the current location where a particular value is stored could be in a pattern variable, which is then modified when a copy occurs (using a nondeterministic choice as to whether to change the pattern variable or not). However, flow equations and recursive path quantifiers allow this behavior to be expressed in a more natural way. Also, user-defined procedures can remove the current variable bindings during a nested query, allowing recursive queries to have different values for the same variable without needing to reassign it along a single path.

Another language feature which can be evaluated is the placement of analysis and optimization specifications. Having them located in a separate file allowed for the direct use of Scheme, including a third-party Scheme interpreter and parser. It also allowed Scheme syntax to be used. However, a syntax which allows specifications to be integrated into the input C++ code would likely be more convenient for users. The specifications would need to be able to come from header files so that they can be reused for separate files and applications. A more concise syntax could also be used, making specifications easier to write. Lastly, integration into ConceptC++ would allow concept-based optimizations to be included with the concepts themselves. These issues are discussed in more detail in Section 3.11 on page 40.

This section is intended to show the analyses and optimizations which are at the limits of the **Pavilion** system's capabilities, and those which showcase its distinctive features. Many simpler analyses, such as those done by previous regular-expression-based systems for program analysis, are also representable. For example, simple bit-vector data flow analyses such as live variable analysis and non-generic forms of constant propagation are easy to express. Previous systems using regular expressions for program analysis typically cannot propagate a constant value (for example) through several variables; providing that feature in the **Pavilion** system is an improvement over those earlier systems. Overall, the **Pavilion** system can represent most standard compiler optimizations. The problems relate to the restriction to power-set lattices (and the consequent lack of widening), the restriction to compile-time analysis and optimization, and the restriction to properties which can be expressed using traces. The first restriction prevents analyses such as value range analysis, which uses the infinite-height lattice of intervals of integers, and requires widening to terminate on many loops. Widening is difficult to implement because it requires the detection of loops in the program, which, although possible, is not part of the "pure" regular expression framework; it is possible, however, to include the program points reached into the trace, allowing this analysis to be expressed using regular expressions and path quantifiers. The **Pavilion** system also does not include any features for integrating run-time information into its analyses, although they could be added in the future. Lastly, the information available to the **Pavilion** system, and thus analyses and optimizations written using it, is restricted to the program trace and how that trace relates to the

program's abstract syntax tree; other information may be added to the trace, however, to remove this limitation.

CHAPTER 8

Conclusion

This thesis presents the **Pavilion** system for specifying compiler analyses and optimizations declaratively, with a focus on optimization of software using abstractions. Regular expressions, with many extensions, are used to express the analyses; although these make some analyses much easier to specify, they have difficulty with certain types of analysis. Regular expressions, with the extensions described in this thesis, are compositional: a regular expression can be built from smaller, independent pieces. This capability allows an optimization for a particular type to be created based on regular expression fragments that are provided by models of various concepts, making the creation of generic optimizations straightforward. In fact, generic optimizations and analyses are exactly analogous to generic algorithms: as an algorithm works by composing functions provided by its input types, an optimization is generated by composing regular expression fragments provided by its input types.

As extensions to the basic regular expression paradigm are required for sophisticated program analysis, the tradeoffs between possible extensions and how they affect expressiveness and implementability become important. Although the **Pavilion** language does not include every possible feature, it includes a reasonable set; and an implementation approach has been defined that supports the features chosen.

The **Pavilion** implementation was used to write several program analyses and optimizations, both based on abstractions provided by functions and those provided by concepts. These were then applied to small test programs, and were found to function correctly. For the most part, the **Pavilion** specifications of the optimizations fairly directly expressed their essential content; a user should be able to write optimizations in the language in a straightforward manner. Some usability features remain to be improved, however, such as the language syntax; also, precise analyses (those applying to only a single trace) are more difficult to write than those that are more approximate (applying to

several traces using path quantification). The overall language is simple to use, and provides a set of features which are likely to be familiar to ordinary library authors.

Future work

One avenue for future work is to define more analyses and optimizations for specific domains using the **Pavilion** language. For example, optimizations could be written specifically for graphs using the concepts defined in the Boost Graph Library [131]. Also, implementing all of the analyses provided by STLlint [60, 61] would be useful. Encouraging library authors to use and evaluate the **Pavilion** language would also be beneficial to determine whether it (and concept-based optimization in general) increases library author productivity and application performance in practice.

The **Pavilion** system is applicable to the problem of testing programs for security flaws and other correctness issues. Security properties are already often verified through model checking [26], and thus violations are given as trace patterns. Regular-expression-based approaches to program analysis have been used for security analysis in the past [101], and one analysis example has been implemented as an example in Section 7.1 on page 84. Implementing a broader range of security analyses using the **Pavilion** language would also be a research opportunity, as well as a way to have more impact on commonly-used applications.

Several previous systems for trace-based program analysis have been applied to concurrent programs, using several different models of parallelism. The **Pavilion** system could also be extended to support analysis and transformation of these programs, increasing its generality. This task would likely be straightforward; it would mainly involve more sophisticated ways of generating traces from input programs and interleaving traces together. However, context-sensitive analysis of fully general parallel programs is undecidable [114]; approximations have been used in the past to achieve decidability [18, 23, 80]. Also, handling analysis of programs over all possible process counts at once, as is done in [34] would be an interesting problem.

A production-quality implementation of the **Pavilion** system would also be beneficial, especially for applying the system to large applications. In particular, the implementation would need to be much more concerned with efficiency of analysis, including reusing computations as much as possible. Also, analyses and optimizations should be compiled into a lower-level language such as

C++, despite the complexity of doing so. Increasing efficiency might also involve using a different automaton representation as the target of the regular expression compiler; supporting as many language features as possible within such a representation would still be an important concern, however. Additionally, new language features might be developed, and the implementation updated to include them.

The **Pavilion** language is applicable to analyzing and transforming programs written in several languages. The analyses themselves may or may not transfer to other source languages; that depends on the set of terminals used in the push-down automaton generator for the input language. Defining a common set of terminals to cover as many languages as possible would aid usability. In addition to source code, the **Pavilion** system could also be generalized to process object code. Traces in the automaton correspond more closely to sequences of instructions than they do to source code, so analyses might be able to be written more directly. A potential problem, however, is that many instructions can perform the same task, and analyses must be general enough to cover all possible sequences.

As stated in Section 2.5.1 on page 14, many current program analysis specification languages use equations over the program data flow to specify analysis behavior, and many standard analyses are specified in this form. The **Pavilion** system is able to express some flow equations using nested path quantification (Section 3.10 on page 37), but is limited to equations that return Boolean values or sets of elements from the program. The ability to define more lattices should be provided, as well as a more direct syntax for writing flow equations. Additionally, the widening technique to allow program analyses to operate on lattices with infinite height should be provided, as it is essential to implement, for example, value range analysis [67].

On a similar note, program flow automata are produced based on the Steffen model of the program, in which the control flow graph is the only information included [138]. This level of analysis neglects the values of variables, for example. More sophisticated models could also be used; other abstract interpreters can be used as the basis of an automaton-based analysis approach [121]. However, adding this feature would require a language to specify abstract interpreters, which would require the equivalent of a full program analysis specification language such as that in [4].

In order to use the technique described in Section 4.4.4 on page 53, a set of functions must be found that is useful for analyses, is closed under operations such as $\wedge$ and $\vee$, and yet has an efficiently computable equality operation. Trying to find as general of a class of functions as possible satisfying these requirements would allow analysis power to be increased. Tree automata might be a good source for such classes of functions [33].

Integrating separate analyses in a common framework would also be an interesting area of future work. For example, aliasing information from one analysis might feed information into a constant propagation analysis, even if the analyses are written independently. In particular, different concepts may provide different information about uses of types that model them, and a program may use different objects that model several concepts; the information derived from those concepts should be combined. Ideally, superanalysis [31] or a similar technique for combining independent analyses [96] should be used so that analyses can integrate their information without imposing any particular direction of flow between them. The factoring of analyses as functions and the ability for them to call each other may provide much of this facility, as could the ability to "plug together" optimizations from simpler pieces.

More sophisticated algorithm recognition would allow optimizations written for generic programs to be applied to non-generic implementations as well. Algorithm recognition refers to a set of techniques for determining whether a particular section of a program implements a particular operation. This feature would allow operations in non-generic code to be matched to the same operation when it occurs in a model of a concept. For example, the type **double** with the operations *min* and + forms an idempotent semiring. Algorithm recognition would allow a concept-based optimization for all such semirings (in this case, the rewrite $a \oplus (a \oplus b) \rightarrow a \oplus b$) to apply to code such as **double** *d = a < c ? a : c; f(a < d ? a : d);*, optimizing it to *f(a < c ? a : c);*. Without recognition of this construct, it would be necessary for the input program to explicitly call a *min* function to enable the optimization.

Several methods of improving analysis efficiency have been developed in previous work, and integrating them into the **Pavilion** system would be worthwhile. Demand-driven analysis allows only those properties of a program relevant to optimizations to be computed. Incremental analysis reduces the amount of program analysis that must be re-done when the program is transformed by

an optimization. Both of these techniques are intended to improve analysis performance, and both are common in previous optimization specification frameworks; combining them with the more sophisticated language provided by **Pavilion** would make that system more useful.

Another way to improve the performance of program analysis is to reduce the precision of the analysis, usually in combination with a strategy to improve the precision when necessary. Several approaches for this purpose have been explored in past work. One approach, used in model checking, is counterexample-guided abstraction refinement (CEGAR) [30]. This approach involves using a rough abstraction of the program at first, and checking to determine if it violates a safety property. If it does, the violating program trace is examined; if it is an artifact of imperfect abstraction, a more precise abstraction of the program, computed specifically to exclude that trace, is used for further iterations of the analysis. Other techniques for controlling analysis precision dynamically would also be interesting to explore.

Running program analyses, and possibly transformations, on parallel computers would also be an effective means to improve analysis performance. An approach in which program analyses are compiled into a lower-level representation provides a good opportunity to parallelize them. Analysis quality might also be improved by allowing more precise analyses to be practical for larger software. Little previous work has been done in this area, and so it is likely a fertile opportunity for new work.

Another avenue of future work is to provide mechanisms to prove that analyses and optimizations written in the **Pavilion** language are correct. Such work has been done for other specification languages by Craig Chambers and his group [97, 98, 120]. Automatic correctness proofs would allow users to ensure that their optimizations preserve program correctness, and would thus make users more confident in using the **Pavilion** system on production software.

It might be possible to apply or extend the **Pavilion** language to specify optimizations that occur partly or wholly during program execution. Such staged optimizations have been proposed by Philipose et al [111]. Analyses have also been implemented using a mix of both static and dynamic techniques, such as in the Program Query Language [104] and trace-based aspects in the AspectBench Compiler [3]. Implementing regular expressions efficiently enough to use at run-time might be difficult, however.

Overall, a specification language for optimizations on generic software components was useful, and regular expressions form a reasonable basic model for specifying such optimizations and their corresponding analyses. However, many features needed to be added to the basic regular expression paradigm to achieve this goal. Many avenues exist for future research, but the basic model should be extensible to cover most of the improvements necessary to enable a broader range of analyses and optimizations to be expressed.

Bibliography

[1] Accellera. *Property Specification Language Reference Manual*, June 2004. Version 1.1. `http://www.eda.org/vfv/docs/PSL-v1.1.pdf`.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] C. Allan et al. Adding trace matching with free variables to AspectJ. In *OOPSLA*, pages 345–364, New York, NY, USA, 2005. ACM Press.

[4] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *Symposium on Static Analysis*, pages 33–50, London, UK, 1995. Springer-Verlag.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, editors. *LAPACK User's Guide*. SIAM, 3rd edition, 1999.

[6] U. Assmann. Graph rewrite systems for program optimization. *ACM Transactions on Programming Languages and Systems*, 22(4):583–637, 2000.

[7] M. H. Austern. *Generic programming and the STL: Using and extending the C++ Standard Template Library*. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[8] O. S. Bagge and M. Haveraaen. Domain-specific optimisation with user-defined rules in CodeBoost. In J.-L. Giavitto and P.-E. Moreau, editors, *Workshop on Rule-Based Programming*, volume 86/2 of *Electronic Notes in Theoretical Computer Science*, Valencia, Spain, 2003. Elsevier.

[9] O. S. Bagge, M. Haveraaen, and E. Visser. CodeBoost — a framework for transforming C++ programs. Technical Report UU-CS-2001-32, Institute of Information and Computing Sciences, Utrecht University, 2001.

[10] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 65–74. IEEE Computer Society Press, 2003.

[11] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *SIGOPS Operating Systems Review*, 40(4):73–85, 2006.

[12] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation*, pages 203–213, New York, NY, USA, 2001. ACM Press.

[13] T. Ball and S. K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 97–103, New York, NY, USA, 2001. ACM Press.

[14] S. Ben-David, D. Fisman, and S. Ruah. Embedding finite automata within regular expressions. In *International Symposium on Leveraging Applications of Formal Methods*, Nov. 2004. http://www.wisdom.weizmann.ac.il/~dana/publicat/Embedding_paper.pdf.

[15] W. C. Benton and C. N. Fischer. Interactive, scalable, declarative program analysis: from prototype to implementation. In *Principles and Practice of Declarative Programming*, pages 13–24, New York, NY, USA, 2007. ACM Press.

[16] D. Beyer, A. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *Static Analysis Symposium*, number 3148 in LNCS. Springer-Verlag, 2004.

[17] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory*, pages 135–150, 1997.

[18] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, pages 62–73, New York, NY, USA, 2003. ACM Press.

[19] J. M. Boyle, T. J. Harmer, and V. L. Winter. The TAMPR program transformation system: simplifying the development of numerical software. In *Modern Software Tools for Scientific Computing*, pages 353–372. Birkhäuser, Cambridge, MA, USA, 1997.

[20] M. Bravenboer and E. Visser. Rewriting strategies for instruction selection. In S. Tison, editor, *Rewriting Techniques and Applications*, volume 2378 of *LNCS*, pages 237–251, Copenhagen, Denmark, July 2002. Springer-Verlag.

[21] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[22] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag New York, New York, NY, USA, 1990.

[23] S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 334–349. Springer, Mar. 2006.

[24] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[25] A. Chauhan and K. Kennedy. Optimizing strategies for telescoping languages: procedure strength reduction and procedure vectorization. In *International Conference on Supercomputing*, pages 92–101, New York, NY, USA, 2001. ACM Press.

[26] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security*, pages 235–244, New York, NY, USA, 2002. ACM Press.

[27] H. Chen, D. Wagner, D. Schultz, and G. Morrison. *MOPS User's Manual*. Computer Science Division, UC Berkeley, Nov. 2004. http://wwwcsif.cs.ucdavis.edu/~wujt/manual.pdf.

[28] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *Programming Language Design and Implementation*, pages 85–95, New York, NY, USA, 2005. ACM Press.

[29] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[30] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, London, UK, 2000. Springer-Verlag.

[31] C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17(2):181–196, 1995.

[32] C. Colby and P. Lee. Trace-based program analysis. In *POPL*, pages 195–207, New York, NY, USA, 1996. ACM Press.

[33] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 1997. Released October 1, 2002.

[34] B. Cook, D. Kroening, and N. Sharygina. Over-approximating Boolean programs with unbounded thread creation. In *Formal Methods in Computer Aided Design*, 2006.

[35] K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *International Conference on Computer Languages*, 1992.

[36] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pøasøareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, New York, NY, USA, 2000. ACM Press.

[37] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Workshop on SPIN Model Checking and Software Verification*, pages 205–223, London, UK, 2000. Springer-Verlag.

[38] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd International Symposium on Programming*, pages 106–130, April 1976. `http://www.di.ens.fr/~cousot/COUSOTpapers/ISOP76.shtml`.

[39] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977.

[40] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems — a case study. In *Programming Language Design and Implementation*, pages 117–126, New York, NY, USA, 1996. ACM Press.

[41] C. de Dinechin. XLR: Extensible language and runtime, nov 2007. `http://xlr.sourceforge.net/`.

[42] O. de Moor, S. Drape, D. Lacey, and G. Sittampalam. Incremental program analysis via language factors, 2002. Unpublished.

[43] O. de Moor, D. Lacey, and E. V. Wyk. Universal regular path queries. *Higher Order Symbol. Comput.*, 16(1-2):15–35, 2003.

[44] S. Debois. Imperative program optimization by partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 113–122, New York, NY, USA, 2004. ACM Press.

[45] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[46] S. Drape, O. de Moor, and G. Sittampalam. Transforming the .NET intermediate language using path logic programming. In *Principles and Practice of Declarative Programming*, pages 133–144, New York, NY, USA, 2002. ACM Press.

[47] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. Softw. Eng. Methodol.*, 13(4):359–430, 2004.

[48] E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[49] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, Oct. 2000.

[50] D. R. Engler. Interface compilation: Steps toward compiling program interfaces as languages. *IEEE Transactions on Software Engineering*, 25(3):387–400, 1999.

[51] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification*, pages 232–247, London, UK, 2000. Springer-Verlag.

[52] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Computer Aided Verification*, pages 324–336, London, UK, 2001. Springer-Verlag.

[53] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Compiler Construction*, pages 85–98, New York, NY, USA, 1986. ACM Press.

[54] M. Flatt. PLT MzScheme: Language manual. Technical Report PLT-TR2006-1-v370, PLT Scheme Inc., 2006. `http://www.plt-scheme.org/techreports/`.

[55] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 2002. ACM Press.

[56] R. Garcia, J. Järvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Proceedings of the 2003 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'03)*, Oct. 2003.

[57] D. Gregor, J. Järvi, M. Kulkarni, A. Lumsdaine, D. Musser, and S. Schupp. Generic programming and high-performance libraries. *International Journal of Parallel Programming*, 33(2), June 2005.

[58] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *OOPSLA*, pages 291–310. ACM Press, October 2006.

[59] D. Gregor and A. Lumsdaine. ConceptGCC. Web page, Apr. 2007. http://www.generic-programming.org/software/ConceptGCC/.

[60] D. Gregor and S. Schupp. Making the usage of STL safe. In J. Gibbons and J. Jeuring, editors, *Proceedings of the IFIP TC2 Working Conference on Generic Programming*, pages 127–140, Boston, 2002. Kluwer.

[61] D. Gregor and S. Schupp. STLlint: Lifting static checking from languages to libraries. *Software: Practice & Experience*, mar 2006.

[62] D. Gregor and B. Stroustrup. Proposed wording for concepts. Technical Report N2193=07-0053, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, March 2007.

[63] E. Gurari. *An Introduction to the Theory of Computation*. Computer Science Press, 1989. Online edition at http://www.cse.ohio-state.edu/~gurari/theory-bk/theory-bk.html.

[64] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Domain-Specific Languages*, pages 39–52, 1999.

[65] S. Z. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 2004.

[66] J. Härdtlein, A. Linke, and C. Pflaum. Fast expression templates. In *Programming Grids and Metacomputing Systems*, volume 3515 of *LNCS*, pages 1055–1063. Springer, 2005. http://dx.doi.org/10.1007/11428848_133.

[67] W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[68] D. Hildum and J. Cohen. A language for specifying program transformations. *IEEE Trans. Softw. Eng.*, 16(6):630–638, 1990.

[69] G. J. Holzmann. Logic verification of ANSI-C code with SPIN. In *Workshop on SPIN Model Checking and Software Verification*, pages 131–147, London, UK, 2000. Springer-Verlag.

[70] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[71] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[72] IBM Corporation. General purpose pragmas. Part of the IBM XL C/C++ compiler manual. http://publib.boulder.ibm.com/infocenter/pseries/v5r3/topic/com.ibm.vacpp7a.doc/compiler/ref/rupragen.htm.

[73] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. Geneva, Switzerland, Sept. 1998.

[74] International Organization for Standardization. *ISO/IEC 9899:1999: Programming languages — C*. Geneva, Switzerland, Dec. 1999.

[75] J. Järvi, J. Willcock, and A. Lumsdaine. Concept-controlled polymorphism. In F. Pfennig and Y. Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *LNCS*, pages 228–244. Springer Verlag, Sept. 2003.

[76] M. Jazayeri, R. Loos, D. Musser, and A. Stepanov. Generic Programming. In *Report of the Dagstuhl Seminar on Generic Programming*, Schloss Dagstuhl, Germany, Apr. 1998.

[77] S. C. Johnson. Lint, a C program checker. In *Unix Programmer's Manual*, volume 2. AT&T Bell Laboratories, 1978.

[78] S. C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[79] B. Joy, K. Kennedy, et al. Information technology research: Investing in our future. Report to the President, President's Information Technology Advisory Committee, Feb. 1999. `http://www.nitrd.gov/pitac/report/index.html`.

[80] V. Kahlon and A. Gupta. On the analysis of interacting pushdown systems. In *POPL*, pages 303–314, New York, NY, USA, 2007. ACM Press.

[81] R. M. Kaplan and M. Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, 1994.

[82] K. Karuri. A framework for automatic generation of code optimizers. Master's thesis, Indian Institute of Technology, Kanpur, May 2001.

[83] K. Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *International Parallel and Distributed Processing Symposium*, pages 297–304, May 2000.

[84] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(2):387–408, Feb. 2005.

[85] G. A. Kildall. A unified approach to global program optimization. In *Principles of Programming Languages*, pages 194–206, New York, NY, USA, 1973. ACM Press.

[86] J. Knoop, O. Rüthing, and B. Steffen. Towards a tool kit for the automatic generation of interprocedural data flow analyses. *Journal of Programming Languages*, 4(4):211–246, 1996.

[87] D. Koes, M. Budiu, and G. Venkataramani. Programmer specified pointer independence. In *Workshop on Memory System Performance*, pages 51–59, New York, NY, USA, 2004. ACM Press.

[88] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983. `http://www.sciencedirect.com/science/article/B6V1G-45TK9RG-T/2/91fe9ef25c099454365566642b762af4`.

[89] O. Kupferman and S. Zuhovitzky. An improved algorithm for the membership problem for extended regular expressions. In *International Symposium on Mathematical Foundations of Computer Science*, LNCS, pages 446–458, London, UK, 2002. Springer-Verlag.

[90] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In *Compiler Construction*, volume 2027 of *LNCS*, pages 52+, 2001.

[91] D. Lacey, N. D. Jones, E. V. Wyk, and C. C. Frederiksen. Compiler optimization correctness by temporal logic. *Higher Order and Symbolic Computation*, 17(3):173–206, 2004.

[92] A. Lal and T. Reps. Improving pushdown system model checking. In *Computer Aided Verification*, 2006.

[93] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Principles of Database Systems*, pages 1–12, New York, NY, USA, 2005. ACM Press.

[94] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Notices*, 39(3):46–55, 2004.

[95] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[96] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–282. ACM Press, 2002.

[97] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 220–231. ACM Press, 2003.

[98] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Principles of Programming Languages*, pages 364–377, New York, NY, USA, 2005. ACM Press.

[99] T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *Symposium on Static Analysis*, pages 280–301, London, UK, 2000. Springer-Verlag.

[100] Y. Li and W. Pedrycz. Regular expressions with truth values in lattice-monoid and their languages. In *Fuzzy Information Processing*, volume 2, pages 572–577. IEEE, 2004.

[101] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu. Parametric regular path queries. *SIGPLAN Not.*, 39(6):219–230, 2004.

[102] Y. A. Liu and S. D. Stoller. Querying complex graphs. In *Practical Aspects of Declarative Languages*, 2006.

[103] E. Marschner, B. Deadman, and G. Martin. IP reuse hardening via embedded Sugar assertions. In *IP Based SoC Design*, Oct. 2002.

[104] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*, pages 365–383, New York, NY, USA, 2005. ACM Press.

[105] B. McNamara and Y. Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming*, October 2000.

[106] F. Nielson, H. R. Nielson, and H. Seidl. Normalizable Horn clauses, strongly recognizable relations, and Spi. In *Symposium on Static Analysis*, pages 20–35, London, UK, 2002. Springer-Verlag.

[107] A. Okhotin. Boolean grammars. *Information and Computation*, 194:19–48, 2004.

[108] A. Okhotin. The dual of concatenation. In *Mathematical Foundations of Computer Science*, volume 3153 of *LNCS*, pages 698–710. Springer Berlin/Heidelberg, 2004. http://www.springerlink.com/content/t27g6rmyxl7jwykq.

[109] K. Olmos and E. Visser. Strategies for source-to-source constant propagation. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies*, volume 70 of *Electronic Notes in Theoretical Computer Science*, page 20, Copenhagen, Denmark, July 2002. Elsevier Science Publishers. http://www.elsevier.nl/locate/entcs/volume70.html.

[110] R. Paige. Viewing a program transformation system at work. In *Programming Language Implementation and Logic Programming*, pages 5–24, London, UK, 1994. Springer-Verlag.

[111] M. Philipose, C. Chambers, and S. J. Eggers. Towards automatic construction of staged compilers. In *Principles of Programming Languages*, pages 113–125, New York, NY, USA, 2002. ACM Press.

[112] D. Quinlan, M. Schordan, Q. Yi, and B. de Supinski. A C++ infrastructure for automatic introduction and translation of OpenMP directives. In M. Voss, editor, *Workshop on OpenMP Applications and Tools*, volume 2716 of *LNCS*. Springer, 2003.

[113] D. Quinlan, M. Schordan, Q. Yi, and B. R. de Supinski. Semantic-driven parallelization of loops operating on user-defined containers. In *16th Workshop on Languages and Compilers for Parallel Computing*, Oct. 2003. http://parasol.tamu.edu/lcpc03/informal-proceedings/Papers/26.pdf.

[114] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *TOPLAS*, 22(2):416–430, 2000.

[115] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Programming Language Design and Implementation*, pages 83–94, New York, NY, USA, 2002. ACM Press.

[116] E. Rice, S. Lerner, and C. Chambers. Automatically inferring sound dataflow functions from dataflow fact schemas. In *Compiler Optimization Meets Compiler Verification*, Apr. 2005.

[117] A. D. Robison. Impact of economics on compiler optimization. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 1–10. ACM Press, 2001.

[118] T. Rus and E. Van Wyk. Using model checking in a parallelizing compiler. *Parallel Processing Letters*, 8(4):459–471, 1998.

[119] D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Principles and Practice of Declarative Programming*, pages 117–128, New York, NY, USA, 2005. ACM Press.

[120] E. R. Scherpelz, S. Lerner, and C. Chambers. Automatic inference of optimizer flow functions from semantic meanings. In *PLDI*, pages 135–145, 2007.

[121] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL*, pages 38–48, New York, NY, USA, 1998. ACM Press.

[122] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, Aug. 2003.

[123] M. Schordan and D. Quinlan. Specifying transformation sequences as computation on program fragments with an abstract attribute grammar. In *Workshop on Source Code Analysis and Manipulation*, pages 97–106, 2005.

[124] S. Schupp, D. Gregor, D. R. Musser, and S.-M. Liu. Library transformations. In *First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), Florence, Italy*, pages 109–121. IEEE, November 2001.

[125] S. Schupp, D. Gregor, D. R. Musser, and S.-M. Liu. User-extensible simplification: Type-based optimizer generators. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 86–101, London, UK, 2001. Springer-Verlag.

[126] S. Schupp, D. Gregor, D. R. Musser, and S.-M. Liu. Semantic and behavioral library transformations. *Information and Software Technology*, 44(13):797–810, 2002.

[127] S. Schupp, D. Gregor, B. Osman, D. R. Musser, J. Siek, L.-Q. Lee, and A. Lumsdaine. Concept-based component libraries and optimizing compilers. Technical report, RPI Computer Science Department Technical Report 02-02, 2002.

[128] S. Schupp, D. P. Gregor, and D. R. Musser. Algebraic concepts represented in C++. Technical Report TR-00-8, Rensselaer Polytechnic Institute, 2000. `http://www.cs.chalmers.se/~schupp/old_projects/simpl/doc/AlgCpp.pdf`.

[129] K. Sen and G. Roşu. Generating optimal monitors for extended regular expressions. In *Run-time Verification*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 226–245, Oct. 2003. `http://www.sciencedirect.com/science/article/B75H1-4DDWKTJ-PT/2/1b3b804534530eb47c98408821182f90`.

[130] M. D. Sharp and S. W. Otto. A class specific optimizing compiler. In *Proceedings of the First Annual Object-Oriented Numerics Conference*, 1993.

[131] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[132] J. Siek, A. Lumsdaine, and L.-Q. Lee. *Boost Graph Library*. Boost, 2001. http://www.boost.org/libs/graph/doc/index.html.

[133] J. G. Siek. *A Language for Generic Programming*. PhD thesis, Indiana University, August 2005.

[134] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. The generic graph component library. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 399–414. ACM Press, 1999.

[135] Silicon Graphics, Inc. *SGI Implementation of the Standard Template Library*, 2004. http://www.sgi.com/tech/stl/.

[136] G. Sittampalam, O. de Moor, and K. F. Larsen. Incremental execution of transformation specifications. *SIGPLAN Notices*, 39(1):26–38, 2004.

[137] A. Srinivasan, T. Ham, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. In *IEEE International Conference on Computer-Aided Design*, pages 92–95, Nov. 1990.

[138] B. Steffen. Data flow analysis as model checking. In *Theoretical Aspects of Computer Software*, pages 346–365, London, UK, 1991. Springer-Verlag.

[139] A. Stepanov. The Standard Template Library — how do you build an algorithm that is both generic and efficient? *Byte Magazine*, 20(10), Oct. 1995.

[140] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[141] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[142] H. Tamaki and T. Sato. OLD resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98, London, UK, 1986. Springer-Verlag.

[143] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide*, version 6.4.1 edition. http://www.haskell.org/ghc/docs/latest/html/users_guide/index.html.

[144] S. W. K. Tjiang and J. L. Hennessy. Sharlit – A tool for building optimizers. In *Programming Language Design and Implementation*, pages 82–93, New York, NY, 1992. ACM Press.

[145] J. S. Uhl and R. N. Horspool. Flow grammars — a flow analysis methodology. In *Compiler Construction*, pages 203–217, London, UK, 1994. Springer-Verlag.

[146] E. Van Wyk, O. de Moor, G. Sittampalam, I. Sanabria-Piretti, K. Backhouse, and P. Kwiatkowski. Intentional programming: a host of language features. Technical Report PRG-RR-01-21, Computing Laboratory, University of Oxford, 2001.

[147] A. Vargun. *Code-Carrying Theory*. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, Dec. 2006. http://www.cs.rpi.edu/~musser/gsd/athena/cct-thesis.pdf.

[148] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[149] T. L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in *C++ Gems*, ed. Stanley Lippman.

[150] T. L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, volume 1505 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[151] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

[152] G. A. Venkatesh and C. N. Fischer. SPARE: A development environment for program analysis algorithms. *IEEE Transactions on Software Engineering*, 18(4):304–318, 1992.

[153] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In D. Rombach and M. L. Soffa, editors, *International Conference on Software Engineering*, pages 172–181, New York, NY, USA, 2006. ACM Press.

[154] E. Visser. Stratego: A language for program transformation based on rewriting strategies. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[155] D. S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.

[156] D. Whitfield and M. L. Soffa. Automatic generation of global optimizers. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 120–129. ACM Press, 1991.

[157] J. Willcock, J. Järvi, A. Lumsdaine, and D. Musser. A formalization of concepts for generic programming. In *Concepts: a Linguistic Foundation of Generic Programming at Adobe Tech Summit*. Adobe Systems, Apr. 2004.

[158] H. Yamamoto. An automata-based recognition algorithm for semi-extended regular expressions. In *International Symposium on Mathematical Foundations of Computer Science*, LNCS, pages 699–708, London, UK, 2000. Springer-Verlag.

[159] K. Yi and W. L. Harrison III. Automatic generation and management of interprocedural program analyses. In *Principles of Programming Languages*, pages 246–259, New York, NY, USA, 1993. ACM Press.

Jeremiah J. Willcock

Center for Applied Scientific Computing, L-550
Lawrence Livermore National Laboratory
Livermore, CA 94551

Phone: (925) 422-7790
Fax: (925) 423-1173
E-mail: willcock2@llnl.gov

Research Interests

- Program analysis and optimization, especially of high-level abstractions
- Analysis of object code
- Optimization of sparse matrix and graph kernels
- Programming languages

Education

Ph.D. Computer Science (in progress), Indiana University, Bloomington
M.S.C.S.E. Computer Science and Engineering, University of Notre Dame du Lac, 2002
B.S. Computer Science (summa cum laude), Northern Michigan University, 1999

Professional Experience

6/2006-present	Student intern and temporary staff member, Lawrence Livermore National Laboratory
5/2004-8/2004	Student guest, Lawrence Livermore National Laboratory
2001-2003	Research assistant, Indiana University, Bloomington
2000-2001	Teaching assistant, University of Notre Dame du Lac

Honors and Organizations

- Department of Energy High Performance Computer Science Fellowship
- Arthur J. Schmitt Fellowship

Selected publications

Journal and Magazine Articles

Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. "An Extended Comparative Study of Language Support for Generic Programming." *Journal of Functional Programming*, 17(2):145–205, March 2007.

Jeremiah Willcock, Andrew Lumsdaine, and Arch Robison. "Using MPI with C# and the Common Language Infrastructure." *Concurrency and Computation: Practice & Experience*, 17(78):895-917, June/July 2005.

Jaakko Järvi, Jeremiah Willcock, Howard Hinnant, and Andrew Lumsdaine. "Function Overloading Based on Arbitrary Properties of Types." *C/C++ Users Journal*, 21(6):25-32, June 2003.

Conference and Workshop Articles

Jaakko Järvi, Douglas Gregor, Jeremiah Willcock, Andrew Lumsdaine, and Jeremy Siek. "Algorithm Specialization in Generic Programming: Challenges of Constrained Generics in C++." In *Programming Language Design and Implementation*, New York, NY, USA, pages 272-282, 2006. ACM Press.

Jeremiah Willcock and Andrew Lumsdaine. "Accelerating Sparse Matrix Computations via Data Compression." In *International Conference on Supercomputing*, June 2006.

Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. "Associated types and constraint propagation for mainstream object-oriented generics." In *Object-oriented Programming Systems, Languages, and Applications*, New York, NY, USA, pages 1-19, 2005. ACM Press.

Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. "Algorithm specialization and concept constrained genericity." In *Concepts: a Linguistic Foundation of Generic Programming* at Adobe Tech Summit, San Jose, CA, April 2004. Adobe Systems.

Jeremiah Willcock, Jaakko Järvi, Andrew Lumsdaine, and David Musser. "A Formalization of Concepts for Generic Programming." In *Concepts: a Linguistic Foundation of Generic Programming* at Adobe Tech Summit, April 2004. Adobe Systems.

Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. "A Comparative Study of Language Support for Generic Programming." In *Object-oriented Programming, Systems, Languages, and Applications*, October 2003.

Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. "An Analysis of Constrained Polymorphism for Generic Programming." In Kei Davis and Jörg Striegnitz, editors, *Multiparadigm Programming 2003: Proceedings of the MPOOL Workshop at OOPSLA'03*, John von Neumann Institute of Computing series, Anaheim, CA, pages 87-107, October 2003.

Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. "Concept-Controlled Polymorphism." In Frank Pfennig and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of LNCS, pages 228-244, September 2003. Springer Verlag.

J. A. Izaguirre, Q. Ma, T. Matthey, J. Willcock, T. Slabach, B. Moore, and G. Viamontes. "Overcoming Instabilities in Verlet-I/r-RESPA with the Mollified Impulse Method". In T. Schlick and H. H. Gan, editors, *Proceedings of 3rd International Workshop on Methods for Macromolecular Modeling*, volume 24 of LNCSE, pages 146-174. Springer-Verlag, Berlin, New York, 2002.

Jeremiah Willcock, Jeremy Siek, and Andrew Lumsdaine. "Caramel: A Concept Representation System for Generic Programming." In *Second Workshop on C++ Template Programming*, Tampa, Florida, October 2001.

Masters' Thesis

Willcock, J., *Concept Representation for Generic Programming*, Masters' thesis, University of Notre Dame du Lac, 2002.