Automatic Staged Compilation

Matthai Philipose

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2005

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Matthai Philipose

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of the Supervisory Committee:

____

Craig Chambers

Reading Committee:

____

Craig Chambers

____

Susan Eggers

____

Daniel Grossman

Date:__________________________

University of Washington

Abstract

Automatic Staged Compilation

Matthai Philipose

Chair of the Supervisory Committee:

Professor Craig Chambers

Computer Science and Engineering

The ability to optimize programs while they execute has become increasingly important in recent years. The primary challenge in such optimization is to keep the run-time overhead of optimization down while maximizing its effectiveness. The widely used solution of Just-In-Time (JIT) compilation keeps run-time overhead low, at considerable engineering cost, by sacrificing performance.

The past few years have seen the emergence of *staged optimization*, which produces run-time optimizations that often have much lower run-time overhead than traditional optimizers, yet do not sacrifice any of their functionality. The key to the technique is a method, called *staging*, to transfer optimization overhead to static compile time from run time. Unfortunately, developing staged variants of individual optimizations has been highly specialized, labor-intensive work; staging pipelines of optimizations even more so.

This dissertation presents a system called the Staged Compilation Framework (SCF), which automatically stages entire pipelines of compiler optimizations at arguably little additional engineering cost beyond building the slower traditional version of the pipeline. SCF harnesses two powerful but traditionally difficult-to-use techniques, partial evaluation and dead-store elimination, to achieve staging. An implementation of SCF shows that staged compilation can speed up pipelines of classical compiler optimizations by up to an order of magnitude, and more commonly by a factor of 4.5 to 5.

Table of Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I would like to thank my advisor Craig Chambers. The work described in this dissertation posed challenges every step of the way. Craig gave me the freedom to work these challenges out at my own pace, a process that transformed me from graduate student to professional researcher. On the other hand, Craig's clarity of thought, mastery of material from the big ideas to the details and his instinct for identifying the central problem ensured that the freedom didn't become an impossible burden. Finally, Craig's obvious love for research has always been an inspiration for me: I don't know of many advisors who routinely check in large quantities of code with no paper deadline in sight, especially after tenure!

For the first few years of graduate school, I was co-advised by Susan Eggers. Susan has been a great mentor, calming influence and reality check. From Susan, I learnt how to break down systems into their components and understand them through rigorous measurement. I am especially grateful to her for supporting me while I cast about for the correct way to formulate the work of this dissertation.

I am grateful to my other committee members, Dan Grossman and Jean-Loup Baer. Dan agreed on short notice to serve on my reading committee and actually read through and vetted the whole dissertation, for which I am especially grateful. Jean-Loup also served on my general exam committee. Thanks to David MacDonald for agreeing to the thankless task of Graduate School Representative.

The work in this dissertation describes the third in a series of dynamic compilation systems I was involved in building at the University of Washington, and is strongly influenced by its predecessors. Joel Auslander was my partner in crime in building the first system. Although Joel and I were classmates, his knowledge of compilers and his mathematical intuition far exceeded mine, and I learned much from him. Brian Grant and Markus Mock were co-creators of the DyC system that was the direct predecessor of the system presented in this dissertation. We spent a couple of years beating the Multiflow compiler into submission: a bonding experience, if any! The Cecil group, under Craig's leadership, provided a wonderful environment for doing research. Thanks to Jonathan, Keunwoo, Sorin, Todd and Vass for all the good times, and for the excellent reading groups, discussion sessions and group meetings. Thanks especially to Sorin and Keunwoo for helping with practicing my defense talk.

Lindsay Michimoto, Melody Kadenko and Frankye Jones have been invaluable in dealing with the university. Lindsay in particular has been infinitely patient while I meandered leisurely through the dissertation writing and defense process. I would also like to thank the computing support staff for the tireless and excellent service during my stay.

I committed the cardinal sin of joining work before I wrote up my dissertation. I am grateful to my managers and colleagues at Intel for having been extremely supportive of me during the three years it took me to finish. I would especially like to thank my managers Gaetano Borriello and James Landay for their support and understanding.

Friendships with my fellow graduate students put things in perspective, even when the pressures of work were intense. Thanks to Joel Auslander, Anhai Doan, Marc Fiuczynski, Brian Grant, E Lewis, Jack Lo, Evan McLain, Omid Madani, Todd Millstein, Sujay Parekh, Tapan Parikh, Kurt Partridge, Don Patterson, Josh Redstone and Zasha Weinberg.

Thanks to Amma and Appa for their unconditional love, support and belief in me all these years. You have been my most influential teachers. Finally, my wife Kerry has held a job, given birth to two children and completed her own doctoral studies while I was buried in my thesis work and dissertation. Through it all, she has showered me with love and affection. Thank you Kerry. Every man should be so lucky.

DEDICATION

To Amma and Appa who made me

and Kerry who completed me

1. Introduction

Information relevant to program compilation becomes known at different stages during program compilation and execution. These stages include:

1. traditional separate compile time, when each single-file piece of a program becomes available,
2. library assembly time, which offers new interprocedural analysis opportunities,
3. program link time, which offers more interprocedural analysis opportunities and possibly closed-world analysis opportunities as well,
4. initial program load time, when details of the execution platform become known,
5. dynamic load time, when knowledge of run-time extensions or changes to the program can be exploited, and
6. run time, which offers opportunities to customize the compiled code to the application's actual run-time behavior.

Exploiting the information available in later stages can lead to much better optimization, in practice as well as theory. For example, link-time compilers can perform interprocedural and whole-program analysis, and run-time compilers can optimize based on dynamic program behavior or target platform characteristics, all with substantial performance gains.

Optimizing on the basis of late-stage information comes with a challenge. The later the stage, the faster (in terms of time per instruction optimized) the optimizer run at that stage needs to be. One reason for this requirement is that information relevant to optimization typically changes more frequently at later stages. Consider the dynamic-loading and the run-time stages, for instance. Typically, each time a module is dynamically loaded, it is run a large number of times. Since the cost of each instance of load-time optimization is amortized over a larger period than that of run time optimization, we are typically willing to incur a higher overhead at load time.

A commonly used technique to achieve fast late-stage optimization is to explicitly design versions of optimizers for the late stage that are "leaner". These optimizers have a carefully chosen subset of the functionality of their early-stage counterparts. For instance, just-in-time compilers have a smaller set of optimizations than typical static optimizing compilers, and the optimizations themselves are often non-iterative and local. Also, link-time optimizers are often flow-insensitive, whereas separate compile time optimizations are flow-sensitive. This technique of sacrificing optimization quality to speed up optimization has proven to be effective in many cases.

A complementary approach to speeding up late stages, called *staged optimization*, uses early-stage computation for pre-planning and partially executing late-stage optimization. The intention is that by thus increasing the *effective* amount of time available to the late-stage optimization (while hopefully not overly burdening the early stage), late-stage optimizations don't have to be as lean as, and can therefore be more effective than, optimizations that run wholly in the late stage.

Staged optimization exploits the fact that for many programs, although precise input values to an optimization may not be available until a late stage, some *approximate knowledge* of these inputs is available at an early stage. For example, it may be known, at separate compilation time, which variables and data structures are likely to have invariant values, which methods are the likely targets of particular dynamic dispatches, and which branches are likely to be biased, but the actual values, methods, or branch paths may be unknown until link time, load time, or run time. It is possible to exploit this early knowledge by designing the optimization so that it executes over many stages. The part that executes at an earlier stage could exploit early knowledge by pre-computing the possible calculations and outcomes of the later-stage parts, and generating a customized version of the later-stage parts that performs only the computations needed to resolve what was unknown in the earlier stage. Since the customized late-stage part needs only to complete the optimization, late-stage optimization costs are lowered. Further, since a given piece of early-stage information may approximate many later-stage instances, the overhead of building a customized optimizer at an early stage can be recouped over many later-stage uses.

Staged optimization has been shown to be both fast and effective for run-time compilation of non-trivial programs [24, 14, 39, 48]. However, although these systems have validated the idea of staging, the complexity of engineering them to stage arbitrary optimizations is a barrier to their widespread adoption. In fact, all but one of these systems only stage only a single optimization: partial evaluation. Anecdotal evidence suggests that despite of a vast literature on staging partial evaluation, building systems that staged effectively even this single optimization was quite challenging.

A predecessor of the staged compilation system presented in this thesis, called DyC [24], demonstrated staged versions of two optimizations beyond partial-evaluation: copy propagation and dead-assignment elimination. However, writing staged compilers based on these algorithms for staging individual optimizations is substantially more difficult and error-prone than writing their unstaged counterparts. The primary difficulty is that where traditional optimizations reason about the behavior of the program to be optimized, a staged optimization has to reason about the behavior of the optimization itself when applied to that program. In effect, the early stage has to perform a kind of case analysis of all the ways that optimizations might proceed at a later stage, and then pre-compute for each case as much of the final result as possible. Reasoning at this "meta" level is a significant burden on the compiler writer. A secondary difficulty is that an algorithm for staging an individual optimization may not work well when the optimization is part of a pipeline. In particular, the DyC algorithms for staging individual optimizations were hardwired to assume that a particular set of optimizations would precede them in the pipeline, thus destroying the compiler designer's ability to modularly re-arrange the optimizations in different orders in the pipeline.

To address these difficulties, we present in this thesis a system called the Staged Compilation Framework (SCF) that can automatically and mechanically construct a staged compiler, given an ordinary single-stage compiler. The compiler writer simply writes program optimizations (that typically input a function and output a transformed version of the function), and composes them in an arbitrary sequence, just they would with a traditional, unstaged compiler. The optimizations are written in a first-order, side-effect-free subset of ML called SCF-ML. Beyond using this language, the compiler writer need not, in princple, be aware that the optimization is to be staged.[1] At any stage, given approximate information about the inputs to the compiler, a compiler *user* (who usually is distinct from the

1. Although in practice, the SCF infrastructure is tuned to perform best when optimizations are written in a specific (arguably natural) "compositional" style.

compiler *writer*) may feed the compiler and the information to SCF, which will automatically produce a version of the compiler specialized for the approximate information.

The design of the system is based on two key insights. First, the approximate information, describing both inputs to and results of any staged optimization in the compiler pipeline, may be viewed uniformly (and independently of the optimizations preceding the optimization in the pipeline) as the *set of possible values* assumed at later stages by the formal parameters and the return value of the optimization function. Later stages have access to "less approximate" information in the precise sense that the set of possible inputs at the later stage is a subset of that at the earlier stage.

Second, the effect of a hand-written staged optimization on its approximate input is similar to that of *systematically specializing an ordinary, unstaged version of the optimization with respect to the approximate input*. In our current design, the specializer (which we call the *stager*) consists of a forward pass which performs a specialization step called online partial evaluation, followed by a reverse pass which eliminates dead store operations. Although both partial evaluation and dead-store elimination are techniques that have been studied extensively, it is notoriously difficult to design online partial evaluators [56] or dead-store eliminators [53] that are both effective and terminate in a reasonable amount of time. Showing how to effectively harness these techniques towards the task of staging programs is at the heart of this thesis.

This thesis makes the following contributions:

1. It motivates and formulates the problem of automatically deriving a staged version of a compiler from an unstaged one.
2. It shows how to represent possible inputs and outputs to all optimizations in a uniform way, and how to use the uniform representation to enable staging of a pipeline of optimizations given techniques to stage an individual optimization.
3. It describes a set of techniques for automatically specializing individual optimizations so as to make them substantially faster. Novel contributions include:

- a simple first-order side-effect free functional language called SCF-ML for specifying optimizations,

- an expressive domain for the online partial evaluator,
- a simple but effective analysis to determine what functions to specialize and on what arguments,
- a context-sensitivity strategy which specializes call-chains of functions and is tailored to the concrete behavior of the input program,
- an "eager" specialization scheme to keep track of correlations when analyzing sets of values, and
- a dead store elimination algorithm that cooperates with the partial evaluator to enable effective dead assignment elimination through commonly used data structures.

4. It evaluates an implementation of the design, targeted at staged compilation of C programs. The evaluation shows that our techniques can produce staged compilers that are several times faster than their unstaged versions, and also analyzes the contribution of the techniques mentioned above to overall speedup.

The rest of the thesis is structured as follows. Chapter 2 places the goals and techniques of SCF in the context of other efforts to achieve fast late-stage compilation. Chapter 3 presents an overview of the design of SCF followed by an example detailing inputs and outputs to various parts of SCF. Chapters 4 through 8 describe how the core components of SCF work. Chapter 9 presents measurements that evaluate both the overall effectiveness of SCF, and the contributions of particular SCF techniques to the overall picture. Chapter 10 presents conclusions and discusses future work.

2. Motivation and Background

Staged compilation is essentially a method for making compilers faster. *Automatic* staged compilation, as advocated in this dissertation, is a way for making it easier to create faster compilers. In this chapter, we discuss why creating faster compilers is useful, how staging compares to other methods for speeding up compilers, and how automated staging compares to other methods for staging.

We present a simple view of a modern compiler in section 2.1. We discuss traditional approaches to fast compilation when a compiler is used according to this simple view. In section 2.2 we introduce the notion of compilation stages, which are distinct points in time when parts of inputs to the compilers are introduced, possibly by different principals (such as kernel function writers, library writers and whole-program writers). We explain how the staged availability of information may demand even faster compilers. In sections 2.3, we introduce an approach, called staged compilation, that allows us to exploit the compilation stages themselves to speed up compilation. In sections 2.4 through 2.6, we discuss three techniques, procrastination, pessimism and preplanning, for implementing staged compilation. The first two techniques are routinely used in existing compilers. The third technique, preplanning, is more powerful in important ways but less developed. We examine three techniques for preplanning-based staging, in order of increasing automation, ending with the highly automated approach proposed in this dissertation.

2.1 A Simple Model of a Modern Compiler

Figure 2.1 is a schematic for a modern compiler [2]. The compiler consists of a sequence, called a *pipeline*, of *optimization phases* O_1 through O_n. In what follows, we refer to the O_i as *optimizations* or as *phases*. Common optimizations include constant propagation,

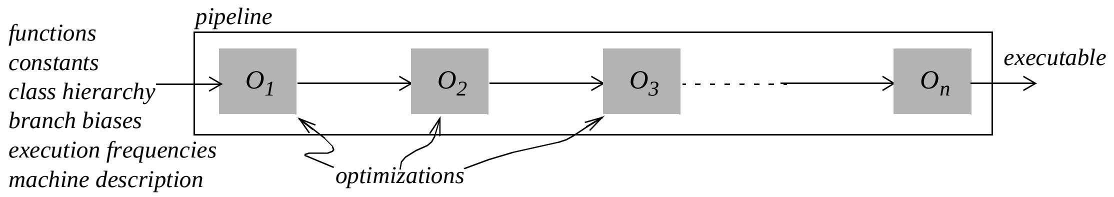


FIGURE 2.1: A Simple View of a Modern Compiler.

dead assignment elimination, function inlining, register allocation, scheduling and linking. Optimizations can be interprocedural, in which case many functions are optimized simultaneously, or intraprocedural, where functions are analyzed one at a time. The pipeline takes as input the functions that comprise the program to be compiled, along with a variety of auxiliary information that helps in compiling the program. We will call the functions *input functions* and the program the *input program* in what follows. The auxiliary information may include, for instance, values of constants in the program, class hierarchies (when the input program is object oriented), the probabilities of various branches in the programs being taken, the execution frequency of input functions and a description of the machine on which the program is intended to be executed. Each optimization takes the set of functions and relevant auxiliary information and produces a transformed version of the functions and sometimes, auxiliary information about the transformed functions. The output of the entire pipeline is an executable program, i.e., a fully compiled version of the input program.

In the simplest usage model for the compiler,[1] a programmer (a *compiler user*) first writes the program to be compiled, gathers all the source code and auxiliary information for the program, and invokes the compiler (typically written by a separate *compiler writer*) to produce the program to be executed. We call this usage model the “single-stage” model, since the compiler is invoked exactly once, when all inputs to it are available. A *program user* may later execute the resulting program. The implicit contract between these three principals is that the compiler writer produces a compiler that does not take too long to compile, but produces binaries that deliver good performance (measured in execution speed, memory footprint, compliance with security policies, etc.). The compiler user selects the features of the compiler (typically through command line flags), invokes it, and waits for as long as necessary to produce sufficiently high quality code for their clients. The program user runs the program with assurance of maximum performance. At the heart of the contract is the capability of compilers to produce binaries of sufficiently high quality within a reasonably small amount of time.

1.Although for reasons of efficiency and practicality this simple model where all source code is submitted at once to the compiler is somewhat simpler than that used by most traditional compilers, it is the one used implicitly to describe many whole-program optimizations.

The difficulty in living up to this contract is that the analyses that underlie optimizations are typically polynomial (traditionally *$O(n^3)$* or worse) in their input size. When the input programs are analyzed one input function at a time, as most compilers still do, this cost is bearable. However, if as many modern optimizations do, the whole program needs to be analyzed at once, improving compilation speed becomes an important research topic. Existing techniques to improve speed fall into three main classes, *approximation*, *summarization* and *selective optimization*, which we describe below.

1. **Approximation.** As per the widely used abstract interpretation [15] view of program analysis, conventional optimizations adopt an abstract view of the set of possible traces (i.e., the sequence of concrete stores) of the program they are optimizing, and interpret the operations of the program as computations on the abstract trace. The challenge of designing an effective optimization lies in picking an abstract view that is precise enough that it preserves relevant features of the traces of the program being approximated, but approximate enough that the interpretation of the program over this abstract domain terminates in a timely fashion. Some common approximations, which may be applied to the abstractions selected for most optimizations, and which are designed to speed up program analyses include *flow insensitivity* [5,69], where the optimization conflates trace values from different program points, *context insensitivity* [12,57], where the optimization conflates trace values entering a function from different calling program points, *heap insensitivity* [22, 59] where the optimization conflates heap-allocated store locations and *bidirectional assignments* [64], where the optimization conflates the store locations on two sides of assignment statements.
2. **Summarization.** As mentioned above, a typical optimization interprets with respect to an abstract domain each operation in the program being optimized. The interpretation function for each operation is sometimes called a *flow function*. The action of an optimization on a program fragment (such as a basic block, function or module) can be viewed as the application of a sequence of flow functions to the abstract store at the entries to the fragment.

In some cases, it is possible to replace the sequence of flow function with a single *summary flow function* which is simpler than the compound one obtained by sequentially invoking the original flow functions. In these cases, the optimization proceeds in two passes. The first pass generates the summaries, and the second pass uses them. When the optimization needs to analyze the summarized program fragment, it can simply consult the summary of the fragment to obtain (in general) a much quicker estimate of the effect of the fragment than re-analyzing the entire fragment. Reps [52], for instance, has shown how to mechanically construct fast summary functions for interprocedural optimizations whose abstract domains meet a particular set of criteria. The T1-T2 analysis of Tarjan [2] is another method for constructing fast summary functions for certain intraprocedural optimizations. Finally, the componential analysis of Flanagan and Felleisen [21] shows how to construct summary-based versions of a broadly useful class of analyses called set-based analyses. Many other analyses, such as synchronization analysis [58] and points-to analysis [17] have benefited from a summary-based formulation.

3. **Selective optimization**. The simplest, and most common, technique for lowering optimization costs is to provide the compiler user with the option of disabling certain optimizations in the pipeline, of applying lean versions of optimizations when desirable, or of enabling optimization only on certain parts of the incoming program, e.g., parts which are executed heavily as per an execution profile.

The above techniques are effective enough that the single-stage model of compilation is quite acceptable in many traditional compiler-use scenarios. In particular, since this model places responsibility for compilation of the whole program on a single compiler user, as long as the user accepts this responsibility, she will presumably be willing to “pay a reasonable price” (where “reasonable” is, say, between linear and quadratic time in input program size) to optimize the whole program. Especially with programs written in “low-level” languages such as C and C++ that benefit relatively little from heavyweight whole-program analyses, and that are not designed to be used as mobile code (where the processor on which they are executed is determined only at run time), the single-stage model is adequate.

When a single compiler user is unwilling to accept compilation overhead for the whole program, when heavyweight whole-program analysis can be profitable (as for programs written in high-level languages such as Java, C#, Lisp and ML), or when all inputs required by the compiler are unavailable (as in programs using dynamic linking and mobile code), the single-stage model breaks down. Given recent software engineering trends towards high-level languages with mobile, dynamically updated programs, and given that programs have long been created by multiple compiler users, it is useful to examine a compilation model that goes beyond a single stage.

2.2 Compilation Stages

The single-stage model assumed that the compiler is invoked exactly once, when all its inputs are available. We now recognize that the inputs to the compilers may be made available in multiple stages (i.e., distinct points in time). Correspondingly, there may be more than one compiler user providing this input. We call this model the multi-stage usage model.

Figure 2.2 is a pictorial representation of the multi-stage model. The figure focuses less on the compiler pipeline (on the right of the figure) than on the stages and inputs to

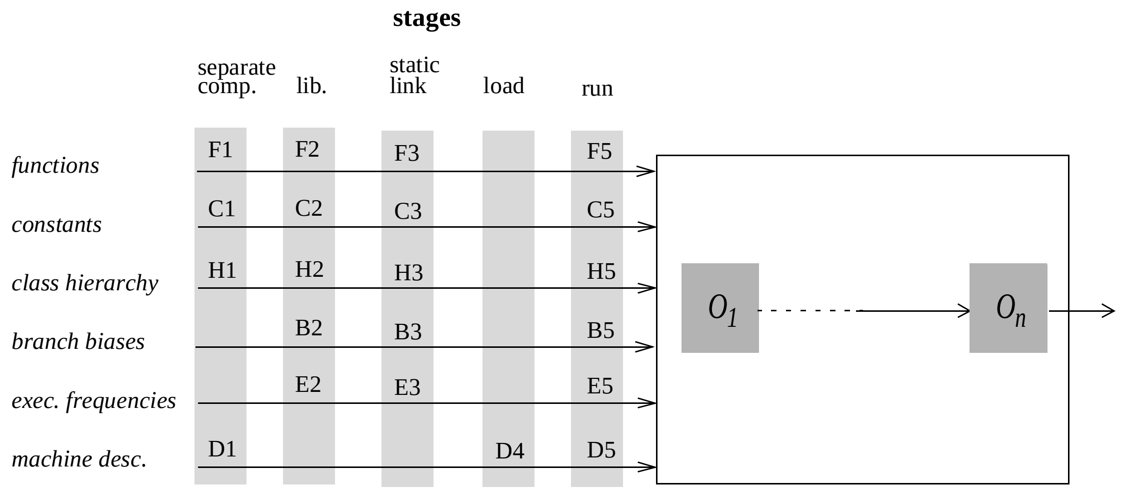


FIGURE 2.2: Compilation Stages.

the compiler (on the left). The inputs and stages form a table, with inputs as rows and stages as columns. Inputs range from functions to machine descriptions, as in the single-stage model. Stages range from separate compilation (labeled “separate comp.”) to program execution (“run”). Each column (highlighted in gray) in the table, represents the additional information for compilation introduced in the stage labeling the column. An entry in the table (such as F2) designates the part of a particular kind of input available at a particular stage (F2 represents the functions available in the second stage, i.e., the library assembly stage). We step through the five stages in turn and discuss the kinds of information that may be available at each. The particular stages, inputs and incremental information portrayed in the figure are not meant to rule out others: they are meant to serve only as a realistic example.

1. The separate compile time stage, when individual functions are available. The functions themselves are denoted by label F1 in the figure. With each function, we often know the values of certain constants (C1) used in these functions, a partial class hierarchy for classes used in the function (H1), and sometimes, a description (D1) of the machine on which the function is intended to be executed on. Note that branch biases and execution frequencies of a function are typically not known until link time, when it is possible to run a version of the program and gain profile information.

2. The library assembly stage, when a set of functions is assembled into a library. Having these functions makes library wide constants (C2) and class-hierarchy extensions (H2) available. Further, if the library has been tested with “representative” clients, many of the branch biases (B2) and execution frequencies (E2) may also be available. It may seem that in order to “test out” the library with a client, it is necessary to execute the library, so that it is dubious to claim that profile information is available before the execution stage. However, note that it is often possible to run the library on a few representative programs to get profile information, and to use the profiles so collected as the input profile when compiling most other programs. For most programs that use the library, therefore, it may be unnecessary to execute the program to get profile information. The same is true of the separate compilation stage, of course. However, it

is unusual to collect profile information for individual functions, although for large very heavily used kernels, it is not inconceivable to do so.

3. The static link stage, when sets of functions and libraries are grouped into whole programs, is very similar to library execution, except that we often have an additional guarantee at this point that no more functions will be added to the program (or to certain parts of the program), so that the compiler can make a strong closed-world assumption.
4. The load stage, when programs are loaded from disk into memory for execution. Details (D4) of the machine on which the program is to be run may be provided only at this point. For instance, a program may be compiled for a particular architecture, but it may be possible to optimize it further for an implementation of the architecture.
5. The execution stage, when the program is run. Based on values computed during execution of the program, additional functions may be dynamically linked and loaded, yielding a more complete view of the program (F5 and H5), the values of variables that are quasi-static (remain fixed for long periods during execution) may be revealed (C5), more accurate execution profiles (E5) and branch biases (B5) may be available, and (in the case of dynamically loaded and mobile code) details of the underlying machine (D5) may be revealed.

The staged availability of information as illustrated above presents two challenges in compiler design. The first challenge is that because we now have multiple compiler users (typically, one at each stage), each user may be willing to pay only for compiling “their part of the program”. For instance, if a user links in large libraries produced by other users into a relatively small program that they have written, they may be unwilling to accept the large compilation overhead to optimize the program as a whole. Using a single-stage compilation strategy would likely be unacceptable in this scenario because the final compiler user would have to pay the cost of optimizing the whole program.

The second challenge is that since the final stage may well be during program execution, the strategy of waiting until all inputs are available before compiling can potentially result in a slowdown of the program to be optimized. Specifically, the overhead of compi-

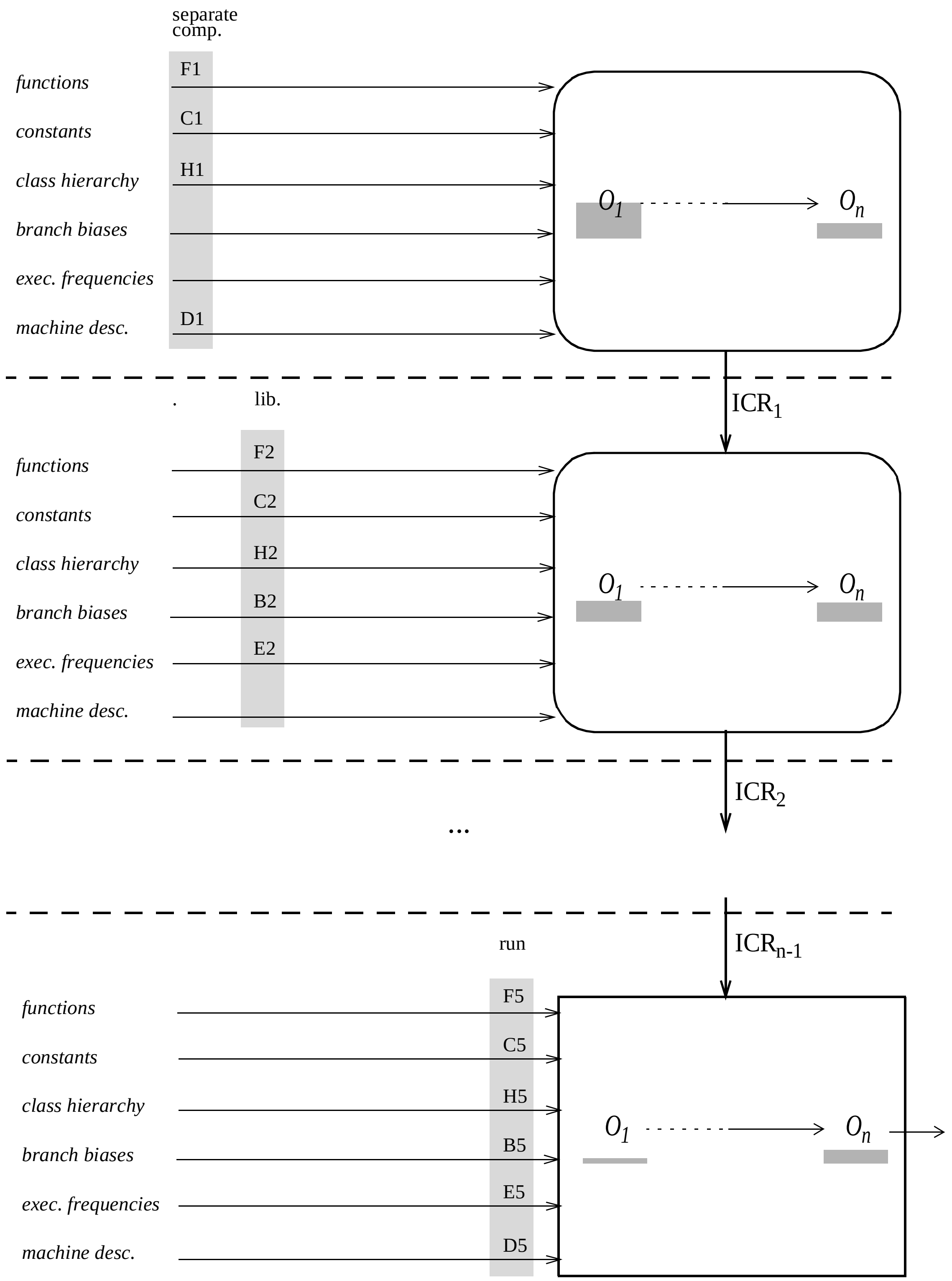


FIGURE 2.3: Staged Compilation: A Schematic View.

lation at the execution stage may overwhelm any benefits from optimization, especially given the high overhead of heavyweight optimization techniques.

2.3 Staged Compilation

Careful use of the techniques from the previous section for speeding up optimizations could help in addressing the above two challenges. Interestingly, however, the staged availability of inputs itself presents a powerful additional opportunity for addressing the challenges. In particular, both the above challenges can be viewed essentially as due to deferring “too much” work to later stages, especially the final stage. Time spent compiling in the final stage may be “disproportionate” to the information added in the final stage. If we could shift some of the compilation burden away from the final stage to the previous stages (by “partially executing” it somehow), and in particular make the work done in each stage correlated with the amount of new information added in that stage, we have the possibility of reducing the amount of work done in later stages.

Figure 2.3 provides a schematic view of how staged compilation may work. The compiler is partially executed on the partial inputs available at each stage before the final stage, and fully executed at the final stage. A single partial execution is a row in the figure; rows are separated by dotted horizontal lines. Partial execution of the compiler is denoted by drawing the compiler pipeline with rounded corners, as for the separate compilation and library assembly stages. Concrete execution is denoted by drawing the pipeline with square corners, as in the run time stage.

The result of partial execution is an intermediate compilation result, denoted ICR in the figure. For instance, partially executing the compiler on information available at the first (separate compilation) stage results in the intermediate result ICR_1. All later compiler executions (whether partial or concrete) take as input the ICR from the previous stage. The intention is that the current stage can use the ICR from the previous stage so as to avoid repeating the work done by the previous stages. Each stage will therefore hopefully do less work than if it had to process the information gathered over all previous stages. Figure 2.3 represents the diminished load on each step by replacing the full gray squares comprising the optimizations of the pipeline with gray rectangular slivers. If staged compilation works well, for instance, each optimization in the final stage will only have to do a very small sliver of the work it would have to do in the unstaged model.

We assumed in this section that there is some way of partially executing the compiler pipeline, and of communicating the result as an ICR. In the next three sections, we examine some concrete approaches for doing so.

2.4 Staging via Procrastination

A common technique for staging is based on the observation that not all optimizations in the pipeline require all inputs. Further, optimizations early in the pipeline often require only information that is available in early stages of compilation. For instance, early phases in a pipeline rarely perform either whole program optimization or machine-specific optimizations, and can do with single functions with inputs. At each stage, some prefix of the pipeline may have all the inputs it needs.

A possible implementation for partial execution, therefore, is to identify the subsequence of optimizations in the pipeline for which all inputs are available at a given stage, but not in the previous stage, and to concretely execute only that sub-sequence. Execution of the rest of the pipeline would be deferred to later stages, a strategy we call procrastination. The input for execution would be the ICR from the previous stage, the result of all phases preceding the current sub-sequence. The execution will produce results to be passed on to the first optimization after the sub-sequence; of course, this optimization does not have all its inputs in the current stage. These results would therefore constitute the ICR fed into the next stage.

Procrastination clearly has the potential to move some compilation overhead to early stages. In fact, in most modern compilers, functions are parsed and many local optimizations performed at the separate compilation stage. Even in just-in-time (JIT) compilers [65, 1], which are optimized to exploit information available at the run-time stage, it is common to compile functions down to the bytecode level in a separate compilation stage that happens well before the program is executed. Compilers that perform whole program optimizations normally also perform parsing and local optimizations before whole program optimizations. If the compiler consists purely of intraprocedural optimizations, then almost the entire pipeline (other than simple "gathering" steps such as archiving and linking) can be moved to the separate compilation stage.

The drawback of procrastination is equally clear. Many lucrative optimizations, for instance whole-program analyses that perform function devirtualization, representation optimizations (such as inlining of fields) and machine-specific optimizations (such as register allocation and scheduling), often require inputs (such as the whole program, execution profiles and machine descriptions) that are often only available at very late stages. Thus, compiler users at late stages, such as link time, may still be saddled with very high overheads of heavyweight program analysis, and exploiting run-time information may still require an excessive overhead.

2.5 Staging via Pessimism

A compilation system based purely on procrastination has the disadvantage that if an optimization has incomplete inputs, the compiler user has no choice but to defer the optimization to a later stage. Pessimism is a technique for partial execution that, unlike pure procrastination, allows an optimization to execute to completion with partially defined inputs. The basic idea behind the technique is simple: where input information is necessary but missing, assume it to be the most conservative possible information. The trick to making pessimism effective is to fall back on missing information sparingly.

Examples of how conventional optimizations may use pessimism to counter lack of information follow.

1. To understand as precisely as possible whether a particular function call side effects a particular local variable, it is in general necessary to analyze the whole program. Potential definitions due to side effects are important to identify in optimizations such as dead assignment elimination and register allocation. In the absence of the whole program, the most pessimistic assumption would be that all local variables may be side-effected at every function call. Intuitively, however, we know that a variable can be side-effected only if its address has *escaped*, i.e., has been passed to a callee, either directly as a parameter or indirectly through the heap. When only a single function or a library of functions is known at a particular stage, it is conventional to perform an

escape analysis [11, 49], and pessimistically assume that only escaped variables are possibly side effected.

2. When optimizing object-oriented programs, it is necessary to determine the possible set of classes that a particular variable use may assume at run time. Identifying the classes can help in eliminating expensive virtual method calls, for example. As with side-effect analysis, the complication is that if the reference escapes some fragment of code (or if its value flows from outside the fragment), there is a possibility that it will be assigned an arbitrary legal value by the code it escapes to (or that its value flows from). Ideally, if all the code the fragment may reference is available, conventional whole program analyses could determine its possible classes. However, if only a function or some libraries of the eventual program are available, it is common to make pessimistic assumptions. The naive pessimistic assumption would be to assume that the variable will be *some* subclass of its declared class at run time; given that classes may be added at later stages to the program, the set of possible classes cannot be bounded at the early stage. A more sophisticated pessimistic approach (called *extant analysis* [63]) is to identify those variables that escape the currently available fragment of code (or to which there is may be value flow from outside), and make the pessimistic assumptions for these. Those variables that do not escape the fragment can be analyzed with the assumption of full knowledge.
3. When performing constant propagation or partial evaluation, it is necessary to have the values for all constants to be propagated, including constants represented by symbols. For instance, a variable may be annotated `final` in a Java program, or a symbol may denote an address to be resolved to a fixed value by the loader or dynamic linker. At a particular stage (such as library assembly time or load time), some subset of such constants may be resolved, and not others. A pessimistic analysis could handle this situation simply by systematically setting the not-as-yet-resolved constants to “non-constant” value and then performing the optimization.
4. When performing backend optimizations such as scheduling and register allocation, it is useful to have as detailed a model of the machine on which the code will be

executing on as possible. However, before the code is executed, it may be the case that the architecture of the processor to be executed is known, but not the implementation. For instance, it may be known that the target is an X86 processor, but not which generation, what multimedia instructions it will support, etc. A pessimistic solution may be to select a conservative machine representation that is guaranteed to execute correctly on all instances of the architecture, albeit at some performance cost.

To enable pessimistic compilation over an entire pipeline, a compiler writer may provide both regular (optimistic) and pessimistic versions of optimizations in a pipeline. At a given stage, in the presence of incomplete information, a compiler user may choose between procrastination and pessimism. If he chooses the former, he would simply pass on the ICR input of the current stage to the next stage. If he chooses the latter, he would execute the pessimistic version of the optimization on the incoming ICR and pass on the results to the next phase. More commonly, the compiler writer would just provide either a pessimistic or an optimistic version of the entire pipeline; in the former case, the compiler user defers no work to the next stage, in the latter he would defer all work.

Pessimism has the advantage that it allows optimizations to be executed at early stages in spite of partial availability of inputs, thus avoiding overly burdening later stages. Its disadvantage is that the late stage speed comes at the expense of lost optimization opportunities.

2.6 Staging via Preplanning

Procrastination and pessimism are less than ideal solutions for the absence of complete information about an input required by an optimization. A compiler user can either completely defer an optimization (adding corresponding overhead to later stages), or completely execute it pessimistically (with accompanying performance loss in compiled code). Essentially, we achieve partial execution of the *pipeline as a whole* in the face of partial inputs at a stage by concretely executing individual optimizations whose inputs are completely defined (albeit perhaps only after a pessimizing step). A third option, which we will call preplanning, allows partial execution of an *individual* optimization given only partial information about its inputs. “Partial execution” in this case means a kind of preplanning, rather than execution in the conventional sense, since conventional execution is impossible without complete inputs. The result of partial execution at a stage can be thought of as a plan for completing the optimization at a later stage, when more information about the optimization's inputs is available, without repeating much of the work from the earlier stage. The advantage of this approach is that it reduces the burden on the late stage(s) without necessarily compromising on accuracy.

Below we discuss two ways in which preplanning-based staging may work. Section 2.6.1 shows how, if only summarizable analyses are to be staged and the only information to be exploited in stages is the set of functions in the program, the summarization technique of section 2.1 may be adapted to achieve partial evaluation of the analysis. Section 2.6.2 discusses ways of staging complete optimizations (which typically comprise both analyses and transformations), as well as pipelines of optimizations on arbitrary kinds of staged inputs.

2.6.1 Summarization as Preplanning

Recall from section 2.1 that one of the ways to speed up an analysis with no loss of quality in the result was *summarization*. Summarization essentially structures the analysis as a two-step process. The *summarization step* processes various fragments of the program (basic blocks, functions, libraries, etc.) separately from the rest of the program to compute a compact, analysis-specific summary for the fragments. The *propagation step* performs the analysis using the fast summaries in place of the original program fragments whenever possible.

The extension to staging is obvious and has been proposed by various researchers [21, 58, 17]: the two steps can be performed in different stages if the summaries from the early stages are treated as the ICR's. For example, given just a library at the library assembly stage, we could construct a summary for the library that captures only its interactions with its clients. At link-time, we could do whole-program analysis across the pre-computed summaries that accompany the fragments of the program being linked together. Once the system reaches fixed point, we could propagate results locally within summaries. In both cases (global and local analysis), summarization allows the number of nodes being analyzed simultaneously to be much smaller than without summarization.

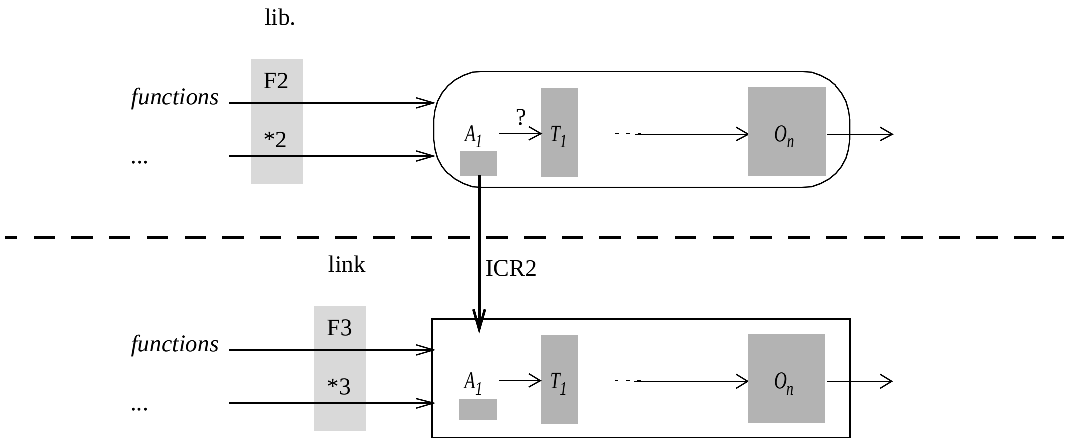


FIGURE 2.4: Staging Analyses by Summarization.

Figure 2.4 shows the benefits and the limitations of staging by summarization. Suppose the first optimization O_1 in the compiler pipeline is an optimization comprised of a summary-based analysis A_1 and a transformation T_1. Let F2 be the subset of input functions known at the library assembly stage, and F3 the subset at the later link stage (the remaining arguments in each stage are collectively labelled *2 and *3 respectively). Assume for this example that all inputs are known at the link stage. Then the summarization step of A_1 may be executed on inputs F2, and the summary saved (as intermediate compilation result ICR2 for the link stage). The link stage will summarize the remaining functions in F3, and perform propagation on all the summaries. Since summarization of all functions available at library assembly time occurs at that stage, the time taken summarizing (and therefore analyzing) in the link stage can be much less than if the whole program had to be summarized.

On the other hand, although the analysis itself can be partially executed at the early stage, the transformation it feeds into and all subsequent optimizations typically cannot. This is because, as figure 2.4 indicates by a question mark between A_1 and T_1, it is unclear what “partial results” A_1 may pass to T_1. In particular, we need to somehow “partially execute” the propagation step at A_1 in addition to the summarization step, so that T_1 may have some “partial knowledge” of possible analysis results. In fact, even though summaries for

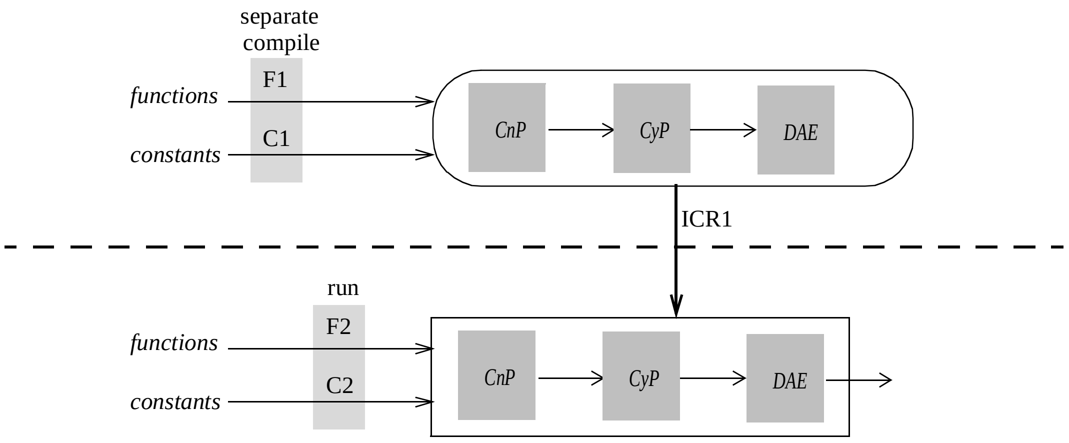


FIGURE 2.5: Example Pipeline to Demonstrate Preplanning on Pipelines of Optimizations.
The pipeline to be staged consists of the three optimizations constant propagation (*CnP*), copy propagation (*CoP*) and dead-assignment elimination (DAE) over two stages, the separate compilation stage and the run time stage.

analyses are stored in files along with program intermediate representations for processing at the later stage, no existing summary-based system stages the optimization phases downstream of the analysis. The applicability of summary-based staging to speeding up late stage optimizations (or pipelines of them) is therefore restricted.

Another limitation of the summary-based approach to staging is that it is designed to exploit the staged arrival of program fragments (functions, libraries, etc.), but not other inputs such as constant values and machine models.

2.6.2 Preplanning for Pipelines of Optimizations

We now discuss three methods for preplanning individual optimizations and complete optimization pipelines. The methods differ most importantly in the amount of effort they require from the principals involved in the compilation process and in the speed of the later stages. The first technique (manual preplanning by compiler users), requires great incremental effort from compiler users *per new program* and less so from compiler writers, but can give the fastest late-stage compilers. The second method (manual preplanning by compiler writers), requires great incremental effort from compiler writers and much less from compiler users *per optimization to be staged*. The third method, (automatic preplanning), requires relatively little incremental effort from both compiler writers and compiler users, given a one-time investment in a staged compiler framework of the kind described in this dissertation.

Figure 2.5 shows the configuration we will use as a running example in this section. Compilation is staged across two stages, the separate compile time stage and the run time stage. The compiler pipeline consists of the three optimizations constant propagation, copy propagation and dead assignment elimination. The pipeline takes two inputs, the function to be optimized and a record that maps each parameter and global variable used in the function to either the value NON_CONSTANT or a value of the form CONSTANT(*v*). The former value indicates that the variable will not have a fixed value at run time; the latter indicates that it will have the fixed value *v*. We assume that the function is fully known at both stages (so that F1 and F2 are identical). For our example, we assume that F1 and F2 are both the function `mul_add` specified in figure 2.6(a). On the other hand, we assume that for some of the parameters or globals in the function, we know at the early stage only that they are constants, without knowing the value of the constant (input C1 in figure 2.5); we assume that the actual values are revealed only at the run-time stage (input C2 in figure 2.5). Figure 2.6(b) shows the value of C1 we use in our example. We know at the early stage that parameters `x` and `y` of function `mul_add` are not constants (and will not be at run-time either). On the other hand, we know that `a` is a constant, but we do not its value at this stage. Figure 2.6(b) shows the value of C2 we use in our example: at the late stage, the fixed value of `a` is revealed to be zero.

```
int mul_add(int x,int y, int a){
  int u = x * a;      //command 1
  int v = u + y;      //command 2
  return v;           //command 3
}
```

(a)

```
{ x = NON_CONSTANT, y = NON_CONSTANT, a = CONSTANT (?)}
```

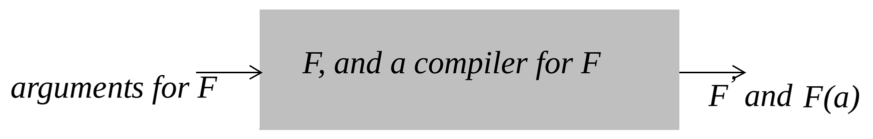
(b)

```
{ x = NON_CONSTANT, y = NON_CONSTANT, a = CONSTANT (0)}
```

(c)

FIGURE 2.6: Inputs to the Example Pipeline.

(a) Input function F1 (same as F2) (b) Input record C1, specifying early-stage values for constants. The question mark indicates that the actual constant value is unknown at this stage. (c) Input record C2, specifying late-stage values for constants.



(a)

```
int mul_add(x, y, a){
  static code *mul_add_Opt = NULL;
  if (mul_add_Opt == NULL) {mul_add_Opt = compile(mul_add_Compile(a));}
  return (*mul_add_Opt)(x,y);}
```

(b)

```
AST mul_add_Compile(a: int){
  switch (a) {
      case 0 :  return |mul_add(x,y,a){                          return y;}|;
      case 1 :  return |mul_add(x,y,a){              v = x * y;  return v;}|;
      case k :  return |mul_add(x,y,a){u = x * `k; v = u * y;  return v;}|;}}
```

(c)

FIGURE 2.7: Manual Staging by Compiler Users.
(a) Schematic for manual staging by compiler users (b) Version of mul_add function that optimizes before executing (c) A hand-written compiler pipeline customized to process F1 given C1.

We are now ready to discuss the three staging techniques based on preplanning.

2.6.2.1 Manual Staging by Compiler Users

Figure 2.7(a) shows one way in which staging could be achieved through preplanning. Essentially, instead of writing a *function F* which provides whatever functionality the application user may be interested in, the compiler (who writes the function and compiles it) instead writes both a *compiler for F* which when given late-stage information will produce a version of *F* optimized to that information, and a *driver* which invokes this compiled version of *F* [33, 51, 66, 62, 32].

Figure 2.7(c) shows what a compiler custom-written for the F1 and C1 of our example would look like. Essentially, the compiler user reasons about the effect of executing the compiler pipeline of figure 2.5 on the function `mul_add` of figure 2.6(a). When `a` has value 0, the three optimizations combine to leave only the return command in the function. When `a` has value 1, they combine to eliminate the first command in `mul_add`. Finally, in the case that `a` has some other value *k*, the returned function is the same as the input function, except that all uses of `a` may be replaced by the constant expression `k`.

The `mul_add_Compile` function in figure 2.7(c) has two constructs that are not part of conventional C code: fragments of code are surrounded by vertical bars |...|, and the reference to `k` is preceded by a backquote character **`**. These two features are standard constructs in meta-programming systems, which are designed to allow programmers to manipulate fragments of code from within programs. The vertical bars can be regarded as on operator which, given the code fragment they surround, generate AST's for that fragment, and the ` is an escape character, such that the expression `` `k `` evaluates to the value of the variable `k` in the scope immediately enclosing the bars surrounding the expression. For readers familiar with the Scheme [32] programming language || and ` are intended to have the same effect as quasiquote and comma and respectively.

When invoked at run time with a particular value of `a`, the `mul_add_Compile` function returns an AST that represents an optimized version of the function `mul_add`. In the process, it would have done much less work than a traditional compiler pipeline, since it just executes a single conditional before producing the appropriate version of code. Figure 2.7(b) shows the driver an application writer would write to invoke the custom optimization pipeline just described. A special function `compile` (called on line 3) first converts the resulting AST into binary format. The application then jumps to the binary code generated (line 4). In all subsequent calls to the `mul_add` function, the conditional ensures that no compilation will be performed, and control will pass directly to the optimized version of the function.

This example illustrates both the strong and weak points of the staging by compiler users. On the positive side, the custom compilers can be extremely fast, since the flow of control and data within them can be optimized by hand. They can also be very effective, since the hand-written compiler may exploit knowledge about the program no generic compiler can deduce. On the negative side, writing custom compilers using meta-programming support is extremely complicated and error prone. Having to reason about the possible behavior of pipelines of optimizations on a function is beyond the abilities of most compiler users (or compiler writers, for that matter), especially under the time constraints of building a typical application. The fully manual approach is therefore most suitable when a small, performance-critical part of an important program needs to be optimized using late-stage information.

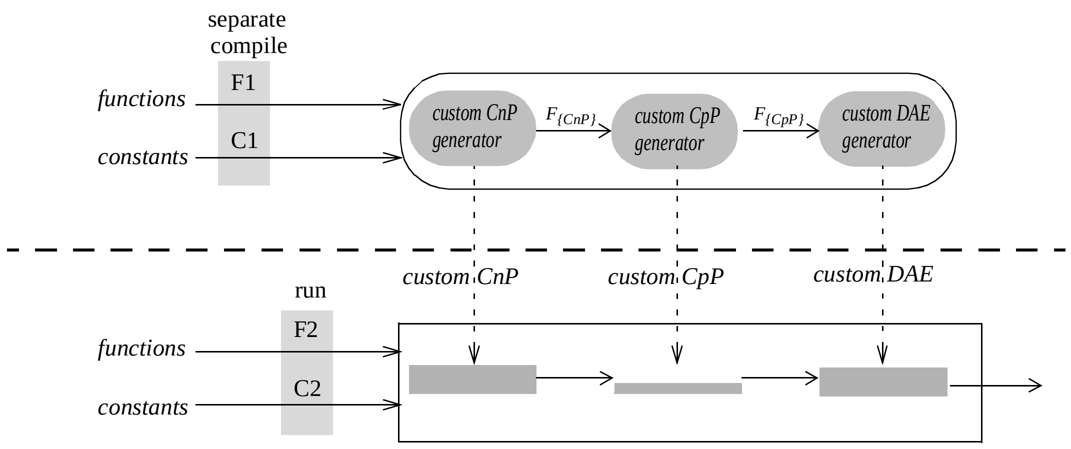
2.6.2.2 Manual Staging by Compiler Writers

An alternative to the compiler user having to write a custom compiler each time she writes a program is to have the compiler writer write a custom *optimization generator* each time she writes a compiler optimization. An optimization generator for a particular optimization would take as input partial information about the program fragment to be optimized and generate a version of the optimization customized to that partial information.

Figure 2.8(a) illustrates how an optimization generator works. Above the dotted line, in the rounded boxes is a sequence of optimization generators, one for each optimization we want to customize. The first optimization generator in sequence is for the constant propagation optimization. It takes as input the partial inputs to constant propagation available in the early stage, and produces two outputs. One output, sent downwards in the figure, is the constant propagator customized to *F1* and *C1* . The second, labeled $F_{\{CnP\}}$ and sent to the right, is a *partial description* of the possible actions performed by the customized constant propagator on the incoming function *F*.

The downward arrows that represent customized optimizations are dotted. We use dotted arrows between two boxes to indicate that one of the boxes either produces or consumes the other. A solid arrow between two boxes indicates that the box at the tail produces a value that is consumed by the box at the head. In this case, the rounded optimization-generator boxes above the dotted line generate the small rectangular boxes that comprise the late-stage compiler.

Figure 2.8(b) shows what the customized constant propagator may look like, using the meta-programming notation introduced in the previous section. The customized optimization consists of a fragment of code for each command of the incoming function. Each fragment conditionally produces a transformed version of the command it corresponds to, and optionally sets a variable (such as `u`) to an analysis value (such as `NON_CONSTANT`



(a)

```
AST mul_add_ConstantPropagator(int a){
  //Custom code for optimizing command 1
  if (a == 0) {u = 0;           c1 = |u = 0;|;}
  else        {u = NON_CONSTANT; c1 = |u = x * ‘a|;}
  //Custom code for optimizing command 2
  if (u == 0) c2 = |v = y;|)
  else        c2 = |v = u + y;|);
  //Custom code for optimizing command 3
  c3 = |return v;|;
  return |mul_add(x,y,z){ ‘c1 ‘c2 ‘c3 }|;
}
```

(b)

```
mul_add
(x_DYN, y_DYN, a_STATIC){
  u = x * a;
  v = u + y;
  return v;
}
```

F_1

```
mul_add
(x_DYN, y_DYN, a_STATIC){
  u = x *_FOLD*(a) a_FILL(a);
  v = u +_FOLD+(u) y;
  return v;
}
```

$F_{\{CnP\}}$

```
mul_add
(x_DYN, y_DYN, a_STATIC){
  u = x *_FOLD*(a) a_FILL(a);
  v = u_REPLACE(u) +_FOLD+(u) y;
  return v_REPLACE(v);
}
```

$F_{\{CpP\}}$

(c)

FIGURE 2.8: Manual Staging by Compiler Writers.

(a) Generating specialized optimizers with hand-written optimization specializers (b) Specialized constant propagator (c) Action annotated functions representing inputs to optimizations.

to indicate that the corresponding variable is not a constant). Analysis variables set for one command are consulted to determine the actions for downstream commands, much as a conventional optimization writes and reads fields of the abstract store when executing the flow function of a command.

The customized optimization thus obtained has a significantly lower overhead than an uncustomized optimization that achieves the same results. The uncustomized version would, for each command, have additional decode-and-dispatch code which branched on various fields of the command to determine which flow and transformation function to execute, followed by an update of a map data structure (instead of a single variable) representing the store, followed by a sequence of accesses to fields of the command that need to be reproduced in the transformed command. Replicating a flow function per command avoids all this overhead. However, comparing the automatically generated customized code for a single optimization phase of figure 2.8 with the hand-written custom code *for the whole pipeline* of figure 2.7, we see that hand customization can potentially produce still faster custom compilers than automatically customized versions.

We have described the process of customization as essentially that of stamping out, in the early stage, a specialized optimization function for each command in the input function. However, customization cannot require that all commands in the input function are fully known at the early stage. In our example, the customizer for constant propagation does indeed have a fully known input function *F*. However, this is not the case with the customizers for copy propagation and dead assignment elimination. In fact, the input functions to these phases will be generated only at the run-time stage, by executing the customized constant propagation of figure 2.8(c). Instead of the particular input function (which will only be known at run time), all these customizers can hope to have at the early stage is a description of the *possible* functions that may result from the execution of the custom optimizations produced by generators before them in the pipeline.

Staging pipelines of optimizations is not a topic much studied before this dissertation. Figure 2.8(c) shows the only method investigated previous to SCF [24] for describing at the early stage the potential results of executing customized optimizers. $F_{\{CnP\}}$ describes the potential results of the constant propagation phase (which will be input to the copy propagator), whereas $F_{\{CpP\}}$ describes the potential results of the constant and copy propagator in sequence, which will be input to the dead code eliminator. For completeness, F_1 shows how the early inputs to the constant propagator are described in this scheme.

The basic technique used to indicate the effect of an optimization is to annotate the incoming function with *action annotations* specific to that optimization which indicated what actions the optimization could take on the function. In figure 2.8(c), for instance, the function $F_{\{CnP\}}$ is annotated with three annotations, `FILL(a)`, `FOLD*(a)` and `FOLD+(u)`. The first annotation is intended to indicate that if the variable `a` has a suitable value, it may be replaced in the later stage by its value. The second annotation indicates that given a suitable value of `a`, the multiply operation may be folded away. The third annotation indicates that the addition operation may be folded away if the value of `u` is suitable (specifically, zero).

The customizer for the copy propagation optimization, given an annotated input, reasons about the possible functions that could result if these incoming annotations were executed, simulates the action of copy propagation on these possible results, and adds its own annotations to the function to indicate the possible additional effects of copy propagation. The annotations added by copy propagation are underlined in the function labelled $F_{\{CpP\}}$ in figure 2.8(c). The annotations $\underline{\texttt{REPLACE(v)}}$ and $\underline{\texttt{REPLACE(u)}}$ are intended to indicate that if `v` (resp. `u`) are copies at the late stage, they will be replaced by the variables they are copies of. The customizer for dead-assignment elimination now has to contend with the function annotated with action-annotations from both upstream customizers.

Downstream customizers thus have to be aware of the semantics of the annotations placed by each upstream customizer, and need to reason about the interaction of the effects of these annotations. The burden on the downstream customizer increases rapidly, to the point that engineers designing downstream customizers need to make conservative estimates of the interaction of effects. In fact, with just three phases, the implementation of the action-annotation based pipeline described above sacrificed opportunities for optimizations in the second and third phases of the pipeline. The coupling between optimizations leads to a system that does not scale very well beyond two or three optimizations. Given that modern compilers can easily have more than ten optimization phases, this limited scaling is a serious problem.

Writing generators for customized optimizations, as described in this section, does help in making staged compilation more suitable for widespread use compared to the completely manual approach described in the previous section. The application writer (or compiler user) can now stick to writing applications rather than writing additional custom compilers for applications. The optimization writer, who presumably also writes the customized optimization generator, only needs to do so once for each optimization he writes, an event that happens far less frequently than writing an application to be compiled on the compiler.[1]

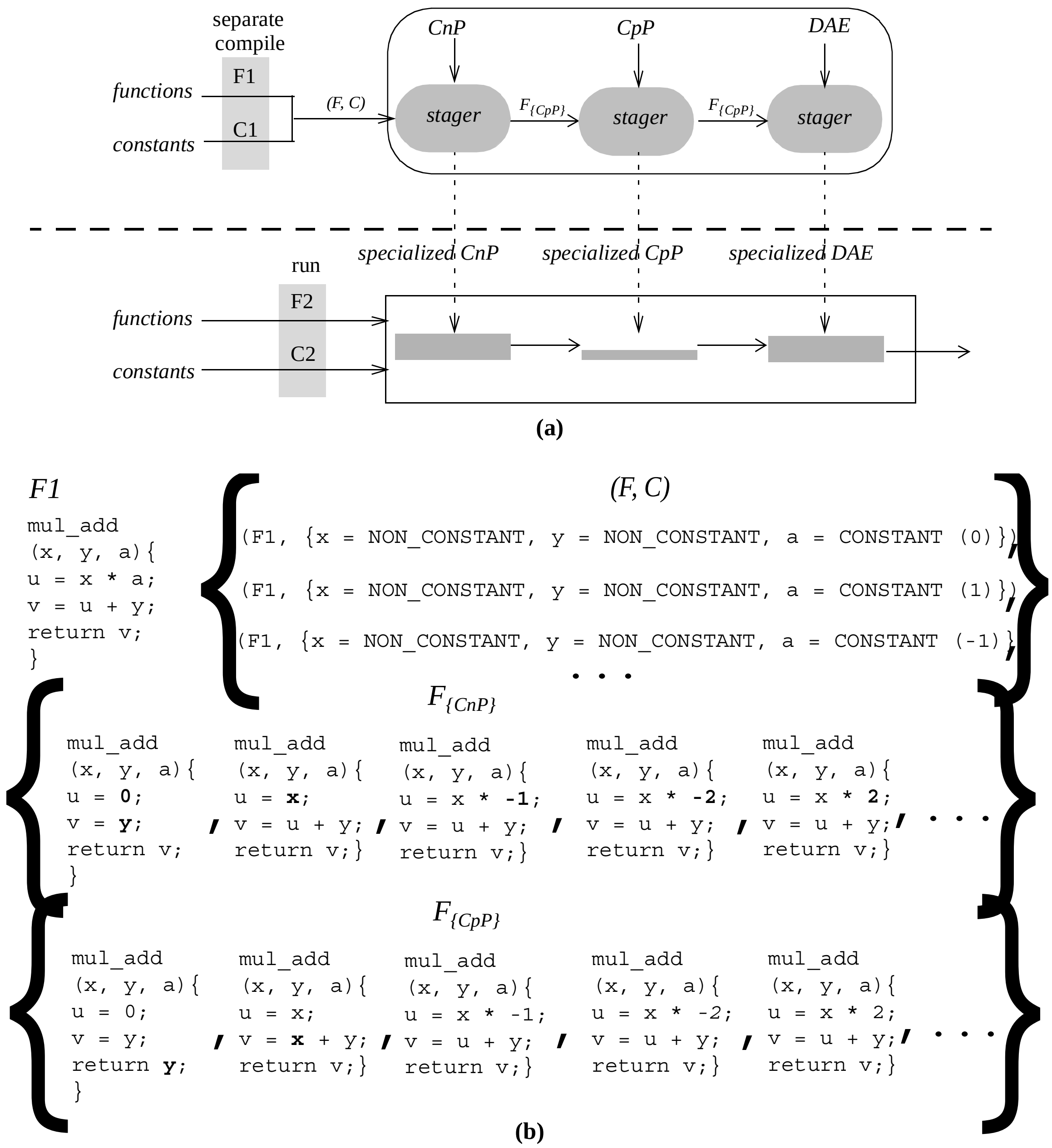
Unfortunately, writing generators in this manner is enough of a black art that each generator written until now has merited a research paper [68, 38, 14, 8, 24]. In practice, writing each optimization generator has taken one or more man years of work by dedicated and highly specialized engineers. Our experience (and informally, that of others) indicates that without substantial progress both in understanding their nature, and automating the process of generating them, generating custom optimizations will remain impractical for mainstream compilers.

In summary, at least based on the state of the art preceding this dissertation, writing one optimization generator per optimization in the compiler was impractical for most compilers. Further, pipelining optimization generators was even less feasible. The one previous proposal of using action annotations, which was work leading to this dissertation, does not scale to conventional compiler pipelines.

2.6.2.3 Automatic Staging

This dissertation presents a system (called the Staged Compilation Framework, SCF) which produces customized versions of pipelines of optimizations while allowing compiler writers to stick to writing optimizations, and compiler users to writing applications. They neither have to write optimization generators, nor have to reason about the effect of previous optimization phases as methods in the previous section did. By showing how custom optimizations can be generated automatically and composed modularly, the dissertation also casts light on the essence of generating custom optimizers. An optimization writer who wishes to write an optimization generator can hopefully benefit from learning how an automatic optimization generator works.

1.Unless the compiler is a research compiler!



```
F1
mul_add
(x, y, a){
u = x * a;
v = u + y;
return v;
}
```

```
(F, C)
{
(F1, {x = NON_CONSTANT, y = NON_CONSTANT, a = CONSTANT (0)}),
(F1, {x = NON_CONSTANT, y = NON_CONSTANT, a = CONSTANT (1)}),
(F1, {x = NON_CONSTANT, y = NON_CONSTANT, a = CONSTANT (-1)}),
. . .
}
```

```
F{CnP}
{
mul_add      mul_add      mul_add        mul_add        mul_add
(x, y, a){   (x, y, a){   (x, y, a){     (x, y, a){     (x, y, a){
u = 0;       u = x;       u = x * -1;    u = x * -2;    u = x * 2;
v = y;     , v = u + y; , v = u + y;   , v = u + y;   , v = u + y;, . . .
return v;    return v;}   return v;}     return v;}     return v;}
}
}
```

```
F{CpP}
{
mul_add      mul_add      mul_add        mul_add        mul_add
(x, y, a){   (x, y, a){   (x, y, a){     (x, y, a){     (x, y, a){
u = 0;       u = x;       u = x * -1;    u = x * -2;    u = x * 2;
v = y;     , v = x + y; , v = u + y;   , v = u + y;   , v = u + y;, . . .
return y;    return v;}   return v;}     return v;}     return v;}
}
}
```

FIGURE 2.9: Using an Automatic Optimization Specializer.
(a) Steps in using an automatic specializer (b) Functions and auxiliary information used and produced.

Figure 2.9 shows how SCF works. It is useful to compare figure 2.9(a) with figure 2.8(a) of the previous section. They key differences between the figures are all above the dotted line separating the compile-time stage from the run-time stage. For one, instead of having a distinct rounded box for each optimization (each of which generated a custom version of an optimization), we now have three instances of a single rounded box, labelled *stager*, which can generate all the customized optimizations. The stager takes two inputs. The first, which enters the stager from above in figure 2.9(a), is an ordinary optimization, and is all an optimization writer has to write. The second, which enters from the left, is an early-stage description of the possible late-stage inputs to the optimization. The stager produces two outputs. The first, the customized version of the optimization is carried downwards by the dotted arrow as in the previous section. We will not further discuss the structure of the specialized compiler in this section, although it is similar to that of figure 2.8(c) of the previous section, and it will discussed in great detail in the next chapter.

The second output, which goes to the right (and is the input to the next invocation of the stager), captures the possible versions of the input function that can result from the optimization at run time. The crucial difference is the representation of these potential results. In the previous section, the effect of an optimization was represented at the early stage by action annotations that (in an optimization-specific way) indicated the action with which each syntactic phrase of the input function may be transformed. As shown in figure 2.9(b), SCF represents the effect by the *set of values* which may result from the execution of optimization at the later stage. These early-stage descriptions of potential later inputs to the phases are labelled $F_{\{CnP\}}$ and $F_{\{CpP\}}$ respectively in the figure.

(F,C) is the early-stage description of the possible later inputs to the entire pipeline. *F* is the function `mul_add` as before. *(F,C)* is the set of pairs such that the first element of the pair is the function *F* and the second is a record with fields *x* and *y* bound to `NON_CONSTANT` and field *a* bound to value `CONSTANT(i)`, for all integer `i`. Similarly, when constant propagation is performed on some pair *(F,c)* from *(F,C)*, it will yield a function which is in the set labelled $F_{\{CnP\}}$. One of these functions will in turn be input to the copy propagator at the late stage, so that the output from the constant propagator will be a member of the set $F_{\{CpP\}}$.

The sets of possible outputs and inputs are represented explicitly as shown in figure 2.9(b) instead of implicitly via action annotations. Dropping the optimization-specific action annotations of the previous section in favor of sets of values frees the stager from reasoning about the semantics of, and interactions between, annotations inserted upstream.

As long as the stager is able to take an optimization and a set of possible arguments to that optimization, and produce the set of possible result values for the optimization, instances of stagers can be chained as shown in the figure with each stager in the chain having no knowledge of what phase precedes it, and what succeeds it.

In the rest of the dissertation, we will describe in detail the structure and functioning of the stager.

2.7 Summary

Staged compilation has the advantage over unstaged compilation in that it can shift some of the overhead of compilation from the late stages to the early stages. As a result, it can potentially complement the benefit from traditional optimization-speeding techniques such as approximation, summarization and selective optimization. Procrastination-based staged compilation allows the early execution of all optimizations in a pipeline if all inputs to these optimizations are available, and defers the rest for later stages. Pessimism-based staging sacrifices optimization opportunities to allow early execution of even those optimizations whose inputs may not be all available. Preplanning-based staging has the potential advantage of allowing early stage partial execution of optimizations whose inputs are not available without necessarily losing optimization opportunities. It does so by producing a plan for fast completion of optimizations in later stages, when more information is available. Summarization can be viewed as a staging technique that applies to analyses, but not pipelines of optimizations. Proposed techniques for preplanning-based staging of whole pipelines of optimizations have suffered from unacceptable engineering complexity. The automatic technique presented in this dissertation is aimed at signficantly lowering the complexity of preplanning-based staging.

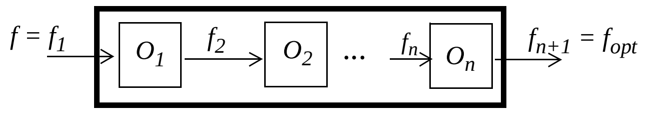
3. Overview and Example

This chapter presents a high-level view of SCF. First, we see via a schematic how the framework may be used for staging an arbitrary pipeline of compiler optimizations. The schematic emphasizes the role of the *stager* (introduced in the previous chapter) as the engine which stages optimizations and is therefore the heart of SCF. Second, we continue the running example of the previous chapter by first describing the form of inputs to the stager, and then showing concretely how the stager is invoked on these inputs. Third, we examine the internal structure of the stager. Internally, the stager consists of two main parts, a *partial evaluator* and a *dead-store eliminator*. We use our example to show the effect a basic version of each of these parts has on its inputs, and to show what the output of the stager as a whole is, on our running example. Succeeding chapters show how the stager produces these outputs, and how they can be improved.

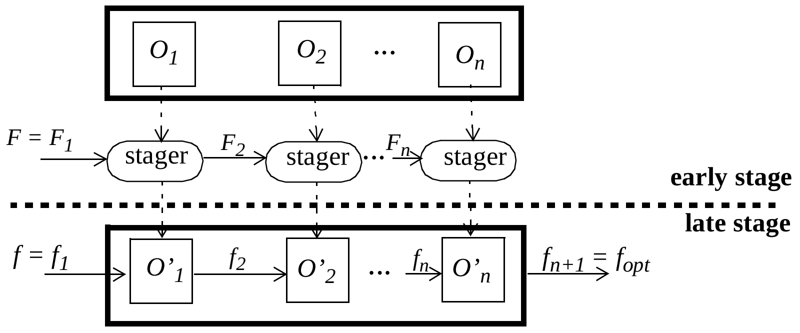
3.1 High-Level Description of SCF

Figure 3.1(a) illustrates the compiler writer's view of SCF. He writes a set of optimizations O_i, some subset of which may be sequenced to form a conventional optimizing compiler pipeline $O_1 \ldots O_n$ that takes in a function f to produce an optimized program f_{opt}. Each optimization O_i is a function transformer that takes in an arbitrary data structure f_i (typically a tree representation of the function to be optimized and possibly additional context information about the properties of the function's formal parameters or the characteristics of the target machine onto which the function is to be compiled) and produces a transformed data structure f_{i+1}. Writing these optimizations should be the same amount and kind of work as writing an optimization pipeline in a regular unstaged compiler.

Figure 3.1(b) illustrates the compiler user's view of staging an initial pipeline of optimizations $O_1 \ldots O_n$ with respect to a partial description F of its eventual inputs f, to produce a specialized pipeline $O'_1 \ldots O'_n$. The partial description F defines the set of possible inputs (i.e., functions and context information) on which the specialized pipeline might be invoked. The specialized pipeline can then be run on any input f that is a member of the set described by F, to produce a corresponding optimized function f_{opt}.



(a) The Compiler Writer's View of SCF



(b) The Compiler User's View of SCF

FIGURE 3.1: High-Level View of SCF.
Solid arrows represent producer-consumer relationships between boxes. Dotted arrows represent input-consumer and producer-product relationships: the stager consumes optimizations O_i and produces optimizations O'_i.

Each specialized optimization O'_i is what is left of the original optimization O_i after all the parts of its work that can be precomputed based solely on information in F_i have been performed; the work that remains will finish the optimization when it is finally given the complete function and context information f_i. When staging a whole pipeline, a pipeline of specialized optimizations is produced. The specialized pipeline $O'_1 \ldots O'_n$ can be run just like the original pipeline $O_1 \ldots O_n$, with exactly the same result f_{opt}, as long as its input f is a member of the set of expected inputs described by F.

The stager does not *run* its input optimization, but rather takes as input the *source code* of the optimization O_i and the partial description F_i of the optimization's possible inputs to produce the *source code* of the specialized optimizer O'_i and a partial description F_{i+1} of the optimization's possible outputs. In the second stage, the specialized optimization pipeline $O'_1 \ldots O'_n$ can be *run* on an input function f to produce an optimized function f_{opt}.

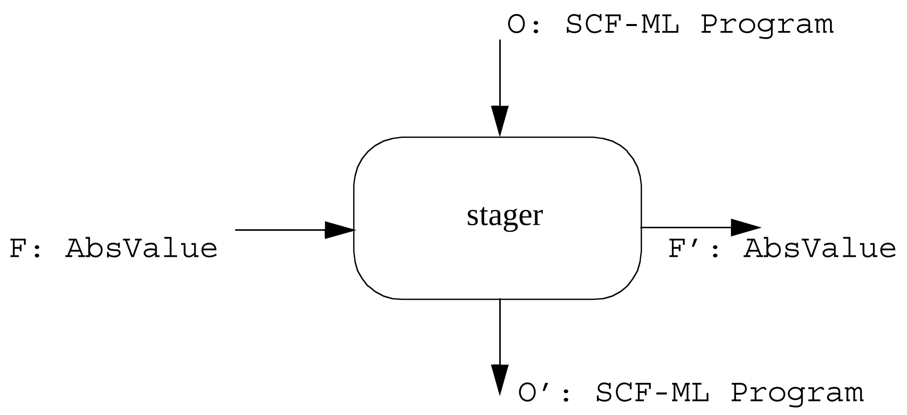


FIGURE 3.2: Signature of the Stager.
The `AbsValue` datatype is defined in table 1, and the SCF-ML `Program` datatype is defined in figure 3.3.

3.2 The Interface to the Stager

Figure 3.2 shows the interface to the stager discussed in the previous section in more detail. *O* and *O'* are the optimization to be specialized and its specialized version, respectively, and *F* and *F'* are the partial descriptions of the possible late-stage inputs and outputs, respectively, of *O'*. So far, we have left the precise format of the *O*s and the *F*s unspecified. In this section, we specify precisely the format of these values and illustrate them with examples.

When we need to use a specific optimization pipeline as an example, we will pick the pipeline of figure 2.9(a), i.e., O_1, O_2 and O_3 are constant propagation, copy propagation and dead-assignment elimination, respectively. For input information *I*, we will use variants of the `mul_add` function and associated constant propagation information of figure 2.9(b).

3.2.1 SCF-ML: The Language for Specifying Optimizations

Optimizations are specified in a first-order, purely functional subset of ML called SCF-ML. SCF-ML provides no support for exceptions. Figure 3.3 specifies the concrete syntax for this language. All valid SCF-ML programs are valid Standard ML [43] programs, with identical semantics.

```
1  P  ∈ program    ::=  M_1...M_n
2  M  ∈ moduleDef  ::=  structure id = struct d_1...d_m g_1...g_n end
3
4  //Type and module definitions
5  d  ∈ definition ::=  datatype dt_n and ... and dt_n
6                       | type id = t
7                       | structure id = MapFn(type key = t_1 type value = t_2)
8                       | structure id = SetFn(type value = t)
9                       | open id
10 dt ∈ datatype   ::=  id = id_1 [of t_1]|...| id_n [of t_n]
11 t  ∈ type       ::=  int | bool | char | string | [id.]set | [id.]map | tn
12                      | (p)
13 p  ∈ prodType   ::=  t_1 *...* t_n
14 tn ∈ typeName   ::=  id | id.tn
15
16 //Function declarations
17 g  ∈ funcGroup  ::=  fun fd_1 and ... and fd_n
18 fd ∈ funcDef    ::=  id pt = e
19 e  ∈ expr       ::=  x | k | (e_1,...,e_n)
20                      | case e of m_1 | m_2 | ... |m_n
21                      | let b_1...b_n in e end
22                      | if e_1 then e_2 else e_3
23                      | SOME e | f e | pr e | e pr e | cf (fn pt => e_1) e_2
24
25 //Value bindings
26 b  ∈ Bindings   ::=  val pt = e
27
28 //Pattern matching
29 m  ∈ match      ::=  pt => e
30 pt ∈ pattern    ::=  _ | id | k | f pt | (pt_1,...,pt_n) | id as pt
31
32 //Primitive domains
33 pr ∈ Primop     ::=  [id.]insert | [id.]find | [id.]remove | [id.]equal
34                      | [id.]union | [id.]delete | [id.]add | [id.]member
35                      | + | - | * | ...
36 cf ∈ CurriedFun ::=  [id.]unionWith | [id.]map
37 k  ∈ Constants  ::=  [id.]empty | NONE | c
38 f,x∈ QualifiedId::=  id | id.x
39 c  ∈ BaseConst  =    Int ∪ Bool ∪ String ∪ Char
40 id ∈ Identifier
```

FIGURE 3.3: Concrete Syntax for SCF-ML Programs.

Briefly, programs (line 1) consist of a set of module definitions; in the style of ML, modules are called *structures* in SCF-ML. Structures (line 2) consist of a sequence of (type and module) definitions followed by a sequence of function declarations. Type definitions (line 5) include definitions of datatypes (line 5) and the definitions of type synonyms (line 6). Datatype definitions define possibly recursive sum types (analogous to union types in C). Type synonyms may be associated with product types (line 13; these are analogous to struct types in C), base types (line 11; note the two special base types `set` and **`map`** that are pre-provided by SCF) and other type names. In general, types may be prefixed by the names of modules where they are defined (lines 11 and 14). For instance, the type `ArrayList.list` would refer to the `list` type defined in the `ArrayList` module. SCF-ML restricts the definition of modules within a module to those representing map data structures, obtained by applying the predefined **`MapFn`** functor (line 7), and those representing sets, obtained by applying **`SetFn`** (line 8). For instance, **`structure IntBoolMap = MapFn(type key = int type value = bool)`**, defines a map from integers to booleans. Finally, SCF-ML allows modules to be opened (line 9), so that their contents are visible within the current module.

Function definitions bind function names to expressions that constitute the body of the functions. Expressions are variables, constants, constructor applications (which create a value of a datatype with a particular tag), tuple constructors (which create a value of a product type), case expressions, let expressions, conditionals, function calls, primitive operation calls, and a special form for applying *map* and *union* operations on map data structures (lines 23 and 35). As in ML, case expressions in SCF-ML perform pattern matching and binding of variables. For instance, given a pair `x` of integers, `case x of (1,y) => y | (-1, y)=> -y| _ => 0` evaluates to the second element of the pair, its negation, or to zero, when the first element is *1*, *-1* and *0* respectively. SCF-ML also provides built-in primitives (lines 33 and 34) for manipulating map and set data structures in addition to the usual primitive operations such as addition and subtraction. The built-in constructor **`SOME`** and the constant **`NONE`** represent values resulting from map

```
datatype int_list =      (* Declare a sum-type with two variants *)
  Empty                  (* A zero-ary variant i.e. scalar *)
| Cons of int * int_list (* A binary variant *)

fun reverse l = reverseHelper(l,Empty)

and reverseHelper(lInput, lResult) =
  case lInput of
    Empty  =>      lResult
  | Cons(i, lInput') =>
      let val lResult' = Cons(i,lResult)
      in reverseHelper(lInput',lResult') end
```

FIGURE 3.4: SCF-ML Program for Reversing a Linked List of Integers

operations. A map lookup for an key not in the map will evaluate to *NONE*, whereas if the key maps to value *v*, it will evaluate to *SOME v*.[1]

For the benefit of those unfamiliar with ML, figure 3.4 shows how a program to reverse a linked list of integers is written in SCF-ML. Comment syntax is similar to that of C, except that the delimiters *(** and **)* are used instead of */** and **/* respectively. It is legal to have apostrophes in SCF-ML identifiers. Thus, instead of variables named `lResult_1` and `lResult_2`, we have `lResult` and `lResult'`.

Built-in support for the map and set data structures is an important feature of SCF-ML. Built-in maps and sets currently have the limitation that they assume that the equality function on map keys and set elements respectively is structural equality. Otherwise, the semantics of the map data structure operations are standard. In particular, the operations and values have the same meaning as those of the `ORD_MAP` and `ORD_SET` interfaces of the SML/NJ Utility Library [7]. Figure 3.5 illustrates the semantics of the built-in map

```
- structure IntMap = MapFn(type key = int type value = int)
  structure IntMap : sig (* signature for SCF-ML maps here*) end

- IntMap.empty
  <>

- val m = IntMap.insert(1,19, IntMap.insert(2,14,IntMap.empty))
  val m = <1->19, 2->14>

- IntMap.find(m,2);
  SOME 14

- IntMap.find(m,3);
  NONE

- IntMap.equal(m, IntMap.empty);
  false

- val m' = IntMap.map (fn x => x * x) m
  val m' = <1->361, 2->196>

- val m'' = IntMap.unionWith (fn (x,y) => if (x > 200) then x else y) (m,m')
  val m'' = <1->19, 2->196>
```

FIGURE 3.5: Map Operations in SCF-ML.

1.The two constructors are identical to those of the *Option.option* type in the Standard ML Basis Library. They are built-in here because maps themselves are built-in.

```
structure AST = struct
type label = int

datatype prog     = prog of id * funs * label

and      funs     = funs_empty
                    | funs of func * funs * label

and      func     = func of func_signature * id * vars * cmd * label

and      cmd      = skip of label
                    | seq of cmd * cmd * label
                    | assign of var * expr * label
                    | decl_cmd of decls * cmd * label
                    | if_else of expr * cmd * cmd * label
                    | switch of expr * cases * label
                    | while_do of expr * cmd * label
                    ...
and ...

end
```

FIGURE 3.6: Defining an Intermediate Representation for a Subset of C in SCF-ML.

operations. The figure shows the results printed by a (hypothetical) SCF-ML interpreter when various map operations are typed in at the prompt (`-`).

Line 1 shows how a map from integers to integers may be defined using the **`MapFn`** functor. As is conventional with Standard ML map lookups, on a **`find`** operation, if the key to be looked up maps to some value *v* in the map, the operation returns the value *SOME v*, otherwise it returns the value *NONE* (lines 11 and 14 respectively in the figure). The **`map`** operation (line 19) takes an anonymous function *f* as first argument, and returns the map that results from applying *f* to each value in the range of the map. The **`unionWith`** operation (line 22) also takes an input function *f* in addition to the maps to merge. The operation returns the union of its argument maps: if some key *k* is in only one of the maps being operated upon, and it maps to value *v*, the mapping *(k, v)* is transferred to the result map. If some key *k* maps to values *v* and *v'* in the two input maps, the mapping *(k, f(v,v'))* is transferred to the result map.

We now turn to how SCF-ML may be used to specify optimizations. We choose to use abstract syntax trees (AST's) as our internal representation for programs. Figure 3.6 shows part of the datatype definition for AST's (intended to represent C functions) used in SCF. A possibly puzzling detail is that every AST node has an integer label attached to it. The label for each node is intended to be used as a handle for associating dataflow facts with that node. Although we expect that most compiler writers will use this pre-defined AST definition, there is nothing to prevent them from defining their own intermediate representation (perhaps because they are representing a language other than C, or they want to use a different internal representation than AST's).

In our example three-optimization pipeline, the compiler writer must write the three optimizations in the pipeline in the SCF-ML language. Figure 3.7 shows how dead-assignment elimination may be written in SCF-ML.

Line 1 of the figure indicates that the code for dead-assignment elimination is being encapsulated in a module named `DAE`. Line 2 opens the `AST` module (defined partially in figure 3.6) to read in the type definitions for the input program abstract syntax tree. Types such as `var`, `label`, `func`, `cmd` and `expr` used at line 14, 21 and 39 respectively are defined in this module. Lines and 7 and 11 define modules by instantiating parameterized modules (called *functors* in Standard ML). Line 7 defines a module `LiveSet` representing the set of live variables maintained by the analysis. Line 12 defines the module `AssignMap`, which is a map from labels on commands in the input AST to a lattice value `live` or `dead`. Lines 12 and 26 illustrate how components of modules can be accessed from other modules: `AssignMap.map` refers to the type named `map` defined in the `AssignMap` module, and `LiveSet.`**`delete`** is the set delete function in the `LiveSet` module.

The entry point to the optimization is the function `optimize`. Note that the figure deviates slightly from the SCF-ML syntax specified in figure 3.3 in that it allows function definitions to be decorated with optional input and return types. As per line 14, for instance, the `optimize` function takes an input of type `func` and evaluates to a value of the same type. The optimization itself is divided into an analysis pass (function `analyze-Fun`) followed by a transformation pass (function `transformFun`).

The analysis pass is a backwards dataflow analysis that computes, using a threaded set (`lSet`) of live variables, and a map (`aMap`) which indicates whether each assignment in the incoming AST is live or dead. Consider, for instance, lines 24-28 of the figure, which show how the analysis processes an assignment statement of the form `v = e` with label

```
1  structure DAE = struct
2  open AST     (* Import declarations of input program representation*)
3
4  datatype liveness = dead | live
5
6  (* LiveSet holds the set of live variables at a program point *)
7  structure LiveSet = SetFn(type value=var)
8  type live_set = LiveSet.set
9
10 (* AssignMap records whether each assignment is live or dead at a program point *)
11 structure AssignMap = MapFn(type key=label type value=liveness)
12 type assign_map = AssignMap.map
13
14 fun optimize(f:func):func =
15   transformFun(f, analyzeFun(f))
16
17 and analyzeFun(func(_,_,c,_):fun):assign_map =
18   let val (_, aMap) = analyzeCmd(c, LiveSet.empty, AssignMap.empty)
19   in aMap end
20
21 and analyzeCmd(c:cmd, lSet:live_set, aMap:assign_map)
22               :(liveSet * assign_map) =
23   case c of
24     assign(v, e, lbl) =>
25       let val lv = if LiveSet.member(lSet, v) then live else dead
26       in (analyzeExpr(e, LiveSet.delete(lSet, v)),
27           AssignMap.insert(aMap, lbl, lv))
28       end
29   | seq(c,c',_)        =>
30       let val (lSet,aMap) = analyzeCmd(c',lSet,aMap)
31       in analyzeCmd(c,lSet,aMap) end
32   | return(e,_)        =>
33       (analyzeExpr(e, lSet), aMap)
34   | while_do(e, c, _)  =>
35       analyzeWhile(lSet, lSet, aMap, e, c)
36   | . . . other cmd cases here . . .
37
38 and analyzeExpr(e:expr, lSet:live_set):live_set =
39   case e of
40     var_ref(v, _)      => LiveSet.add(lSet, v)
41   | primop(op,es)      => analyzeExprs(es, lSet)
42   | const _            => lSet
43   | . . . other expr cases here . . .
44
45 and analyzeExprs(es:exprs, lSet:live_set):live_set=
46   case es of
47     exprs(e,es)        => analyzeExprs(es,analyzeExpr(e,lSet))
48   | exprs_none         => lSet
```

FIGURE 3.7: Specifying Dead Assignment Elimination in SCF-ML.

`lbl` (written `assign(v,e,lbl)` in the figure). In line 25, the analysis checks if `v` is in the set of downstream live variables computed by the analysis so far. If so, the analysis

```
49 and analyzeWhile(live_fix:live_set, live_exit:live_set, assigns:assign_map,
50                  test:expr, body:cmd)
51       :(live_set * assign_map) =
52   let val live_head             = analyzeExpr(test, live_fix)
53       val (live_taken, assigns')= analyzeCmd(body, live_head, assigns)
54       val live_fix'             = meet(live_taken, live_exit)
55   in
56     if LiveSet.equal(live_fix, live_fix') then
57       (live_head, assigns')
58     else
59       analyzeWhile(live_fix', live_exit, assigns', test, body)
60   end
61
62 and meet(live1:live_set, live2:live_set):live_set =
63   LiveSet.union(live1, live2)
64
65 and transformFun(func(fname, formals, c, lbl):func,
66                  aMap:assign_map):func =
67   func(fname, formals, transformCmd(c, aMap), lbl)
68
69 and transformCmd(c:cmd, assigns:assign_map):cmd =
70   case c of
71     assign(v, e, lbl) =>
72       case AssignMap.find(assigns, lbl) of
73         SOME dead => skip(lbl)          (* replace with empty cmd *)
74       | SOME _    => c                  (* leave unoptimized *)
75   | seq(c, c', lbl) =>
76       seq(transformCmd(c,aMap),transformCmd(c',aMap),lbl)
77   | while_do(e, c, lbl) =>
78       while_do(e, transformCmd(c), lbl)
79   | return _ => c
80   | ... other cmd cases here ...
81
82 end (* structure DAE *)
```

FIGURE 3.7 (continued): Specifying Dead Assignment Elimination in SCF-ML

deduces that the assignment is live (and sets `lv` to liveness value `live`), otherwise it deduces the assignment is dead (and sets `lv` to `dead`). In line 26, it removes `v` from the set of live variables and then checks the right hand side of the assignment (via the call to `analyzeExpr`) to add all variables used there to the set of live variables; this set is returned as the set of variables live before the assignment. Finally in line 27, the analysis updates (in the functional sense) `aMap` to record that the statement labelled `lbl` has liveness value `lv`.

Lines 75 through 79 perform the actual pruning of dead assignments. Given assignment `assign(v,e,lbl)` the optimization looks up the liveness value associated with label `lbl` in the assignment map (line 77). If the assignment has been deemed dead by the analysis, it is replaced by a no-op (the `skip` command of 78), otherwise the command is left unchanged (line 79).

The example illustrates some important points. First, optimizations are typically divided into analysis and transformation functions. Second, the analysis function typically computes a map from program labels to properties that hold at that label. This map is then consulted by the transformation. This map, and related ones, are propagated throughout the program. Third, two classes of recursive functions are prominent in optimizations. The functions that analyze individual commands and expressions often call each other recursively. In this case, the recursive call has as its argument a sub-command/expression of the command/expression that was the argument to the original call. On the other hand, commands that cause iteration (such as the `while_do` command) are processed by recursive fixpoint functions (such as the `analyzeWhile` function of the figure) where the recursive call to the fixpoint function is on the same command as the initial call. SCF contains machinery to handle accurately these two kinds of recursive calls. Finally, the optimization is specified quite naturally using SCF-ML. This has been our experience with other intraprocedural analyses we have implemented in SCF-ML.

We end this section with a few details on the other two optimizations in the pipeline. These details will be useful in understanding how exactly the stager is invoked at the early stage. In line with the example of chapter 2 (see figure 2.3 for instance), we assume that the constant propagation optimization (implemented, say, in module `CnP`) has a slightly non-traditional interface. The optimization uses the traditional constant-propagation lattice with elements that belong to the data type `lattice_val = NON_CONSTANT | CONSTANT of int | UNDEFINED`. However, in addition to the body of the function *f* to be optimized, the entry function `CnP.optimize` of the constant propagator takes as argument a list of bindings providing the constant-propagation lattice values to which the formals of *f* should be bound at the beginning of constant propagation on *f*, i.e., `CnP.optimize: AST.fun * (CnP.lattice_val list) -> AST.fun`. Finally, we assume the copy propagation pass is implemented as a conventional function-to-function transformer in module `CpP`, i.e., `CpP.optimize: AST.fun-> AST.fun`.

```
An integer:
  1

The empty list []:
  nil

The list [12]:
  cons(12, nil)

A tuple with three elements, the second of which is a tree:
  (1, tree(13, empty_tree), false)

An association list-based implementation of the map [1 -> 19, 2 -> 14]:
  cons((1,19), cons((2,14), nil))

A representation of the command x = y + z:
  assign(var("x"),primop(op_add,exprs(var("y"), exprs(var("z"), expr_none))))
```

FIGURE 3.8: Concrete Values That May Be Defined in SCF-ML Programs.

3.2.2 Augmented Regular Tree Expressions

In the previous section, we discussed how one of the inputs to the stager, the optimization program, is represented using the SCF-ML language. In this section, we specify how the other input, the abstract value that captures possible late-stage inputs, is represented.

As discussed in section 2.9, the abstract value represents the set of possible values that may be input to the optimizer at the late stage. A representation for the abstract value must therefore be able to represent sets containing any input to an optimization program. Now since an optimization may be an arbitrary SCF-ML program, its input may be any value (called a concrete value) an SCF-ML program may manipulate. To understand the requirements on abstract values, we must therefore first understand the structure of concrete values in SCF-ML.

We can use the structure of SCF-ML types (as specified in figure 3.3) as a guide to the structure of SCF-ML values. Since the type of an expression in SCF-ML is either a primitive constant (integer, boolean or string), a sum-type, or a product type, all values manipulated by SCF-ML are either primitive constants, values tagged by constructor names, or tuples of values. Figure 3.8 illustrates the kinds of concrete values manipulated by SCF-ML. The figure is intended to convey the fact that a combination of tuples, tagged variants and atomic types is sufficient to encode most data structures including lists, maps, trees and program fragments in a straightforward fashion.

3.2.2.1 The Concrete Domain

Intuitively, the concrete values manipulated by SCF-ML programs are primitives, tuples and tagged values. Formally, the concrete value domain is the Herbrand Universe *H*, given by the smallest set satisfying the equation:[1]

$$H = \{c(t_1,...,t_{arity(c)}) \mid c \in Constructor \wedge t_i \in H\} \cup \{(t_1,...,t_n) \mid n > 1 \wedge t_i \in H\}$$

Integer, boolean, character and string constants are viewed as constructors with arity 0.

For concreteness, assume that built-in maps have an association list-based implementation[2]. In ML, one may write:

```
type (''a,'b) map = (''a * 'b) list
```

Sets may be implemented by maps representing their membership function, and will not be discussed explicitly below:

```
type ''a set = (''a, bool) map
```

Built-in maps and sets therefore have a concrete representation in *H*. In what follows, we will always print concrete maps as association lists, although in practice SCF uses a more efficient internal representation for maps.

3.2.2.2 Syntax and Semantics of Abstract Values

Given that we are trying to represent sets of possible concrete values using abstract values, an abstract value represents a member of the set $AbsValue = 2^H$, the powerset of the above concrete domain. Much prior work in abstract interpretation and set-based program analysis [54, 31, 4, 26] has gone into formalisms for representing this domain. We choose as our representation a modified version of the Regular Tree Expression (RTE) representation of

1.This is a slightly augmented form of the traditional Herbrand Universe since the latter does not contain untagged tuples.
2.Those familiar with ML will notice that the type parameter `''a` is an equality type, i.e., equality on these values is given by structural equality. In fact, built-in maps assume the equality testing criterion between two concrete values that are map keys is structural equality. The user cannot define his own equality function over keys. We have not found this restriction to be too onerous. On the other hand, the precision of abstract map operations in SCF depends crucially on this restriction.

Table 1: Augmented Regular Tree Expressions (Abstract Values): Syntax and Semantics
The set $\underline{v}$ of concrete values represented by a closed abstract value v is $\cup_{\gamma \in I \times H} \Psi(v, [], \gamma)$; $[]$ is the empty substitution.

	$v \in AbsValue ::=$	Set represented by v under substitutions σ, γ: $\Psi(v, \sigma, \gamma) \in 2^H$
no values	$\mathbf{0}$	$\{\}$
all values	$\mathbf{1} \# i$	$\{t \mid t = \gamma[i] \wedge t \in H\}$
tuples	$(v_1,...,v_n) \# i$	$\{t = (t_1,....,t_n) \mid t = \gamma[i] \wedge \forall_{j \in 1...n} . t_j \in \Psi(v_j,\sigma,\gamma)\}$
constants	$c \in$ 0-ary Constructors	$\{c\}$
tagged values	$(c\ v')\# i$	$\{t = c\ t' \mid t = \gamma[i] \wedge t' \in \Psi(v', \sigma, \gamma)\}$
alternative values	$(v_1 \mid ... \mid v_n)\# i$	$\{t \mid t = \gamma[i] \wedge t \in \Psi(v_1, \sigma, \gamma) \cup ... \cup \Psi(v_n, \sigma, \gamma)\}$
recursively defined values	$(fix.\ v') \# i$	$\{t \mid t = \gamma[i] \wedge$ $t \in$ least fixpoint of $T = \cup_{\gamma' \in I \times H} \Psi(v', \sigma[rec \rightarrow T], \gamma')\}$
	$rec \# i$	$\{t \mid t = \gamma[i] \wedge t \in \sigma[rec]\}$
maps	$map($ $must[(u_1,u_1')...(u_n,u_n')],$ $may[(v_1,v_1')...(v_1,v_m')]) \# i$	$\{t = [(t_1, t_1') ... (t_p, t_p')] \mid$ $t = \gamma[i] \wedge$ $(\forall_{k \in 1...n} \cdot$ $s_k = \Psi(u_k, \sigma, \gamma) \wedge$ $s_k' = \Psi(u_k',\sigma, \gamma) \wedge$ $\exists_{j \in 1...p} \cdot$ $\{t_j\} = s_k \wedge t_j' \in s_k') \wedge$ $(\forall_{j \in 1...p}$ s.t. $t_j \notin \cup_{i = 1...n} \Psi(u_i, \sigma, \gamma).$ $\exists_{k \in 1...m} \cdot$ $s_k = \Psi(v_i, \sigma, \gamma) \wedge$ $s_k' = \Psi(v_i', \sigma, \gamma) \wedge$ $t_j \in s_k \wedge t_j' \in s_k')$ $\}$

Aiken and Murphy [3]. The syntax and semantics of our augmented version of RTE's, simply called "abstract values" below, is in table 1.

On the left of the table is the syntax of abstract values *v*. These may include a special "variable" named *rec*.[1] Further, some abstract values *v* are tagged with integers *i* as denoted by the notation *v#i*. We will call these integers *abstract value identity tags* or simply *ID's* in what follows. On the right of the table is the set, written $\Psi(v, \sigma, \gamma)$, of concrete values represented by v under substitutions γ from ID's to concrete values (i.e., elements of H), and σ which may contain a binding from *rec* to a set of concrete values (i.e., to an element of 2^H). We say that an occurrence of *rec* is *bound* if it is enclosed in a *fix* value, otherwise we say it is *free*. If abstract value v has no free occurrences of variable *rec*, we say that v is *closed*. In the rest of the dissertation, we assume that all abstract values we deal with are closed. For closed v, we say that concrete value *t conforms to v* iff $t \in \bigcup_{\gamma \in I \times H} \Psi(v, [], \gamma)$, where [] is the empty substitution. As a convenience, in what follows we will write the set of conforming concrete values of abstract value v as $\underline{v}$.

3.2.2.3 Abstract Value Identity Tags

Abstract value identity tags (ID's) are an innovation particular to SCF. To gain some intuition on their role, recall that abstract values encode at the early stage a set of values such that the corresponding late stage value will be an element of this set. The basic idea behind ID's is that if two abstract values are tagged with the same ID at the early stage, it is guaranteed that the late stage value corresponding to the one will be the same as that for the other, i.e., the abstract values are "correlated." If, at the early stage, we needed to evaluate an equality test on two such values, we could therefore determine that the values will definitely be equal at the late stage. Since equality testing is commonly used in the optimizations being staged (in order to check if the dataflow analysis has reached fixpoint), accurate early equality testing is important.

The mapping γ enforces that if two abstract values v and v' have the same ID i, and are not nested in different *fix* values, then they are guaranteed to represent the same concrete value $\gamma[i]$. The ID tags therefore correlate concrete values that conform to different abstract values. As a simple example, if $v = (1 \mid 2)\ \#\ 31$ and $v' = (1 \mid 2)\ \#\ 31$, then the possible concrete values represented by $(v, v')\#734$ are $(1, 1)$ (in which case γ is the map $\{31\text{->}1, 734 \text{->} (1,1)\}$) and $(2, 2)$ ($\gamma = \{31\text{->}2, 734 \text{->} (2,2)\}$), but not $(1, 2)$ or $(2, 1)$.

1.In fact, since the variable always has the name *rec*, we may as well call it a keyword. However, traditional presentations of RTE's allow variables of arbitrary names, a feature not included in abstract values. We choose to preserve the name in order to highlight the parallel.

Although table 1 seems to consider arbitrary mappings γ from ID's to values, since the semantics of a given value *v* are specified in terms of the union over the value of $\Psi(v, [], \gamma)$ under all values of $\gamma \in I \times H$, the rules of table 1 implicitly impose a set of consistency requirements on γ. For instance, the rule for tagged values implies that an abstract value *(cons* ***1****)#21* has no valid interpretation under a γ that maps ID *21* to *nil*. Again, in the example of the previous paragraph, γ may only map tag *31* to the integer *1* or *2* (as per the rule in the table for processing alternates $v_1|...|v_n$) and not, for instance, to an arbitrary *343*. As a final example, the rule for tuples in the tables prohibits the mapping *{31->1, 734 -> (2,2)}*.

The rule for processing *fix* abstract values has a subtlety that deserves explanation. First, consider how *fix* values work in conventional RTE's (i.e., those without ID's). The *fix* form could be used to represent the set of all lists that contain the integers *1* or *2* by writing *fix(nil | cons(1| 2, rec))*. This expression expands to the set *{nil, cons(1, nil), cons(2, nil), cons(1, cons(1, nil)), ...}*. The set corresponding to the expression *fix. v* would be the least fixpoint of $T = \Psi(v, \sigma[rec \text{ -> } T])$. Intuitively, we generate values in the set by recursively replacing all occurrences of *rec* in *v* by the *fix* expression itself.

When we now add ID's to the abstract value, we are faced with the question of what value of γ to use when processing these recursively substituted values. We could use the γ we were using for the original *fix* expression, so that $\Psi(fix.\ v, \sigma, \gamma) = \Psi(v, \sigma[rec \text{ -> } T], \gamma)$. Consider a version of the above example with ID's: *(fix(nil | (cons((1| 2)#11, rec#12)#13)#14)#15)#16*. A consistent value for γ is *{16->cons(1, nil), 14->cons(1, nil), 11->1, 12->nil}*. Note, however that using these semantics, we cannot specify a γ where the ID *16* is mapped to value *cons(1, cons(2, nil))*, since this would entail ID *11* being mapped to both *1* and *2*. In order to allow different versions of "alternates" (such as *1|2* above) to be chosen in different recursive instantiations of a *fix* expression, we remove the restriction that the recursively instantiated *fix* expression is evaluated with the same γ as the original, and instead allow the recursive instantiations to be evaluated with arbitrary γ, leading to the rule of table 1 for $\Psi(fix.\ v, \sigma, \gamma)$.

Figure 3.9 gives further examples of abstract values and the set of concrete values that each abstract value represents. In particular, the figure demonstrates that ID's increase

Abstract Value, v	**Conforming Concrete Values, <u>v</u>**
(1 | 2 | 17)#23	*{1, 2, 17}*
(cons((1|2)#7, ***1****#234)#717)#22*	*{cons(i, j) | i ∈ {1,2}, j ∈ H}*
*(**1**#212, **1** # 212)#765*	*{(i,i) | i∈ H}*
((fix.((4,rec#12)#34|4)#27)#3, 0, (fix.((4,rec#13)#35|4)#28)#3, 0, (fix.((4,rec)#14|4)#29)#3)#1171	$\{\ \overbrace{(4,(4,...(4,4)))}^{n\ 4\text{'s}},\ 0,\ \overbrace{(4,(4,...(4,4)))}^{n\ 4\text{'s}},\ 0,\ \overbrace{(4,(4,...(4,4)))}^{n\ 4\text{'s}} \mid n > 0\}$
(fix. ((tree(“a” , rec#1)#2)#3 | empty_tree)#4)#5	*{empty_tree, tree(“a”, empty_tree), tree(“a”, tree(“a”, empty_tree)), ...}*
map(must [(1,(7|19)#1)], may [((2|14)#2, 9)])#113	*{[(1,7)], [(1,7), (2,9)], [(1,7), (14,9)], [(1 ,7), (2,9), (14,9)], [(1,19)], [(1,19), (2,9)], [(1,19), (14,9)], [(1,19), (2,9), (14,9)]}*

FIGURE 3.9: Examples of Some Abstract Values and the Sets they Represent.

fundamentally the expressiveness of abstract values relative to the RTE’s on which they are based: RTE’s are known to be equivalent to context-free grammars (CFG’s) [13] in expressivity. It is known further that CFG’s cannot encode the set of strings $4^n04^n04^n$ [29]. The fourth example of figure 3.9, on the other hand, shows how to encode this set of strings using abstract values. ID’s therefore add a form of context sensitivity to abstract values.

To ease reading and writing abstract values, in the rest of this dissertation if the ID of an abstract value is irrelevant to the discussion, we will simply drop it. The first abstract value of figure 3.9 would thus be written simply as *(1 | 2 | 17)*. We will also use *italics* when we write abstract values.

3.2.2.4 Abstract Maps

A second distinctive feature (relative to conventional Regular Tree Expressions) of our abstract values is the special representation for sets of possible map data structures. This representation consists of two association lists. Both map abstract values to abstract values. The first list, tagged *must* in the figure, is used for keys that are required to be in any conforming concrete map and that are known exactly, i.e., that are singleton abstract values. If an abstract value pair (*v*,*v’*) is in the *must* list of abstract map *m*, then every concrete map that conforms to *m* must map the single member of $\underline{v}$ to a member of $\underline{v}$'. The second list, tagged *may*, records more approximate key/value bindings. For every entry (*t*,*t'*) in a conforming concrete map such that *t* is not in the domain of the *must* list, there must exist a pair (*v*,*v'*) in the *may* list such *t* is in $\underline{v}$ and *t'* is in $\underline{v}$'. Thus, any key in a conforming map that is not in the *must* list is required to be in the *may* list, but not all keys in the *may* list need be in a conforming map.

The last row of figure 3.9 gives an example of how sets of maps are encoded. On the left side of the row is an abstract map, and on the right is the set of maps represented by that map. Because *1* is a key in the *must* list, all the maps on the right are required to have *1* as a key. On the other hand, since *2|14* is only a *may* key, the maps on the right may have one, both or neither of *2* and *14* as keys.

```
{u = x * i | i∈ Int} ∪ {u = y * i | i∈ Int}
```

(a) Set of Input Commands to be Represented

```
assign("u",
    primop( op_times,
          exprs( var("x", (label 23))| var("y", (label 24)),
                exprs( const( int(1), (label 77)),
                      exprs_none,
                      (label 76)),
                (label 75)),
          (label 78)),
    (label 79)
```

(b) Abstract Value Representing the Above Set

```
u = (x | y) * 'Int
```

(c) Compact Notation for the Above Abstract Value

```
mul_add (x, y, a){
u = (0 | x | (x * 'Int));
v = (y | ((u|x) + y));
return (v | y);}
```

(d) Compact Notation for the Set $F_{\{CpP\}}$ of Figure 2.9(b)

FIGURE 3.10: Representing Input Functions as Abstract Values.

3.2.2.5 Some Conventions for Writing Abstract Values

One class of abstract values that will come up often in the rest of the thesis is that of abstract values representing sets of AST fragments (such as expressions, commands and functions). As figures 3.10(a) and 3.10(b) illustrate, these abstract values can get very tedious to write. Figure 3.10(a) shows a set of commands to be represented by an abstract value. Figure 3.10(b) is an abstract value representing this set of commands, and is difficult to read because of the profusion of identity tags and labels. Figure 3.10(c) introduces a compact notation for writing abstract values that represent sets of input (AST) functions or their fragments. The basic idea is to omit writing the tags for the non-scalar abstract values, to omit labels on the incoming instructions, and to write all operations infix as the concrete syntax of the input language would dictate. In addition to the vertical bar separating variables `x` and `y` (which parallels that in the abstract value), the compact notation uses a special terminal `'Int` to represent the set of all integers. In future uses of this notation, we use terminals `'String` and `'Bool` to represent the universal sets of all strings and booleans respectively. In keeping with our previously established convention, we use *italic* font to write the abstract value, except that we denote that it is a an abstract value denoting input code by using courier font in addition: the result is the *`courier italic`* font.

As a further illustration of the notation, figure 3.10(d) shows an abstract value representing the set of functions $F_{\{CpP\}}$ of figure 2.9 written in the compact notation. In what follows, when tag numbers of abstract values and label numbers of input program fragments are unimportant, we will use the compact notation to represent the fragments.

3.2.3 How the Stager is Invoked

Having defined precisely the format of inputs to SCF, we now turn to how the compiler *user* stages a pipeline containing the three example optimizations. The user is interested in compiling and running a C program, of which the `mul_add` function of figure 2.3 is a part. Typically, at static compile time, the user would have determined via profiling that the function is heavily used by their C program, and is therefore worth dynamically optimizing. Further, value profiling may have revealed that the variable `a` changes infre-

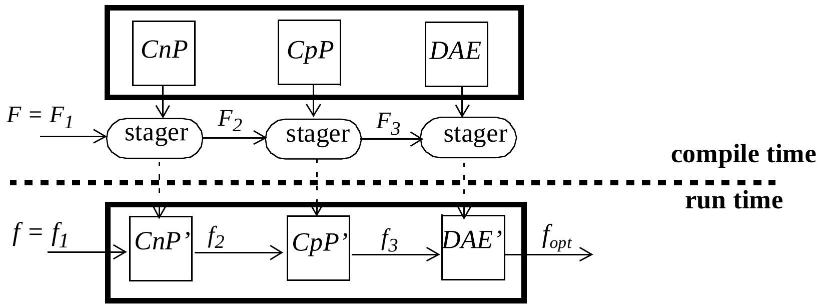


FIGURE 3.11: Staging a Three-Phase Compiler Pipeline.

```
let val f: AST.func = AST.parse "mul_add.c"
    val args =
          ["CnP.NON_CONSTANT", "CnP.NON_CONSTANT", "CnP.CONSTANT('Int)"]
    val F1: AbsValue = toAbsValue (f, args)
    val O1: SCF_ML.Program = SCF_ML.parse "const_prop.scf-ml"
    val O2                 = SCF_ML.parse "copy_prop.scf-ml"
    val O3                 = SCF_ML.parse "dae.scf-ml"
    val (O1', F2)          = stager (O1, F1)
    val (O2', F3)          = stager (O2, F2)
    val (O3', _)           = stager (O3, F3)
in (SCF_ML.print(O1', "const_prop.staged.scf-ml");
     SCF_ML.print(O2', "copy_prop.staged.scf-ml");
     SCF_ML.print(O3', "dae.staged.scf-ml"))
end
```

FIGURE 3.12: Invoking the Stager at Static Compile Time.

quently and is therefore a good candidate to be designated as a run-time constant. At this point, the user might want to use the SCF framework to stage the optimization pipeline $O_1 \ldots O_n$ and thereby produce a version of this pipeline specialized to optimize the `mul_add` function under the assumption that its third argument `a` is *some* integer constant, whose value will be revealed at run time.

Figure 3.11 (which is essentially the same as figure 2.9(a) from the previous chapter) illustrates the steps involved in staging our example compiler pipeline. At compile time (above the dotted line in the figure), the user invokes the stager in a sequential manner to generate the specialized optimization phases *CnP'*, *CpP'* and *DAE'*. At run time (below the dotted line in the figure), the specialized pipeline is invoked to produce the optimized program F_{opt}.

The stager in SCF is implemented in Standard ML. Figure 3.12 shows the Standard ML expression that implements figure 3.11. We assume that the modules `CnP`, `CpP` and

```
( int mul_add(int x, int y, int a) {
    int u = x * a;                  //command 1
    int w = u + y;                  //command 2
    return w;} ,                    //command 3

  (NON_CONSTANT, NON_CONSTANT, CONSTANT(1)))
```

(a) F_1: The Abstract Value Input to Constant Propagation

```
int mul_add(int x, int y, int a) {
  int u = (0 | x | (x * 'Int));
  int v = (y | (u + y));
  return v; }
             (i)
```

```
int mul_add(int x, int y, int a) {
  int u = (x | (x * 'Int));
  int v = ((u | x) + y);
  return v; }
             (ii)
```

(b) F_2: After Staged Constant Propagation on F_1. Version (i) assumes a more aggressive optimization, version (ii) a less aggressive one.

```
int mul_add(int x, int y, int a) {
  int u = (0 | x | (x * 'Int));
  int v = (y | ((u | x) + y));
  return (v | y); }
             (i)
```

```
int mul_add(int x, int y, int a) {
  int u = (x | (x * 'Int));
  int v = ((u | x) + y);
  return v; }
             (ii)
```

(c) F_3: After Staged Copy Propagation on F_2. Versions (i) and (ii) correspond to inputs (i) and (ii) respectively from figure 3.13(b).

FIGURE 3.13: Abstract Values Input and Output by the Stager at Compile Time.

DAE are defined in files const_prop.scf-ml, copy_prop.scf-ml and dae.scf-ml, respectively. In lines 2-4, the expression constructs, using a call to the function toAbsVal, an abstract value F1 representing the early-stage input to the first optimization in the pipeline, i.e., constant propagation. toAbsVal is a function that takes an AST representing an input function, and a list of strings (each string encoding as an abstract value an argument to the function), and returns an AbsValue that represents the set of tuples with the function as the first component and a list of concrete arguments as the second. In lines 5-7, the ML expression reads in the SCF-ML programs corresponding to the three optimizations. In lines 8-10, it invokes the stager on the optimizations Oi in a sequential manner to produced specialized optimizations Oi'. The values F2 and F3, which represent the possible results of executing optimizations O1 and O2, are used as inputs to stage the next optimization in the pipeline. Finally (lines 11-13), the expression writes the specialized optimization programs to disk.

```
1 fun optimize_mul_add_on_arg_a(a:int): code array =
2 let val f1    = AST.parse "mul_add.c"
3     val f2    = CnP_Staged.optimize
4                     (P1,[CnP.NON_CONSTANT,CnP.NON_CONSTANT, CnP.CONSTANT a])
5     val f3    = CpP_Staged.optimize f2
6     val fopt  = DAE_Staged.optimize f3
7 in CGen.codegen fopt end
```

FIGURE 3.14: Stub Function Providing Run-Time Interface to Staged Optimizers.

The variables `F1`, `F2` and `F3` denote the early stage abstract inputs to constant propagation, copy propagation and dead-assignment elimination, respectively. We saw the sets of values represented by these abstract values (called F_A, $F_{\{CnP\}}$ and $F_{\{CnP,CpP\}}$) in figure 2.9 of the previous chapter. Figure 3.13(a), (b)(i) and (c)(i) respectively show how these abstract inputs may be represented as abstract values, using the compact notation for abstract values discussed previously. For future reference, we also show in figure 3.13(b)(ii) and (c)(ii) the corresponding output abstract values if we assume that the constant propagator does not replace a multiply-by-zero with the constant expression 0.

We have thus far discussed what the user of SCF needs to do at compile time. We now describe how the results of the compile-time stage are used at run time. At run time, when the C program containing `mul_add` is about to invoke it with some concrete value of argument `a` for the first time, it first invokes the specialized optimizers (written to the files suffixed "`.staged.scf-ml`" in figure 3.12) by calling the stub function of figure 3.14 with the value of `a` as argument, and then jumps to the array of machine code returned by this function call. The stub function is generated (at static compile time) specifically to allow its caller to optimize function `mul_add` with respect to the late-stage value of `a` and the pipeline discussed above. The stub is typically generated as part of the expression in figure 3.12. The step generating the stub is omitted from that figure for clarity.

The run-time optimizer of figure 3.14 is not as efficient as it could be. Although for simplicity we show (in line 2) the optimizer as parsing the `mul_add` function from disk, since `mul_add` was known at static compile time, it could have been parsed at that stage, and its resulting AST stored in the text segment of the C program (or the specialized optimizer) so that run-time parsing could be avoided.

The last action of the stub is to generate (line 7) machine code for the optimized `mul_add` function via the `codegen` function. This illustrates a useful level of flexibil-

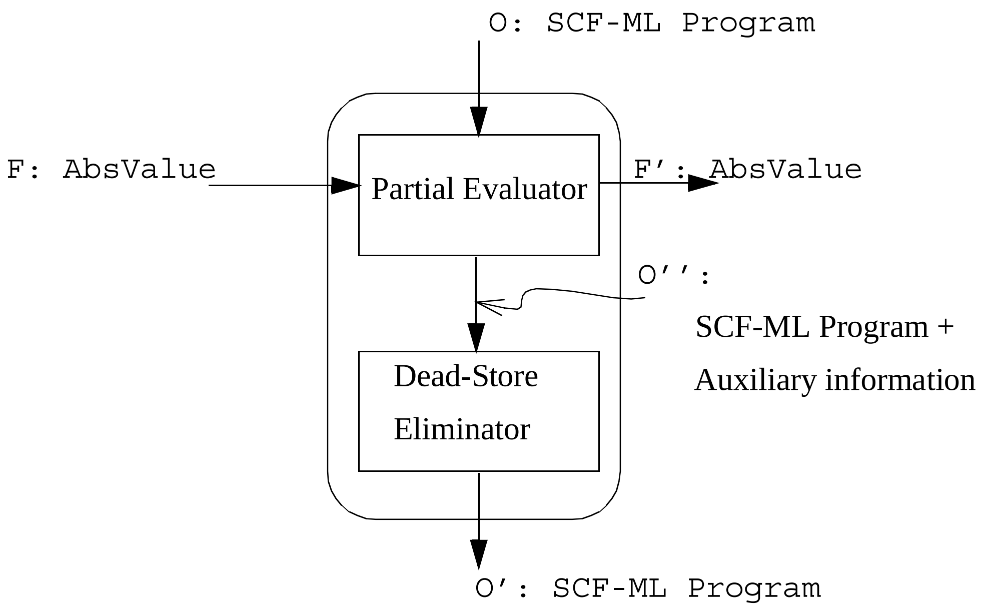


FIGURE 3.15: Internal Structure of the Stager.

ity provided by our framework: a staged pipeline may be succeeded by unstaged optimizations.

3.3 Internal Structure and Interfaces of the Stager

We have so far specified the input and output formats of the stager. In particular, the stager takes two inputs (a SCF-ML program and an abstract value) and produces two outputs (a specialized SCF-ML program and an output abstract value). In figure 3.7, we gave an example of an input SCF-ML program. In figure 3.13, we showed examples of input and output abstract values. We have so far not described in any detail the specialized SCF-ML optimization produced by the stager. In this section, we examine the structure of these optimizations (the modules *CnP'*, *CpP'* and *DAE'* of figure 3.11), and show how these specialized optimizations relate to their unspecialized variants. We explain the net specialization effect as a combination of the individual effects of the two main constituents of the stager, the *partial evaluator* and the *dead-store eliminator.*

Figure 3.15 shows the internal structure of the stager. The incoming abstract value *F* and optimization program *O* are fed into the partial evaluator, which produces the output abstract value *F'* (which represents the possible results of executing *O* on values in the set represented by *F*), a *partially evaluated* version *O”* of *O*, and some auxiliary information related to *O”*. The dead-store eliminator takes as input these two values and produces optimization program *O’*, which is the specialized program output by the stager.

In what follows, we will assume that the abstract value *F* is the value *F3* of figure 3.13(c)(ii), i.e., the early stage input to dead assignment elimination in our example pipeline. Further, we will assume that the optimization *O* is the dead-assignment elimination (DAE) module defined in figure 3.7.

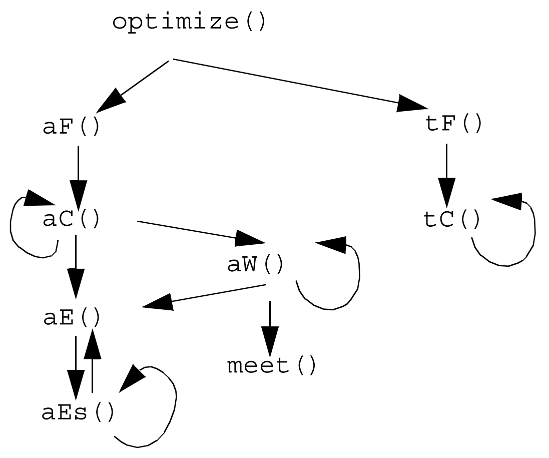
Figure 3.16 illustrates the effect of the partial evaluator on its inputs. On the left of the figure are the inputs to the partial evaluator, presented in a format different from those used so far. Figure 3.16(a) is the call-graph of the optimization *O*. Each node in the graph corresponds to a function in the module, and is labelled by an abbreviation of the name of the function (function names `analyze[Fun|Cmd|Expr|Exprs]` are abbreviated to `a[F|C|E|Es]` respectively, and `transform[Fun|Cmd]` to `t[F|C]`). We draw an edge between two nodes labelled *f* and *f’* iff a call exists from function *f* to function *f’* in the definition of the module.

Figure 3.16(b) is a graphical representation of the abstract input value *F*. The tree is simply a parse tree for *F* with respect to the abstract syntax of table 1. Some of the nodes are labelled with numbers 1 through 7 using the notation (**:0**, **:1** through **:7**). We use these numbers simply as a way to name the different nodes in what follows.

Figure 3.16(c) shows the callgraph for *O”*, the result of partially evaluating the optimization *O* with respect to *F*. The figure may be understood as the “unrolling” of the graph of figure 3.16(a) over the tree of figure 3.16(b) in the following sense. For each AST node in *F*, the partial evaluator produces a corresponding specialized version of the analysis and transformation functions for that node. For example, for the AST node numbered **0** (which happens to represent a command), the partial evaluator produces the functions `analyzeCmd0` (written `aC`**`0`**`()`) and `transformCmd0` (`tC`**`0`**`()`) respectively.

Each specialized function thus corresponds to a node in the input AST. The body of the specialized function is specialized to the argument node in two primary ways. First, whenever there is a call in an analysis/transformation function specialized to node ***i*** to analyze/transform a node ***j*** that is the child of ***i***, then we specialize the callee to node ***j***. For example, since commands **1** and **2** are children of the sequence command **0**, the recursive `analyzeCmd` calls to analyze the first and second commands in the sequence are replaced by calls to `analyzeCmd1` and `analyzeCmd2` respectively. Note that in figure 3.16(c), nodes labelled `aC1()` and `aC2()` are children of the node labelled `aC0()`.

Figure 3.17 shows the second way in which analysis and transformation functions are specialized to their argument nodes. The basic functionality can be thought of as a form of constant folding. Whenever an expression evaluates to a constant value, it is replaced by a



(a): Callgraph for Input DAE Module (*O*)

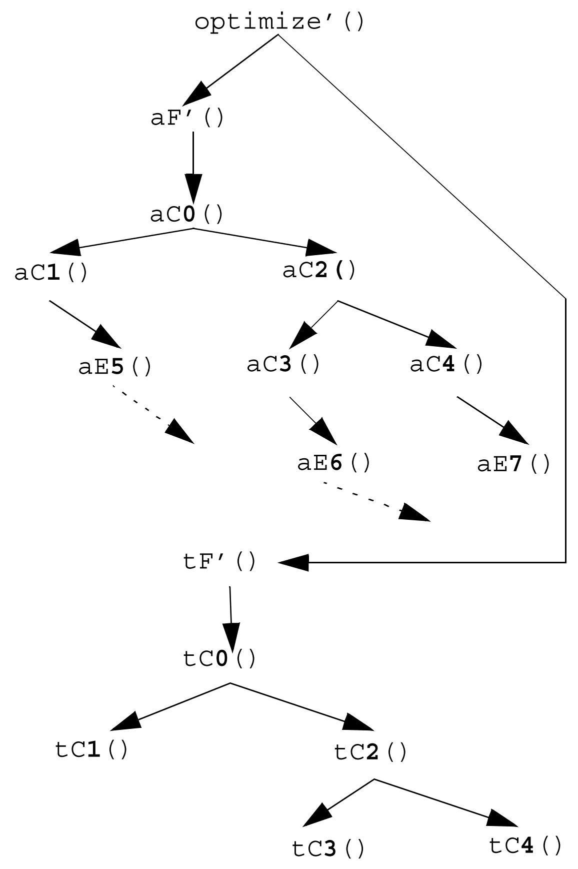
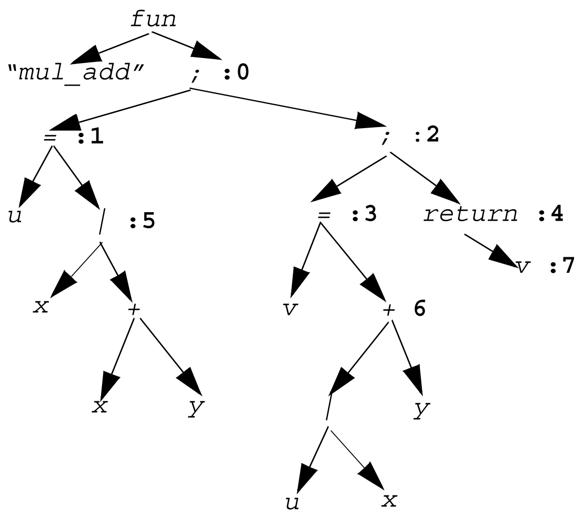


FIGURE 3.16(c): Callgraph for Partially Evaluated Dead-Assignment Elimination Module (*O*”)



(b) Input Abstract Value (*F*)

FIGURE 3.16: Inputs To, and Output From, the Partial Evaluator.

```
1  ...
2  and analyzeCmd3 (c, lSet, aMap) =
3    case c of
4      assign(v, e, lbl)   =>
5          let val lv = ~~if LiveSet.member(lSet, v) then live else dead~~ live
6          in (analyzeExpr6(e, LiveSet.delete(lSet, v)),
7              AssignMap.insert(aMap, ~~lbl~~ 3, ~~lv~~ live))
8          end
9      ~~| seq(c',c'',_) ...~~
10 ...
11
12 and transformCmd3 (c, aMap) =
13   case c of
14     assign(v, e, lbl) =>
15       ~~case AssignMap.find(assigns, lbl) of~~         }
16         ~~SOME dead => skip(lbl)~~                      } c
17       ~~| SOME _ => c~~                                 }
18   ~~| ...~~                                             }
```

FIGURE 3.17: Partially Evaluated Dead-Assignment Elimination .

simpler expression that evaluates to that value. In the figure, these simpler expressions are indicated in ***`bold italics`***.

The figure shows the bodies of the analysis and transformation functions for command 3. Recall that command 3 is the assignment `v = (u|x) + y`. Since, in the function `mul_add` being analyzed, command 3 is followed by command 4, i.e., `return v`, the variable `v` is guaranteed to be live at command 3, so that the membership test (`LiveSet.member(...)`) on the live-variables set on line 5 is guaranteed to evaluate to true, and the whole expression is guaranteed to evaluate to the constant value *live*. The if-expression on line 5 can therefore be folded away (as indicated by the strike-through) and replaced with the expression ***`live`***. Also, since `analyzeCmd3` is the analysis function for command 3, the value of variable `lbl` is guaranteed to be 3, as indicated by the replacement on line 7 (where the variable `lv` is also replaced by its constant value). Finally, `c` is guaranteed to be an assignment statement (of the form `assign(v,e,lbl)`), as tested in line 4. All other options on the form of `c` can also therefore be folded away, as in line 9.

The transformation function for command 3 (beginning at line 12) also benefits from constant folding. Since the analysis command for command 3 associated the constant value *live* with label 3 in the assignment map (line 7), the lookup on the assignment map

```
1  ...
2  and analyzeCmd3 (c, lSet, aMap) =
3    case c of
4      assign(v, e, lbl)   =>
5          let val lv = live
6          in (analyzeExpr6(e, LiveSet.delete(lSet, v)),
7              AssignMap.insert(aMap, 3, live))
8          end
9  ...
10
11 and transformCmd3 (c, aMap) =
12   case c of
13     assign(v, e, lbl) =>
14       c
```

FIGURE 3.18: Analysis and Transformation Functions After Partial Evaluation.

(at line 15) is guaranteed to return value *SOME(live)*, so that (as per the case on line 17) the entire case statement may be replaced by the expression **`c`**.

A final point to note is that as a result of folding, certain variables that were bound in pattern matches now become dead. Variable `lbl` of line 4 and variables `e`, `v` and `lbl` of line 13 all fall into this category. Since these variables are dead, we can avoid binding them in the patterns, and thereby avoid a potential load operation for each of them.

Figure 3.18 shows the versions of functions `analyzeCmd3` and `transformCmd3` that result from these transformations. The figure illustrates the benefits of partial evaluation, but reveals opportunities for eliminating a few dead operations. First, the dead binding of `lbl` on line 4 can be avoided. Instead we indicate that we "don't care" what the value in this place is by inserting a wild card match _. Second, consider the binding of variable `lv` on line 5. Since `lv` is not used downstream, the binding is dead and can be eliminated. Third, consider the case statement of lines 12-14. Since variables `v`, `e` and `lbl` are not used downstream, and the sole purpose of the case statement is to bind these variables, the entire case statement may be folded away and replaced its body, the expression `c`[1]. These two transformations are fairly routine instances of dead-store elimination.

The map insertion at line 7 is a more subtle, and very important, instance of a dead store. The insertion associates the value *live* with the label 3. The intention is that the

1.Unlike Standard ML, SCF-ML has undefined behavior when input values are unexpected variants of expected types. In particular, where traditional ML would raise a "non-exhaustive match failure" if `transformCmd3` were invoked with the wrong variant of command as argument, the SCF-ML program could (as a result of the current optimization) simply return the argument command.

```
...
and analyzeCmd3 (c, lSet, aMap) =
case c of
  assign(v, e, _)      =>
  (analyzeExpr(e, LiveSet.delete(lSet, v)), aMap)
...

and transformCmd3 (c, aMap) = c
```

FIGURE 3.19: Analysis and Transformation Functions After Dead-Store Elimination.

downstream transformation function for command 3 can refer to the liveness value for label 3 in the assignment map `aMap`, when deciding whether command 3 is live or dead. For command 3, this lookup in the map happens in line 15 of figure 3.17. However, since that read is folded away, it turns out that the binding inserted into the map has no readers, and can be omitted. The map insertion into map `aMap` is simply replaced by an expression that returns `aMap` unchanged.

We have so far not discussed at all the nature of the "auxiliary information" returned by the partial evaluator as per figure 3.15. At this point, we will just mention that this information is used for eliminating dead map accesses such as those described in the previous paragraph. The format and use of this map will be specified precisely in the next chapter.

Figure 3.19 shows the analysis and transformation functions for command 3 after partial evaluation and dead-store elimination. Comparing with figure 3.17, it is clear that the staged version of the optimization executes significantly fewer instructions than the unstaged version.

3.4 Summary

In this chapter, we have specified the format of inputs and outputs of the stager, and described the two major components of the stager in terms of their desired effect on an example. In the next three chapters, we will describe how these components achieve these effects, beginning with the partial evaluator.

4. The Partial Evaluator

In the previous chapter, we explained that the stager is composed of a partial evaluator and a dead-code eliminator. We described the partial evaluator only by its desired effect. In this chapter, we describe in detail the structure of the partial evaluator.

The partial evaluator in SCF is an *online* partial evaluator. An online partial evaluator may be understood as a program interpreter with two differences. First, instead of evaluating program fragments (such as functions and expressions) to get their *concrete* values, the partial evaluator performs a form of abstract interpretation where program fragments evaluate to *abstract* values which represent sets of concrete values. Correspondingly, while interpreting the fragment, instead of maintaining a *concrete environment* mapping variables to concrete values, partial evaluators maintain an *abstract environment* mapping variables to abstract values. Each abstract environment *E* represents a set of concrete environments $\underline{E}$. We say that the concrete environments in $\underline{E}$ conform to *E*. Second, for each fragment it evaluates in a given abstract environment, the partial evaluator produces, in addition to the abstract value of the fragment, a version of the fragment (called a *residualized* version) specialized to the abstract environment.

Figure 4.1 shows the signature for the partial evaluator module using Standard ML (SML) syntax. In what follows, we write all code that is part of the definition of the stager in *italics*. All SML keywords are in ***bold italics***. In all examples that follow, modules that will be precisely defined later in the chapter are *underlined* the first time they are mentioned.

The signature specifies that all modules that conform to it are required to provide a single function PE_p (the subscript *p* stands for "program" to indicate that the SCF-ML fragment to be partially evaluated is the incoming program), which given an incoming SCF-ML optimization program and input abstract value returns a residual optimization

```
signature PARTIAL_EVALUATOR = sig
  val PE_p :  SCF_ML.program * AbstractValue.value ->
              SCF_ML.program * AbstractValue.value * LabelAbstractValueMap.map

end
```

FIGURE 4.1: Signature of the Partial Evaluator

program, the abstract value that the input program evaluates to given the abstract input, and a map from labels to abstract values which will be explained later in this chapter and in chapter 7. The last data structure is the same as the “auxiliary information” of figure 3.15.

4.1 Signatures of Some Key Data Structures

The conventional design [56] of an online partial evaluator is as a worklist algorithm over the functions in the program being evaluated. As is typical with these algorithms, our par-

```
datatype ‘a option = SOME ‘a | NONE
datatype ‘a choice = MUST ‘a | MAY ‘a | NOT

signature ABSTRACT_ENVIRONMENT = sig
  type env
  val empty:   env
  val find:    env * SCF_ML.var -> AbstractValue.value
  val insert:  env * SCF_ML.var * AbstractValue.value -> env
  val meet:    env * env -> env
  val shadow:  env * env -> env
end

signature CACHE = sig
  type cache
  type contour_key
  type cache_info =
    {argVal:AbstractValue.value, retVal:AbstractValue.value, resExp:SCF_ML.exp, numEvals: int}
  val empty:   cache
  val find:    cache * contour_key -> cache_info option
  val insert:  cache * contour_key * cache_info -> cache
  val flush:   cache -> SCF_ML.fun list
end

signature WORKLIST = sig
  type worklist
  type contour_key
  type worklist_info = {key: contour_key, value: AbstractValue.value} choice
  val empty:   worklist
  val add:     worklist * worklist_info -> worklist
  val take:    worklist -> (worklist * worklist_info) option
end

structure AbstractEnvironment   : ABSTRACT_ENVIRONMENT
structure Cache                 : CACHE
structure WL                    : WORKLIST
```

FIGURE 4.2: Signatures of Primary Data Structures Used by the Partial Evaluator

tial evaluator maintains three primary data structures, which we will call *abstract environment*, *cache* and *worklist*. Figure 4.2 specifies the signatures, named *ABSTRACT_ENVIRONMENT*, *CACHE* and *WORKLIST* respectively, for these modules. Since these data structures are standard, we will only define formally their signature and not their implementation, and summarize informally their required behavior.

Lines 1 and 2 define two types that will be used frequently in what follows. The first is the parameterized type *option*. It has two variants, the unary variant *SOME* and the zero-ary *NONE*. The type is intended to describe expressions that either return a value in some domain or no value at all (such as a map lookup that either succeeds with some value or fails with none). The second type, *choice,* is similar to *option*, but has three variants. It is intended to be used in situations where a value represents one of a variety of actions, which must, may and should not be performed (represented by the *MUST*, *MAY* and *NOT* constructors respectively). We will see examples of both types below.

Lines 4 through 11 of the figure delimit the signature *ABSTRACT_ENVIRONMENT* that must be satisfied by the abstract environment module. As per line 5, the module must provide a type named *env*, which will correspond to the variable-to-abstract-value map that constitutes the environment. In addition (line 6), the module is required to provide a value named *empty* (which is intended to stand for the empty map) of type *env*. Line 7 requires an accessor function *find*, which given an environment and a variable returns the value corresponding to the variable in the environment. It is undefined what happens if the variable is not in the environment. Line 8 requires an *insert* function which, given an environment, variable and value, returns the result of updating the environment with the variable-value binding. Line 9 requires a *meet* function, which given two abstract environments, each representing a set of concrete environments, returns a third abstract environment that represents (possibly a superset of) the set of environments comprising the union of the incoming two sets. Finally, line 10 requires a function *shadow*, which given two environments *E* and *E'* as arguments, returns the result of replacing any key-value pair *(k, v)* in *E* with the pair *(k, v')* if *k* is bound to *v'* in *E'*. In what follows, we will assume a module *AbstractEnvironment* (line 33) that conforms to this signature.

The cache is essentially a map from function names to a pair of abstract values representing the inputs and outputs calculated thus far by the partial evaluator for each function.

It is standard [46, 60, 58, 25] to implement interprocedural analyses using partial transfer functions represented by caches of this kind. Lines 13 through 22 delimit the definition of the *CACHE* signature which defines the abstract interface of the cache. Lines 15 through 17 define the domain and range of the map underlying the cache. The domain is represented by an opaque type *contour_key* (line 15). The contour key type is typically a tuple consisting of the name of a function in addition to other information that distinguishes invocations of the function made in different contexts. We will see an example contour key later in this section. The range of the map is a record type *cache_info* (lines 16-18), which has four fields: *argVal* and *retVal,* the argument and return abstract values respectively computed for the current contour, *resExp*, the residualized version of the expression constituting the body of the function represented by the contour key, and *numEvals*, an integer counting the number of times the cache entry for this contour key has been modified.

The *find* function (line 19) on the cache returns a value of type *cache_info option*. This is an instantiation of the parameterized type *'a option* discussed above. The intention is that if the *find* fails in the map, it returns *NONE*, otherwise it returns value *SOME ci*, assuming *ci* is the cache information found. The final function in the signature for the cache is the *flush* function which essentially returns the list of all the residualized functions in the cache. In what follows, we will assume a module *Cache* (line 34) that conforms to this signature.

The worklist (lines 23-30) is typically either a stack or queue of elements of type *worklist_info*. Each element contains a choice of a contour (which in turn, as mentioned above, typically contains a function name) and the argument value with which to partially evaluate the function named by the contour. The *add* function adds an element to the worklist. As indicated by the *option* in the return type of the *take* function, invoking this function on worklist *wl* optionally returns *NONE* if the worklist is empty, and *SOME(wl', wli)* where *wli* is an element of *wl* and *wl'* is the result of removing *wli* from *wl*. In what follows, we will assume a module *WL* (line 35) that conforms to this signature.

```
P  ∈ program    ::=  d_1,...,d_m g_1,...,g_n
d  ∈ typeDec    ::=  tn = k_1 of t_1, ..., k_n of t_n
t  ∈ type       ::=  int | bool | string | char | tn | p
p  ∈ prodType   ::=  t_1 * ... * t_n

g  ∈ funDef     ::=  f x = e
e  ∈ expr       ::=  x | c | c e | (e_1,...,e_n) | case e of m_1 m_2 ... m_n| f e |
                     pr e | cf (fn x => e_1) e_2
m  ∈ match      ::=  pt => e
pt ∈ pattern    ::=  _ | x | c | c pt | (pt_1,...,pt_n) | x as pt

pr ∈ primop     ::= map_insert | map_find | map_equal | map_remove
                    | + | - | ...
cf ∈ curriedFuns::= map_unionWith | map_map
c  ∈ constructors = Identifiers ∪ Int ∪ Bool ∪ String ∪ Char
f, x, tn ∈ Identifiers
```

FIGURE 4.3: Abstract Syntax for SCF-ML Programs

4.2 Core SCF-ML and Notational Conventions

We discuss below the abstract syntax of SCF-ML (in which optimizations are written), followed by some notational conventions for abbreviating Standard ML (in which SCF itself is written).

SCF desugars the concrete SCF-ML syntax of figure 3.3 into the abstract syntax of figure 4.3. Modules are eliminated by inlining. The `let` and `if` special forms are desugared into the more general `case` form. All operations on set data structures are replaced by their implementation using map operations; the abstract syntax has only built-in map operations.

In what follows, we will be specifying various parts of SCF itself. SCF is written in Standard ML. We write all Standard ML code comprising SCF in *italics* with keywords additionally in ***bold***. We use certain abbreviations throughout. We will uniformly omit the ***val*** keyword in ***let*** bindings. The Standard ML expression ***let val*** *x = g 3* ***in*** *f x* ***end*** will instead be written ***let*** *x = g 3* ***in*** *f x* ***end***. Further, when we wish to perform an assignment as part of a sequence of let bindings, instead of using the notation ***let*** *... _ = (x := ...) ...* ***in ... end***, we will write ***let*** *... x := ... ...* ***in ... end***. Finally, we use vertical bars |...| to delimit either patterns or constructor applications written in concrete syntax, possibly omitting uninteresting parts. For instance, instead of writing *SCF_ML.constructor(c, e)*, we will write |*c e*|, and instead of *AbsValue.tuple(vs, id*), we write |*(vs)*|. By omitting explicit constructors, the notation potentially confuses different entities that share the same concrete syntax. When the context does not resolve the ambiguity, we use explicit constructors.

4.3 Initialization and Fixpointing

In this section, we specify the fixpoint loop that iterates over the functions in the incoming SCF-ML program. The underlying structure of the loop is common to many interprocedural analyses, and this structure has been proposed for online partial evaluators as well [56]. However, the partial evaluator in SCF provides more aggressive versions of traditional modules used by the fixpoint loop, and performs pre-processing steps not common to conventional fixpoint-based online partial evaluators. We use the precise specification of the fixpoint loop to highlight both similarities with, and differences from, existing schemes. Where the differences are important, we will go into them in more detail later in the chapter.

Figure 4.4 specifies the entry function and the fixpoint loop of the partial evaluator module. Lines 1-5 of the figure define global variables which (as indicated by the ***ref*** keyword) may be side-effected. In this section, we will focus on the first three definitions (lines 1-3), which define the cache *c*, worklist *wl* and program *p* to be partially evaluated as described above. These data structures are represented as global variables that are side-effected so as to avoid having to thread them throughout the partial evaluator as done in conventional presentations. Lines 4 and 5 define variables that store the pre-processing and analysis results, and will be discussed later in this chapter.

Lines 8-15 define the entry function PE_P. On line 8, we use pattern-matching to split the incoming program into type-declarations *ds* and function declarations which are delimited by the vertical bars ||, as discussed in the previous section. We digress briefly to elaborate on the notation using line 8 as example. Pattern |*g1*,...,*gn*| would match against a list of *n* elements with *g1* and *gn* matching the first and last elements. |*f x* = *e*,...,*gn*| requires that the first element in this list be a function whose name is matched against variable *f*, formal against *x* and body against *e*. Similarly, if variables *f*, *x* and *e* are bound to function name, formal name and SCF-ML expression respectively, we will often write |*f x*

```
val c      = ref Cache.empty
val wl     = ref WorkList.empty
val p      = ref Program.empty
val lvm    = ref LabelAbstractValueMap.empty
val fm     = ref FinitenessAnalysis.emptyFinitenessMap   (*FinitenessAnalysis written as FA below*)

(* See figure 4.1 for signature of function $PE_P$ *)
fun $PE_P$ (p as program(ds, | f x = e,...,gn|),v) =
  let fm  := FA.analyze p
      _    = RematStrategy.reset();
      ck   = ContourKey.mkContourKey (f, !fm, v, ContourKey.empty)
      wl  := WorkList.add(WorkList.empty, MUST(ck, v))
      _    = fixpoint ()
      gs'  = Cache.flush !c
  in (RematStrategy.postProcess(program(ds, gs')), !lvm) end

(* fixpoint: unit -> unit *)
and fixpoint () =
  case WorkList.take !wl of                                  (*ContourKey written as CK below*)
       NONE                       => ()
     | SOME(wl', MAY(ck, v))      => (wl := wl'; processFun(ck, v, false, Map.find(!fm,CK.func ck))
     | SOME(wl', MUST(ck,v))      => (wl := wl'; processFun(ck, v, true, Map.find(!fm,CK.func ck))

(* processFun: ContourKey.contour_key * AbstractValue.value * boolean * FA.FP -> unit *)
and processFun(ck, v, mustExec, fp) =
  let  ($vArg_{old}$, $vRet_{old}$, e, i)   = case Cache.find(!c, ck) of
                                      SOME ci   => (ci#argVal, ci#retVal, ci#resExp, ci#numEvals)
                                    | NONE      => (v, AbstractValue.top, |()|, 1)
      c                           := Cache.insert(!c, ck, {argVal = $vArg_{old}$, retVal =$vRet_{old}$,
                                                      resExp = e, numEvals = i})
      vArg'                        = AbstractValue.meet(v,$vArg_{old}$)
      vArg                         = Widening.widen(vArg', i, fp)
  in if (not mustExec) andalso AbstractValue.mustBeEqual(vArg, $vArg_{old}$) then
      fixpoint ()
    else let  ( e', vRet)   = $PE_f$ (ck, vArg)
              newInfo       = {argVal = vArg, retVal = Widening.widen(vRet,i,fp),
                                      resExp = e',numEvals = i+1}
              c            := Cache.insert(!c, ck, newInfo)
         in if not AbstractValue.mustBeEqual(meet($vRet_{old}$,vRet),$vRet_{old}$) then
              (addClrs ck;
               fixpoint ())
            else fixpoint ()
         end
  end

and addClrs (ck:ContourKey.countour_key):unit =
    ...add calling contours of ck to worklist !wl...
```

FIGURE 4.4: Initialization and Fixpoint Loop for Partially Evaluating Programs.

= *e*| (instead of *SCF_ML.funDef(f,x,e)*) for the SML expression constructing an SCF-ML function.

The body of PE_P essentially does some pre-processing before performing the fixpoint analysis at the heart of the partial evaluator. In particular, it performs a pre-processing analysis to be discussed later (line 9), resets a persistent data structure (line 10), constructs a contour key containing the name of the entry function of the incoming SCF-ML program (line 11), pushes a pair containing this contour key and the incoming abstract value *v* on the worklist (line 12), performs the fixpoint computation which side-effects its results into cache *c* (line 13), and extracts (line 14) and returns (line 15) the results of the fixpoint computation. Before returning the program, it performs (on line 15, via the *RematStrategy.postProcess* call) some post-processing to be described later.

The fixpoint computation pops a task off the worklist (lines 18-22) and processes it (lines 25-44). Each task is a choice (*MAY* or *MUST*) of a contour key and an abstract value. Given a contour key, we first (lines 26-28) check in the cache to see if the key has been processed before, and if so what the cache information corresponding to the key was. In particular, we are interested in the abstract argument value (*argVal*), result value (*resVal*) and the number of times this contour has been evaluated (*numEvals*), and therefore read these fields off the cache-info record *ci* (the notation *r#f* reads field *f* of record *r*). If no entry exists in the cache for the key (line 28), we construct suitable default cache information. Before analyzing the contour, we record (lines 29-30) in the cache that we are about to analyze it; this step is necessary so that if no entry existed in the cache for the contour, we will now have an entry mapping its return value to *top*, thus avoiding unbounded recursive analysis of the contour. Given the previous abstract argument $vArg_{old}$ with which the contour was processed, we (line 31) add in the current argument *v* by invoking the lattice *meet* function on values *v* and $vArg_{old}$. The abstract value is next (lines 32) *widened* as necessary. Widening enables abstract interpreters to terminate when their domains are infinitely tall.

Apart from the particular meet function and widening strategy used (which will be discussed later), all other steps are standard. The abstract interpretation performed by the partial evaluator has as its domain the power set of the (infinite) Herbrand Universe of section 3.2.2.1, with set containment as the lattice ordering relation. The interpretation lattice is therefore infinitely tall, necessitating a widening-based abstract interpretation. The meet operation on this lattice is simply set union. In chapter 6, we will discuss how to widen while preserving sufficient accuracy, and also how to compute meets on abstract values efficiently. The *Widening* module invoked as a pre-processor in the figure aids in the former task, whereas the *AbstractValue* module aids in the latter.

Lines 33-38 complete the processing of the worklist task. In the case that the widened abstract value is one that was previously processed and the task is not a *MUST* task, there is nothing more to be done; we just ignore the task and revert to the fixpoint loop to get the next task (line 34). Alternately (lines 35-43), we perform partial evaluation of the individual function (line 35) with a call to PE_f, and record the result in the cache (lines 36-38).

Finally, if the return value for the contour just analyzed changed, we place on the worklist for re-analysis all contours whose analysis depended on the return value of the contour just analyzed, and return to the fixpoint loop (lines 39-41). The helper function that adds contour keys *ck* for re-analysis, defined only with a brief comment on line 47, first determines the argument value *v* recorded for *ck* in the cache and places the task *MUST(ck,v)* on the worklist. As described above, the *MUST* task forces re-evaluation of the corresponding function even if the argument of the function has not changed since the last time it was processed, to account for the fact that the return value for one of its callees may nevertheless have changed. If the return value is unchanged, we omit placing callers on the worklist and directly return to the fixpoint loop (line 42).

Again, these steps for processing worklist tasks are standard, although for our particular representation of abstract values (Augmented Regular Tree Expressions), implementing the *mustBeEqual* test in an efficient, conservative but sufficiently effective way requires careful design.

4.4 Partial Evaluation of Individual Functions

In the previous subsection, we described a fixpoint loop that processes (contour keys corresponding to) individual functions. Each function was processed by a call to the undefined function PE_f. In this section, we specify PE_f, as usual paying particular attention to how it differs from standard versions.

Figure 4.5 defines PE_f and its main helper functions, in particular the function PE_e, which is responsible for partially evaluating expressions. Given, in PE_f, a contour key *ck* corresponding to the function $f\,x = e$ to be specialized and the abstract value *v* of the function argument, we simply invoke PE_e to partially evaluate *e* in an environment where variable *x* is bound to value *v* (lines 1-7).

Lines 11-47 define the function PE_e. Given an expression to be partially evaluated, an abstract environment in which to evaluate this expression and a contour key defining the context in which to evaluate, it returns a pair consisting of the residualized version of the expression and the abstract value of the expression. In the rest of this section, we discuss how each type of expression is processed, going into detail where we deviate from typical online partial evaluators.

4.4.1 Literals

Line 11 shows how scalar literal expressions are handled. Recall that the residualized version of an expression is just a simpler (in terms of number of steps needed to evaluate it) version of the expression that nevertheless computes the same values as the expression. Scalar literals are already in the simplest possible form, and the residual expression is the same as input expression *e*. The abstract value of a scalar expression *c* is simply the singleton abstract value *c*.

4.4.2 Variables

Lines 13-15 handle variables. If *v* is the value of a variable *x* in abstract environment *E*, the abstract value of the expression is simply *v*.

The residualized expression replacing variable *x* is determined by the *rematerialization strategy*, encapsulated in the *RematStrategy* module referenced on line 50. The module decides how to replace a given expression by a simpler one. When value *v* is a singleton scalar (by scalar, we mean integers, characters, booleans and strings), for instance, it replaces the variable *x* with the literal expression e_v, the literal that evaluates to

```
(* PE_f: ContourKey.contour_key * SCF_ML.val -> SCF_ML.exp* SCF_ML.val *)
and PE_f (ck , v)=
   let   |f x = e| = findFun ck
         (e', v') = PE_e (e, AbstractEnvironment.insert(AbstractEnvironment.empty,x,v), ck)
   in (e', v') end

and findFun (ck: ContourKey.contour_key): AST.fun = ... find the function definition for key ck...

(* PE_e:  SCF_ML.expr * AbstractEnvironment.env * ContourKey.contour_key ->
          SCF_ML.expr * AbstractValue.value *)
and PE_e (e as |c|, _, ck) = (e, |c|)

| PE_e (e as |x|, E, ck) =
   let   v = AbstractEnvironment.find(E, x)
   in rematerialize(e,v) end

| PE_e (|(es)|, E, ck) =
   let   evs        = List.map (fn e => PE_e(e, E, ck)) es
         (es', vs)  = List.unzip evs
   in (|(es')|, |(vs)|) end

| PE_e (|c e|, E, ck) =
   let (e',v) = PE_e (e, E, ck)
   in (|c e'|, |c v|) end

| PE_e (|map_ insert e|, E, ck) =    ... defined in figure 4.6 ...
| PE_e (|map_ find e|, E, ck)    =    ... defined in figure 4.6 ...
|PE_e (|map_ equal e|, E, ck)  =    ... defined in figure 4.6 ...
| PE_e (|p eArg|, E, ck) =
   let   (eArg', vArg)   = PE_e (eArg, E, ck)
         v               = AbstractValue.evalPrimop(p, vArg)
   in rematerialize (|p eArg'|, v) end

| PE_e (|case e of ms|, E, ck) =
   let   (e',v)       = PE_e (e, E, ck)
         (ms', v')    = PE_ms (ms, v, E, ck)     (* PE_ms defined in figure 4.7 *)
   in rematerialize (|case e' of ms'|, v') end

| PE_e (|f e|, E, ck) =
   let   (e', v) = PE_e (e, E, ck)
          ck'    = ContourKey.mkContour(f, !fm, v, ck)
         v_ret   = case Cache.find(!C, ck') of
                          SOME ci   => ci#retVal
                          | NONE     => AbstractValue.top
         wl      := WorkList.add(!wl, MAY(ck', v))
         f'      = ContourKey.getSpecializedFName ck'
   in rematerialize (|f' e'|, v_ret) end

(* When v is singleton, return a simpler version of e equivalent to e under all feasible contexts *)
and rematerialize(e: expr, v: value): (expr * value) = (RematStrategy.rematerialize(e,v), v)
```

FIGURE 4.5: Partial Evaluation of Functions and Expressions in SCF.

the lone element of $|v|$. This replacement satisfies the requirement that the new expression is simpler to evaluate than the first. Rematerialization is more complex in other cases, and section 6.4 discusses the matter further.

4.4.3 Tuples and Constructors

Partially evaluating tuples and applications of constructors (17-24) is a straightforward matter of partially evaluating sub-expressions and composing the results. For instance, partially evaluating a tuple of expressions (lines 17-20) consists in first partially evaluating the component expressions to obtain a list of residual expression-result value pairs (via the *map* functional of line 18), and reconstituting this list of pairs into a pair of lists (of expressions and values) via the *unzip* functional of line 19. These lists are packed into a tuple expression and a tuple abstract value respectively in line 20.

4.4.4 Primitive Operations

Expressions representing arguments of primitive operations are first evaluated abstractly to get their value *vArg*. The abstract application of the primitive operation is performed by the *evalPrimop* function of the *AbstractValue* module (line 31). As with variables, if the resulting abstract value v is a singleton scalar, we return as the residualized version the corresponding literal expression, otherwise we return an application of the primitive operator. We return v as abstract value of the expression. This conventional sequence of steps is detailed on lines 31-32 of figure 4.5.

As shown in figure 4.6, the map operations *map_insert*, *map_find* and *map_equal* are treated specially, in anticipation of the dead-code elimination pass that follows the partial evaluator in SCF. In particular, while these primitive operations are abstractly evaluated and residualized just as other operations, we also record for each of them an abstract value representing the set of map keys that will, in the later stage, be necessary for performing the operation on the incoming map (or maps). We call this set of keys the *live keys*.

In the case of the *map_insert* and the *map_find* operations, the live keys are exactly the incoming abstract key v_{key} being inserted and found respectively. In the case of *map_equal*, however, we use the special function *findLiveKeys* of the *AbstractValue* mod-

```
1  | PE_e (e as |map_insert eArg|, E, ck) =
2  let  (eArg', vArg)              = PE_e (eArg, E, ck)
3       v_key                      = AbstractValue.match(|(_,x,_)|, vArg)).find(|x|)
4       v                          = AbstractValue.evalPrimop(|map_insert|, vArg)
5       e'                         = |map_insert eArg'|
6       lvm                        := LabelAbstractValueMap.insert(!lvm, labelOf e', v_key)
7  in rematerialize (e', v) end
8
9  | PE_e (e as |map_find eArg|, E, ck) =
10 let  (eArg', vArg)          = PE_e (eArg, E, ck)
11      v_key                  = AbstractValue.match(|(_,x)|, vArg)).find(|x|)
12      v                      = AbstractValue.evalPrimop(|map_find|, vArg)
13      e'                     = |map_find eArg'|
14      lvm                    := LabelAbstractValueMap.insert(!lvm, labelOf e', v_key)
15 in rematerialize (e', v) end
16
16 | PE_e (e as |map_equal eArg|, E, ck) =
18 let  (eArg', vArg)   = PE_e (eArg, E, ck)
19      v_keys          = AbstractValue.findLiveKeys vArg
20      v_bool          = AbstractValue.evalPrimop(|map_equal|, vArg)
21      e'              = |map_equal eArg'|
22      lvm             := LabelAbstractValueMap.insert(!lvm, labelOf e', v_keys)
23 in rematerialize (e', v_bool) end
```

FIGURE 4.6: Three Special Cases of $\mathrm{PE_e}$ of Primitive Operations

ule to return an abstract value that represents the set of keys whose equality (or lack thereof) is not fully determined at the early stage. For instance, if the incoming abstract maps are *map(must[(1, 2), (3, (7|19))], may[23, 12])* and *map(must[(1, 2), (3, (7|19))], may[23, 27])*, *findLiveKeys* should return the value *3|23*, since we can determine at the early stage that the value corresponding to key *1* will be equal for the two maps, and this key therefore does not have to be compared later on.

Recall from figure 4.1 that the partial evaluator returns a map from SCM-ML labels to abstract values. This map is implemented as a global variable *lvm* and initialized as a reference to an empty map in figure 4.4, line 4. We are now ready to explain what the map contains: it records, for each residualized map operation *map_insert*, *map_find* and *map_equal*, the live keys for that operation. The operations are identified by their SCF-ML labels. As shown in lines 6, 14 and 22, we side-effect the gloal variable *lvm*[1] to add a new mapping from the label of the residual operation created to the abstract value repre-

1.The variable name `lvm` is an abbreviation for "live variable map".

```
1  (* PE_ms : SCF_ML.match list * AbstractValue.value * AbstractEnvironment.env *
2             ContourKey.contour_key -> SCF_ML.match list * AbstractValue.value *)
3  and PE_ms([], v, E, ck) =
4  ([], AbstractValue.top)
5
6  | PE_ms(_, AbstractValue.top, _, _) =
7  ([] , AbstractValue.top)
8
9  | PE_ms(m::ms, v, E, ck) =
10 case PE_m(m, v, E, ck) of
11   SOME (m', v_val, v_rem) => let   (ms', v_val') = PE_ms(ms, v_rem, E, ck)
12                              in    (m'::ms', AbstractValue.meet(v_val,v_val')) end
13 | NONE                    => PE_ms(ms, v, E, ck)
14
15
16 (* PE_m : SCF_ML.match * AbstractValue.value * AbstractEnvironment.env * CK.contour_key ->
17           (SCF_ML.match * AbstractValue.value * AbstractValue.value) option*)
18 and PE_m(|p => e|, v, E, ck) =
19 case AbstractValue.match(p,v) of
20   NONE              => NONE
21 | SOME (E', v_rem)  => let (e', v')  = PE_e (e, AbstractEnvironment.shadow(E, E'), ck)
22                        in SOME (|p => e'|, v_rem, v') end
```

FIGURE 4.7: Partially Evaluating Pattern Matching

senting the live keys. The next chapter shows how we use the live keys for a map operation (in conjunction with other information) to determine whether the operation can be declared dead and removed altogether from the residualized optimization program.

4.4.5 Case Expressions

Lines 34-37 of figure 4.5 process case expressions. The partial evaluator first computes (on line 35) the abstract value v of the controlling expression e of the case expression. Function PE_{ms} partially evaluates the sequence of matches in the case expression using v as the incoming abstract value to match against (line 36).

Figure 4.7 defines PE_{ms}. It shows how a sequence ms of matches is partially evaluated. The incoming value v is partially evaluated with respect the first match m in sequence m via a call to PE_m (line 10). Recall that matches m are of the form $p => e$, where p is the pattern guarding execution of expression e.

- If it is certain that no concrete value in v matches the pattern guarding m (line 13), we discard m and continue matching against the remaining matches in ms.

```
case i of
  Int 0 => 0          (* match m1 *)
| Int x => x + 1      (* match m2 *)
```

(a) Expression e_0 to be Evaluated

{i -> Int (0 |1)}

(b) Environment E for Evaluating e_0

```
m' = Int 0 => 0,    vval = 0,     vrem = Int 1     (* processing m1 *)
m' = Int x => 2,    vval = 1,     vrem = 0         (* processing m2 *)
```

(c) Results of Evaluating Matches m_1 and m_2

```
m' = Int 0 => 0,      vval = 0,        vrem = Int 0|1   (* processing m1 *)
m' = Int x => x + 1,  vval = Int 1|2,  vrem = 0         (* processing m2 *)
```

(d) Alternate Results of Evaluating Matches m_1 and m_2

FIGURE 4.8: Partially Evaluating Case Expressions: An Example

- If *m* is a possible match (line 11), PE_m returns the residualized match *m'*, the abstract value v_{val} of expression *e* and a value v_{rem} which is (possibly a superset of) the part of *v* that did not match *p*. We use v_{rem} as the value to match against the remaining matches in *ms* (line 11). If v_{val}' is the abstract value resulting from partially evaluating the remaining matches in the sequence, the abstract value for whole sequence is the meet of v_{val} and v_{val}' (line 12). Similarly, if *ms'* represents the residualized version of the remaining matches, we get the residualized version of the entire sequence by prepending *m'* onto *ms'*.

Figure 4.8 gives an example of how case expressions are handled, and in particular, why getting an accurate estimate of the unmatched part v_{rem} is important. Consider invoking PE_e with an input expression (call it e_0) and environment *E* as shown in figures 4.8(a) and (b). We first evaluate the controlling expression (which in this case is simply the variable `i`) to get its value $v = Int(0|1)$. Next we invoke PE_{ms} with a list of matches $ms = [m_1, m_2]$ with m_1 and m_2 as shown in figure 4.8(a), and $v = Int(0|1)$. The first lines of figures 4.8(c) and (d) show two possible results of specializing match m_1 to $v = Int(0|1)$ (via a call to PE_m as per line 10 of figure 4.7). The two versions agree on the residual version *m'* of the match, and on the value v_{val} of the literal 0 on the right-hand-side of the match, but differ on v_{rem}, which is supposed to represent the part of *Int(0 | 1)* that did not match pattern `Int 0` of match m_1. In particular, the first version is able to constrain v_{rem} to a smaller set of values than the latter.

The benefit of the increased accuracy is evident from comparing the second lines of figure 4.8(c) and (d), which show the results of partially evaluating match m_2 with the new incoming value *v* set to the v_{rem} of the preceding match. In the former case, the variable `x` of pattern `Int x` is bound to *1* whereas in the latter case, it is bound to *0|1*. As a result, in the former case we are able to replace the expression *1* with the simpler residual expression *2*, whereas in the latter case, we have lost this opportunity.

As line 19 of figure 4.7 specifies, the essential computation of matching an abstract value against a pattern and determining the remainder value v_{rem} is performed by the *match* function of the *AbstractValue* module. We will discuss this module and function in more detail in chapter 5. We will discuss an alternate technique based on aggressive specialization to improve residual case expressions in section 6.2.2.

The scheme described above for pattern matching against abstract values is novel to the best of our knowledge. It is most similar to the methods used in class hierarchy analysis [18] of object oriented languages. In that case, functions are defined in terms of a sequence of cases distinguished by the classes of their formals. Given a set of possible values for the classes of the actuals, the analysis divides the set of classes up into subsets such that each subset is associated with a case. A value in a subset is associated with a particular case if that case is the most closely matching. SCF-ML, on the other hand, has a “first-match” rather than a “closest match” semantics, matches against patterns rather than classes, and represents sets of values implicitly with abstract values rather than explicitly as sets of values. SCF-ML also makes different decisions in how to approximate the match process. In particular, it avoids intersection of sets since intersection is expensive on abstract values.

4.4.6 Function Calls

On encountering a function call |*f e*| we first partially evaluate expression *e* (line 40 of figure 4.5), construct a new contour key for the callee function *f* given the current context represented by contour key *ck* and value *v* of *e* (line 41), look up any previously computed return value (if one exists, otherwise use the abstract value ***0***) recorded for *ck* in the cache (line 43-44), and schedule the function corresponding to *ck* for re-execution on argument value *v* by pushing the tagged pair *MAY(ck,v)* on the worklist (line 45). As explained in the previous section, the *MAY* tag ensures that when the worklist task is processed, if the cache already has an entry for *ck* with argument *v* (or an abstract value lower than *v* in the abstract value lattice), the task will not be re-processed. To this end (line 31, figure 4.4), the partial evaluator meets the new value *v* with the existing value $vArg_{old}$ in the cache and optionally widens the result (line 32, figure 4.4); only if the result of meeting and widening is not different from *v* does the partial evaluator process the value.

The above steps are, for the most part, standard when processing function callsites in context-sensitive interprocedural analysis. One deviation is the manner in which the contour key is constructed, i.e., how the *mkContour* function works. This function is explained in chapter 6.

A more subtle issue is that of “breadth-first” processing of contours, i.e., the fact that we place a callee contour on the worklist instead of processing it immediately. We discuss in section 6.3 an unexpected negative interaction between this order of processing and our widening strategy, and a workaround for this problem.

4.4.7 Special Forms for Manipulating Maps

Figure 4.9 shows how the two remaining kinds of expression are partially evaluated. In particular, it shows how the *map_map* and *map_unionWith* special forms are evaluated. Each of these forms contains an anonymous function which needs to be handled with care to maximize the accuracy of evaluating the form. In what follows, we will focus on the *map_map* form. The *map_unionWith* form has a similar implementation.

Recall the intended concrete semantics of the *map_map* form. Given, for instance, a map `m` equal to *[(1, 13), (23,22)]*, the expression `map_map (fn x => x * 2) m`

```
| PE_e (e_in as |map_map (fn x => e) e_map|, E, ck) =
  let  (e'_map, v_map)    = PE_e (e_map, E, ck)
       (e_out, v_out)     = execMapMapConservatively(e_in, E, v_map, e'_map, ck)
  in if isLeafExp e then
       let  v'_out      = AbstractValue.mapMap
                              (fn v => let  E'     = AbstractEnvironment.insert(E, x, v)
                                            (_, v'')  = PE_e (e, E', ck)
                                       in v'' end)
                              v_map
       in (e_out, v'_out) end
     else (e_out, v_out)
  end

| PE_e (e_in as |map_unionWith (fn x => e) e_maps|, E, ck) =
  let  (e'_maps, v_maps)  = PE_e (e_maps, E, ck)
       (e_out, v_out)     = execMapUnionWithConservatively(e_in, E, v_maps, e'_maps, ck)
  in if isLeafExp e then
       let  v'_out    = AbstractValue.mapUnionWith
                            (fn vsTuple => let  E'       = AbstractEnvironment.insert(E, x, vsTuple)
                                                (_, v'')  = PE_e (e, E', ck)
                                           in v'' end)
                            v_maps
       in (e_out, v'_out) end
     else (e_out, v_out)
  end

and execMapMapConservatively(|map_map (fn x => e) e_map|, E, v_map, e'_map, ck) =
  let  (_, v_rng)      = AbstractValue.collapseMap v_map
       E'              = AbstractEnvironment.insert(E, x, v_rng)
       (e'', v'')      = PE_e (e, E', ck)
       v_out           = AbstractValue.mapMap (fn _=>AbstractValue. freshenIds v'') v_map
  in (|map_map (fn x => e'') e'_map|, v_out) end

and execMapUnionWithConservatively(|map_unionWith (fn x => e) e_maps|, E, v_maps, e'_maps, ck) =
  let  (v_map,v'_map)  = let E = AbstractValue.match(|(m_1, m_2)|, vArg))
                           in (E.find(|x|), E.find(|y|)) end
       (_, v_rng)      = AbstractValue.collapseMap v_map
       (_, v'_rng)     = AbstractValue.collapseMap v'_map
       E'              = AbstractEnvironment.insert(E, x, |(v_rng,v'_rng)|)
       (e'',v'')       = PE_e (e, E', ck)
       v_out           = AbstractValue.mapUnionWith (fn _=> AbstractValue.freshenIds v'') v_maps
  in (|map_unionWith (fn x => e'') e'_maps|, v_out) end

and isLeafExp (e: SCF_ML.expr): bool = ... true if e calls no functions, else false...
```

FIGURE 4.9: Partial Evaluation of Map Iterators in SCF

evaluates to the map *[(1, 26), (23,44)]*. The *map_map* special form is therefore an iterator to map the range of the incoming map.

Given the above example, a traditional ML interpreter for this expression would first evaluate the anonymous function (to get a closure as its value), evaluate the map `m` (to *[(1, 13), (23,22)]*), and then apply the primitive operation `map_map` to these two argument values.[1] In particular, the primitive operation would iterate through the range of the second argument value (the map), replacing each element in the range with the result of applying the first argument value (the closure) to it. This is a *higher-order* approach since it assumes a way to represent functions as values. This approach allows the interpreter to avoid special rules for the many special ways in which we may use an anonymous function. In particular, the module for representing concrete values (which presumably performs primitive operations on these values) can provide a single *apply* function of type *primop * value -> value* which evaluates all primitive operations, including ones with function-valued arguments.

Figure 4.10(a) shows how the `map_map` primitive operation is handled in the higher-order case. As the code emphasizes, the `map_map` operation does not have to be handled differently from any primitive operation *p*. The code assumes that the module that represents concrete values is called *Value*.

The higher-order approach assumes the ability to represent functions as values. If, however, an interpreter cannot represent functions as values, it can still handle particular

```
I_e(|p e|, E) =
let  v = I_e(e, E) in Value.apply(p, v) end
```

(a) Higher-Order Approach to Evaluating Primops on Function-Valued Expressions

```
I_e(|map_map (fn x => e) e'|, E) =
let  v_map = I_e(e', E)
in Value.apply_mapMap (fn rngVal =>
                          let E' = Environment.insert(E, x, rngVal)
                          in I_e(e, E') end)
                        v_map
end
```

(b) First-Order Approach to Evaluating Primops on Function-Valued Expressions

FIGURE 4.10: Partial Evaluation of Map Iterators in SCF

1.In fact, assuming as we do that `map_map` is a primitive operation, a traditional ML interpreter would first construct a closure to represent the *curried* application of the primitive operation `map_map` to the anonymous function, and apply this closure to get the effect we describe. The currying, however, is incidental in our case, and we could just as well write `map_map f e` as `map_map(f,e)`.

special forms that contain the syntax of anonymous functions on a case-by-case basis. We will call this the *first-order approach* to interpreting anonymous functions. Figure 4.10(b) shows how this approach works.

Suppose the *Value* module which implements primitive operations on concrete values has an additional function called *apply_mapMap* of type *(value -> value) * value -> value* which takes a function *f* and a value v_{map} (*m* is required to represent a map), iterates through v_{map} and applies *f* to each value in v_{map}'s range, and returns the resulting map. Now the interpreter could, given the form *map_map (fn x => e) e'* to evaluate, and environment *E* to evaluate it in, first evaluate *e'* to get its value v_{map} (line 2), and then invoke *apply_mapMap* on this map with the appropriate function as argument (lines 3-6). The closure, which will be applied to every value in the map, first augments the environment to contain the range value (line 4), and then calls back to the core interpreter to evaluate the expression *e1* in this augmented environment (line 5).

In the last two paragraphs, we have discussed concretely evaluating (SCF-)ML expressions. For concrete evaluation, representing functions as values is well understood, and a modern language that featured anonymous functions would likely use the higher-order approach above. For abstract evaluation, however, the picture is quite different. There is no standard approach for directly representing sets of functions.[1] In particular, the Regular Tree Expression formalism on which our Abstract Value representation is based does not represent sets of functions. The partial evaluator in SCF-ML therefore uses a variant of the first-order approach outlined above. Comparing lines 2 and 5-9 of figure 4.9 with lines 2-6 of figure 4.10(b), we see that except for the use of the *AbstractValue* and *AbstractEnvironment* modules instead of the *Value* and *Environment* modules respectively, the partial evaluator uses precisely the first-order approach described in the previous paragraph.

1.Constraint-based approaches such as set-based analysis [26] and various type-inference algorithms (such as the popular Hindley-Milner [42] scheme for type inference of ML programs) represent sets of functions implicitly by gathering constraints on the domain and range of the function. It is unclear, however, how to perform these analyses in a controlled context-sensitive manner. As we will discuss later in the dissertation, SCF decides whether to analyze a function in a new context based on the abstract values computed so far for the function argument. The only kind of context-sensitivity we are aware of that is provided by constraint-based approaches is the parametric polymorphism of the Hindley-Milner algorithm. At first glance, this kind of context-sensitivity seems insufficient for our purposes.

map(
must [(1,7), [32,12],
may [])
(a)

map(
must [(1,8), [32,13],
may [])
(b)

map(
must [(1,(8|13)), [32,(8|13)],
may [])
(c)

FIGURE 4.11: Partial Evaluation of Map Iterators in SCF

A caveat is that for reasons of efficiency, we use the first-order approach directly only when the anonymous function involved has no callees, i.e., it is a leaf function (as tested in line 3 of figure 4.9). The reason for this is that iterators like *mapMap* evaluate the body of the anonymous function in a highly context-sensitive way. In particular, *for each value v in the range of the map it is iterating over*, it evaluates the body of the anonymous function with *v* as the argument value of the anonymous function. If the body of the anonymous function called some other function *f*, then each time the call-site to this function is evaluated, SCF would place a task on the worklist (as per line 44 of figure 4.5). Further, each time a task is processed, it is likely that the result of callee *f* would change (since the anonymous function calling *f* was invoked with a different value *v* from the range of the map). By line 43 of figure 4.4, a *MUST* task would then be added to the worklist to re-analyze the function containing the anonymous function. Given that an abstract map can easily have dozens of entries, this scheme results in functions that use map operations being re-analyzed a large number of times.

Such excessive re-analysis of functions hurts in two ways. First, it makes the partial evaluator slow. Second, because the widening strategy of the partial evaluator is triggered essentially by the number of times a function (actually the related context key) is re-analyzed, it is possible that the re-execution will trigger widening. Widening stops re-analysis, but often sacrifices precision drastically in return.

To avoid excessive re-analysis, in the case that the body of the anonymous function has callees, we merge all the abstract values in the range of the map into one abstract value (line 28 of figure 4.9), evaluate the anonymous function on the resulting abstract value to get an “abstract range value” *v”* (lines 29 and 30), and conservatively replace each value in the range of the map with a fresh version of *v”* (line 31). The fresh version differs from the original version in that all its IDs are distinct from any previously used by the partial evaluator. Recall that if values have the same ID, by the semantics of abstract values, they are correlated to represent the same concrete values, a correlation which does not exist among the elements of the range of the map. By freshening IDs, we avoid the correlation.

Figure 4.11 illustrates the loss of accuracy that results from range merging. Consider invoking *mapMap* with function *(fn x => x + 1)* on the abstract map shown in figure 4.11(a). Figure 4.11(b) shows the map that would result from using the accurate first-order approach. If, on the other hand, we used the conservative range-merging step of the previous paragraph (say because the map function was *(fn x => inc x)* where *inc* is a user-defined function that adds one to its input), the abstract map that results is the less accurate one shown in figure 4.11(c).[1]

We have focused so far on the abstract value produced when evaluating the special form, and not on the residual value. The complication here is in deciding what the residual form of the body of the anonymous function should be. In the case that we use the range-merging technique described above, we get a single aggregate residualized expression as the result of partially evaluating the body of the anonymous function (line 32 of figure 4.9) on the aggregate abstract value we first construct. We simply use this expression as the body of the residual anonymous function (line 11).

When the range-merging technique is not used, the anonymous function is partially evaluated separately on each of the abstract values in the range of the map (lines 4-8), giving rise to many corresponding residual versions of the function. There is no simple and clearly profitable way to aggregate the many residual versions so generated into a single residual function. As specified on line 6 of figure 4.9, SCF currently ignores these residual versions of the function. Instead (line 9), it generates the residual version by using the range-merging technique. Even when it uses the more precise technique for generating the resulting abstract values, SCF therefore uses the less precise range-merging technique to generate the residual anonymous function. An inspection of residualized code reveals that for current optimizations and their inputs, this decision does not miss any profitable opportunities for residualization.

1.This example shows that it is possible to reduce the number of circumstances in which the less accurate rule is used by pre-processing the incoming program. In particular, we can inline away callee functions (call-trees rooted at the callee, in general).

4.5 Summary

In this chapter, we described the structure of the core partial evaluator module. The module is structured as a fixpoint loop that partially evaluates individual functions. Partially evaluating functions requires partially evaluating the expressions that comprise their bodies. Most types of expressions were straightforward to evaluate; case expressions and forms related to built-in maps were the exception.

The module makes crucial use of helper modules that define abstract values, define the context-sensitivity strategy, decide what values to rematerialize, and define the widening strategy. In the next two chapters, we show how these modules are implemented.

5. Implementing Abstract Values

The partial evaluator of the previous chapter assumes a module *AbstractValue* that represents sets of concrete values. In this chapter, we specify the interface *ABSTRACT_VALUE* required of the *AbstractValue* module, the semantics required of these operations, the internal representation of abstract values, and the algorithms used in SCF to implement the operations required of the *ABSTRACT_VALUE* interface.

As discussed in section 3.2.2, abstract values are based on the Regular Tree Expression (RTE) representation of Aiken and Murphy [3]. Even though the expressiveness and complexity of RTEs have been studied elsewhere, the description of this section is of interest for two reasons. First, SCF augments conventional RTEs with new features that enable more precise manipulation of abstract values of interest to the partial evaluator. We motivate and describe precisely the innovations involved. Second, SCF avoids certain tra-

```
signature ABSTRACT_VALUE = sig
  type value

  val  top            : value
  val  bottom         : value

  val mkInt           : int     -> value
  val mkBool          : bool    -> value
  val mkString        : string  -> value
  val mkChar          : char    -> value
  val mkEmptyMap      : unit    -> value

  val meet            : value * value -> value
  val mustBeEqual     : value * value -> bool
  val isSingleton     : value         -> bool
  val isScalar        : value         -> bool

  val mkTuple         : value list              -> value
  val mkTagval        : string * value          -> value
  val evalPrimop      : SCF_ML.primop * value -> value
  val match           : SCF_ML.pattern * value -> (AbstractEnvironment.env * value) option

  val mapMap          : (value -> value) * value          -> value
  val mapUnionWith    : (value -> value) * value * value  -> value
  val collapseMap     : value                             -> value * value
  val findLiveKeys    : value                             -> value
  val freshenIds      : value                             -> value
end
```

FIGURE 5.1: Signature of the *AbstractValue* Module.

ditional features of RTEs and also implements the remaining features in a relatively simple way. The simpler implementation is nevertheless sufficient to allow effective partial evaluation. We describe precisely this simple but effective implementation.

5.1 Interface

Figure 5.1 specifies the *ABSTRACT_VALUE* interface. The types, functions and values are those referenced by the partial evaluator. Function calls *mkTuple vs* and *mkTagval(c,v)* are written using the syntactic sugars |*(vs)*| and |*c v*| respectively in lines 20 and 24 respectively of figure 4.5.

Table 2 specifies the behavior required of the functions in the *ABSTRACT_VALUE* interface. Given an invocation of the form specified in the left column of the table, the right column specifies the set of possible results for the invocation. An implementation of the *AbstractValue* module is only required to produce a result value from this set, in order to be conservative.

The table uses the following notation. Given abstract value *v* and abstract environment *E*, $\underline{v}$ and $\underline{E}$ respectively are the corresponding sets of concrete values and concrete environments under the correspondence Ψ of section 3.2.2.2. If *F* is a function operating on abstract values, then $\underline{F}$ is the set of functions such that if $f \in \underline{F}$ maps concrete value *t* to *t'*, then there is some *v*, *v'* such that *F(v) = v'* and $t, t' \in \underline{v}, \underline{v}'$ respectively. Finally, *match*, *mapMap* and *mapUnionWith* are the obvious concrete-domain versions of the corresponding functions on the left-hand column.

Returning conservative results as specified by the right-hand column gives the interface functions considerable leeway. Consider the *meet* function, for instance. According to the specification, given abstract values $v_1 = 1 \mid 2 \mid 3$, and $v_2 = 12 \mid 24$ (so that $\underline{v}_1 = \{1, 2, 3\}$ $\underline{v}_2 = \{12, 24\}$), the result of $meet(v_1, v_2)$ may be any *v* such that $\{1, 2, 3, 12, 24\} \subseteq v$. Thus $v = 1 \mid 2 \mid 3 \mid 12 \mid 24$ is a correct result, and so is $v = \mathbf{1}$ (i.e. $\underline{v}$ is the universal set).

In the case of functions *mustBeEqual*, *isSingleton* and *isScalar*, it is always conservative to return *false* as an answer. In other words, if these functions return *true* as an answer, we can be sure that the corresponding abstract values represent equal sets, singleton sets

Table 2: Conservative Specifications for Functions of the AbstractValue Interface

Function	Allowed Results of Function
mkInt i	$\{v \mid \underline{v} = \{i\}\}$
mkBool b	$\{v \mid \underline{v} = \{b\}\}$
mkString s	$\{v \mid \underline{v} = \{s\}\}$
mkChar c	$\{v \mid \underline{v} = \{c\}\}$
mkEmptyMap()	$\{v \mid \underline{v} = \{[]\}\}$
meet(v_1, v_2)	$\{v \mid \underline{v_1} \cup \underline{v_2} \subseteq \underline{v}\}$
mustBeEqual(v_1,v_2)	$\{ b \mid b \Rightarrow \underline{v_1} = \underline{v_2}\}$
isSingleton v	$\{b \mid b \Rightarrow \lvert\underline{v}\rvert = 1\}$
isScalar v	$\{b \mid b => \forall t \in \underline{v} . t \in Int \vee t \in Bool \vee t \in String \vee t \in Char\}$
mkTuple $v_1,\ldots,v_n$	$\{v' \mid \forall t_1 \ldots t_n \in \underline{v_1},\ldots,\underline{v_n}. (t_1,\ldots, t_n) \in \underline{v'}\}$
mkTagVal(*c*, *v*)	$\{v' \mid \forall t \in \underline{v}. c\ t \in \underline{v'}\}$
evalPrimop(*p*,*v*)	$\{v' \mid \forall t \in \underline{v}. p(t) \in \underline{v'}\}$
match(*pt*,*v*)	$\{(E, v') \mid \forall t \in \underline{v}.\ if\ match(pt,t) = SOME\ e\ then\ e \in \underline{E}\ else\ t \in \underline{v'}\ \}$
mapMap(*F*, *v*)	$\{v' \mid \forall (f, m) \in \underline{F} \times \underline{v}.\ mapMap(f, m) \in \underline{v'}\}$
mapUnionWith(*F*,v_1,v_2)	$\{v' \mid \forall (f, m_1, m_2) \in \underline{F} \times \underline{v_1} \times \underline{v_2}.\ mapUnionWith(f, m_1, m_2) \in \underline{v'}\}$
collapseMap v	$\{(v_1, v_2) \mid \forall m \in \underline{v} . \forall (t_1, t_2) \in (dom\ m) \times (rng\ m).\ (t_1, t_2) \in \underline{v_1} \times \underline{v_2})\}$
findLiveKeys v	$\{v' \mid \forall t_k \in H . not\ (\forall (m_1, m_2) \in \underline{v} . m_1(t_k) = m_2(t_k)) \Rightarrow t_k \in \underline{v}'\ \}$
freshenIds v	$\{v' \mid v'$ *is the result of replacing each ID in v with a fresh one*$\}$

and sets of scalars respectively. Otherwise, we have to assume that nothing is known on these fronts.

In fact, according to table 2, any time one of the functions returns an abstract value, it is acceptable to return ***1*** as the result (except for the *freshenIds* function). Although functions implemented in this manner would be correct, simple and extremely efficient, they would also lose an unacceptable amount of information. In particular, the partial evaluator would never be able to profitably residualize an expression evaluated using these functions since the abstract value for the expression would never be singleton. The challenge

```
1  datatype value =
2      top                               (* Written as 0 in this dissertation *)
3    | bottom  of valueID                (* Written as 1 in this dissertation *)
4    | tuple   of value list * valueID
5    | tagval  of string * value * valueID
6    | alt     of value list * valueID
7    | fix     of value * valueID
8    | rec
9    | map     of must_map * may_map * valueID
10   | Int of int | Bool of boolean | String of string | Char of char
11
12 and must_map  = must   of (value * value) list
13 and may_map   = may    of (value * value) list
14 withtype valueID = string
15
16 fun mkInt i        = Int i
17 fun mkBool b       = Bool b
18 fun mkString s     = String s
19 fun mkChar c       = Char c
20 fun mkEmptyMap() = map(must_map [], may_map [], newId())
```

FIGURE 5.2: The Abstract Value Datatype.
We assume a function *newId()* that generates a unique string identifier on each invocation.

in implementing the interface functions is to ensure that they are correct, simple enough that they are feasible to implement, efficient enough that they do not add an unacceptable overhead to partial evaluation and accurate enough that they expose as many residualization opportunities to the partial evaluator as possible.

5.2 Internal Representation

Figure 5.2 specifies how abstract values are represented internally in SCF. The internal representation corresponds directly to the definition of abstract values of table 1 of the previous chapter. This correspondence is not required. For instance, Aiken and Murphy [3] use an equational representation called *leaf linear form* as their internal representation for regular tree expressions. The figure also specifies (on line 20) that an empty map is simply a map with no “may” or “must” entries.

The definition leaves unspecified some key aspects of the internal representation. Recall, for instance that keys in the *must* lists of maps are required to be singleton. More broadly, SCF maintains all abstract values in a *normal form*. Only a subset of the values representable by the *value* type of figure 5.2 are in normal form. We define the normal

$\mathbf{0} \mid v$	$\rightarrow v$	[OR-TOP]
$\mathbf{1} \mid v$	$\rightarrow \mathbf{1}$	[OR-BOT]
$v \mid (v_1 \mid \ldots \mid v_n)$	$\rightarrow v \mid v_1 \mid \ldots \mid v_n$	[OR-ASSOC]
$(v_1 \mid \ldots \mid v_n) \mid v$	$\rightarrow v \mid v_1 \mid \ldots \mid v_n$	[OR-ASSOC’]
$v \mid v'$	$\rightarrow v' \mid v$, *if* $v' < v$	[OR-LT]
$v \mid v'$	$\rightarrow v$, *if* $v = v'$	[OR-EQ]
$(v_1 , \ldots , v_n) \mid (u_1 , \ldots , u_n)$	$\rightarrow (v_1 \mid u_1, \ldots , v_n \mid u_n)$	[OR-TPL]
$m_1 \mid m_2$	$\rightarrow$ *meetMaps*(m_1 , m_2), *where* m_i *is a map*	[OR-MAP]
$(\ldots, \mathbf{0} , \ldots)$	$\rightarrow \mathbf{0}$	[ANDZ-TPL]
$c\ \mathbf{0}$	$\rightarrow \mathbf{0}$	[ANDZ-TAG]
$fix\ .\ \mathbf{0}$	$\rightarrow \mathbf{0}$	[ANDZ-FIX]
$map(must(\ldots, (\mathbf{0}, v), \ldots), \ldots)$	$\rightarrow \mathbf{0}$	[ANDZ-MUST-DOM]
$map(must(\ldots, (v, \mathbf{0}), \ldots), \ldots)$	$\rightarrow \mathbf{0}$	[ANDZ-MUST-RNG]
$map(\ldots, may(\ldots,v_i, (\mathbf{0}, v), v_{i+2},\ldots))$	$\rightarrow map(\ldots, may(\ldots,v_i, v_{i+2},\ldots))$	[ANDZ-MAY-DOM]
$map(\ldots, may(\ldots,v_i, (v, \mathbf{0}), v_{i+2},\ldots))$	$\rightarrow map(\ldots, may(\ldots,v_i, v_{i+2},\ldots))$	[ANDZ-MAY-RNG]
$map(\ldots, (v_i, v'_i), (v_{i+1}, v'_{i+1}), \ldots)$	$\rightarrow map(\ldots, (v_{i+1}, v'_{i+1}), (v_i, v'_i), \ldots)$, *if* $v_{i+1} < v_i$	[MDOM-LT]
$map(\ldots, (v_i, v'_i), (v_{i+1}, v'_{i+1}), \ldots)$	$\rightarrow map(\ldots, (v_i, v'_i), \ldots)$, *if* $(v_i, v'_i) = (v_{i+1}, v'_{i+1})$	[MDOM-EQ]
(v)	$\rightarrow v$	[TPL-ELIM]
$fix.\ v$	$\rightarrow v$, *if rec not free in v*	[FIX-ELIM]

FIGURE 5.3: Rewrite Rules for Normalizing Abstract Values.
Function *meetMaps* is defined in figure 5.5.

form in terms of a set of rewrite rules which convert to normal form an arbitrary value of type *value*. For reasons discussed below, we divide the rules into a subset called the “approximating rules” (comprising the OR-MAP and OR-TPL rules) and the “precise rules” (comprising the remaining rules).

The first seven rules, from OR-TOP through OR-TPL restrict the form of *alt* abstract values. The first two rules follow from the interpretation of *alt* as set union and that of abstract values **0** and **1** as the empty and universal set respectively. These two rules have the effect of keeping abstract values compact. The third and fourth rule constrains *alt* values to be maximally flat: a value *alt [1, alt [2, 3]]* is rewritten as *alt [1, 2, 3]*.

The OR-LT and OR-EQ rules assume the presence of a total order < (with corresponding equality relation =) on abstract values. We discuss a possible definition of this order at the end of this section. Given the order, the OR-LT rule says that abstract values constituting an or-value should be sorted under the order, and the OR-EQ rule says that the or-value should not contain duplicates. These two rules make or-abstract values relatively immune to the order in which their components were unioned into them, and to the number of times a given abstract value is unioned in to a result value.

Rules OR-TPL and OR-MAP perform "eager merges" to prevent two distinct *tuple*, *fix* or *map* abstract values from co-existing at the top level of an *alt* value. The tuple rule, for instance, converts the abstract value *(1, 2) | (3, 4)* to *(1|3, 2|4)*. These rules are different from the previous ones, in that the abstract values they produce may be approximations of their original versions. However, these rules too are aimed at reducing the size of the abstract values produced. The motivation for these rules is somewhat less obvious, since it seems that at first glance, an application of the rule such as the above example seems unlikely to reduce the size of abstract value by much (in the above example, for instance, assuming list-based implementations, the two forms occupy the same amount of memory). The big win from these rules becomes clear only when they are combined with other rules, and when we take into account a common pattern of abstract values that arises when partially evaluating program optimizations.

In particular, as exemplified by the specification of the dead assignment elimination optimization in figure 3.7, program optimizations often pass around tuples containing one or more maps. These maps have as their domain program structures such as variables or labels in the AST being optimized. Now recall that *alt* values are created when the *meet* function is called on abstract values. This occurs most commonly either when the partial evaluator has arrived at a program point in the optimization via two or more sides of a case statement (line 12 of figure 4.7) or when the evaluator is re-analyzing a function that is already in the cache (line 31 of figure 4.4). In either case, the two tuples being met typically contain maps with many common entries. For instance, in the former case, the case statement may handle a particular phrase in the AST in different ways, but this phrase (being a small part of the AST) will only result in differences in a small part of the map. Normalizing tuples by meeting the maps they contain (and thereby avoiding duplication of values) can therefore save a substantial amount of space.

The OR-MAP rule is specified in terms of the unspecified *meetMaps* function, which given two maps, returns a map. The only requirement on this function is that the resulting set of maps be a superset of the incoming maps. An implementation of the function is discussed later in the chapter.

Unlike tuples and maps we do allow two tagged values with the same tag to co-exist in an *alt* value. For instance *cons(1, nil) | cons(2,nil)* is in normal form, and we do not rewrite for instance to *cons(1|2, nil)*. This exception is again motivated by the pragmatics of partially evaluating optimizations. It is common when optimizing to have *alt*'s of tagged abstract values representing two different versions of a program phrase. For instance, branch folding may result in two alternate sides of branches, represented as `x = y | p = y + 1` (say the incoming AST performed two completely different actions on two sides of a constant branch). If we merged the two options prematurely in an effort to normalize, the result of the optimization would instead be represented as `(x|p) = y | (y + 1)`. If the copy propagation optimization were then staged on this result, it would have to account for the possibility that the incoming AST performs the copy `p = y`, and produce residual code for downstream uses of `p` to propagate this copy if necessary.

In summary, we perform eager merging in order to reduce the size of abstract values, but avoid it when the cost in accuracy is too high.

Rules ANDZ-TPL through ANDZ-MDOM simply exploit the fact that the empty set is a "multiplicative zero". Take rule ANDZ_TPL for instance. Recall that the abstract tuple $v = (v_1,...,v_n)$ stands for the set $\underline{v}$ of all concrete tuples $(t_1,...,t_n)$ where for all i, $t_i \in \underline{v}_i$. However, if for some i, $v_i = \boldsymbol{0}$, it follows that $\underline{v}_i = \{\}$, there is no possible satisfying value of t_i, $\underline{v} = \{\}$, and v is equivalent to $\boldsymbol{0}$. These rules preserve the set of conforming values of the abstract values being re-written: the set is always the empty set. Normalizing in this manner reduces the space taken by abstract values, and speeds up operations in the not-uncommon case that an abstract value has no conforming value[1].

Rules MDOM-LT and MDOM-EQ sort abstract maps by their abstract key values. The benefits of this step are similar to those of sorting *alt*-values. Two maps that map structurally identical keys to identical values are guaranteed to be identical, allowing flexible equality testing via purely structural comparison.

The final two rules remove operators. TPL-ELIM, simply replaces tuples with a single field with the contents of that field. FIX-ELIM gets rid of the *fix* operator when it applies to values with no free *rec* expressions. For example, *fix. 1 | 2* and *fix. fix. (nil | cons(1, rec))* are rewritten to *1 | 2* and *fix. cons(1, rec)* (in the latter case, although the outer *fix* contains a *rec*, the *rec* is not free). A rule implicit in the syntax of the abstract values is that "or" expressions with one disjunct are rewritten to just the disjunct.

We have not specified the total order < used while normalizing. Any total order on abstract values that, ignoring IDs of abstract values, designates two structurally identical abstract values as equal is sufficient for the purposes of normalization. In what follows, we will assume a function *compare*: *value * value -> order* (where *order = LESS | GREATER | EQUAL)* that specifies this total order. The particular function used in SCF is *O(n)*.

5.3 Implementing Operations on Abstract Values

In this section, we describe how the functions that comprise the *AbstractValue* interface are implemented. The descriptions are of interest because they show precisely how to implement sufficiently aggressive but reasonably efficient versions of these operations. In particular, it points out the minimum accuracy required by our application domain for various operations, shows how to maintain the normal form incrementally and exploit it, and shows how to maintain abstract value IDs such that the resulting values are semantically consistent. As implemented below, these operations all terminate in polynomial time, as opposed to exponential lower-bounds on accurate versions of some of these operations on RTEs [3].

1.This is not an uncommon case because the first time a function call is abstractly executed, it returns value top (i.e. **0**) as per line 44 of figure 4.5.

5.3.1 *meet*

As per the specification of the *meet* function in table 2, the *meet* of two abstract values *v* and *v'* intuitively results in (a superset of) the union of the sets $\underline{v}$ and $\underline{v}$'. Also, as per the specification of the *alt* form in table 1 of the previous chapter, the abstract value *v* | *v'* rep-

```
(* meet: value * value -> value *)
fun meet(v, v') =
if identical(v, v') then v
else if compare(v,v') = EQUAL then freshenIds v
else meet'(v,v')

(* meet': value * value -> value *)
and meet'(top, top)                 = top
 |  meet'(top, v)                   = v
 |  meet'(v, top)                   = v
 |  meet'(bottom _, _)              = bottom newId()
 |  meet'(_, bottom _)              = bottom newId()
 |  meet'(tuple(vs, _), tuple(vs', _)) =
      let vs'' = ListPair.map meet (vs, vs')
      in tuple(vs'', newId()) end
 |  meet'(u as tagval _, alt (vs, _))   = alt (merge([u], vs), newId())
 |  meet'(alt(us, _), v as tagval _)    = alt (merge([v], us), newId())
 |  meet'(u as tagval_ , v)             = alt (merge([u], [v]), newId())
 |  meet'(u, v as tagval _)             = alt (merge([u], [v]), newId())
 |  meet'(alt (us, _), alt (vs, _))     = alt (merge(us, vs), newId())
 |  meet'(alt (us, _), v)               = alt (merge(us, [v]), newId())
 |  meet'(u, alt (vs, _))               = alt (merge([u], vs), newId())
 |  m eet'(u as fix _, v as fix _)      = alt (merge([u], [v]), newId())
 |  meet'(u as map _, v as map _)       = meetMaps(u, v)

and merge(us, vs) = merge the sorted lists us and vs to get a sorted list without duplicates

... meetMaps as defined in figure 5.5...

and identical(v: value, v': value): boolean=
  identicalAtoms(v, v') orelse idOf v = idOf v'

and identicalAtoms(v, v'): boolean =
  ... true iff v and v' are the same int, bool, string, char or zero-ary tagged value ...

and idOf (v: value): int =
  ... return the id of abstract value v if it has one (undefined otherwise)...

and freshenIds(v: value): value =
  ... replace every ID of v with a fresh one, preserving duplicates...
```

FIGURE 5.4: The *meet* Function.

resents precisely the union just mentioned. Intuitively then, the role of the *meet* function, is simply to produce to the value *alt(v, v', id)*, subject to normal-form requirements, and with an appropriate ID *id*. Figure 5.4 specifies the function. We summarize the key points below.

If the two abstract values being met have the same ID, or if they are the identical integer, character, string, boolean or zero-ary tagged singleton value (e.g. the empty list *nil*), it follows that they are structurally identical. The comparison function over abstract values *compare* must then determine that *compare(v, v') = EQUAL*, so that by the OR-EQ rule of figure 5.3, we return *v* as the meet (line 3).

In the case that the IDs are unequal, but the two values are structurally equal, we return a value that is the structurally equivalent to the input values with fresh IDs. We need to freshen the IDs because leaving an old ID (say from the first argument value) would indicate a correlation between the result value and the old abstract value, when the meet function actually destroys the correlation (the resulting concrete value at run-time may well flow in through the second abstract value).

Lines 8-12 implement the OR-TOP and OR-BOT rewrite rules. A detail is that when implementing the rule $v \mid \mathbf{1} \rightarrow \mathbf{1}$, we tag the new $\mathbf{1}$ with a fresh ID, since we cannot assume any correlation between the new set and the incoming sets. For instance, suppose the *meet* function is being invoked to produce a result value given that two sides a branch evaluate abstractly to *(12 | 54) # 313* and ***1****#312*, all we can conclude is that the resulting value belongs to the set $\underline{\mathbf{1}}$. We cannot, for instance ascribe ID *312* to the new result, since the result may be produced via the first abstract value.

Lines 13-15 implement rewrite rule OR-TPL. Lines 17-23 implement the OR-LT and OR-EQ rules. A subtlety is that at first sight, it seems that lines 17-23 may take two values such as *1 | (2 , 3)* and *(3, 4)* and (by line 20, in this case), produce the value *1 | (2, 3) | (3, 4)* by merging sorted lists *[1, (2 , 3)]* and *[(3, 4)]*. The resulting value is not in normal form, since the OR-TPL rule could be applied to merge the tuples. This case does not arise because SCF-ML follows the same strong typing discipline as ML. In particular, the type discipline prohibits a given value from being possibly both an integer and a tuple, so the value *1 | (2, 3)* could never have formed.

```
type vv_list = (value * value) list

(* meetMaps: value * value -> value*)
fun meetMaps(m as map(must ps, may qs,_), m' as map(must ps', may qs', _)) =
  let  (musts, mays)      = meetMusts([], [], ps, m')
       mays'              = meetMays(mays, qs, m')
       (musts', mays'')   = meetMusts(musts, mays', ps', m)
       mays'''            = meetMays(mays'', qs', m)
  in map(must musts', may mays''', newId()) end

(* meetMusts: vv_list* vv_list * vv_list * value -> vv_list * vv_list *)
and meetMusts(musts, mays, [], _) = (musts, mays)
| meetMusts(musts, mays, (k,v)::ps, m) =
  case find(m, k) of
     MUST v'  => meetMusts(insertSoft(musts, k, meet(v, v')), mays, ps, m)
   | MAY v'   => meetMusts(musts, insertSoft(mays, k, meet(v,v')), ps, m)
   | NOT      => meetMusts(musts, insertSoft(mays, k, v), ps, m)

(* vv_list * vv_list * value -> vv_list *)
and meetMays(mays, [], _) = mays
| meetMays(mays, (k,v)::ps, m) =
  case find(m, k) of
     MUST v'  => meetMays(insertSoft(mays, k, meet(v,v')), ps, m)
   | MAY v'   => meetMays(insertSoft(mays, k, meet(v,v')), ps, m)
   | NOT      => meetMays(insertSoft(mays, k, v), ps, m)

(* vv_list * value * value -> vv_list *)
and insertSoft(ms, k, v) =
  ...suppose kvs = {(k,v)} ∪ {(k', v') ∈ ms| not (mustBeDisjoint(k,k'))};
     remove every (k', v') in kvs from ms;
     insert (k'', v'') into ms where  k'' is the result of meeting all k's in kvs and
                                      v'' is the result of meeting all v's in kvs...

(* find: value * value -> value choice *)
and find(m, k) =
  ...see figure 5.14 for complete definition:
     return MUST v  if (k, v) must be in m
            MAY v   if (k,v) may be in m
            NOT     if k is definitely not in m

(* mustBeDisjoint: value * value -> boolean*)
and mustBeDisjoint(v: value, v': value) =
  ... true if sets v and v' can be guaranteed to be disjoint, false otherwise ...
```

FIGURE 5.5: The *meetMaps* Function.

The *meetMaps* function which merges maps is defined separately in figure 5.5. Recall that as per the rewrite rules, this function is simply required to return an abstract map representing a superset of its incoming maps. Because abstract maps are a novelty of SCF, we

```
1  v  = map( must [(1, 19 | 121), (7, 34)], may[(2 | 3, 39)])
2
3  v’ = map( must[(1, 17 | 23), (8, 1)], may[(3 | 7, 12 | 13), (99, 100)])
4
5  After meetMusts call of line 5, figure 5.5:
6    musts: [(1, 17 | 19 | 23 | 121)]
7    mays: [(7, 12 | 13 | 34)]
8
9  After meetMays call of line 6, figure 5.5:
10   musts: [(1, 17 | 19 | 23 | 121)]
11   mays: [(7, 12 | 13 | 34), (2| 3, 12 | 13 | 39)]
12
13 After meetMusts call of line 7, figure 5.5:
14   musts: [(1, 17 | 19 | 23 | 121)]
15   mays: [(7, 12 | 13 | 34), (2| 3, 12 | 13 | 39), (8, 1)]
16
17 After meetMays call of line 8, figure 5.5:
18   musts: [(1, 17 | 19 | 23 | 121)]
19   mays: [(2 | 3 | 7, 12 | 13 | 34 | 39), (8, 1), (99, 100)]
20
21 The resulting abstract value meetMaps(v, v’):
22   map( must [(1, 17 | 19 | 23 | 121)], may [(2 | 3 | 7, 12 | 13 | 34 | 39), (8, 1), (99, 100)])
```

FIGURE 5.6: Example Showing How the *meetMaps* Function Works.

go into some detail on how this operation is implemented. We use the figure 5.6 to show how the *meetMaps* function works.

The intuition is that given two input abstract maps *v* and *v’*, a key *k* appears in the *must* list of the result only if it is in the *must* list of both inputs. All other keys end up in the *may* list of the result. In both cases, we need to do the appropriate book keeping to gather the values corresponding to the keys. The *meetMaps* function achieves this (lines 5-8 of figure 5.5) by processing the *must* and *may* lists of the incoming maps in turn.

For instance, line 5 of figure 5.5 compares elements of the *must* list of the first map to the second map using the *meetMusts* helper function. Given our example, the latter function would recognize that the key *1* must be in both maps, although it maps to the values *19| 121* and *17 | 23* in the two maps. Since the result map is supposed to represent the union of the two maps, the *meetMusts* function maps the key *1* to the *meet* of these two values (line 15 of figure 5.5). Since *1* must be in both maps, the pair *(1, 17 | 19 | 23 | 121)* is inserted into the *musts* list. On the other hand, since the key *7* must be in the first map (where it maps to *34*), but only may be in the second (where it maps to *12 | 13*), the pair (*7, 12 | 13 | 34)* is inserted into the *mays* list. Lines 6 and 7 of figure 5.6 show these results.

The *meetMaps* function feeds the resulting *mays* list (along with the *may* list *qs* of the first map, and also the second map *m'*) into the *meetMays* function as per line 6 of figure 5.5. Since the key *2 | 3* of the *may* list may overlap with the key *3 | 7* of the map (and the two map to values *39* and *12 | 13* respectively), *meetMays* adds the mapping *(2 | 3, 12 | 13 | 39)* to the *mays* list. We assume a function *mustBeDisjoint: value * value -> boolean* that can tell us conservatively if (the sets corresponding to) two abstract values are definitely disjoint. Lines 10 and 11 of figure 5.6 show the resulting *may*s and *musts* lists.

The *meetMaps* function then processes the *may* and *must* lists of the second input map by calling the *meetMusts* and *meetMays* functions respectively on these lists. The results of these calls are on lines 14-15 and 18-19 respectively of figure 5.6. The resulting lists are packaged into a *map* value to give the result of the entire *meetMaps* function on the example inputs (line 22).

To summarize, the *meet* function has two main complications. First, it needs to preserve the normal form of its inputs, and in particular it needs to implement the *meetMaps* function. Second, it needs to tag its results with fresh IDs where necessary, and more importantly use the old IDs where possible.

5.3.2 *mustBeEqual*

```
fun mustBeEqual(v, v') =
  case compare(v, v') of
    EQUAL => true
  | _        => false
```

FIGURE 5.7: The *mustBeEqual* Function.

Figure 5.7 shows how the *mustBeEqual* function is implemented. Recall that the ordering function *compare(v, v')* on abstract values *v* and *v'* returns EQUAL if *v* and *v'* are structurally identical. Ordinarily, structural equality would be far too restrictive to check equality of sets. For instance, we may be testing the sets represented by the two values *1 | 2* and *2 | 1* for equality, and structural equality would declare that the two are unequal. This is where, as discussed previously, normal form comes to the rescue. The normal form of both values is structurally identical (say *1 | 2*). As discussed previously, normal form rewrites many semantically equal but structurally unequal sets to be structurally equal. The rewrite thus allows abstract values that are structurally unequal but semantically equal to be compared for equality using structural comparisons.

5.3.3 *isSingleton*

The *isSingleton* function, given a value *v*, determines if the set $\underline{v}$ is singleton. Its implementation simply checks if its incoming abstract value contains an *alt* form, a ***1*** or a *map* form with non-empty *may* list within it. If not, it declares the value singleton.

5.3.4 *isScalar*

```
fun isScalar(Int _ | Bool _ | String _ | Char _ )    = true
| isScalar(alt(vs, _))                              = List.all isScalar vs
| isScalar _                                        = false
```

FIGURE 5.8: The *isScalar* Function.

An abstract value is a scalar if it definitely represents a set of scalar concrete values, i.e., integers, booleans, strings or characters. Figure 5.8 shows the linear-time test for this.

5.3.5 *mkTuple*

```
1  fun mkTuple (vs: value list) : value =
2     if List.length vs = 1 then hd vs
3     else if List.exists (fn x => x = top) vs then top
4     else tuple(vs, newId())
```

FIGURE 5.9: The *mkTuple* Function.

Given a list $vs = v_1...v_n$of abstract values, *mkTuple vs* returns a normal-form abstract value representing concrete elements whose *i*th element $t_i \in \underline{v}_i$. Figure 5.9 shows how *mkTuple* is implemented. If *vs* contains only one value, *mkTuple* simply returns that value as required by rewrite rule TPL-ELIM (line 2). If any *v* in *vs* is the empty abstract value *top* (which we often write as ***0***), then as per normal form rule ANDZ-TPL, the entire tuple has value *top* (line 3). If the above two conditions do not hold, the incoming values can be packaged with a new ID to form the required tuple.

```
1  fun mkTagVal(c: string, v: value): value =
2    if v = top then top
3    else tagval(c, v, newId())
```

FIGURE 5.10: The *mkTagVal* Function.

5.3.6 *mkTagVal*

Given abstract value *v* and string *c*, *mkTagVal(c, v)* returns an abstract value *v'* s.t. if $t \in \underline{v}$, $c\ t \in \underline{v}'$. Figure 5.10 does the job in constant time. If *v* is *top*, the *mkTagVal* function returns *top* as the resulting value (line 2), as per normal form rule ANDZ-TAG.

5.3.7 *evalPrimop*

Figure 5.11 shows how the *evalPrimop* function is implemented. The function is given the primitive operation (abbreviated to "primop" below) to be evaluated and an abstract value *v* representing the argument of the primitive operation,[1] returns value *v'* s.t. for each $t \in \underline{v}$, $p\ t \in \underline{v}'$.

Lines 2-3 handle some common but straightforward cases. If argument *v* is *top*, i.e., $\underline{v}$ is empty, so is $\underline{v}'$, regardless of the identity of the primop (line 2). Note that this is not necessarily so: if we had a primop that ignored its inputs and returned a constant value, we could add an additional case for this operation.

If *v* is *bottom*,[2] so is *v'*, regardless of the operator. There are two subtleties here. First, consider the case that the operator is *sgn: int -> bool*, i.e., a function which gets the sign of its input. In this case, even if the argument is *bottom*, we could deduce that the return value has to be *true | false*, and not *bottom*. By returning *bottom*, we seem to lose accuracy and allow arbitrary integers for instance to leak into the return value (since all integers are part of the set $\underline{bottom}$). The typing discipline of SCF-ML again comes to the rescue: the result of this primop can only be used in a context where a boolean value is expected so any non-boolean value included in the abstract value can be ignored at that point.

The second subtlety is that as specified the returned set has a fresh ID, signifying that its abstract values may not be correlated with any value computed so far. It is conceivable

1.For those unfamiliar with ML, the fact that the primitive operation takes only one value as argument does not restrict it to being a unary operation: the single value may be a tuple of arbitrary arity.
2.Recall that we have been using the notation **1** to represent *bottom*.

```
1   (* SCF_ML.primop * value -> value *)
2   fun evalPrimop(_, top)          = top
3   | evalPrimop(_, bottom _ )      = bottom (newId())
4
5   | evalPrimop(|~|, Int i)        = Int (~i)
6   | evalPrimop(|~|, alt (us, _))  =
7     ...compute evalPrimop(|~|, u) for all us in u, and meet the results...
8   ...
9   | evalPrimop(|=|, |(Int i, Int j)|) = if i = j then Bool true else Bool false
10  | evalPrimop(|=|, |(alt(us, id), alt(vs, id’))|) =
11    if id = id’ then
12      Bool true
13    else
14      ...compute evalPrimop(|=|,|(u, v)|) for all u, v in us, vs in cross product, and meet the results...
15  |...
16  | evalPrimop(|map_insert|, |($v_{map}$, $v_{key}$, $v_{val}$)|)  = execMapInsert($v_{map}$, $v_{key}$, $v_{val}$)
17  | evalPrimop(|map_find|, |($v_{map}$, $v_{key}$)|)           = execMapFind($v_{map}$, $v_{key}$)
18  | evalPrimop(|map_equal|, |($v_{map}$, $v_{key}$)|)          = execMapEqual($v_{map}$, $v_{key}$)
19
20  ... execMapInsert defined in figure 5.12 ...
21  ... execMapFind defined in figure 5.13...
22  ... execMapEqual defined in figure 5.15...
```

FIGURE 5.11: The *evalPrimop* Function.

that the result value may not need a fresh ID: if we had a primop *id: ’a -> ’a* (which simply returned its input), we could have a special rule *evalPrimop(id, v) = v* which leaves the ID of the incoming value unchanged.

Lines 5-9 and 14 are straightforward implementations of the semantics of table 2. Primops are assumed to be the traditional arithmetic and logical operators, along with built-in SCF map operations. We first describe how traditional primops are processed. A primop applied to a singleton tuple abstract value is just the singleton abstract value that results from applying the primop to the corresponding concrete tuple (lines 5 and 9). If the tuple has alternates as its components, the result is just the meet of applying the primop to all tuples formed by picking one alternate from each component (lines 7 and 14). For instance *evalPrimop(|+|, |(1 | 25, 7 | 12)|)* would return *8 | 13 | 32 | 37*, since the primop would be applied to the tuples *(1, 7)*, *(1, 12)*, *(25, 7)* and *(25, 12)*.

An instructive special case is the treatment of the primop for equality if the input tuple is not singleton, but the two component abstract values of the incoming abstract tuple have the same ID (lines 11-12). Since two abstract values with the same ID are guaranteed to be correlated to represent the same concrete value, the equality operation can return the singleton abstract value *true* in this case. This case (and especially its counterpart for checking equality for maps) can be important for evaluating fixpoint equality tests accurately.[1]

As lines 16-18 show, we special-case primitive operations on maps. We discuss in some detail below how these are implemented.

5.3.7.1 The mapInsert Primitive Operation

Figure 5.12 specifies how the abstract insert operation on maps, *execMapInsert*, is implemented. Given three abstract values v_{map}, v_{key} and v_{val}, *execMapInsert*(v_{map}, v_{key}, v_{val}) produces abstract value v such that if for all $m, k, t \in \underline{v}_{map}, \underline{v}_{key}$ and $\underline{v}_{val}$, *map_insert(m, k, t)* $\in \underline{v}$, where *map_insert*, operates on concrete values.

Lines 2-5 address the case where the incoming abstract map is bottom, i.e., it can be any map. Regardless of the incoming map, if $\underline{v}_{key}$ is singleton (say $\underline{v}_{key} = \{k\}$), then we know that *k must* map to one of the elements of $\underline{v}_{val}$ in the result. In line 4, therefore, we handle this case by requiring the *must* binding (v_{key}, v_{val}) be in the resulting map. Since the rest of the bindings on the result map are unconstrained, we also require the *may* binding *(bottom, bottom)* in the result.

One may wonder why, in the case that v_{key} is not singleton, we cannot require a *may* binding (v_{key}, v_{val}) in the result map, in addition to the *(bottom, bottom)* map. The answer lies in the semantics of the *may* list, as specified in table 1 of chapter 3. Essentially, a concrete binding (t_k, t_v) may be in $\underline{v}_{map}$ if, for *any (k, v)* in the *may* list of the abstract map, t_k, $t_v \in \underline{k}, \underline{v}$, as long as *k* is not a key in the *must* list. Thus, for instance, if the map has the form *map(must [], may[(1|2,3), (bottom, bottom)]),* looking up *1* in the map will yield *bottom* (since *1* is not in the *must* list, and $1 \in \underline{bottom}$). As a result, whenever *(bottom, bottom)* is in the *may* list, it is redundant to add any entries to the may list (although currently we do not take special effort to avoid this redundancy).

1.In practice, the equality test is almost always on aggregate data structures (maps or sets, not scalars) that represent the abstract store. These are handled specially in SCF as discussed in the following subsections.

```
1  (* execMapInsert: value * value * value -> value *)
2  fun execMapInsert(bottom _, $v_{key}$, $v_{val}$) =
3    if isSingleton $v_{key}$ then
4      map(must[($v_{key}$, $v_{val}$)], may[(bottom (newId()), bottom (newID()))], newId())
5    else bottom (newId())
6  | execMapInsert(map(must musts, may mays, _), top, _) = ... should not reach here ...
7  | execMapInsert(map(must musts, may mays, _), _, top) = ... should not reach here ...
8  | execMapInsert(map(must musts, may mays, _), $v_{key}$, $v_{val}$) =
9    if isSingleton $v_{key}$ then
10     let musts' = insert(remove(musts, $v_{key}$), $v_{key}$, $v_{val}$)
11     in map(must musts', may mays, newId()) end
12   else
13     let  (musts', values) = removePossibleMatches(musts, $v_{key}$, [], top)
14          $v_{in}$              = meetAll($v_{val}$ :: values)
15          mays'           = insert(mays, $v_{key}$,$v_{in}$)
16     in map(must musts', may mays' , newId()) end
17
18 (* vv_list * value * vv_list * value -> vv_list * value *)
19 and removePossibleMatches([], _, $musts_{out}$, $v_{out}$) = ($musts_{out}$, $v_{out}$)
20 | removePossibleMatches((k, v)::musts, $v_{key}$, $musts_{out}$, $v_{out}$) =
21   if mustBeDisjoint(k, $v_{key}$) then
22     removePossibleMatches(musts, $v_{key}$, $musts_{out}$@(k,v), $v_{out}$)
23   else
24     removePossibleMatches(musts, $v_{key}$, $musts_{out}$,, meet(v, $v_{out}$))
25
26 and insert(ms: vv_list, key: value, v: value): vv_list =
27   ...insert pair (key, v) into list ms, keeping ms sorted and duplicate-free wrt keys;
28     if some mapping (key', value') already exists s.t. key = key',
29       replace it with (meet(key, key'), meet(v, value'))...
30
31 and remove(kvs: vv_list, $v_{key:}$value): vv_list =
32 ... remove key-value pair (k,v) from kvs if k and $v_{key}$ are not disjoint; return the new kvs...
34
35 and meetAll (vs : value list): value = ... meet together all the values v in vs ...
```

FIGURE 5.12: The *execMapInsert* Function.

When the value being inserted into a *bottom* map is not singleton, we therefore simply return a *bottom* abstract map, taking care to give it fresh ID to avoid correlating it with the incoming (or any other) map.

The assertions of lines 6 and 7 follow from the ANDZ-TPL rule: if v_{key} were ***0***, the abstract tuple value *(v_{map}, v_{key}, v_{val})* of line 16, figure 5.11 would have been ***0***, so line 2 of figure 5.11 would have returned ***0*** without even invoking the *execMapInsert* function.

Lines 8-16 handle the common case where non-*top* abstract keys and values are inserted into a non-*bottom* map. In the case where the abstract key is a singleton (lines 9-

11), we just add the binding *(v_{key}, v_{val})* to the *must* list, replacing in the process any existing binding *(v_{key}, v'_{val})* of v_{key} in the *must* list (line 10). The replacement is necessary because normal form requires that map keys are unique (so we cannot have two entries with the same key), and appropriate because the old bindings are overridden by the new ones (so we do not, for instance, meet the old value with the new one). We leave the *may* list unchanged. It is unnecessary to remove bindings for v_{key} in the *may* list since *must* bindings shadow *may* bindings.

The case where the incoming key is not singleton is trickier (lines 12-16). Consider, for instance, adding the binding *(1|2, 88)* to *map(must [(1, 13), (34,12)], may [(2 | 3, 55), (323, 175)])*. A naive approach might simply add the above binding to the *may* list to produce the map *map(must [(1, 13), (34,12)], may[(1 | 2, 88), (2 | 3, 55), (323, 175)])*. A later *map_find* operation on this map with key *1* would return the value *13*, since the binding *(1, 13)* in the *must* list shadows other bindings of *1*.

Reasoning about the required semantics in the concrete domain, we see that the naive approach above is not quite correct. In the case that the incoming concrete map is *[1 -> 13]* $\in$ *map(must [(1, 13), (34,12)], may [(2 | 3, 55), (323, 175)])*, and the concrete binding inserted is *(2, 88)* $\in$ *(1|2, 88)*, the resulting map is *[1 -> 13, 2 -> 88]* and it is indeed correct for a *find* operation on key *1* to return *13*. However, if the concrete key inserted were *(1, 88)* $\in$ *(1|2, 88)*, then we would require the subsequent find operation to return *88*.

Alternatively, we might first remove all bindings *(k, v)* from the *must* list such that *k* is not disjoint with the key v_{key} being added. The resulting map would be *map(must [(34,12)], may [(1 | 2, 88), (2 | 3, 55), (323, 175)])*. This solution does not work in the case above where concrete key *(2, 88)* was inserted: it does not account for the possibility that the old mapping *(1, 13)* was left untouched.

A correct solution is to first remove all bindings *(k, v)* in the *must* list such that *k* overlaps with the key v_{key} being added (line 13), compute the meet of the values *values* so gathered and the incoming value v_{val} into value v_{in} (line 14), and insert the binding *(v_{key}, v_{in})* into the *may* list (line 15). In the example above, the resulting map would be *map(must [(34,12)], may [(1 | 2, 13 | 88), (2 | 3, 55), (323, 175)])*.

```
fun execMapFind(v_map:value, v_key:value): value =
  case find(v_map, v_key) of
    MUST v => mkTagVal("SOME", v)
  | MAY v  => meet(mkTagVal("SOME", v), mkTagVal("NONE", mkTuple []))
  | NOT    => mkTagVal("NONE", mkTuple [])
```

FIGURE 5.13: The *execMapFind* Function.

5.3.7.2 The execMapFind Primitive Operation

Figure 5.13 shows how the *execMapFind* operation on maps, which implements the abstract *map_find* operation, is implemented. Given abstract values v_{map} and v_{key} representing sets of concrete maps and keys respectively, *execMapFind(v_{map},v_{key})* returns an abstract value *v'* such that for each concrete t_{map}, $t_{val} \in \underline{v}_{map}$, $\underline{v}_{key}$, *map_find*(t_{map}, t_{val}) $\in$ $\underline{v}$'.

To understand the functionality required of the *execMapFind* function consider the desired result for *map(must [(1, 2)], may [(1|7, 3)])* (which corresponds to the set of concrete maps *{[1->2], [1->2, 7->3]}*) and key *1, 9* or *1|7*.

- With abstract key *1*, we would like to get back the abstract value *SOME 2*. The concrete value *1* maps to the concrete value *2* in all conforming concrete maps. Generalizing,

```
(*find: value * value -> value choice *)
and find(map(must ms, may ms', _), k) =
  case findMust(ms, k) of
    SOME v => MUST v
  | NONE   => case findMay(ms@ms',k, top) of
                  top  => NOT
                | v    => MAY v

and findMust([], _) = NONE
| findMust((k, v)::kvs, k') =
    if mustBeEqual(k, k') then SOME v
    else findMust(kvs, k')

and findMay([], _, v) = v
| findMay((k, v)::kvs, k', v') =
  let v'' = if mustBeDisjoint(k, k') then v'
            else meet(v,v')
  in findMay(kvs, k', v'') end

... mustBeDisjoint defined in figure 5.5...
```

FIGURE 5.14: The *find* Function on Abstract Maps.

therefore, if *(k, v)* is in the *must* list of v_{map} and *k* must be equal to v_{key}, the *execMapFind* should return abstract value *SOME v*. Line 4 of the *find* helper function of figure 5.14 detects this case, and line 3 of the *execMapFind* returns the appropriate value.

- With abstract key *9*, we would like to get back abstract value *NONE*, since *9* is not in the *must* or *may* list of the incoming map, and therefore does not figure as a key in any of the conforming concrete maps. In general, if v_{key} is disjoint from all keys in v_{map}, the returned abstract value should be *NONE*. Line 6 of the *find* helper function and line 5 of the *execMapFind* function detect this case and return the correct value respectively.
- With abstract key *1|7*, we would like to get back either *SOME 2* (when the concrete key is *1* and either concrete map is used), *SOME 3* (when the concrete key is *7* and the second concrete map is used), or *NONE*. Summing up, we would like to get the abstract value *NONE | SOME 2 | SOME 3* as the result. One way to get (essentially) the latter return value is, given key *1|7*, find all bindings with overlapping keys (in this case *(1, 2)* and *(1|7, 3)*), and meet the range values of these bindings (in this case *2* and *3*). If the meet returns abstract value *v*, return *NONE | SOME v* as the result of the whole *execMapFind* operation. Lines 7 and 4 of the *find* and *execMapFind* operations respectively combine to achieve this effect.

5.3.7.3 The execMapEqual Primitive Operation

Figure 5.15 shows the *execMapEqual* operation on maps, which implements the abstract *map_equal* operation. Given two abstract values v_{map} and v'_{map} representing two sets of concrete maps, *execMapFind*(v_{map}, v'_{map}) returns an abstract value *v'* such that for each concrete $t_{map}, t'_{map} \in \underline{v}_{map}, \underline{v}'_{map}$, $map\_equal(t_{map}, t'_{map}) \in \underline{v}'$.

This operation is different from the *mustBeEqual* operation described previously. That operation, given abstract maps, returns a boolean which is true if the sets represented by the maps are guaranteed to be equal, and false otherwise. The current operation compares the maps in the sets in all pairwise combinations, and returns an abstract value whose value is the result of meeting the pairwise results. Thus, given abstract map v_{eg} =

map (must[], may[(2, 3)]) (which represents the set *{[], [2 -> 3]}* of concrete maps), *mustBeEqual(v_{eg}, v_{eg})* returns concrete value *true,* whereas *execMapEqual(v_{eg}, v_{eg})* could return abstract value *true* | *false* (comparing map *[]* to itself gives *true*, whereas comparing it to *[2->3]* gives *false*). We cannot therefore use structural equality alone for computing *execMapEqual*, as we did for *mustBeEqual*.

The example of the previous paragraph raises the question of whether *execMapEqual* can ever return singleton abstract value *true* when comparing two non-singleton abstract values v_{map} and v'_{map}. After all, if $\underline{v}_{map}$ = *{m_1, m_2, ...}* and $\underline{v}'_{map}$ = *{m_1', m_2', ...}* then if *execMapEqual(m_1, m_1') = true* then it must be true that *execMapEqual(m_1, m_2') = false*. This intuition holds in the absence of value ID tags. The additional correlations provided by tags, however, allow even non-singleton abstract maps to return a singleton abstract value *true* on pairwise comparison. In the case of the example with v_{eg} above, invoking *execMapEqual(v_{eg}, v_{eg})* in fact leads to the comparison of two abstract values with the same ID. Having the same ID guarantees that not only will the sets of concrete maps cor-

```
(* execMapEqual: value * value -> value *)
fun execMapEqual(m as map(must ps, may qs, id), m' as map(must ps', may qs', id')) =
  if id = id'                                          then Bool true
  else if mustBeSingleton(m) andalso mustBeSingleton(m')  then Bool (mustBeEqual(m, m'))
  else if mapsDefinitelyDifferInMusts(m, m')           then Bool false
  else if qs = [] andalso qs' = [] andalso
        haveIdenticalStructureAndId(ps, ps')           then Bool true
  else                                                 meet(Bool true, Bool false)

(* value * value -> boolean*)
and mapsDefinitelyDifferInMusts(m, m') =
  there exists key k s.t. k maps to disjoint values v and v' in the must lists of m and m' respectively
  orelse
  the must list of m is disjoint from the domain (as per must and may lists) of m' or vice-versa

(* vv_list * vv_list -> boolean*)
and haveIdenticalStructureAndId([], [])               = true
| haveIdenticalStructureAndId     ([], _)             = false
| haveIdenticalStructureAndId     (_, [])             = false
| haveIdenticalStructureAndId     ((k,v)::ps, (k', v')::ps') =
  mustBeEqual(k, k')
  andalso ((mustBeSingleton v andalso mustBeSingleton v' andalso mustBeEqual(v,v'))
           orelse (idOf v = idOf v'))
  andalso haveIdenticalStructureAndId(ps, ps')
```

FIGURE 5.15: The *execMapEqual* Function.

responding to the two values will be the same, the concrete values are also guaranteed to be identical. SCF would, in fact, return singleton value *true* in this case.

Line 3 of figure 5.15 exploits the correlation provided by IDs to assert that *execMapEqual* on two correlated values gives singleton abstract value *true*. Otherwise (line 4), in the case that the two maps are singleton, we can just test for structural equality to see if the maps are equal or not. Otherwise (line 5), we check if the maps impose mutually incompatible results via their *must* lists, and if so we return singleton abstract value *false*. If (line 12) both maps *must* contain singleton key *k*, but *k* maps to two values *v* and *v'* in two maps such that *v* and *v'* are disjoint, we can conclude that equality over all pairs is definitely false in every case. Alternately (line 14), for each singleton key *k* in the *must* list of one map, if *k* is disjoint with all the keys of the other map, we can be sure that all pairwise comparisons will give false.

If the *must* lists are not definitely incompatible, we make a final attempt to determine that the two maps are definitely correlated. We check to see if the *may* lists are empty. If so, we check (lines 17-23) for two maps that have the same IDs on all internal non-singleton abstract values, and return singleton *true* in this case. For example, *map(must [(1, 2|3#21), (3,4)], may [])#232* and *map(must [(1, 2|3#21), (3,4)], may [])#177*. If even this test fails, we conservatively return *true|false*, i.e., equality may be *true* or *false*.

The case of two maps having identical IDs being compared does not seem to come up in practice. All other cases occur in our benchmarks.

5.3.8 *match*

Figure 5.16 shows how the *match* function is implemented in SCF-ML. Intuitively, *match* performs pattern-matching on abstract values. Given pattern *pt* and abstract value *v*, *match(pt, v)* is intended to return optionally (i.e., in the case that there is a possible match) an abstract environment *E* mapping variables in *pt* to matching abstract values, and an abstract value *v'* which is the "part of *v* that was not matched by *pt*." For instance, if pattern *pt* is *|binary(x, _)|* and *v* = *|(binary(1, 2) | unary 7)|*, *E* would be the map *[x -> |1|]* and *v'* would be *unary 7* (since the latter part of the abstract value does not match the pattern) and *match* is required to return *SOME([x -> |1|],|unary 7|)*.[1]

```
1  structure Env = AbstractEnvironment       (* Define abbreviation for AbstractEnvironment module*)
2
3  (* match: SCF_ML.pat * value -> (Env.env * value) option *)
4  fun match(|_|, _)      = SOME (Env.empty, top)
5
6  | match (|x|, v)       = SOME (Env.init(|x|,v), top)
7
8  | match (pt, top)      = NONE
9
10 | match (pt, bottom) = SOME (bindIDsToBottom pt, bottom(newID()))
11
12 | match(|c pt'|, v_in as |c' v|) =
13    if c = c' then
14       case match(pt', v) of
15          SOME(E, top)  => (E, top)
16          | SOME(E, v')  => (E, |c v'|)
17          | NONE         => NONE
18    else
19       (Env.empty, v_in)
20
21 | match(pt as |c pt'|, alt(vs, _)) =
22    let (E_out, v_out, possibleMatch) = matchAlternatives(pt, vs, Env.empty, top, false)
23    in if possibleMatch then SOME(E_out, v_out) else NONE end
24
25 | match(|(ps)|, v_in as |(vs)| ) =
26    let (E_out, v_out, possibleMatch) = matchTuple(ps, vs, Env.empty, top, false, 1, vs)
27    in if possibleMatch then SOME(E_out, v_out) else NONE end
28
29 | match(pt, v_in as fix(v, _)) =
30    let v' = replaceRecs(v, freshenIDs v_in)
31    in match(pt, v') end
```

FIGURE 5.16: The *match* Function.
Helper functions are defined in figure 5.17.

If it can guarantee that no match is possible, *match* may return *NONE*. As per the definition of table 2, however, it is conservative for *match* to return *SOME(E, v')* even when no match is possible, as long as $\underline{v}$' contains $\underline{v}$.

Line 4 handles the case that *pt* is the "wildcard" pattern _ (which matches all values matched against it). In this case, since *pt* contains no variables, we return an empty environment *E*, and since the match is complete, we return *top* (which corresponds to the empty set) as the unmatched part. Similarly (line 6), if *pt* is the variable *x*, we return an environment where *x* is bound to *v*, and again return *top* for the unmatched part.

1.More precisely, it is required to return *SOME([x->v], v'), where* $\underline{|1|} \subseteq \underline{v}$ and $\underline{|unary\ 7|} \subseteq \underline{v'}$.

```
(* matchAlternatives: pattern * value list * Env.env * value * bool ->
                              Env.env * value * bool *)
and matchAlternatives(_, [], E_out, v_out, b_out) = (E_out, v_out, b_out)
| matchAlternatives(pt, v::vs, E_out, v_out, b_out) =
   case match(pt, v) of
      NONE         => matchAlternatives(vs, E_out, meet(v,v_out), b_out)
   |  SOME(E, v') => matchAlternatives(vs, Env.meet(E, E_out), meet(v',v_out), true))

(* matchTuple: pattern list * value list * Env.env * value * boolean * int * value list ->
                     Env. env * value * bool *)
and matchTuple([], [], E_out, v_out, b_out, _, _) = (E_out, v_out, b_out)
|  matchTuple(pt::pts, v::vs, E_out, v_out, b_out, i, vs_orig) =
   case match(pt, v) of
      NONE         =>
         matchTuple(pts, vs, E_out,
                           meet(v_out, mkTupleWithField(vs_orig, i, v)), b_out, i+1, vs_orig)
   |  SOME(E, v') =>
         matchTuple(pts, vs,
                           Env.meet(E, E_out), meet(v_out, mkTupleWithField(vs_orig, i, v')), true, i+1, vs_orig)

and replaceRecs(v: value, v': value): value= ... replace every free occurence of "rec" in v with v'...

and mkTupleWithField(vs: value list, i: int, v: value) =
   ... replace the i'th element of vs with v, and make a tuple from the resulting list ...
```

FIGURE 5.17: Helper Functions for the *match* Function.

5.3.8.1 Matching Against alt Values

Lines 21-23 of figure 5.16 (along with the helper function *matchAlternatives* of figure 5.17) show how we match a pattern against an *alt* value, i.e., a value that is an “or” of values. The rule seems straightforward: to match a set of alternative values, match the individual alternatives separately, and meet the environments and unmatched parts that result. For instance, when we match pattern *unary x* against value *unary 11 | unary 322 | binary(**1**, 23),* invoking *match* on the individual alternatives could return *SOME([x -> 11], top), SOME([x -> 322], top)* and *NONE* respectively. Note that a *NONE* returned value implies that the whole incoming value, in this case *binary(**1**, 23),* is unmatched. Meeting the returned environments gives *[x -> 11 | 322]* and meeting the unmatched values gives *binary(**1**, 23).*

The above rule, although intuitive, has an adverse consequence. Consider the environment that results from matching pattern *binary(x, y)* against value *binary(1, 2) | binary(*

12, 13). The individual components return environments *[x -> 1, y -> 2]* and *[x -> 12, y -> 13]*. Meeting these environments yields *[x -> 1 | 12, y -> 2 | 13]*. In doing so, unfortunately, we lose some information. In particular, it now seems possible that a match may yield concrete environment *[x -> 1, y -> 13]*, which is clearly impossible since according to the incoming abstract value, *y* is always *2* when *x* is *1*.

The problem is that we wish to correlate the value of a variable in the abstract environment with that of another. Abstract value ID tags currently allow us to express correlations where two variables (or fields thereof) have equal values, but in this case we want to set up a mapping other than equality. Although it is possible that more sophisticated identity-tagging of abstract values can help preserve these correlations too, in SCF we address this problem by avoiding such information-losing *match* statements where possible and appropriate. We discuss the techniques involved in the next chapter.

5.3.8.2 Matching Tuple Patterns Against Tuple Values

Lines 25-27 of figure 5.16 (along with the helper function *matchTuple* of figure 5.17) show how to match a tuple pattern against a tuple value. Unlike the case with *alt*, where the problem was in constructing accurate-enough result environments, in this case the trouble is constructing the abstract value *v'* representing the unmatched part. Consider matching the tuple pattern *(unary x, unary y)* with value *(unary 0 | binary(1, 2), unary 121)*. As usual, we proceed in a compositional fashion and match the components of the tuple pattern to those of the value. Say the results are *SOME([x -> 0], binary(1,2))* and *SOME([y -> 121], top)* respectively. The question now is how to construct the unmatched part for the tuple, given the unmatched parts for its components.

A naive approach may be to simply accumulate the unmatched parts of the components into a tuple. In the above case, the result would be the tuple *(binary(1,2), top)* (or, in normal form, *top*). This would imply that there are no concrete tuples that do not match the given pattern and conform to the incoming tuple abstract value. This is plainly not the case. For instance, the concrete value *(binary(1, 2), unary 121)* conforms to the value, but does not match the pattern. The key is that when one of the components of a tuple has unmatched value v'_i, then none of the concrete tuples in $(v_1,...,v'_i,...,v_n)$ matches. The correct rule is, therefore, that if $v'_1,...,v'_n$ are the unmatched parts for components $v_1,...,v_n$ of the tuple, $meet_{1 \leq i \leq n}$ *(*v_1*,...,*v'_i*,...*v_n*)* is the unmatched part of the tuple as a whole. The helper function *matchTuple* computes this meet. The helper function *mkTupleWithField* (lines 23-24) creates each of the tuples being met.

5.3.8.3 Matching Against fix Values

The final subtle case in the *match* function of figure 5.16 is for *fix* abstract values lines 29-31. Consider matching value *v = fix. (unary 9 | binary(unary ((5 | 12) # 131), rec)* against pattern *pt = binary(x, binary(y, _))*.

The first step is to convert the incoming value into a form without a *fix* at the top level. We “unroll” the *fix* value once as per the *replaceRecs* function of line 21, figure 5.17. Naively, the unrolling would replace every instance of *rec* in *v* with the value *v* itself. The result would be: *unary 9 | binary(unary ((5 | 12)# 131), fix.(unary 9 | binary(unary ((5 | 12) # 131), rec)*. Matching pattern *pt* above using the code from lines 21-23 returns *SOME([x -> ((5 | 12) # 131), y -> ((5 | 12) #131)]),* a result that implies that the values of *x* and *y* are correlated (since they have the same IDs).

However, the semantics of *fix* values that we give in table 1 of chapter 3 do not guarantee any such correlation between versions of an abstract value derived from two different “unrollings” of a recursive value. Our solution is to renumber *fix* abstract values with fresh ID numbers before replacing instances of *rec* with these values. While doing so, we preserve any correlations within the body of the *fix*. The call to *freshenIds* function (line 30, figure 5.16) effects this renumbering. With this scheme, the result of the unrolling above would be *unary 9 | binary(unary ((5 | 12) # 131), fix.(unary 9 | binary(unary ((5 | 12) # 2986), rec))* (assuming *freshenIds* replaces ID *131* with ID *2986*). Given this input value, the abstract pattern match would return environment *SOME([x -> ((5 | 12) # 131), y* -> *((5 | 12) #2986)]),* and *x* and *y* are not correlated.

```
fun mapMap(map(must musts, may mays, _): value, f: value -> value): value =
  let musts'   = List.map (fn (k, v) => (k, f v)) musts
      mays'    = List.map (fn (k, v) => (k, f v)) mays
  in map(must musts', may mays', newId()) end
```

FIGURE 5.18: The *mapMap* Function.

5.3.9 *mapMap*

Given abstract value *v* which represents an abstract map and a function *f* that maps abstract values to abstract values (and that thereby represents a set of functions on concrete values as discussed in section 5.1), the *mapMap* function returns an abstract value *v'*. Value *v'* represents the set of maps that can result when each map in $\underline{v}$ has has its range mapped according to a function in $\underline{f}$.

For example, suppose the incoming map *v = map(must[(1, 2|3), may [(4,5)])* and *f x = evalPrimop(|+|, |(x, 11|21)|)*. The set of concrete maps conforming to the incoming map is $\underline{v}$ *= {[1 -> 2], [1 -> 3], [1 -> 2, 4 -> 5], [1 -> 3, 4 -> 5]}*. The set of incoming concrete functions is $\underline{f}$ *= {f x = x + 11, f x = x + 21}*. Mapping the map *[1 -> 3, 4 -> 5]* according to the function *f x = x + 21* results in the new map *[1 -> 24, 4 -> 26]*. Doing this with all the pairings of incoming concrete maps and functions gives the output set of maps: *{[1 -> 13], [1 -> 14], [1 -> 13, 4 -> 16], [1 -> 14, 4 -> 16], [1 -> 23], [1 -> 24], [1 -> 23, 4 -> 26], [1 -> 24, 4 -> 26]}*.

The set of concrete maps produced as a result in the above example can be represented by the abstract map *map(must[(1, 13|14|23|24), may [(4,16|26)])*. This abstract map the result of applying *f* abstractly on each abstract value in the range of the incoming map. For instance, *f(2 | 3) = 13|14|23|24*. In general, to produce an abstract value representing the application of (the concrete instances of) the abstract map function, it is sufficient to apply the abstract map function to the values in the range of the abstract map. This is what the implementation of figure 5.18 does.

5.3.10 *mapUnionWith*

Given abstract values v_1 and v_2 that each represent an abstract map and a "resolution" function *f* that maps a pair of abstract values to an abstract value, the *mapUnionWith* function returns an abstract value *v'* representing the set of maps that result when the two sets of maps in $\underline{v}_1$ and $\underline{v}_2$ are unioned pairwise. The functions $\underline{f}$ (i.e., the set of concrete functions conforming to abstract function *f*) resolve the case when two concrete maps being merged have some common key *k* which maps to possibly different values u_1 and u_2 in the two maps. In particular, the entry corresponding to the key in the resulting map is $f_{concrete}(u_1, u_2)$, where $f_{concrete} \in \underline{f}$.

For example, suppose v_1 = *map(must[(1, 2)], may [(4,5)])* and v_2 = *map(must[(1, 17), (23, 65)], may [])*, and *f (x, y) = x + y*. The conforming sets of concrete maps then are $\underline{v}_1$ = *{[1 -> 2], [1 -> 2, 4 -> 5]}* and $\underline{v}_2$ = *{[1 -> 17, 23 -> 65]}*, and $\underline{f}$ = *{ fn(x,y) => x+y}*. Given these sets, and picking u_1 = *[1 -> 2, 4 -> 5]*, u_2 = *[1 -> 17, 23 -> 65]*, we get the map *[1 -> 19, 4 -> 5, 23 -> 65]* as the resulting concrete map. Note that since the key *1* was in both maps (with corresponding values *2* and *17* in the two maps), the value corresponding to the key in the result map is *2 + 17 = 19*. On the other hand, since key *4* was

```
(* mapUnionWith: value * value * (value -> value) -> value*)
fun mapUnionWith(m as map(must ps, may qs,_), m' as map(must ps', may qs', _), f) =
  let (musts, mays) = unionWithMusts([], [], ps, m', f)
      (musts, mays) = unionWithMays(musts, mays, qs, m', f)
      (musts, mays) = unionWithMusts(musts, mays, ps', m, f)
      (musts, mays) = unionWithMays(musts, mays, qs', m, f)
  in map(must musts, may mays, newId()) end

(* unionWithMusts: vvlist* vvlist * vvlist * value * (value * value -> value) -> vvlist * vvlist *)
and unionWithMusts(musts, mays, [], _, _) = (musts, mays)
| unionWithMusts(musts, mays, (k,v)::ps, m, f) =
  case find(m, k) of
     MUST v'   => unionWithMusts(insert(musts, k, f(v,v')), mays, ps, m, f)
   | MAY v'    => unionWithMusts(insert(musts, k, meet(v, f(v, v'))), mays, ps, m, f)
   | NOT       => unionWithMusts(insert(musts, k, v), mays, ps, m, f)

(*unionWithMays: vvlist * vvlist * vvlist * value * (value * value -> value) -> vvlist * vvlist *)
and unionWithMays(musts, mays, [], _, _) = (musts, mays)
| unionWithMays(musts, mays, (k,v)::qs, m, f) =
  case find(m, k) of
     MUST v'   => unionWithMays(insert(musts, k, meet(v', f(v,v'))), mays, qs, m, f)
   | MAY v'    => unionWithMays(musts, insert(mays, k, meet(v, meet(v', f(v,v')))), qs, m, f)
   | NOT       => unionWithMays(musts, insert(mays, k, v), qs, m, f)

... insert as in figure 5.5, find as in figure 5.14...
```

FIGURE 5.19: The *mapUnionWith* Function.

only in one of the maps, its value *5* passes through unscathed. Processing all possible pairings for u_1 and u_2, we get the output set *{[1 -> 19, 23 -> 65], [1 -> 19, 4 -> 5, 23 -> 65]}*.

The above set of maps conforms to abstract value *map(must[(1, 19), (23,65)], may[(4, 5)),* which can be derived systematically from the incoming map as follows. The keys in the *must* or *may* list of the result abstract map are those from the respective lists in either of the two incoming maps. This makes sense, since if abstract key *k* must be in either map, the corresponding singleton concrete key is required to be in any result concrete map; it is therefore correct to place *k* in the *must* list of the result map. On the other hand, since if a key appears in an input map, it is also required to appear in the result, it makes sense to place all remaining keys in the *may* list.

We now consider the values corresponding to the keys *k* above in the resulting abstract map. Consider an abstract key *k* that appears in the *must* list of an incoming map. If *k* is also in the *must* list of the other incoming map, with corresponding abstract values *v* and *v'*, then per the semantics of *unionWith*, we invoke *f(v, v')* to compute the resolved key (line 13). If *k* is only in the *may* list of the other, then we have to account for the possibility that it does not appear in the second map. We do so by meeting the resolved value with unresolved key *v* (since if *k* does not appear in the second map, we can use the value from the first map without resolution). This case is handled on lines 14 and 21. If *k* definitely does not appear in the second map, no resolution is necessary (line 15).

If *k* appears in the *may* list of both maps, we again need to perform the *meet* as above to account for the possibility that the resolution function may or may not need to be invoked (line 22). Finally, if *k* is definitely not in one map, but is in the *may* list of the second, its corresponding value needs no resolution, but only *may* appear in the result (line 23).

```
fun collapseMap(map(must musts, may mays, _): value): (value * value) =
   collapseMapLists(musts@mays, top, top)

and collapseMapList([], v_dom, v_rng)            =(v_dom, v_rng)
| collapseMapList((k,v)::kvs, v_dom, v_rng)    = collapseMapList(kvs, meet(k, v_dom), meet(v, v_rng))
```

FIGURE 5.20: The *collapseMap* Function.

5.3.11 *collapseMap*

The *collapseMap* function, given an input abstract value v_{map} that represents a map, is supposed to return a pair of abstract values (v_{dom}, v_{rng}) such that $\underline{v}_{dom}$ represents all concrete values in the domain of $\underline{v}_{map}$ and $\underline{v}_{rng}$ represents all those in the range. The implementation of figure 5.20 achieves this by meeting together the abstract values in the key and value (a.k.a. domain and range respectively) positions of the incoming map.

5.3.12 *findLiveKeys*

The purpose of the *findLiveKeys* function is to determine, given an input abstract value that represents a tuple of two abstract maps *m* and *m'*, an abstract value *v* representing the *live keys* under equality of the maps. As per section 4.4.4, "live keys" refers to the set of concrete keys such that each key in the set belongs to one or other of the maps, but no key in the set belongs to the *must* list of both maps and has the same singleton value in both maps. In other words, the function returns the set of keys of the incoming maps whose values are not definitely equal, a set called the "live keys" in section 4.4.4.

```
(* findLiveKeys: value -> value *)
and findLiveKeys (bottom _) = bottom (newId())
| findLiveKeys(|(map(must ps, may qs, id), map(must ps', may qs', id')|) =
   let (ps_out, ps'_out)= removeIdenticalEntries(ps, ps', ps, ps')
   in collapseDomain ( ps_out @ ps'_out @ qs @ qs') end

and removeIdenticalEntries([], _, ps_out, ps'_out) = (ps_out, ps'_out)
|  removeIdenticalEntries((k, v)::ps, ps', ps_out, ps'_out) =
   case findMust(ps', k) of
      SOME v' =>
         if isSingleton v andalso isSingleton v' andalso mustBeEqual(v,v') then
            removeIdenticalEntries(ps, ps', remove(ps_out, k), remove(ps'_out, k))
         else
            removeIdenticalEntries(ps, ps', ps_out, ps'_out)
      | NONE =>
            removeIdenticalEntries(ps, ps', ps_out, ps'_out)

and collapseDomain kvs =
   let (v, _) = collapseMapList(kvs,top,top) in v end

... findMust as in figure 5.14, remove as in figure 5.12, collapseMapList as in figure 5.20...
```

FIGURE 5.21: The *findLiveKeys* Function.

The code of figure 5.21 finds the live keys in two steps. It first uses the *removeIdenticalEntries* function to step through the *must* list of the two maps, removing keys that appear in both maps and have identical values (lines 7-16). It then appends the resulting filtered *must* lists to the original *may* lists, and meets together the keys of the resulting list (using the *collapseDomain* helper on lines 18-19).

5.4 Summary

In this chapter, we specified the abstract interface required of abstract values, the semantics required of the functions in this interface, a concrete representation for abstract values, and algorithms implementing the interface in a manner that conforms to the required semantics. The semantics allow the interface functions to be approximate, and therefore potentially more efficient than if they had to be exact. However, for practical use in simulating optimizations, the functions need a minimum level of accuracy. Where relevant, we described via examples the minimum accuracy required , and showed how our algorithms provide this level of accuracy while preserving correctness.

6. Strategies for Accurate and Effective Partial Evaluation

Partial evaluation results in faster programs because the partial evaluator replaces expressions in the incoming program with simpler residual expressions. Partial evaluators, given abstract values representing possible arguments to the incoming program, compute the abstract values representing the possible values of every expression in the incoming program. Expressions guaranteed to produce a singleton abstract value are then *folded*, i.e., replaced with a simpler residual expression that computes the same value. In this simple model, a fragment of code in the incoming program would correspond to at most one residual fragment in the result program.

In practice, to ensure that generated residual programs are as fast as possible, the partial evaluator addresses three key additional issues.[1]

First, it can be profitable to replicate fragments of the original program while abstractly executing it. For instance, if the original program had an expression that could possibly evaluate to two concrete values, the expression could not be folded using the simple partial evaluator above. But a partial evaluator that uses replication may process two versions of that expression, each evaluating to a single concrete value which. The partial evaluator must therefore have a *specialization strategy* which decides when code is replicated before partial evaluation.

Second, it can be profitable for the partial evaluator to have a precise but infinitely tall lattice. It is possible that when the functions being evaluated are recursive, the partial evaluator may be invoked on a particular piece of code an unbounded number of times. It is the role of the *widening strategy* to ensure that the partial evaluator terminates while not sacrificing too much precision.

Third, even when a given expression has a singleton abstract value, it is not trivial to determine whether it is profitable to replace the expression with the corresponding constant expression. The role of the *rematerialization strategy* is to determine when to replace expressions with equivalent constant expressions, and how to make this replacement as profitable as possible.

1.These issues arise when designing whole-program variants of many optimizations.

We will refer to these three strategies as *improvement strategies*, since they are improving the quality of partial evaluation. We explore the potential costs and benefits of these strategies through a detailed example in section 6.1 below. Sections 6.2 through 6.4 describe how SCF implements the three strategies to address these trade-offs. Section 6.5 summarizes the section and discusses related work.

6.1 Improvement Strategies at Work: A Detailed Example

In this section, we use a running example to illustrate the role of improvement strategies. Although the partial evaluator in SCF applies to optimization functions (such as the dead-assignment eliminator of figure 3.7) written in SCF-ML, we will use as our example in this section a simpler SCF-ML function. The function is much smaller than a typical optimization, but nevertheless manifests all the relevant complications.

In section 6.1.1, we show how partial evaluation progresses in the absence of improvement strategies. In section 6.1.2, we show the benefits of adding function specialization. In section 6.1.3, we show how a slightly modified version of the example benefits from aggressive rematerialization. In section 6.1.4, we show the benefits of expression specialization. Finally, in section 6.1.5, we show that specialization can cause the partial evaluator not to terminate, and illustrate the strategies used by SCF to ensure termination.

6.1.1 Partial Evaluation With No Improvement Strategies

Figure 6.1 presents a function, `inc`, that takes as input a list of integers and outputs the list that results from incrementing each integer in this list. In this section, we will focus on partially evaluating this function on abstract argument value *[2|3]*, i.e., a list with a single element of value *2* or *3*.

The table of figure 6.2 shows the values produced when `inc` is partially evaluated with abstract input `vs` = *[2|3]*. Each row of the table represents the values of variables

```
(* Increment each element of the incoming list *)
fun inc (vs: int list): int list =
case vs of
  []       => []
| v'::vs' => (v'+1)::(inc vs')
```

FIGURE 6.1: Function to Be Partially Evaluated On Input `vs` = *[2|3]*.

Invocation Number	`vs`	`v'`	`vs'`	`result`
1	*[2|3]*	*2|3*	*[]*	***0***
2	*[2|3] | []*	*2|3*	*[]*	*[]*
3	*[2|3] | []*	*2|3*	*[]*	*[3|4] | []*
4	*[2|3] | []*	*2|3*	*[]*	*[3|4] | []*

FIGURE 6.2: Partially Evaluating the `inc()` Function With No Improvement Strategies.

(`vs`, `v'` and `vs'`), and the result of the function as a whole, due to a distinct invocation of the partial evaluator on function `inc`.[1]

Below, we step through the invocation corresponding to each row in some detail. This detailed view is useful for understanding the design and functioning of improvement strategies.

1. On the initial call to the partial evaluator (invocation number 1), the variable `vs` has value *[2|3]*. The value of `vs` matches the branch of the case expression on line 4 of figure 6.1. The pattern match binds `v'` to *2|3* and `vs'` to *[]*. The recursive call `inc vs'` (with value *[]* for `vs'`) is then evaluated via a cache lookup (as per figure 4.5, line 42) for the value corresponding to function `inc`. The lookup returns the placeholder value ***0*** (since the function has not been evaluated before). Recall that ***0*** is the abstract value representing the empty set of concrete values, i.e., the top of the abstract value lattice. Evaluating the expression `(v'+1)::(inc vs')` then gives value ***0*** as the result of the invocation as a whole.
2. The recursive call of the previous paragraph results in a task being added to the partial evaluator's worklist (as per figure 4.5, line 45). The task requires partial evaluation of function `inc` on input value `vs` = *[]*. The task is now popped off the worklist and processed (invocation 2). First the argument *[]* of the recursive call is met with that of the original call (i.e., with *[2|3]*) to get the argument value (*[2|3]*| *[]*) for the new invocation. This time, both sides of the case expression hold. The side guarded by pattern `[]` evaluates to value *[]*, while the other side again returns ***0***. Meeting the two possibilities we get *[]* for the result of the invocation.

1. The invocation we refer to is the call to PE_f on line 35 of figure 4.4.

```
fun  inc (vs: int list): int list =
case vs of
  []      => []
| v'::vs' => (v'+1)::(inc vs')
```

FIGURE 6.3: Residual Function Resulting From Partial Evaluation Without Improvement.

3. Because the result of the function changed from the earlier value ***0*** to the new value *[]*, and the function calls itself recursively, the partial evaluator needs to re-evaluate the function.[1] This time around (invocation 3), both sides of the case expression hold again, but the lower branch evaluates to *[3|4]* (since the cache lookup for the recursive call to `inc` returns *[]* this time), and meeting with the upper branch of the case, the invocation returns *[3|4] | []*.
4. The change of result necessitates another partial evaluation of the function, but this time (invocation 4) the result sticks at *[3|4]|[]* and the partial evaluator terminates.

Figure 6.3 shows the residualized function that results from the final invocation of the partial evaluator on the input function. Note that in the final invocation of the partial evaluator, none of the variables `vs`, `v'` and `vs'` evaluates to a singleton scalar abstract value, and therefore neither does any of the expressions of the incoming function. As a result, the partial evaluator cannot rematerialize any of the expressions in the incoming function, and the residual version of the function is identical to the original one; partial evaluation provides no benefit at all.

6.1.2 Adding Function Specialization as an Improvement Strategy

In this section, we examine how the partial evaluation proceeds when we add just function specialization.

Figure 6.4 shows the sequence of abstract values that results from specializing function `inc` during partial evaluation. We evaluate `inc` separately (i.e., without first meeting in argument values from previous invocations) on each abstract value *v* it is invoked on, producing corresponding residual functions $\texttt{inc}_v$. Each distinct specialized version of the analyzed function is called a *contour*. The many contours for a given function are distin-

1. The partial evaluator performs the "results-changed?" check on line 39 of figure 4.4, determines that the result has indeed changed, and places the callers of function `inc`, i.e., `inc` itself, on the worklist for further processing.

invocation number	contour	`vs`	`v'`	`vs'`	result
1	$inc_{[2\|3]}$	*[2|3]*	*1*	*[]*	***0***
2	$inc_{[]}$	*[]*	*n.a.*	*n.a.*	*[]*
3	$inc_{[2\|3]}$	*[2|3]*	*1*	*[]*	*[3|4]*

FIGURE 6.4: Partial Evaluation With Function Specialization.

guished by a *contour key* (the value *v* in this example). Rows of the table again denote the distinct invocations of the partial evaluator on the `inc` function. The column labelled “contour” specifies which particular contour of the `inc` function is processed by the invocation.

The initial partial evaluation of the function results, as before, in a recursive invocation of function `inc` with abstract argument `vs'` = *[]*. Unlike in the unspecialized case above, we do not merge this new argument value of `inc` with its original value before re-analyzing `inc`. Rather, we analyze `inc` separately for this argument value, i.e., we analyze the callee contour $\mathtt{inc}_{[]}$. The analysis results in the case on line 3 of figure 6.1 being taken. Invocation 2 therefore returns *[]* as a result for the whole function. As before, the changed return value of the callee contour results in the re-analysis of the calling contour $\mathtt{inc}_{[2|3]}$ (invocation 3), which returns this time the value *[3|4]* and terminates.

Figure 6.5(a) shows the residual functions that result, under the assumption that the partial evaluator only residualizes primitive scalars, i.e., integers, booleans, characters and strings, in SCF-ML. Analyzing the two different contours results in two different residual versions of the `inc` function. Since each version is partially evaluated with respect to just a subset of the abstract argument values of the original function, the corresponding residual version can be better optimized than before. Note that although the total amount of *static code* produced may be more than that produced by the unspecialized partial evaluator, the amount of code *dynamically executed* is smaller. In particular, as per line 3 of figure 6.5(a), the residualized version of the function does not have to compare the incoming value to pattern `[]`, whereas (as per line 3 of figure 6.3) this comparison needs to be done in the unspecialized version. Also, as per line 6, when the recursive call is executed, the specialized version can simply produce the empty list with no need for testing the input value.

```
fun  inc[2|3] vs =
case vs of
  v'::vs'  => (v'+1)::inc[] vs'

and inc[] vs' =
[]
```

(a)

```
fun  inc[2|3] vs =
case vs of
  v'::vs'  => (v'+1)::[]
```

(b)

FIGURE 6.5: Residual Function Resulting From Function Specialization.

Finally, as figure 6.5(b) shows, each contour key may not have a corresponding residualized function. If the partial evaluator can rematerialize user-defined scalars (i.e., zero-ary constructors) in addition to primitive scalars, then it would replace the call to `inc[]` with an expression constructing the singleton result of that call, in this case the expression `[]`. The resulting residual function executes fewer instructions than the one in figure 6.5(a).

6.1.3 Adding Rematerialization as an Improvement Strategy

In the previous section, we considered the case where the partial evaluator is restricted to rematerializing scalars. The appeal of this approach is that a literal expression computing a scalar is guaranteed to be no more expensive to evaluate than any expression it replaces.

The simplicity of the approach is at the expense of quality of code produced, however. To demonstrate the inefficiency that results, we alter our running example slightly. Instead of partially evaluating function inc on input value *[2|3]*, we instead evaluate it on the input *[2|3, 7]*. Figure 6.6(a) shows the residual functions that result, assuming purely scalar rematerialization. The key point is that although the partial evaluator computes that the invocation to function `inc[7]` results in the singleton abstract value *[8]*, it cannot replace the invocation with an expression constructing this value, since the list is not a scalar.

```
fun  inc[2|3,7] vs =
case vs of
  v'::vs'=> (v'+1)::inc[7] vs'

and inc[7] vs =
case vs of
  v::vs' => 8::[]
```

(a)Scalar Rematerialization

```
fun  inc[2|3,7] vs =
case vs of
  v'::vs'=> (v'+1)::8 ::[]
```

(b) Non-Scalar Rematerialization

FIGURE 6.6: Rematerializing Non-Scalars.

Figure 6.6(b) illustrates a more aggressive rematerialization strategy, which allows the rematerialization of *arbitrary* singleton abstract values. Given this strategy, the partial evaluator can replace the call to `inc`$_{[]}$ with an expression (line 3 of figure 6.6(a)) with the expression `8::[]`, which generates the singleton list *[8]*. Avoiding the function call clearly results in savings in this case: the residual code avoids the execution of an extra function call and case expression.

Although rematerializing arbitrary singleton values can be profitable, it is not necessarily so. Later in this chapter, we will see an example where the expression resulting from rematerialization is more expensive than the one it replaces.

6.1.4 Adding Expression Specialization as an Improvement Strategy

Until now, we have considered the function call to be the unit of code replication: we have replicated functions so as to specialize them for particular input values. We demonstrate in this section that specializing at a granularity smaller than functions (and in particular, at the level of expressions) can result in faster residual code.

In order to demonstrate expression specialization, we once again change the input value to the `inc` function. This time, we assume that the input value is *[2,3]|[7,8]*.

Figure 6.7(a) shows the result of function specialization with non-scalar rematerialization, given these inputs. When the partial evaluator is first invoked on the contour *inc*$_{[2,3]|[7,8]}$, the variable `vs'` is bound to abstract value *[3]|[8]*. The residual function corresponding to this contour is shown on lines 1-3 of the figure. The recursive call to `inc` now results in the contour *inc*$_{[3]|[8]}$. The result of evaluating this contour is *[4|9]*, an abstract value that is not singleton. Since the result of evaluating the contour is not singleton, the call cannot be rematerialized. The resulting residual call to `inc`$_{[3]|[8]}$

```
1 fun  inc_{[2,3]|[7,8]}vs =
2 case vs of
3   v'::vs'=> (v'+1)::inc_{[3]|[8]} vs'
4
5 and inc_{[3]|[8]} vs =
6 case vs of
7   v::vs' => (v'+1)::[]
```

(a)

```
1 fun  inc_{[2,3]|[7,8]}vs =
2 case vs of
3   2::vs' => 3::4::[]
4 | 7::vs' => 8::9::[]
```

(b)

FIGURE 6.7: Expression Specialization.

is on line 3 of figure 6.7(a), and the corresponding residualized function body is on lines 5-7.

Figure 6.7(b) illustrates an alternate possibility. When analyzing the original contour $\mathtt{inc}_{[2,3] | [7,8]}$, the partial evaluator detects that the abstract value being matched against has two components (*[2,3]* and *[7,8]*) that match against the same pattern, namely `v'::vs'`. It then specializes this pattern to produce two distinct corresponding patterns (`2::vs'` and `3::vs'`), such that each new pattern matches exactly one of the two components. Next, it specializes separately on the two component values the expression being guarded by the pattern, i.e., it evaluates `(v'+1)::inc vs'` twice: first with `(v',vs') =` (*2, [3]*) and next with *(7,[8])*. Adding the additional test for the constant values in the pattern allows, in each case, the body of the case to be replaced by an expression that constructs a constant singleton list.

The code of figure (b) executes fewer case expressions, function calls and arithmetic operations than that of figure (a). On the other hand, for the one case expression that it does execute, figure (b) performs an extra check (to determine whether the first element of the incoming list is 2 or 3). Although in this example the benefits exceed the extra cost of discriminating between the values being specialized on, this is not always the case. Addressing this trade-off is a key part of expression specialization strategies.

6.1.5 Ensuring Termination in the Face of Improvement Strategies

The `incPadded` function of figure 6.8 is a slightly modified version of the `inc` function of figure 6.1. Given an input list of integers `vs` and integer `n`, `incPadded(vs, n)` produces the same result as `inc(vs)`, except that it prepends `n` copies of the first element of `vs` to this result. For instance `incPadded([23,343], 2)` would yield `[23, 23, 24, 344]`. This small modification in the function being partially evaluated can potentially cause a partial evaluator operating on it to loop forever.

Consider partially evaluating `incPadded` on abstract input *([2] , 0|1)*. Suppose we use the technique from the previous sections of using the entire function argument as the contour key. Then, as shown in figure 6.8(b), the partial evaluator will be invoked with an unbounded number of contour keys. The problem is that the second abstract argument of

```
(* Increment each element of vs by 1. Prepend n copies of the first
element of vs to the result *)
fun  incPadded (vs: int list, n: int): int list =
case vs of
  []       => []
| v'::vs' =>
   if n = 0 then (v'+1) :: (incPadded (vs', n))
   else v':: (incPadded (vs, n-1))
```

(a) Function to be partially evaluated on input vs = *[2]* , n = *0|1*

contour key	vs	n	vs'	v'	result
[2], 0|1	*[2]*	*0|1*	*[]*	*2*	***0***
[2], -1|0	*[2]*	*-1|0*	*[]*	*2*	***0***
[2], -2|-1	*[2]*	*-2|-1*	*[]*	*2*	***0***
[2], -3|-2	*[2]*	*-3|-2*	*[]*	*2*	***0***
[2], -4|-3	*[2]*	*-4|-3*	*[]*	*2*	***0***
[2], -5|-4	*[2]*	*-5|-4*	*[]*	*2*	***0***
…					

(b) Using the entire argument as the contour key

contour key	vs	n	vs'	v'	result
[2]	*[2]*	*0|1*	*[]*	*2*	***0***
[2]	*[2]*	*-1|0|1*	*[]*	*2*	***0***
[2]	*[2]*	*-2|-1|0|1*	*[]*	*2*	***0***
[2]	*[2]*	*-3|-2|-1|0|1*	*[]*	*2*	***0***
[2]	*[2]*	*-4|-3|-2|-1|0|1*	*[]*	*2*	***0***
[2]	*[2]*	*-5|-4|-3|-2|-1|0|1*	*[]*	*2*	***0***

(c) Using the finite part of the argument as the contour key

contour key	vs	n	vs'	v'	result
[2]	*[2]*	***1***	*[]*	*2*	***0***
[]	*[]*	***1***	*n.a.*	*2*	*[]*
[2]	*[2]*	***1***	*[]*	*2*	*[3]*
[2]	*[2]*	***1***	*[]*	*2*	*[3] | [2,3]*
[2]	*[2]*	***1***	*[]*	*2*	*[3] | [2,3] | [2, 2, 3]*
[2]	*[2]*	***1***	*[]*	*2*	*[3] | [2,3] | [2, 2, 3] | [2, 2, 2, 3]*
…					

(d) Widening non-finite arguments

contour key	vs	n	vs'	v'	result
[2]	*[2]*	***1***	*[]*	*2*	***1***
[]	*[]*	***1***	*n.a.*	*2*	***1***

(e) Widening non-finite results

```
fun incPadded[2] n =
  if n = 0 then [3]
  else 2 :: (incPadded[2] (n-1))
```

(f) Residual version of function

FIGURE 6.8: Challenges of Function Specialization.

`incPadded` (corresponding to formal `n`) is not *finite*. During partial evaluation, it can take on an unbounded number of values, thus leading to an unbounded number of contours.

A known solution to this problem is to omit the non-finite part of the contour key, and only use the value of `vs` as contour key. Unfortunately, as shown in figure 6.8(c), although this solution results in a finite number of contours, a single contour may still be slated for analysis on an unbounded number of non-finite argument values. Since, for a given contour, we meet together all argument values on which the contour has been invoked thus far, the size of the input value keeps increasing with each invocation.

A standard solution to this new problem is to *widen* the non-finite arguments. We preemptively set the value of the non-finite arguments to an abstract value whose size is guaranteed not to increase indefinitely. In figure 6.8(d), we set the abstract value corresponding to argument `n` to the bottom of the abstract value lattice, ***1***. Unfortunately, as the figure shows, this exposes another problem. Recall that (as per the core partial evaluator of figure 4.4), a contour may be re-analyzed either if it is invoked with arguments different from those it has previously been analyzed with, or if the return-value of one of its callees has changed. As figure 6.8(d) shows, since `incPadded` is its own callee, and each invocation to `incPadded` adds one entry to a result set of unbounded size, the number of such invocations is unbounded.

A solution to this problem is to widen possibly unbounded results (in addition to widening arguments, as above). As shown in figure 6.8(e), if we set the return value to ***1*** on every invocation of the function, the partial evaluator terminates after two invocations of the function. The combination of restricting contour keys to finite parts of incoming arguments, argument widening and return value widening is known to guarantee termination of the partial evaluator.

An unintended (and detrimental) consequence of our return-value widening strategy is that we know absolutely nothing about the possible values that may be returned by the partially evaluated function. For instance, although we can tell by inspection that any returned value is a list that is either empty, or has some number of 2s followed by a 3, our return value ***1*** hides even the fact that the returned value has to be a list. As we shall see, performing widening while limiting loss of information is crucial for performance in SCF.

A final observation is that, comparing figure 6.8(f) to 6.8(a), in spite of the very conservative return values that result from widening, the residualized code obtained executes significantly fewer operations than the original code. The benefit is wholly due to the fact that we specialized using `vs` as a contour key. If we had used no contour key (and therefore performed no specialization at all), we would still have needed the widening techniques to guarantee termination, but would have ended up with a residual program identical to the incoming program. The additional step of identifying finite arguments and specializing on them is therefore profitable even in the face of conservative widening.

6.2 Specialization Strategies

The examples of the previous section showed the benefits of specialization, both at the granularity of entire functions. For function specialization, we saw the importance of selecting an appropriate contour key. For expression specialization, we saw that discriminating between specialized versions of the expression can add to the number of instructions executed, so it is important to be careful in selecting when to perform expression specialization.

In what follows, we describe how SCF's specialization strategies are tailored for effective specialization of optimizations. In section 6.2.1, we describe how SCF performs function specialization. In particular, we describe how it identifies finite arguments and picks contour keys. In section 6.2.2, we describe how SCF performs expression specialization.

6.2.1 Function Specialization

As discussed in the example of section 6.1.5, when specializing functions, it is important to pick *finite* parts of function arguments for contour keys. In other words, we want to discover those arguments of functions that are guaranteed to take on a finite set of values any time the program containing the function is partially evaluated. In section 6.2.1.1 below, we specify SCF’s *finiteness analysis*, which computes the finite part of function arguments.

In the preceding example, we used the finite part of the argument of a function as the contour key for the function. We will argue in section 6.2.1.2 that in order to better partially evaluate optimization programs, it is useful to use contour keys that are the result of composing the finite parts of many function arguments, yielding contour keys that are *chains* of finite argument values.

6.2.1.1 What Values to Specialize On: Finiteness Analysis

The goal of finiteness analysis is to determine for each function in the incoming program, the parts of the function argument that are guaranteed to assume a finite number of abstract values, over all invocations of the function during partial evaluation. In what follows, we first define the problem to be solved by finiteness analysis precisely, then explain the intuition behind SCF’s analysis, and finally describe the analysis itself.

PROBLEM STATEMENT

The analysis represents the finite parts of function arguments using *finiteness patterns*. Syntactically, a finiteness pattern has the form shown in figure 6.9(a). The structure of finiteness patterns parallels that of abstract values[1]. This is for good reason, since the

$$
\begin{array}{lll}
\phi \in \mathit{FinitenessPattern} = & & \\
\quad F & \text{// finite} \\
\mid\ I & \text{//infinite} \\
\mid\ (\phi_1, \ldots, \phi_n) & \text{//tuple} \\
\mid\ c\ \phi & \text{//construct} \\
\mid\ c_1\ \phi_1 | \ldots | c_n\ \phi_n & \text{//alt}
\end{array}
$$

(a): Syntax

$$
\begin{array}{l}
F^{-1}\ v = v \\
I^{-1}\ v = \mathbf{1} \\
\\
(\phi_1, \ldots, \phi_n)^{-1}\ (v_1, \ldots, v_n) = (\phi_1^{-1}\ v_1\ , \ldots,\ \phi_n^{-1}\ v_n) \\
(\phi_1, \ldots, \phi_n)^{-1}\ \mathbf{0} = \mathbf{0} \\
(\phi_1, \ldots, \phi_n)^{-1}\ \mathbf{1} = (\mathbf{1}, \ldots, \mathbf{1}) \\
\\
(c\ \phi)^{-1}\ \mathbf{0} = \mathbf{0} \\
(c\ \phi)^{-1}\ \mathbf{1} = c(\phi^{-1}\ \mathbf{1}) \\
(c\ \phi)^{-1}(c\ v) = c(\phi^{-1}\ v) \\
(c\ \phi)^{-1}(c_1\ v_1 | \ldots | c_n\ v_n) = c_i\ (\phi^{-1}\ v_i) \text{ where } c = c_i \\
\\
(\phi_1 | \ldots | \phi_m)^{-1}\ v = \phi_1^{-1}\ v | \ldots |\ \phi_m^{-1}\ v
\end{array}
$$

(b): Semantics

FIGURE 6.9: Finiteness Patterns.

intention is to use finiteness patterns as templates to indicate "interesting parts" (and in particular, finite parts) of abstract values. For example, we argued in the example of figure 6.8(a) that the argument of function `incPadded` was a pair whose first component (`vs`) was finite, and the second (`n`) infinite. Correspondingly, the finiteness pattern for this function would be the tuple pattern $\phi_{incPadded} = (F, I)$. On the other hand, if we had a function f with one argument, a list of type `'a list`, and we deduced that only the head of the list is finite, we would ascribe the finiteness pattern $\phi_f = nil \mid cons(F, I)$ to f.

Once the finiteness analysis deduces a particular finiteness pattern for a function, the partial evaluator uses the pattern to extract the "finite parts" of incoming abstract values so that they can be used as contour keys. Figure 6.9(b) shows how this projection works. Given a finiteness pattern ϕ and abstract value v, the figure defines $\phi^{-1}v$, the abstract value that results from applying the pattern to the value. Intuitively, the scheme ensures that if two abstract values v and v' are identical in their finite parts, then $\phi^{-1}v$ and $\phi^{-1}v'$ are structurally identical. The basic idea is to replace all parts of v and v' that correspond to infinite parts of the function with the same abstract value (we pick the abstract value $\mathbf{\mathit{1}}$).

Now that we have specified the output of the analysis precisely, we are ready to can state the requirements of the analysis itself. Let P be a program with functions $F = \{f_1,...,f_n\}$. Let $V = \{v_1, v_2, v_3, ...\}$ be the possible abstract inputs to P. When P is partially evaluated with respect to some abstract input v_i, let the sequence of abstract arguments with respect to which function f_j is partially evaluated be $U_{ij} = u_{ij1}, u_{ij2},..., u_{ijk}$, ... The job of the finiteness analysis is to compute, for each function f_j in F, a finiteness pattern ϕ_j such that for all v_i in V, the set $V_{ij} = \{\phi_j^{-1}(u_{ijk}) \mid u_{ijk} \in U_{ij}\}$ is finite. We say that V_{ij} *closes* V with respect to input v_i and function f_j. Intuitively, we require that although the number of abstract inputs u_{ijk} to a function may be unbounded, the "finite parts" of these inputs must be bounded.

1.We omit recursive finiteness patterns here, although they can be made to have a consistent semantics, and provide functionality beyond the patterns shown here. However, since they are not essential in SCF, and add significant complexity, we omit them here.

```
1  ...
2  and analyzeCmd(c:cmd, lSet:live_set, aMap:assign_map)
3                 :(liveSet * assign_map) =
4    case c of
5      assign(v, e, lbl) =>
6        let val lv = if LiveSet.member(lSet, v) then live else dead
7        in (analyzeExpr(e, LiveSet.delete(lSet, v)),
8            AssignMap.insert(aMap, lbl, lv))
9        end
10   | seq(c,c',_)       =>
11       let val (lSet,aMap) = analyzeCmd(c',lSet,aMap)
12       in analyzeCmd(c,lSet,aMap) end
13   | ...other cmd cases here...
14
15 and analyzeExpr(e:expr, lSet:live_set):live_set =
16   case e of
17     var_ref(v, _)     => LiveSet.add(lSet, v)
18   | primop(op,es)     => analyzeExprs(es, lSet)
19   | const _           => lSet
20   | ...  other expr cases here...
21
22 ...
```

FIGURE 6.10: A Fragment of Dead-Assignment Elimination.

THE INTUITION BEHIND SCF'S FINITENESS ANALYSIS

In designing the finiteness analysis for SCF, we keep in mind that the end effect we are striving for is to "unroll" the optimization over the incoming abstract Abstract Syntax Tree (AST). More precisely, we wish to specialize each analysis or transformation function to each node of the incoming (abstract) AST that it analyzes or transforms. We argue below that, given a conventional style of writing optimizations, this requirement can be phrased as a finiteness requirement. The style of these optimizations also points to a simple implementation for the finiteness analysis.

As a concrete example of how an optimization is written, figure 6.10 replicates a small fragment of the SCF-ML specification of dead-assignment elimination from chapter 3. The main point is that the optimization consists of analysis and transformation functions[1] (collectively called "optimization functions" below), where each type of AST node (e.g., commands, expressions, expression lists) has its own optimization function (`analyzeCmd`, `analyzeExpr` and `analyzeExprs` in this example). Each of these functions has as one of its arguments the AST node being analyzed (e.g., the formal `c` of line 2

1.Only analysis functions are shown in the figure.

and the formal `e` of line 15). Further, given their argument AST nodes, the optimization functions typically extract the sub-trees of these nodes and invoke the corresponding optimization functions on them, or perform simple non-recursive processing. For example, lines 5 and 10 of figure 6.10 extract the components of assignment and sequence commands respectively, whereas lines 17 through 19 extract components of variable, primop and constant expressions respectively. To summarize, typically each optimization function has a formal parameter that represents a node of the AST being optimized; further, this node is purely the result of applying a series of projection operations to the AST input to the optimization program as a whole.

If the input AST to be optimized were a fully concrete AST, it is straightforward to see that for each optimization function f_j, if the q'th formal represents the AST node it is analyzing, then the FP $\phi_j = (I,...,F,...I)$ (an n-tuple where only the q'th entry is F, otherwise I) is a valid (as per the previous section) result of the finiteness analysis. For any incoming AST v_i, let S_i be the set of all sub-trees of v_i.[1] Let V_{ij} be the set $\{(\mathbf{1},...,s,...,\mathbf{1}) \mid s \in S_i$ *, where s is the q'th element of an n-tuple with* $\mathbf{1}$ *in all other positions*$\}$. Let U_{ij} be the set of all abstract values with which the partial evaluator invokes function f_i. Clearly, applying any sequence of projections to v_i will result in a sub-tree of v_i and will therefore be in S_i. For any tuple in U_{ij} of actual parameters $u_{ijk} = (a_1,...,a_q,...,a_n)$ with which f_j is partially evaluated, therefore, since a_q is the result of applying some sequence of projections to v_i, a_q must belong to S_i. $\phi_j^{-1}(u_{ijk}) = (\mathbf{1},...,a_q,...,\mathbf{1})$ must therefore belong to V_{ij}. We therefore have that $\{\phi_j^{-1}(u_{ijk}) \mid u_{ijk} \in U_{ij}\} \subseteq V_{ij}$. Finally, since S_i and therefore V_{ij} are finite, we conclude that the ϕ_j as defined above are valid finiteness patterns as per the last section.

If the input AST is an abstract AST (as is true in SCF), matters become more complex. In what follows, we assume only that the input AST is some abstract value from the set *AbsValue* of all abstract values. If we exclude *alt* and *fix* values from *AbsValue* (call this new set of values *AbsValue'*), the remaining abstract values are identical in structure

1.Not all functions in the optimization program are associated with a unique AST node; some functions, such as the `meet` function, are helper functions. We associate FP I with all functions that are not optimization functions with AST nodes as arguments.

to input AST's above, except that the variants ***1*** and ***0*** are added as possible leaves of the tree representing each value. In this case, for any $v \in AbsValue'$, we can generate V_v that closes *AbsValue'* with respect to v using the same sub-tree based construction as in the previous paragraph.

If we include *alt* values (call this new set *AbsValue"*), we need a definition of sub-tree that accounts for values of the form $v = c_1\ v_1 \mid ... \mid c_n\ v_n$. We assume without loss of generality that $c_i \neq c_j$ if $i \neq j$. For our purposes, the sub-trees of the above value then are just $v_1,...,v_n$. Note that applying a projection operator to v gives one of the v_i. Given this definition of sub-trees, for any v in *AbsValue"*, we can again use the same sub-tree based construction as before to close *AbsValue"* with respect to v.

Finally, consider including *fix* values. The trouble with *fix* values is that applying a projection operation on a *fix* value can result in an abstract value that is not structurally a sub-tree of the original tree. For instance, if $v = fix(c_1 \mid c_2(c_3, c_4\ rec))$, then $c_2^{-1}(v)$ is $(c_3, c_4\ fix(c_1 \mid c_2(c_3, c_4\ rec)))$. The result comes from replacing the *rec* token of the sub-tree $(c_3, c_4\ rec)$ with the original value v. This last observation actually provides a method to generate systematically all the abstract values possibly generated by projection, even in the presence of *fix* values. Consider all structural sub-trees of the input value v to the program. For every sub-tree with an unbound *rec* token, replace the token with the *fix* value that originally bound the token. The resulting set V_v of abstract values closes *AbsValue* with respect to v.

So far, we have established that any function argument such that all paths from program entry to the function entry perform only projection operations is guaranteed to be finite. Regardless of how long the paths are, and how many times the operators are applied, the partial evaluator will only see a finite set of abstract values as the function argument. This observation suggests a dataflow analysis for detecting finite arguments. For each function in the program, begin by assuming initially that the argument is finite (F). For each projection expression (of the form $c^{-1}\ e$ or e^{-i}) in a function, if e is finite, designate the expression to be finite (F) and infinite (I) otherwise. Designate all other expressions to be infinite. If the expression is a function call, $f\ e$, if e is infinite then set the finiteness of *f*'s formal to infinite too. Repeat the process until no function argument or expression changes finiteness. Since there are only two possible finiteness patterns (*F* and *I*), and a program fragment designated *I* can never change back to an *F*, the above iterated process is guaranteed to stop.

The above analysis decides for each function whether its formal is entirely finite or not. There is no notion of a part of the formal being finite and the rest infinite. A two-valued finiteness pattern ($\phi \in \mathit{FinitenessPattern} ::= F \mid I$) therefore suffices in this case. In many cases, however, it is useful to keep track of the parts of expression that are finite. For instance, if *x* is the finite formal of a function, the expression c^{-1} *(c x)* always evaluates to the value of *x*, and should also therefore be designated finite. However, by the analysis we just described, the application of the constructor *c* will cause sub-expression *c x* to be designated *I*, and thereby the whole expression to be *I*.

An option would be to extend the definition of finiteness patterns to ($\phi \in \mathit{FinitenessPattern} ::= F \mid I \mid c\ \phi$). We could then, given the FP *F* for *x*, ascribe the finiteness pattern *c F* to the sub-expression *c x*, and interpret the subsequent projection c^{-1} *(c x)* as "projecting" out the finiteness pattern *F*. A similar argument applies to the construction and projection of tuples, leading the finiteness pattern ($\phi \in \mathit{FinitenessPattern} ::= F \mid I \mid c\ \phi \mid (\phi_1,...,\phi_n)$).

A complication that arises when we allow recursive finiteness patterns is that the resulting set of finiteness patterns becomes infinite. In particular, it is possible that during the iterative process outlined above, a program fragment may assume an infinite sequence of finiteness patterns (such as *F, c F, c c F, c c c F, ...*, or *F, (F, F), (F, (F, F)), ...*). A solution to the problem would be to allow the finiteness analysis to declare at some point that it is in an infinite loop, and to allow it to *widen* select finiteness patterns to *I* so as to ensure termination of the loop. Given the complexity of widening, SCF adopts a pragmatic solution to the problem.

The key is that for optimization programs, it is most important to have *tuple* finiteness patterns, i.e., those of the form $(\phi_1,...,\phi_n)$, and much less important to have *construct* of patterns, i.e., those of the form $c\ \phi$. The reason is that every "multi-argument" function call results in the formation of argument tuples, part of which are finite and part of which are not. Note, for instance, that the `analyzeCmd` function of the dead-assignment eliminator (figure 3.7) takes as argument a tuple, the first argument of which is a fragment of the incoming AST, and the others a set and a map. The first component of the tuple is finite (since the AST only has a finite number of fragments), whereas the rest are not.

Even if we restrict ourselves to tuple patterns, syntactically it is possible for tuple finiteness patterns to form infinite chains such as $F, (F, F), (F, (F, F)), ...$. For a given ML program, however, the chain can contain only a finite (and generally small) number of elements. In particular, the nesting depth of tuples in the finiteness pattern for the argument of a function cannot be any deeper than the nesting depth of product types in the ML type of that argument. For instance, if a function has argument type `(int * (boolean * char))`, it is impossible for the function to have a finiteness pattern with tuples more than two deep. The finiteness of the lattice simplifies the design of the analysis algorithm: we do not have to make provisions for detecting and handling (via widening) infinite loops in the analysis.

THE FINITENESS ANALYSIS IN SCF

The actual finiteness analysis used in SCF, specified in figure 6.11, uses the finiteness patterns $\phi \in FinitenessPattern ::= F \mid I \mid (\phi_1,...,\phi_n)$. We conservatively represent all finiteness patterns of the form $c\ \phi$ by I.

The algorithm maintains a cache *C* (line 4) which represents the currently known map from function names to their finiteness patterns. For convenience of notation, *C* is actually a reference to the cache, and is manipulated via side-effect by the algorithm. When the algorithm terminates, it returns (line 8) the cache referred to by *C*. The returned cache is (a slightly richer form of) the finiteness map mentioned above. For brevity, we write $C[f]$ for looking up function name *f* in cache *C*, and $C[f \rightarrow v]$ for the result of side-effecting key *f* to value *v* in cache *C*.

The algorithm begins by assuming (line 7) optimistically that the entry function has FP *F*. Given the finiteness pattern for the arguments of a function, the algorithm proceeds (lines 18-29) by computing the FP for each sub-expression of the body of the function. Intuitively, for each expression, the FP computed for the expression denotes the finite parts of that expression. While doing so, it maintains an *environment* that maps bound

```
structure FinitenessAnalysis =
$\phi \in$ FinitenessPattern = FP  = F | I | ($\phi_1$, ..., $\phi_n$)
E $\in$ AbstEnv                      = (Id, FP) map
C $\in$ Cache = FinitenessMap ref = ((funName, (FP * FP)) map) ref
val C := ref Cache.empty
fun analyze (p: program): FinitenessMap =
   let _ = $FA_f$ (entry function of p, F)
   in !C end

and $FA_f$ (f: funName, $\phi$: FP): FP =
   let   ($\phi_{arg}$ , $\phi_{ret}$) = case C[f] of SOME pr => pr | NONE => (F, F)
         $\phi_{new}$          = meet($\phi_{arg}$,$\phi$)
         |f x = e|      = the definition of function f
   in if $\phi_{new}$ <> $\phi$ then let $\phi'_{ret}$ = $FA_e$(e, [x -> $\phi_{new}$]) in (C[f -> ($\phi_{new}$, $\phi'_{ret}$)] ; $\phi'_{ret}$) end
      else $\phi_{ret}$
   end

and $FA_e$ (|k|: expr, E: AbstEnv): FP = F
| $FA_e$ (|x|, E)             = E[x]
| $FA_e$ (|c e|, E)           = let $\phi$ = $FA_e$ (e, E) in I end
| $FA_e$ (|($e_1$,...,$e_n$)|, E)   = let [$\phi_1$...$\phi_n$] = List.map (fn e => $FA_e$(e, E)) [$e_1$,...,$e_n$] in |($\phi_1$,...,$\phi_n$)| end
| $FA_e$ (|case e of $m_1$,$m_2$,...,$m_n$|, E)=
                              let $\phi$          = $FA_e$ (e, E)
                                  [$\phi_1$...$\phi_n$] = List.map (fn |pt => e'| =>
                                                    let E' = match(E, pt, $\phi$) in $FA_e$(e' , E') end)
                                                  [$m_1$,...,$m_n$]
                              in List.foldl meet F $\phi_1$...$\phi_n$ end
| $FA_e$ (|f e|, E)           = let $\phi_{arg}$ = $FA_e$ (e, E) in $FA_f$ (f, $\phi_{arg}$) end
| $FA_e$ (|p e|, E)           = let $\phi_{arg}$ = $FA_e$ (e, E) in I end

and meet (I: FP, _: FP) : FP    = I
| meet (_, I)                   = I
| meet (F, $\phi$)                   = $\phi$
| meet ($\phi$ , F)                  = $\phi$
| meet (|($\phi_1$,...,$\phi_n$)|, |($\phi'_1$,... ,$\phi'_n$)|) = ListPair.map meet ([$\phi_1$,...,$\phi_n$] , [$\phi'_1$,...,$\phi'_n$])


and match (E:AbstEnv, pt: Pattern, I: FP): AbstEnv = E[pt ->I]
| match (E, pt, F)                                 = E[pt ->F]
| match (E, |x|, $\phi$)                                = E[x ->$\phi$]
| match (E, |_|, $\phi$)                                = E
| match (E, |($pt_1$,...,$pt_n$)|, |($\phi_1$...$\phi_n$)|)           = ListPair.foldl (fn((pt,$\phi$),E)=> match(E, pt, $\phi$))
                                                             E ([$pt_1$,...,$pt_n$] [$\phi_1$, ..., $\phi_n$])

and projectFiniteArgs (f: funName, C: FinitenessMap, v:value): value list=
let   ($\phi_{arg}$ , _)       = C[f]
in extract (as per figure 6.9(b)) the parts of v determined to be finite by finiteness pattern $\phi_{arg}$ end

end (* structure FinitenessAnalysis *)
```

FIGURE 6.11: Finiteness Analysis.

variables to the FP's for those values (just as a conventional interpreter would maintain an environment mapping variables to the values they are bound to). We write $[x \rightarrow \phi]$ to denote an environment with singleton binding from x to ϕ, $E[x]$ for the field x of environment E and $E[x \rightarrow \phi]$ to denote environment E augmented with x bound to ϕ. The rules for processing expressions are straightforward for the most part, and we discuss them in order below.

- If the expression is a constant (line 18), it is wholly finite (the partial evaluator will evaluate the expression to a single value).
- If the expression is a variable (line 19), its FP is that ascribed to it in the environment.
- If the expression is a value constructor of the form *c e* (line 20), then the expression is conservatively ascribed FP *I*. As discussed previously, the finiteness analysis of SCF does not track finite parts of constructed values. We still analyze the body *e* of the expression, since *e* may invoke a function and thereby contribute to its FP.
- If the expression is a tuple (line 21), the FP for the expression is simply a tuple-FP whose components are the FP's for the individual components of the incoming tuple.
- If the expression is a case expression (lines 22-27) of the form ***case** e **of** matches*, we proceed as follows. We first compute the FP ϕ for expression *e*. Now, each match in *matches* has the form *pattern => expression* (where *pattern* may contain a finite set of binding variables).

  For each match, we first compute finiteness patterns for the variables bound in the pattern via the *match* function (called on line 25, implemented on lines 38-43); the extra variable-to-FP bindings are added to the environment in which the *expression* part of the match is processed.

  The *match* function has five cases. If the pattern ϕ above is *I*, i.e., the controlling expression of the case expression is infinite, so are the values bound by its patterns (line 38). We write $E[pt \rightarrow I]$ to indicate that all variables in pattern *pt* are bound to FP *I* in environment *E*. If ϕ above is *F*, i.e., the controlling expression of the case expression is finite, then so is every variable projected out of it (line 39). Note that we are allowed to do this because projection operators preserve closure. If the incoming pattern is a variable *x*, then in the resulting environment, *x* is bound to the same FP ϕ as the controlling expression *e* being matched (line 40). If the pattern is a wildcard, the resulting environment is the same as the incoming one (line 41). Finally, if the pattern is a tuple pattern of the form $(pattern_1,...,pattern_n)$, the incoming FP ϕ must be of the form $(\phi_1,...,\phi_n)$, and resulting environment is simply the union of the environments that result from matching ϕ_i to $pattern_i$ (lines 42-43). No other case is legal for the match. For instance, patterns of the form *c p* must match FP's of the form *F* or *I*, since we do not have FP's of the form $c\ \phi$; we therefore do not have to special-case these patterns.

Given the augmented environments for each match, we compute (line 25) the FP's for the expressions constituting the bodies of each match as usual. Now, if $\phi_1,...,\phi_n$ are the FP's for the matches, then the FP for the case expression is the *meet* of these FP's.

The *meet* function is defined on lines 31 through 35. It determines the composite FP of a variable that may have two different FP's when computed along two different paths. The meet of any pattern with *I* is *I* (lines 31-32). Intuitively, if a variable can be computed along two different paths, along one of which it may assume during partial evaluation an infinite number of values and along the other a finite number, the variable itself may have an infinite number of values. The meet of any pattern ϕ with *F* is ϕ (lines 33-34). Intuitively, if a variable can be computed along two different paths, along one of which it is restricted to assume a finite number of abstract values during partial evaluation, then the finiteness of the variable is the same as that of implied by the other path. In dataflow analysis terms, *I* and *F* are the bottom (most conservative) and top (most precise) of the lattice of finiteness patterns. Finally, if the two abstract values both have tuple FP's, the FP of the resulting abstract value is simply the tuple whose individual components are the pointwise meet of the incoming FP's (line 35).

- If the expression is a function call of the form *f e* (line 28), we first find the FP ϕ for *e*, and immediately reprocess the function *f* with new argument FP ϕ to determine the new FP for its arguments and return values.

  Analyzing a function is fairly standard (lines 10-16). Essentially, we check if the incoming FP has been handled before; if so, we return the return value previously computed. If not, we analyze the body of the function using an argument FP that is the meet of the previously recorded finiteness pattern for the function, and the new one, and return the resulting FP. The meet on the argument FP ensures that the argument is only designated as finite only if it is finite along all paths leading to the function call.

- If the expression is a primitive operation, we assume that the result is infinite. The rationale for this is very similar to that for constructors: unbounded application of many primitive operations, such as addition, subtraction, division and multiplication, can result in an unbounded number of result values given a single input value. Since primitive operations work on atomic primitives and on maps, and not on the crucial AST nodes we wish to specialize on, we conservatively label infinite any values of these types.

6.2.1.2 How Much to Specialize: Contour Chains from Lists of Finite Values

Once we identify the finite arguments of functions, we need to decide how to use the finite arguments to form contour keys. So far, we have simply used the abstract value of the finite argument as contour key for each function. In SCF, this strategy is not quite enough to give us the desired level of specialization. In what follows, we explain how this simple strategy is inadequate, and then describe a more complex strategy that addresses these inadequacies.

PROBLEM 1: SPECIALIZING NON-FINITE FUNCTIONS

The first problem with using finite arguments as contour keys is that some functions may not have any finite arguments. We will call such functions *non-finite functions* below. The `meet` function of the dead code elimination optimization (figure 3.7, line 62) is an example of a non-finite function. Consider applying dead-code elimination (from figure 3.7) to

```
1 int foo(int x){
2   while(read()){x=7;}1
3   x = 10;
4   while(read()){x--;}2
5   return x;}
```

FIGURE 6.12: Example Function to Be Optimized.

the function `foo` of figure 6.12. As per the specialization keys derived from the finiteness analysis, the `analyzeWhile` function for analyzing *while* loops will be replicated, with one copy for each of the loops on lines 2 and 4 (since the finiteness analysis has the effect of declaring any function argument that is a sub-tree of the incoming AST as finite). We have labelled the two while loops with numbers 1 and 2, and write the corresponding replicas of the `analyzeWhile` function as $\texttt{aW}_1$ and $\texttt{aW}_2$ respectively. However, the `meet` function will not be replicated (since its arguments are maps, and therefore not sub-trees of the input AST), so that both replicas of `analyzeWhile` will invoke the same instance of the meet function, as shown in figure 6.13.

In the case of while loop 2, the partial evaluator will be invoked on the meet function with an abstract live-variables set containing variable `x`. This is because the return command immediately after the second while loop uses `x`, and therefore keeps `x` live. On the other hand, in the case of while loop 1, the partial evaluator will be invoked with an abstract set that does not contain `x`, since the assignment command immediately after the first while loop defines `x`. These two possible abstract arguments are shown in figure 6.13 as annotations on the corresponding edges[1].

Unfortunately, the partial evaluator will first meet the information from the two sources before analyzing the body of the `meet` function. The result is a “smearing” of

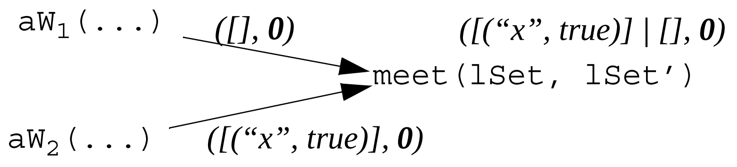


FIGURE 6.13: A Problem With Not Specializing Non-Finite Functions.

1.The figure represents the abstract live-variables sets as abstract maps. The set *{“x”}* is represented as the map *[(“x”, true)]*. The empty set is represented as the empty map. This translation is necessary because SCF has no direct mechanism for representing abstract sets.

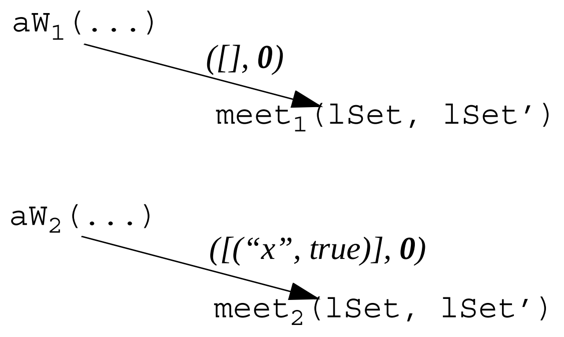


FIGURE 6.14: Specializing Non-Finite Functions Using the Context Key of Their Callers.

information from the two callsites: the partial evaluator will conclude that the incoming live-variables set may or may not contain the variable `x`, and therefore that `x` may or may not be live in either loop body. The corresponding removal of dead assignments will then result in optional removal of assignments to `x` in both loop bodies, when a more accurate analysis should have resulted in unconditional removal of the assignment in loop 1 but not the one in loop 2.

Now that we have established that there is a clear cost to not specializing non-finite functions, and that these costs are applicable in fairly common situations, we need to address the question of how to specialize these functions. What contour keys should we use for specialization? In SCF, we view non-finite functions as helper functions to the finite functions that call them. Correspondingly, we attempt to analyze a distinct copy of a non-finite function for each finite caller of the function. We achieve this effect by using the contour key of the finite calling function to specialize non-finite callee functions. The partial evaluator, while evaluating the body of any function, keeps track of the contour key for that function explicitly. The contour key is the argument of type *Contour-Key.contour_key* as specified on line 9 of figure 4.5 (which specifies the type signature for the partial evaluator for expressions). At every callsite, the partial evaluator uses the contour key of the caller function as one of the ingredients of the contour key for the callee (line 41 of figure 4.5). Figure 6.14 illustrates the results of the new scheme: notice that the smearing across callsites has disappeared because each callsite has its own version of the callee.

```
1   int foo(int x){
2    int x = x + 1|2;
3    while(read()){
4     print(x);}1
5    return;}
```

FIGURE 6.15: Modified Example Function to Be Optimized.

PROBLEM 2: SPECIALIZING FINITE RECURSIVE FUNCTIONS

Specializing non-finite functions by using the keys of their callers improves the accuracy of partial evaluation, but still loses precision in an important case. Consider the case where a node in the incoming function needs to be analyzed iteratively to fixpoint. In SCF-ML, such analysis would typically be achieved by a recursive analysis function. An example is the `analyzeWhile` function in the dead-assignment elimination example of figure 3.7, which iteratively analyzes `while` nodes. Since the `analyzeWhile` function has a finite argument (the command being analyzed),[1] by our specialization strategy so far, we would produce a distinct contour key for each `while` command being analyzed. We would thus avoid merging the analysis-values corresponding to different `while` commands in the program. However, since the `analyzeWhile` function may recursively be called many times on the same `while` command, we may still merge analysis values from the evaluation of these distinct calls. The resulting loss of accuracy adds significant overhead to the residual version of the analysis.

Figure 6.15 illustrates this problem. Consider analyzing the function `foo` of figure 6.15. In particular, consider the while loop (labelled "1") of lines 3 and 4. As per the dead-assignment elimination optimization of figure 3.7, the loop will be analyzed by invoking the `analyzeWhile` function. The `analyzeWhile` function calls itself recursively if fixpoint has not yet been reached (figure 3.7, line 64), and also calls the `meet` helper function.

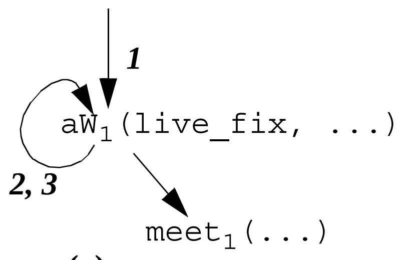
Figure 6.16(a) shows the specialized callgraph that would result from the specialization strategy we have discussed so far. We would analyze separately a version of the `ana-`

1.As actually implemented in figure 3.7, the original while loop is not the argument the `analyzeWhile` function. Instead, we pass two arguments corresponding to the loop: the exit condition of the loop (named `test`), and the body of the loop (named `body`). The corresponding contour key will be the result of projecting these two arguments out using finiteness pattern *(I, I, I, F, F)*.

`lyzeWhile` and `meet` functions using the contour key corresponding to the while loop. We write the corresponding distinct versions of the `analyzeWhile` and `meet` functions as `aW`$_1$ and `meet`$_1$ respectively (the subscript “1” here corresponds to the fact that the finite argument to `analyzeWhile`, i.e., the while command, has label 1). The recursive call to `analyzeWhile` in the original `analyzeWhile` function is translated to a call to the `aW`$_1$ function: since the recursive call to the `analyzeWhile` function has as finite argument the same while loop as the initial call, its target would also be the function `analyzeWhile`$_1$, resulting in the back-edge of figure 6.16(a).

The edges of the callgraph of 6.16(a) are labelled by the order in which the partial evaluator processes them. The partial evaluator is initially invoked on `analyzeWhile` from a callsite outside the `analyzeWhile` function (this call is labelled ***1***), and the next two calls to the partial evaluator (labelled **2** and ***3***) are triggered by the recursive call to `analyzeWhile` from within its own body.

Figure 6.16(b) tracks the value of the argument `live_fix` of `aW`$_1$ over successive invocations to the partial evaluator. Recall that argument `live_fix` represents the set of live variables immediately after the loop-exit test. The first time the loop is analyzed, the argument `live_fix` evaluates to the empty set *{}* (since no variables are live downstream of the loop of figure 6.15). This value is labelled ***call1 args*** in the figure. The `analyzeWhile` function then analyzes the controlling expression and body of the loop, concludes that a new live set *{x}* is now applicable, and recursively invokes `analyze-While` with this argument. Unfortunately, since we are analyzing the same contour `aW`$_1$ of `analyzeWhile` in both the initial and recursive calls, the partial evaluator does a meet of the arguments before doing the recursive analysis. This is the meet done on line 31



(a)

call1 args: *(live_fix = {}, ...)*

call2 args: *(live_fix = {x} | {}, ...)*

call3 args: *(live_fix = {x} | {}, ...)*

(b)

FIGURE 6.16: A Problem With Maintaining One Contour Key Per Finite Argument.

of figure 3.7. The new abstract argument value of $\texttt{aW}_1$ is therefore *{}|{x}* (the result of meeting *{}* with *{x}*). This value is labelled ***call2 args***. In processing this new function call, the partial evaluator again encounters a recursive invocation of $\texttt{aW}_1$. However, this time around the `live_fix` argument has the same abstract value as in the previous iteration (***call3 args***), and the partial evaluator terminates its analysis of the while loop with an abstract live-variables set of *{}|{x}*.

The result abstract value for the loop of *{}|{x}* implies that it is possible that the live-variables set may have value *{}* at the entry to the loop. The partial evaluator thus has to reckon with the possibility that at the head of the loop, no variables at all are live. The downstream transformation code that uses the value of this live-variables set has to take this possibility into account (by generating residual code to conditionally eliminate the assignment statement `x = x + 1| 2` preceding the while loop). It is easy to see that in practice the live-variables set is guaranteed to have variable `x` in it, and that the transformation step should therefore unconditionally leave the assignment untouched. The approximation introduced by the meets performed at the recursive calls to the `analyze-While` function therefore result in less efficient residual code being produced.

One way to solve the problem of excessive merging is to generate a separate contour key for the invocations of the `analyzeWhile` function that we want to keep separate, so that we never meet the results from two such invocations to the function. As mentioned above, the merging we want to avoid is that which results from using the same contour key for various recursive invocations of a function. For example, we would like the partial evaluator to use one contour key when the `analyzeWhile` function is initially invoked on the while loop (and, in general when any finite recursive function is invoked), a different contour key for the recursive call to the `analyzeWhile` function on the same loop, yet another key if the `analyzeWhile` function is invoked recursively from within this second recursive call, and so on.

In SCF, we generate a separate contour key per invocation as follows. If the partial evaluator is invoked on a call-chain $f_1,...,f_n$ of functions with finite arguments $v_1,...,v_n$ (we will call the pair $k_i = (f_i,v_i)$ a *specialization key* for function f_i in what follows), then we use the list $[k_j,...,k_1]$ of specialization keys as the *contour key* for invocation f_j.

A side benefit of generating the contour keys in this manner is that the technique automatically ascribes to non-finite functions (essentially) the contour key of their callers. If we invoke a non-finite function *f* from a function with contour key *ck*, then the key for *f* will be *(f,**1**)::ck*.[1] Each finite function is therefore guaranteed to get a distinct copy of its helper functions, as required in the previous section.

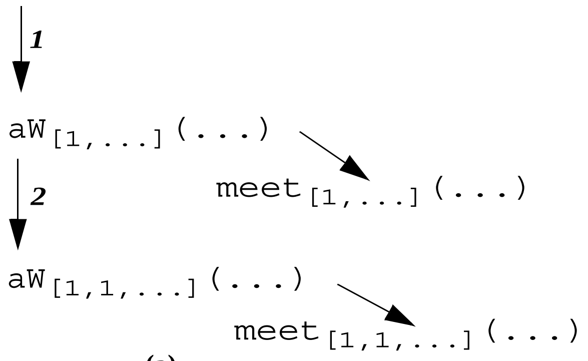
There is, however, a problem with the new scheme for generating contour keys. Even though the number of specialization keys is bounded (as guaranteed by the finiteness analysis), the number of contour keys is no longer bounded. A function may be invoked recursively with a bounded number of finite abstract arguments, but an unbounded number of non-finite abstract arguments, and each of these invocations will result in a new contour key.

The solution adopted in SCF is to bound the number of times a particular specialization key may appear in a contour key (or in effect, the number of times a recursive function may be "unrolled"), with a small integer, *k*. The fact that the augmented contour keys are aimed at specializing recursive functions that analyze loops in the incoming program (e.g. the `analyzeWhile` function analyzes while loops in incoming program) provides us with a useful constraint in this direction. In particular, it has been observed [2] that although in the worst case many dataflow analyses may iterate $O(n^2)$ times when processing functions of size *n*, in practice they tend to iterate a small, constant number or times: often, 2 or 3 times. In SCF, therefore, we use $k = 2$ as the depth to which recursion is specialized.

Figure 6.17 shows the contours produced by this approach when analyzing function `foo`. Where we originally had the single contour $\texttt{aW}_1$ any time `analyzeWhile` was invoked on the while loop, we now maintain separate contours $\texttt{aW}_{[1,...]}$, $\texttt{aW}_{[1,1,...]}$, etc. for each distinct recursive invocation of the `analyzeWhile` function on the incoming while loop.[2] Figure 6.17(a) shows the resulting specialized callgraph. Figure 6.17(b) shows the abstract values for the arguments on this callgraph. The important case is that of

1.Recall that (figure 6.9(b)) the finite argument ascribed to a non-finite function is ***1***.

2.The ellipses represent the chain of specialization keys corresponding to the chain of function calls that led to the call to `analyzeWhile` being specialized.



call1 args: *(live_fix = {}, ...)*

call2 args: *(live_fix = {x}, ...)*

(b)

FIGURE 6.17: Using Chains of Finite Arguments as Contour Keys.

call arg2. Since call 2 results in a different contour than call 1, we do not need to meet the abstract argument values for the two calls. The result is that in this case, the `live_fix` argument of ***call arg2*** has the abstract value *{x}* (composed to the previous less precise value of *{}*|*{x}*). The more precise abstract value obviates the need for another recursive call to `analyzeWhile`, and returns *{x}* as the set of live-variables for the entire while loop.

The addition of key lists makes the worst-case running time (and size) of the analysis exponential in the size of the result of performing purely finite-argument-based specialization on an optimization program (or equivalently, in the size of the incoming abstract function). A program that consists of n nested while loops can result in k^n specialized versions of the `analyzeWhile` function. However, in the common case, after specialization on finite arguments, the resulting optimization program tends to have a tree-shaped callgraph with relatively few merges (equivalently, the nesting depth of programs to be optimized is typically independent of the size n of the program, and bounded by a small, fixed number e.g. 2 or 3). With a tree-shaped callgraph, call-chain-based specialization costs drop from exponential to linear.

THE CONTOUR KEY MODULE IN SCF

In the previous two subsections, we have described two problems (non-finite functions, and recursive finite functions) in the design of contour keys, and outlined the solutions to these problems. In this subsection (see figure 6.18) we provide the precise specification of

```
1  structure ContourKey =
2  struct
3  structure FA = FinitenessAnalysis
4  type specialization_key  = SCF_ML.funName * value
5  type contour_key         = specialization_key list
6
7  val k = ref 2
8  val emptyCtrKey = []
9
10 fun mkContourKey (f: SCF_ML.funName, fm: FA.FinitenessMap, v: value, ck:contour_key):
11                         contour_key =
12   let v_finite   = FA.projectFiniteArgs (fm, f, v)
13       sk      = (f, v_finite)
14       k'      = List.length (List.find (fn(sk') => sk = sk') ck)
15   in
16       if k' < !k then sk:: ck
17       else findCKSuffix (sk, ck)
18   end
19
20 and findCKSuffix (_: specialization_key, []: contour_key): contour_key = []
21 | findCKSuffix (sk, ck as (sk'::ck')) =
22   if sk = sk' then ck
23   else findCKSuffix(sk, ck')
24
25 and getSpecializedFName (ck: contour_key): funName =
26   ... return a unique function name for each ck...
27
28 fun func = getSpecializedFName      //An abbreviation used in figure 4.4
29 end (* structure ContourKey *)
```

FIGURE 6.18: Module Implementing Contour Keys.

the contour key module in SCF. Although the criteria to be satisfied by contour keys are somewhat tricky, the solution that results is straightforward to implement.

As described in the preceding section, a *specialization key* (line 4) represents a function and the finite abstract value it was invoked with. A *contour key* (line 5) is a list of specialization keys. The number of times a particular specialization key may appear in a contour key is *k* (line 7). As a degenerate case, $k = 0$ results in disabling function specialization.

The main function in the module is *mkContourKey*, which creates new contour keys, given the name *f* of the function to be specialized, the finiteness map *fm* for the program, the abstract value *v* representing the argument of the function to be specialized, and the contour key *ck* of the caller function. To generate the new contour key, we first use *fm* to project out the finite value v_{finite} corresponding to argument v (line 12), and create a new specialization key *sk* by pairing the current function name with v_{finite} (line13). Next (line 14), we determine the number of times, *k*', that the new specialization key has already appeared in the contour key. If *k*' is below the threshold *k* then we prepend *sk* to the old contour key to produce the new contour key list (line 16). Otherwise, we return as contour key the longest suffix of *ck* starting in *sk* (line 17). Function *findCKSuffix* (lines 20-23) computes this suffix. Returning the suffix in this manner ensures that although the first *k* recursive invocations of a function result in distinct specialized versions of the function, all deeper recursive invocations result in a call to the *k*'th specialized version.

A final detail is that client modules (e.g. the partial evaluator module of figure 4.5, line 46) require a function *getSpecializedFName* (detailed on lines 24-25) to generate a unique function name , given the contour key for a function. This can be done in a number of ways, including hashing.

6.2.2 Expression Specialization

It is sometimes beneficial, while partially evaluating an expression, to first replicate the expression and partially evaluate each replica separately for a distinct context, i.e., for a distinct abstract environment. In this section, we first present an example showing how expression specialization can benefit partial evaluation of optimizations, and then discuss how SCF provides the specialization required.

6.2.2.1 Example: Expression Specialization and Dead-Assignment Elimination

Consider computing the set of live variables for the abstract expression *x+1|2+x* by invoking[1] the `analyzeExpr` function from the dead-assignment eliminator of figure 3.7. Figure 6.19(a) shows an abbreviated trace of the function invocations that result from this invocation, and their input and output values. The first column of the figure is the invocation number, indicating the sequence in which the invocations are processed by the

1.When we refer to function “invocations” here, we are of course referring to instances when the function is executed abstractly by the partial evaluator, not to function calls in a concrete execution.

no.	function	input expr(s)	input live set	output live set
1	analyzeExpr	*+[x,1] | +[2,x]*	*must[], may[]*	*must[], may[(x, true)]*
2	analyzeExprs	*[x,1] | [2,x]*	*must[], may[]*	*must[], may[(x, true)]*
3	analyzeExpr	*2|x*	*must[], may[]*	*must[], may[(x, true)]*
4	analyzeExprs	*[1] | [x]*	*must[], may[(x, true)]*	*must[], may[(x, true)]*
5	analyzeExpr	*1|x*	*must[], may[(x, true)]*	*must[], may[(x, true)]*
6	analyzeExprs	*exprs_none*	*must[], may[(x, true)]*	*must[], may[(x, true)]*

(a)

```
fun analyzeExprs(es:exprs, lSet:live_set):live_set=
 case es of
   exprs(e,es)        => analyzeExprs(es,analyzeExpr(e))
  | exprs_none         => lSet
```

(b)

```
fun analyzeExprs[x,1] | [2,x](es:exprs, lSet:live_set):live_set=
  case es of
    exprs(e as const _,es)=> analyzeExprs[2,x](es,analyzeExpr(e))
   | exprs(e,es)        => analyzeExprs[x,1](es,analyzeExpr(e))

```

(c)

FIGURE 6.19: Expression Specialization Applied to Dead-Assignment Elimination.
(a) Partial execution trace of dead-assignment elimination (b) The `analyzeExprs` function before expression specialization (c) The `analyzeExprs` function after expression specialization on the abstract expression. *[x,1] | [2,x]*.

partial evaluator. The second column is the name of the function invoked. The third column is the abstract value representing the expression or expression list input to the function. The last column is the abstract value representing the live variables set that results from the invocation. As discussed previously (section 3.2.2.2), abstract sets are represented by their membership functions in SCF: a value is a member of a set if and only if it maps to *true* in the map representing the set.

We assume we start with an empty input live set. To find the live variables in the abstract add operation *x+1|2+x* (call 1), we find the live variables in the abstract expression list *[x,1]|[2,x]* that is the argument of the add operation (call 2). For convenience, figure 6.19(b) reproduces the definition of the `analyzeExprs` function from figure 3.7(b), which is responsible for analyzing expression lists.

```
| $PE_e$ (|case e of ms|, E, ck) =
   let  (e',v)    = $PE_e$ (e, E, ck)
        ms        = discriminate(ms, v)
        (ms', v') = $PE_{ms}$ (ms, v, E, ck)
   in rematerialize (|case e' of ms'|, v') end
```

FIGURE 6.20: Invoking the expression specializer from the partial evaluator.

Consider partially evaluating this function on the expression list *[x,1] | [2,x]*. Recall that *[x,1]* and *[2,x]* are compact notation for *exprs(var("x"), exprs(const(1),exprs_none))* and *exprs(const(2),exprs(var ("x"),exprs_none))* respectively. Matching these abstract values against the pattern exprs(e,es) from line 3 of figure 6.19(b), we bind value *x | 2* to variable e and *[1] | [x]* to variable es. Unfortunately, these bindings lose correlation implicit in the incoming value: in particular, they imply (over-conservatively) that e may have concrete value *2* when es has value *[1]*, i.e., that the incoming expression list could be *[2,1]*. The latter possibility implies that *x* may not be live in the incoming expression list: note that the returned abstract live set (line 2 column 5) is *([must[], may[(x, true)])*, which includes the concrete set *{}* as a conforming value.

Figure 6.19(c) shows how expression specialization can fix the problem. Essentially, the specializer inserts *discriminating patterns* to prevent bindings that lose correlation. In particular, the extra match of line 3 only matches the expression list *[2,x]* so the match of line 4 will only match expression list *[x,1]*. Analyzing these values individually will result in abstract live-variables sets *([must[(x, true)], may[])* for each branch of the case expression, resulting in the set *([must[(x, true)], may[]])* for the whole case expression. Note that the latter set has exactly one conforming concrete set, *{x}*, so that it excludes the possibility that *x* may not be live downstream. Note also that expression specialization eliminates matches that can have no possible concrete values matching against them: the exprs_none case of figure (b) is not in figure (c).

6.2.2.2 Expression Specialization Support in SCF

Expression specialization in SCF is limited to avoiding loss of correlation from case expressions. The interface between the partial evaluator and the specializer is via the *discriminate* function which takes the list of matches, i.e., pattern-expression pairs of the case expression, and the abstract value being matched against, and returns a new list of matches. The new list will, in general, contain more matches with discriminating patterns, and will therefore result in less loss of correlation. Figure 6.20 shows how the part of the partial evaluator that handles SCF-ML case expressions is modified. The additional code (relative to the version of figure 4.5) required to achieve expression specialization is underlined in line 3 of figure 6.20. The new code replaces, in a modular way, the old list of matches with a new, more discriminating list of matches.

Figure 6.21 specifies the *discriminate* function. The important activity is in the *foldl* iterator of lines 2 through 6. Given (line 6) the incoming list of matches *msIn*, and the list *vs* of abstract values that need to be discriminated, the iterator steps through each match *m* accumulating discriminating matches as follows. Let *p* be the pattern for the match *m*, let *ms* be the discriminating matches accumulated so far, and let *vs* be the list of possible abstract values to match against (line 2). Using the *findPossibleMatches* function, we first identify the part *vs'* of *vs* that have at least a partial match with pattern *p* and the part *vs"* of *vs* that are might not match *p* (line 3). We then use the *discriminateMatch* function to produce a list of discriminating matches *ms'*, such that matches in *ms'* discriminate between values in *vs'* (line 4). Finally, we append the new list *ms'* of discriminating matches to the list *ms* of matches accumulated thus far (line 5). The possibly unmatched abstract values *vs''* are reserved for successive iterations.

```
1  fun discriminate(msIn: SCF_ML.match list, alt(vsIn, _): value): SCF_ML.match list=
2    let (ms,_) = foldl (fn(m as |p => _|, (ms,vs))=>
3                          let   (vs', vs'')   = findPossibleMatches(p, vs)
4                                ms'           = discriminateMatch(m,vs')
5                          in (ms@ms', vs'') end)
6                  ([],vsIn) msIn
7    in ms end
8  | discriminate (msIn, vIn as |fix _|)  = discriminate(msIn, (unroll vIn))
9  | discriminate (msIn, _)               = msIn
```

FIGURE 6.21: The *discriminate* Function for Specializing Case Expressions.

```
fun findPossibleMatches(p: SCF_ML.pattern, vs: value list):(value * value) list =
  foldl   (fn (v,(vsMatched, vsMismatched)) =>
              case AbstractValue.match(p,v) of
                 SOME(_, top)   => (v::vsMatched, vsMismatched)
               |  SOME (_, v')   => (v::vsMatched, v'::vsMismatched)
               | NONE            => (vsMatched, v::vsMismatched)
          ([],[]) vs
```

FIGURE 6.22: The *findPossibleMatches* Function.

```
1  fun discriminateMatch (m: SCF_ML.match, vs: value list): SCF_ML.match list =
2    let numMatchOps = numMatchOpsInMatch m
3        maxMatchOps = numMatchOps + !maxExtraMatchOps
4        ms          = genMatches(m, vs, numMatchOps, maxMatchOps, [])
5    in ms end
```

FIGURE 6.23: The *discriminateMatch* Function for Generating Discriminating Matches.

IDENTIFYING VALUES TO DISCRIMINATE BETWEEN

Figure 6.22 specifies the *findPossibleMatches* function. Given pattern *p* and list *vs* of abstract values that need to be discriminated, the function iterates through each value *v* in *vs*, testing if *v* matches *p* (line 3), maintaining lists *vsMatched* (for those values processed so far that at least partially match *p*), and *vsMismatched* (for those values that may at least partially not match *p*). If the match is a total match, i.e., there is no part of *v* that does not match *p* (line 4), then we add *v* solely to *vsMatched* (recall that the *match* function returns the "unmatched part" as the second component of its return tuple, so a value of *top* for this component implies the match was complete), if the match is partial, with *v'* the "unmatched part", we add *v* to *vsMatched*, and *v'* to *vsMismatched* (line 5), and if there is definitely no match (line 6), we add *v* solely to *vsMismatched*.

GENERATING SEQUENCES OF DISCRIMINATING MATCHES

Figure 6.23 specifies the *discriminateMatch* function. Given a match *m* and a list *vs* of values to discriminate among, the function returns a set of matches that discriminate between the values as much as possible, such that the extra matching overhead is bounded (and typically, small). The bound is determined by a global parameter: *maxExtraMatchOps* is the maximum number of extra matching operations each discriminating match is allowed to perform. For example, the pattern `exprs(const _,es)` of figure 6.19(c) performs more matching operations than the pattern `exprs(e,es)` it replaced. Before performing its central task of generating the discriminating matches (via the helper function *genMatches*), *discriminateMatch* first computes the number *numMatchOps* of operations necessary to perform the incoming match *m* (line 2), adds on *maxExtraMatchOps* to determine the maximum number *maxMatchOps* of operations a generated discriminating match is allowed to perform, and invokes *genMatches* to generate discriminating matches that perform at most *maxMatchOps* operations.

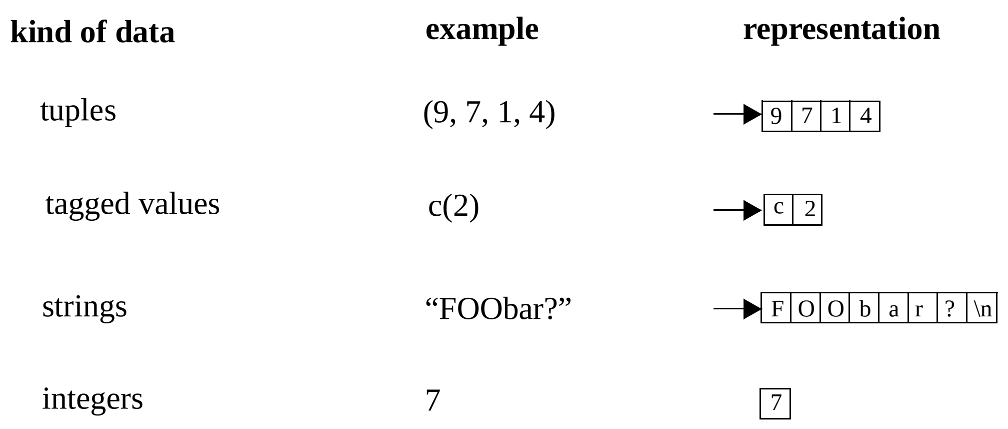


FIGURE 6.24: Data Representation Assumed by SCF.

ESTIMATING THE COST OF DISCRIMINATORS

A question that needs to be resolved at this point is that of counting the number of operations a match operation performs. For instance, how many more operations does the pattern `expr(const(2),es)` of the previous paragraph perform at run time compared to the pattern `expr(e,es)` that it replaces? In SCF, we interpret “number of operations” to mean the number of operations that will be executed on a modern RISC machine.

Given that the pattern matches are ultimately compiled into a series of access and comparison operators on the data being matched against, it is necessary to know how the data being matched is represented in memory. We assume the layout shown in figure 6.24. The representation is a standard [6] “boxed” representation. All values except atomic values (integers, booleans and characters) are assumed to be represented by pointers to contiguous words in memory. An *n*-tuple is represented by *n* contiguous words in memory (we ignore word-size issues). A tagged value is represented by an adjacent pair of words, with the first word containing the tag and the second the value. Strings are represented as a null-terminated sequence of words, each word containing a character. Atomic types are represented by a single word.

The above assumptions on data layout lead to the algorithm of figure 6.25 for estimating the cost of executing a pattern match. The estimate is a conservative (worst-case) estimate, and assumes that the entire pattern is traversed in search of a match before the match fails. The algorithm steps through the input pattern recursively adding up the cost to load

```
1  fun numMatchOpsInMatch (m as |p => _|) = numMatchOpsInPattern p
2
3  and numMatchOpsInPattern |c _|    = 2
4  | numMatchOpsInPattern |c p|      = 3 + numMatchOps p
5  | numMatchOpsInPattern |(ps)|     =
6     let numNonWildCards = List.foldl (fn (|_|,i) => i| (_, i) => i+1) 0 ps
7     in List.foldl (fn (p,i) => numMatchOpsInPattern p + i) numNonWildCards ps end
8  | numMatchOpsInPattern |x as p|   = numMatchOpsInPattern p
9  | numMatchOpsInPattern |"s"|      = (String.length s)*5
10 | numMatchOpsInPattern |c|        = 2
11 | numMatchOpsInPattern |_|        = 0
```

FIGURE 6.25: Counting the Number of Operations to Implement a Pattern Match.

and compare against the corresponding parts of the value being matched against. When the pattern is a tagged-value comparison, if the pattern representing the value is a wildcard, the cost is just that for loading the tag and comparing it, i.e., 2 operations (line 3). If the value-pattern is not a wildcard, then there is an additional offset load for accessing the carried value itself, and the recursive cost of matching it (line 4). Similarly, when matching against a tuple, wildcards in the tuple incur no cost, whereas all other places incur a 1-operation cost for loading the corresponding value; the recursive cost of matching against these values is then added (lines 5-7). Variable bindings incur no cost (line 8). Matching against a string requires iteration over the characters of the string; each iteration requires a counter update, two loads and two compares (five operations in all), so that the total worst-case cost is five times the length of the string (line 9). Atomic constants require a load and compare (line 10). Wildcards require no action (line 11).

COMPUTING INDIVIDUAL MINIMAL DISCRIMINATING MATCHES

Figure 6.26 sketches how the *genMatches* function works. Recall that the role of the function is, given a match *m* (of the form *p=>e*), a list of values *vs* that have at least a partial match with *m*, a current number of match operations allowed *n*, a maximum number of match operations allowed *max*, and a list of discriminating matches generated so far *msSoFar,* to augment *msSoFar* with versions of the match *m* that discriminate between as many of the values in *vs* as possible. All match operations in *m* must have a cost between *n* and *max*.

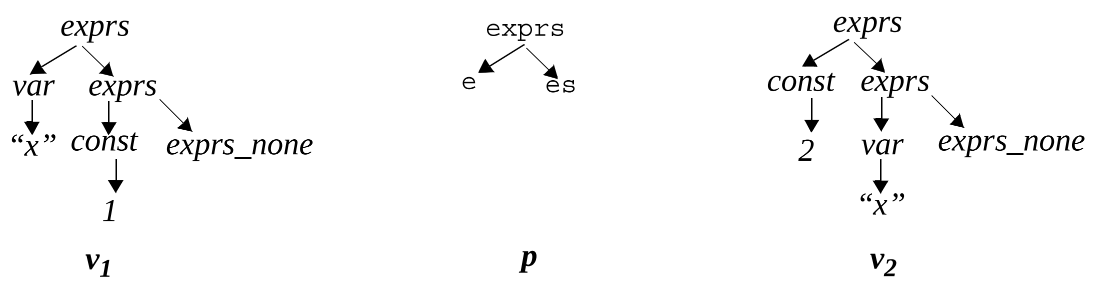
The function takes a brute-force generate-and-test approach. It iterates through values of *n* between the cost of pattern *p* and *max*. For each *n*, for each value *v* in *vs*, it generates

```
1  fun genMatches(m as |p=>_|:match , vs:value list, n:int, max:int, msSoFar:match list):(match list) =
2  if n > max then msSoFar @ [m]
3  else
4  let ps          = generatePrefixPatterns(n, v, p)
5     (ps', vs')   = testPrefixPatterns(ps,vs)
6     ms           = makeMatchesFromPrefixPatterns(m, ps')
7     msSoFar'     = msSoFar @ ms
8  in if vs' = [] then msSoFar'
9     else genMatches(m, vs', n+1, max, msSoFar')
10 end
11
12 and generatePrefixPatterns(n: int, vs: value list, p: pattern): pattern list=
13 // for each value v in vs, generate the list ps of all size-n patterns at least as specific as p that prefix it.
14 //return the list derived from concatenating all such lists.
15
16 and testPrefixPatterns(ps, vs): (pattern list * value list) =
17 //initialize the returned-patterns-list ps' to [], and the returned-value list vs' to vs
18 //for each pattern p in ps, check if p matches some unique value v in vs
19 //     if so, add p to ps', and remove v from vs'
20 //return (ps', vs')
21
22 and makeMatchesFromPrefixPatterns(|p=>e|:match, ps:pattern list): match list =
23 //For each pattern p' in ps, create a match |p' => e|
24 //return the resulting list of matches
```

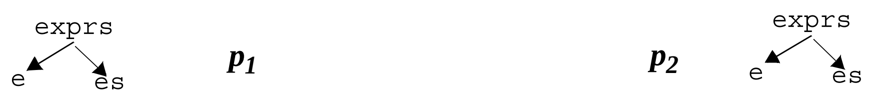
FIGURE 6.26: Computing Minimal Discriminating Matches Via Generate and Test.

all patterns of size *n* that result from merging *p* with some "prefix" of *v* (line 4 of figure 6.26), filters out those patterns that match with exactly one of the values in *vs* (line 5), converts these patterns into matches (line 6), and accumulates them (line 7) into the running result. Any values in *vs* that do find a unique match in line 5 are excluded from the next iteration. If this exclusion results in an empty *vs* set, the iteration terminates early and returns the accumulated matches (line 8), otherwise the iteration proceeds with the next higher value of *n* (line 9). If the maximum number of comparisons allowed is exceeded, the algorithm conservatively adds on the original match as a "catch-all" for any values that have not been matched by the matches generated in the previous iterations (line 2).

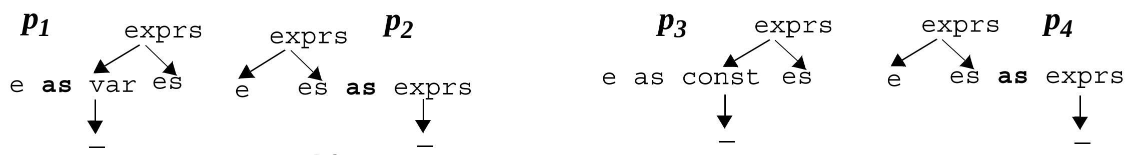
Figure 6.27 illustrates how the *genMatches* function works on the example of figure 6.19. Figure 6.27(a) specifies the pattern *p*, and the two abstract values v_1 and v_2 such that



(a) Argument Values



(b) Patterns Generated for *n = 5*



(c) Patterns Generated for *n = 7*

```
fun analyzeExprs(es:exprs, lSet:live_set):live_set=
  case es of
    exprs(e as var _,es)=> analyzeExprs(es,analyzeExpr(e))
  | exprs(e as const _,es)=> analyzeExprs(es,analyzeExpr(e))
```

(d) The `analyzeExprs` Function After Expression Specialization

FIGURE 6.27: Example: Computing Minimal Discriminating Matches.

vs = [v1,v2]. The values and pattern are illustrated using their parse trees. The cost of pattern *p* is 5 operations in this case.

Figure 6.27(b) shows the first patterns generated by the *generatePrefixPatterns* function: patterns p_1 and p_2 are generated corresponding to values v_1 and v_2 respectively. Each pattern consists of a prefix of the abstract value (in this case, just the root node *exprs*), merged with the incoming pattern *p*. Since p_1 and p_2 both match v_1 and v_2, the *testPrefixPatterns* function of figure 6.26, line 5 returns an empty set *ps'* of unique matching patterns. No new matches will be added to *msSoFar*, and the *genMatches* function proceeds iteratively with *n = 6*.

The next interesting case is for *n = 7*. As shown in patterns p_1 through p_4 of figure 6.27(c) using the additional 2 pattern-matching operations now available, SCF can now perform an additional tag equality test on the incoming trees. Because the left child of v_1 has tag *var*, the *generatePrefixPatterns* function adds a tag test to determine if the left child of the incoming value has tag `var`. The resulting pattern is p_1. Similarly, patterns p_2, p_3 and p_4 are based on v_2 and test for tags `exprs`, `const` and `exprs` respectively. The *testPrefixPatterns* function will now determine that patterns p_1 and p_3 match values v_1 and v_2 uniquely. It will therefore return the list *ps'* = *[*p_1, p_3*]* and *vs'* = *[]* (since no values are left to be matched). The *makeMatchesFromPrefixPatterns* will, given these patterns, produce the two matches of lines 3 and 4 of figure 6.27(d). At this point, since *vs'* is empty, *genMatches* and *discriminateMatch* will return.

The *discriminate* function will now call *discriminateMatch* with the remaining match *m* = `exprs_none => lSet` of figure 6.19(b) and an empty list *vs* of values to discriminate. Functions *genMatches, discriminateMatch* and *discriminate* will return at this point, the latter with the list of matches `[|(e` **`as`** `var _,es) => ...|, |(e` **`as`** `const _, es) => ...|]` as return value. Figure 6.27(d) shows the resulting function that the partial evaluator sees.

6.2.2.3 Discussion: Sacrificing Specialization Opportunities for Simplicity

The expression specialization strategy described in the previous section is designed to be simple while specializing optimization programs well. An appealing feature of SCF's specialization strategy is that it does not require analysis of the expression guarded by the matches being specialized (or worse, functions invoked by this expression). Below, we list some of the design issues that SCF sidesteps in the quest for simplicity:

1. **Maintaining correlations across repeated variable use.** On any single execution of an expression, when a given variable (or location, in general) is used multiple times in the expression, the concrete value of the variable at all these uses will be the same. In principle, we could perform an analysis to check if the current expression being analyzed performs this kind of re-use, and if so specialize the expression to the different concrete values the variable may assume. To minimize the size of the expression produced, we could replicate only the smallest sub-expression that

contained all uses. In SCF, however, we consider this case too uncommon to merit the extra complexity in analysis.

2. **More rigorous cost-benefit analysis.** Even when we restrict ourselves to specializing correlation-losing case expressions, the decision to specialize needs to be based on a cost-benefit analysis. Costs include both the overhead of the discriminating matches introduced and the overhead induced by the increased size of the program (due to replication). The main benefit is the increased efficiency of residual code. Ideally, we would specialize whenever the benefit exceeded the cost.

   In SCF, we make no effort to estimate total costs or benefit. In particular, although we estimate the increased cost of each pattern match, we make no effort to estimate the total extra costs across all new matches. We make no effort to estimate the cost of increase in code size or the benefit of better-specialized code that may result. Our philosophy is that if we keep the per-match overhead low enough, and only add matches when it prevents a loss in correlation, we are unlikely to lose. Further, estimates of costs and benefits have low accuracy, and very high cost: the size of the specialized code will depend on the values being specialized on. Further, each discriminating match results in replication not just of the expression guarded by the match, but potentially also of the functions called by the expression. In general, therefore, it is impossible to get an accurate estimate of code size without executing the partial evaluator in its general interprocedural mode. Doing so at every possible expression specialization site is clearly prohibitive. If more accurate estimation of costs and benefits emerges as an important issue, it may be worth studying more elaborate approaches from the partial evaluation community [45].

3. **Generating more efficient discriminating patterns.** The high-level description of figure 6.26 hides at least one simplifying design decision adopted by SCF. Since the *testPrefix* function generates discriminating patterns with strictly increasing *n*, and no provision for backtracking, SCF loses opportunities for efficiency. Consider specializing pattern `c1  _` to abstract value *v* = *c1(c2,c2)* | *c1(c2,c3)*| *c1(c3,c2)*| *c1(c3,c3)*. The SCF algorithm would discard the *n=5* patterns `c1(c2,_)`,

`c1(_,c2),c1(c3,_),c1(_,c3)` because each of these match at least two of the component values of v above, and would therefore fail the *testPrefixPattern* test of figure 6.26. The algorithm would then proceed to $n=7$ and suggest the discriminating patterns `c1(c2,c2)`, `c1(c2,c3)`, `c1(c3,c2)`, `c1(c3,c3)`. Unfortunately, the optimal mix of discriminators combines the $n=7$ patterns with the $n=5$ ones: `c1(c2,c2),c1(c2,_),c1(_,c2),c1(_,c3)`.

6.3 Widening Strategy

In section 6.1.5, we showed how the infinitely tall lattice of abstract values used by the partial evaluator in SCF can lead to an unbounded number of re-analyses of functions. We sketched how a *widening strategy* can guarantee termination by lowering precision of abstract values, but pointed out the importance of limiting the amount of precision sacrificed.

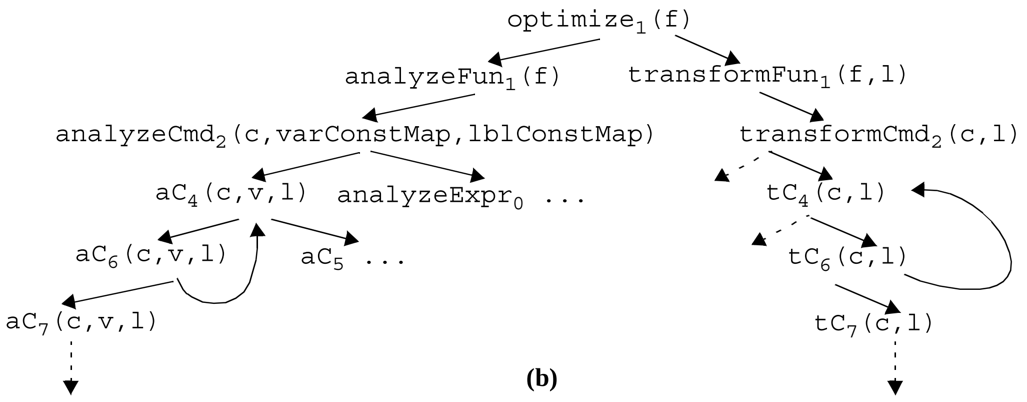
In particular, if v_1, v_2, v_3, ... is the sequence of (argument or return) abstract values corresponding to invocations of the partial evaluator on the function, the ideal widening strategy would be one that, in some finite number of steps, can predict the limit of this sequence, i.e., can compute $v_\infty = lim_{i \to \infty} v_i$. This problem is undecidable in general: it amounts to predicting, for an arbitrary function, the set of values a variable may assume over a program execution. A practical widening strategy therefore restricts itself to computing, after some finite number of steps, an approximation $v_{>\infty}$ of v_∞ such that $\underline{v}_{>\infty} \supseteq \underline{v}_\infty$, while trying to ensure that $\underline{v}_{>\infty}$ is not “too big”, i.e., that it does not include key elements missing from $\underline{v}_\infty$.

A widening strategy needs to address two questions. When should the abstract store at a given program point be widened? Given the decision to widen, what should the widened value be?

In practice, the question of when to widen and what value to widen to depends strongly on the program to be abstractly interpreted. In SCF, of course, we are interested in abstractly interpreting optimization programs. At any function call, the state of optimization programs is determined by three main data structures: the AST node being optimized, the map (or set) representing the abstract store, and the AST node being generated (for

$\mathit{int\ foo(int\ y)}\{$
$\mathit{int\ x} =_3 -1 +_0 1;_2$
$\mathit{fix}_4\ (\{\}_5 \mid x =_7 x +_8 1;_6\ \mathit{rec})$
$\mathit{return}_9\ x +_{10} y\}_1$

(a)



(b)

FIGURE 6.28: Motivating Example for Widening.
An abstract function (a) and constant propagation specialized to that function (b)

transformation functions). As discussed in the previous section, we expect each function to be invoked on a finite number of incoming AST nodes, and we intend to analyze the function separately for each of these nodes. We should not therefore have to widen the value of the AST node being analyzed (and indeed, of any finite argument). SCF's heuristics therefore focus on when and how to widen maps and generated AST's so as to avoid excessive loss of information. Other kinds of values are widened conservatively.

In section 6.3.1, we introduce an example that shows the value of intelligently widening input maps and output AST nodes. In section 6.3.2, we use the example to motivate our widening strategy, with a detailed look at how to handle maps.

6.3.1 Motivating Example: Widening and Constant Propagation

To understand the nuances of widening input maps and output AST nodes intelligently, consider partially evaluating the constant propagation optimization on the abstract function of figure 6.28(a). We label with subscripts the abstract values representing the AST nodes of interest to us. The abstract function represents the set of all concrete functions that have the assignment `x = -1 + 1` as the first statement, followed by zero or more

Invocation Number	v	l	v′	l′
1	*(must[x->0], may[])*	*(must[0->0], may[])*	*(must[x->ϕ], may[])*	*(must[0->0], may[])*
2	*(must[x->0|1], may[])*	*(must[0->0], may[8->1])*	*(must[x->0|1], may[])*	*(must[0->0], may[8->1])*
3	*(must[x->0|1|2], may[])*	*(must[0->0], may[8->1|2])*	*(must[x->0|1|2], may[])*	*(must[0->0], may[8->1|2])*
4	*(must[x->0|1|2|3], may[])*	*(must[0->0], may[8->1|2|3])*	*(must[x->0|1|2|3], may[])*	*(must[0->0], may[8->1|2|3])*
...				

FIGURE 6.29: Argument and Return Values Produced While Evaluating `analyzeCmd`$_4$.

increments of `x`, followed by the return statement `x + y`. Figure 6.28(b) shows the callgraph for the version of constant propagation specialized to this input function. Note, as before that we have essentially one contour per incoming AST node. In particular, the *fix* value in the incoming AST results in the recursive calls from `aC`$_6$ to `aC`$_4$, and from `tC`$_6$ to `tC`$_4$.

Every function in the callgraph has as first parameter the node being processed. The node parameter is followed in some cases by map `varConstMap` (abbreviated as `v`), which is a variables-to-constants map representing the values of all variables defined before the current AST node. Whenever the constant propagation analysis determines that a variable *x* has constant value *k*, it adds binding *x*->*k* to `varConstMap`. The final parameter in each case is the map `lblConstMap` (abbreviated as `l`) which is a map from AST node labels to their constant value. Whenever the constant propagation analysis detects an expression labelled *l* that can be replaced by a constant *k*, it adds the binding *l*->*k* to `lblConstMap`.

Once the analysis is complete, the optimization uses the `transform` functions to transform the incoming AST, while consulting the `lblConstMap` built up by the analysis. Whenever the constant propagation transformation detects a node labelled *l* such that *l* maps to constant *k* in `lblConstMap`, it replaces the node with the constant expression *k*.

We now study how SCF handles this example, focusing mainly on the maps produced when processing the *fix* value of figure 6.28.

Consider partially evaluating the contour `analyzeCmd`$_4$ of figure 6.28(b) with the appropriate variable-to-constants map (`v`) and label-to-constant map (`l`). Recall that the `analyzeCmd` function takes as its arguments the command `c` to be analyzed along with `l` and `v`, and returns a modified variable-to-constants map `v'` and a modified label-to-constants map `l'`. Figure 6.29 shows a possible sequence of invocations of the function, and the input and output values for each invocation.

On the first invocation of `analyzeCmd`$_4$, map `v` contains just *x->0* (from processing assignment $x =_3 -1 +_0 1$), and map `l` has *0->0* (since the expression $-1 +_0 1$, with label *0*, can be replaced by the constant expression `0`). The invocation results in the analysis of the two sub-commands, $x =_7 x +_8 1;_6$ fix_4 $(\{\}_5 \mid x =_7 x +_8 1;_6$ $rec)$[1] and $\{\}_5$:

1. Processing sub-command 6 results in sequentially processing command 7 and then command 4. Processing command 7 results in *x* being re-bound to the constant *1* in `v` and label *8* being bound to *1* (indicating that the expression $x +_8 1$ can be replaced by the constant expression *1*) in `l`. Processing command 4 results in a recursive call to `analyzeCmd`$_4$ being placed on the worklist with values *(must[x->1], may[])* and *(must[0->0, 8->1], may[])* respectively for arguments `v` and `l`. A cache lookup returns a value of ***0*** for the return value of the recursive call and therefore for command 6 as a whole.
2. Processing sub-command 5 results in the two maps passing through unchanged.

Meeting the result values from the two sides, the final returned value `v'` of `v` is *meet((must[x->1], may[]),* ***0****) = (must[x->1], may[])*, and `l'` is *meet((must[0->0], may[]),* ***0****) = (must[0->0], may[])*.

On the second invocation (invocation 2 of figure 6.29) of `analyzeCmd`$_4$, the partial evaluator pops the above argument values (`v` *= (must[x->1], may[]),* `l` *= (must[0->0, 8->1], may[])*) from the worklist, and meets it with the old values (*(must[x->0], may[]), (must [0->0], may[])*) from the previous invocation, to get the new abstract input values

1.This command is obtained by "unrolling" the original command, fix_4 $(\{\}_5 \mid x =_7 x +_8 1;_6$ $rec)$, once.

(`v` = *(must[x->0|1], may[])*, `l` = *(must[0->0], may[8->1]*). The new invocation results in a fresh recursive call to $\texttt{analyzeCmd}_4$ with argument `v` = *(must[x->1|2], may[])*, `l` = *(must[0->0], may[8->1|2])* pushed on the worklist, leading eventually to invocation 3. Figure 6.29 lists the argument and return values for the first four invocations in the unbounded series of invocations that follows.

The example highlights the two key requirements for a widening strategy for SCF. First, the strategy needs to widen indvidual fields of tuples. Second, maps need to be widened carefully. In particular, maps tend to grow in values not in keys, so that widening growing *alt* abstract values to ***1*** will often affect only individual values in the maps, and not the keys.

6.3.2 Reducing Information Loss While Widening

A simple strategy the partial evaluator can adopt to avoid unbounded analysis of a contour is to keep track of the number of times the contour has been analyzed and, if this number exceeds a small fixed threshold, to widen both input and result values of the contour to the abstract value ***1***.

"Maximal-widening" strategies such as the one above are extremely conservative. In the particular context of our examples, if widening happens when analyzing some command *c*, both result maps `l` and `v` will be widened to ***1***, so the analysis of all downstream commands could have both input and output values of these maps be ***1***. Worse, since the transformation function is typically downstream of the analysis function, partial evaluation of the transformation function may have to proceed under the worst-case assumption that map `l` has value ***1***.

A standard strategy in this situation is to perform "sufficient widening" instead of maximal widening. The value to widen to is typically picked using a heuristic parameterized by the sequence of values seen so far for arguments and results. Intuitively, these strategies observe the sequence of values so far and tries to guess their lower bound. Ideally, such a strategy would guess the greatest lower bound (g.l.b.) of the sequence. Computing the g.l.b. is, in general, undecidable. Most sufficient-widening strategies, therefore, are iterative: after an initial number n_1 of steps, they guess a lower bound based on the values seen so far. If, after some number $n_2 > n_1$ of steps, the series of values has not converged, they guess another one, and so on. Finally, for some bounded integer *k*, if the series has not converged after *k* steps, they revert to the maximal widening strategy.

6.3.3 Widening in SCF

Figure 6.30 shows how the widening strategy is implemented in SCF. In particular, SCF uses *k=2*, and n_1 and n_2 are fixed before the partial evaluator executes; the typical values used by SCF are *10* and *11* respectively (lines 4 and 5 of the figure).

The widening function, *widen* (lines 7-12), takes as argument the abstract value *v* to be widened, the number *n* of times the value has been widened (including the current instance), and the finiteness pattern *fp* for the value. If the maximal widening threshold has been reached, if the incoming value is already ***1***, no widening is performed (line 9), otherwise, the result returned is the maximally widened value ***1***. If only the sufficient widening threshold is reached, but not the maximal threshold, then the function *widen'* is invoked (line 10) to achieve sufficient widening of non-finite components. Below the sufficient-widening threshold, no widening is performed (line 12).

The *widen'* function essentially traverses the abstract value and the corresponding finiteness pattern, widening each sub-value with respect to each sub-pattern.

- The base case for widening is the replacement of infinite *alt* and *fix* values with ***1*** (line 29-30). For instance, given non-finite value *2|3*, the result of widening would be ***1***.
- On the other hand, if an abstract value is known to be finite, then it is not widened, for reasons discussed above (lines 14-15).
- Widening of tuple values whose finiteness is represented by a tuple finiteness-pattern happens component-wise (lines 21-24, and 48-49). For example, widening *(12|78, 2|3)* given finiteness pattern *(F, I)* will result in abstract value *(12|78, **1**)*. If the pattern were *(I, I)*, the result would be *(**1**,**1**)*.
- Widening of infinite tuples results in each component of the tuple value being assumed infinite, and therefore widened with respect to *I* (lines 17-20). *List.enumerate(i, x)*

```
structure Widening = struct
structure FA = FinitenessAnalysis
type widening_result = WIDENED of value | UNWIDENED
val n1 = ref 10
val n2 = ref 11

fun widen(v: value, n: int, fp: FA.FinitenessPattern): widening_result =
  if n >= !n2 then                              //maximal widening
    if v = bottom _ then UNWIDENED
    else                  WIDENED (bottom (newId()))
  else if n >= !n1 then widen' (v, fp)          //sufficient widening
  else if n < !n1   then UNWIDENED              //no widening yet

and widen'(v: value, FA.F: FA.FinitenessPattern) =
                                      UNWIDENED
| widen'(bottom _: value, _)      =   UNWIDENED
| widen'(tuple(vs, _), fp as FA.I) =
  case (widenTuple (vs, List.enumerate (List.length vs, fp))) of
    SOME vs                     =>  WIDENED vs
  | NONE                        =>  UNWIDENED
| widen'(tuple(vs, _), |(fps)|) =
  case (widenTuple (vs, fps)) of
    SOME vs                     =>  WIDENED (tuple(vs, newId()))
  | NONE                        =>  UNWIDENED
| widen'(tagVal(c,v,_), _) =
  case widen' (v,FA.I) of
    WIDENED v'                  =>  WIDENED tagVal(c,v',newId())
  | _                           =>  UNWIDENED
| widen'(alt(vs,_), _)          =   WIDENED (bottom(newId()))
| widen'(fix(v,_), fp)          =   WIDENED (bottom(newId()))
| widen' (map(must uus, may vvs, _) =
  case (widenMap uus, widenMap vvs) of
    (NONE, NONE)                =>  UNWIDENED
  | (SOME uus', NONE)           =>  WIDENED(map(must uus', may vvs, newId()))
  | (NONE, SOME vvs')           =>  WIDENED(map(must uus, may vvs', newId()))
  | (SOME uus', SOME vvs')      =>  WIDENED(map(must uus', may vvs', newId()))
| widen' (v, _)                 =   UNWIDENED

and widenMap (vvs: (value * value) list): (value * value) list option =
let uus        = //replace each (v,v') in vvs with (u, u'), where u and u' result from widening v and v'
   isWidened= //true if any u, u' above actually required widening
   uus'       = //sort tuples (u,u') in uus' by u;
                 //replace maximal sub-sequences (u,u'_1),...,(u,u'_n) with (u, meet u'_1,...,u'_n)
in if isWidened then SOME uus'
  else NONE
end

and widenTuple(vs:value list, fps: FA.FinitenessPattern list): (value list) option =
  ... widen v_i w.r.t. fp_i to get u_i; if no u_i is widened, return NONE else SOME us ...
end //structure Widening
```

FIGURE 6.30: Widening in SCF.

constructs a list of length *i* with all elements set to *x*. Widening *(12|78, 2|3)* results in value ***(1*, *1)***.

- Since finiteness patterns do not currently model the internals of tagged values in detail, any time we widen an infinite tagged value, all we know is that the entire value may be infinite (i.e., has finiteness pattern *I*). We therefore widen all sub-components of the value assuming they are infinite (line 26).
- Maps consist of two sorted lists of pairs of abstract values. Widening a map (lines 31-36) consists of traversing the lists, widening each abstract value in a pair. We then re-sort the resulting lists by the domain values of the pairs. Any time a list thus sorted has a series of values *(u, u_1')*, ... , *(u, u_n')*, i.e., pairs with duplicate keys, we replace the whole set with the single pair *(u, meet(u_1',...,u_n'))*.
- In every case where widening of sub-components is attempted, if no sub-component actually needs widening, the entire value is deemed not to need widening (lines 20, 24, 28, 33).

When applied to the example of figure 6.29, the widening strategy avoids the need for an unbounded number of rows. In particular, invocation 10 will trigger widening and produce the following values for *(v, l, v', l')*: *(must[x->**1**], (must[0->0], may[8->**1**]), must[x->**1**], (must[0->0], may[8->**1**]))*. Invocation 11 will again result in widening being triggered, and the same maps being generated. Since the old and new values are the same, the partial evaluator can avoid re-analysis of the contour (line 36 of figure 4.4), and therefore any of the callee contours. Partial evaluation will therefore terminate with the above relatively accurate widened values of the map.

6.4 Rematerialization Strategy

The central benefit of partial evaluation comes from replacing expressions that compute constant values with simpler residual expressions that compute the same value, much like a traditional constant propagation optimization. In SCF, the decision of when to perform the replacement and what expression to replace with is made by the rematerialization strategy.

```
signature REMATERIALIZATION_STRATEGY = sig
val reset          :unit -> unit

val postProcess    :SCF_ML.program -> SCF_ML.program

val rematerialize  :SCF_ML.expr * AbstractValue.value -> SCF_ML.expr
end
```

FIGURE 6.31: Signature Required of the Rematerialization Strategy Module.

All rematerialization strategies used in SCF conform to the *REMATERIALIZATION_STRATEGY* signature of figure 6.31. The signature requires three functions:

1. A *reset* function invoked before partial evaluation commences (see line 10 of figure 4.4 to understand precisely where SCF invokes it), which lets the conforming module reset data structures that need initialization before the partial evaluator is used.
2. A *postProcess* function that takes as input the program resulting from partial evaluation and returns the program that results from performing any rematerialization-related post-processing on this program. This function is invoked at line 17 of figure 4.4.
3. A *rematerialize* function that takes as input an expression and an abstract value for that expression, and returns a residual version of the expression.

SCF supports two rematerialization strategies. The simpler and more conventional one (figure 6.33(a)), which we call *scalar rematerialization*, is identical to that used by the traditional constant propagation optimization. Any expression that evaluates to a singleton scalar (i.e., integers, characters, booleans and strings; in general any value that can be generated by a literal expression in SCF-ML) is replaced by the SCF-ML literal that evaluates to that scalar (lines 3-7 of the figure). For instance, given expression `x+1` and abstract value *Int 17*, the rematerialized expression will simply be the literal `17`.

SCF also provides a more aggressive rematerialization strategy that allows rematerialization of non-scalars. The complication with rematerialization of non-scalars is that it is possible for the rematerialized expression to be more expensive to compute than the original version. Consider, for instance, partially evaluating expression *x* in an environment where *x* is bound to the non-scalar singleton abstract value *cons(1, cons(3, cons(232,*

```
structure RematStrategy = struct
(* Replace singleton scalars by the literal expressions that generate them *)
fun rematerialize(e: SCF_ML.expr, v: AbstractValue.value): SCF_ML.expr =
   if AbstractValue.isSingleton v andalso AbstractValue.isScalar v then genLiteral v else e

and genLiteral (v: value):SCF_ML.expr =
   ... for the single concrete value s in v, return the SCF_ML literal that evaluates to s...

and reset() = ()

and postProcess(p: SCF_ML.program): SCF_ML.program= p

end //structure RematStrategy
```

FIGURE 6.32: Scalar Rematerialization.

nil))). If `x` is replaced by the residual expression `cons(1, cons(3, cons(232,nil)))`, the replacement would have the effect of slowing the program down. On the other hand, we saw in section 6.1.3 that rematerializing non-scalars is sometimes profitable.

The challenge in rematerialization of non-scalars is therefore to find a replacement expression that is guaranteed to cost less than the original one to compute. SCF has a simple technique that enables it to make this guarantee for any expression *e* that has a singleton constant value *v*. The technique is outlined in figure 6.33. Let e_v be the constant expression that evaluates to *v*. Then SCF replaces *e* with a variable (of the form `__rematVar__`*`i`* where *i* is an integer), and inserts the binding `__rematVar__`*`i`* `=` e_v at the top level of the program. We call the inserted variables *rematerialization variables*. In the actual implementation, SCF generates the variable (line 6 of the figure), and stores the binding between the variable and the residual expression in a rematerialization map (lines 7 and 8). When the *postProcess()* function is called on the program (after partial evaluation is completed), SCF examines the body of the program to determine all rematerialization variables actually referenced, and prepends the program with bindings between these variables and the corresponding rematerialization expressions. When a rematerializable expression is a sub-expression of a larger rematerializable expression, the rematerialization variable for the former is subsumed by the one for the latter. SCF's check to see if rematerialization variables are actually referenced is designed to avoid adding bindings to such variables.

Figure 6.34 shows how SCF-style non-scalar rematerialization could work when the dead assignment elimination optimization of figure 3.7 is partially evaluated. Consider the case where the function being optimized has the compound statement $\texttt{x =}_{21}\ \texttt{y + z;}_{22}$ $\texttt{return}_{23}$ `x`. Figure 6.34(a) shows the transformation functions corresponding to each of the commands (function `transformCmd_21` is responsible for transforming the command labeled 21 and so on), in the absence of non-scalar rematerialization. Since command 21 (the assignment to `x`) is live in the above fragment (since the succeeding command uses `x`), we expect `transformCmd_21` to return the command unchanged (line 12, figure 6.34(a)). However, since the returned value is not a scalar, the scalar rematerialization strategy cannot rematerialize away the function call, even though the returned value has a constant value.[1] When non-scalar rematerialization is enabled, the function

```
structure RematStrategy = struct
val rematMap = ref VarNameToExpMap.empty : (VarNameToExpMap.map ref)

fun rematerialize(e: SCF_ML.expr, v: AbstractValue.value): SCF_ML.expr =
  if AbstractValue.isSingleton v then
    let vName      =   genNewVarName()      //vars have unique prefix __rematVar__
        exp        =   (* build an expression that generates value v*)
        rematMap   :=  (!rematMap.insert(vName,exp)
    in SCF_ML.var(vName) end
  else (e,v)

and reset() = (rematMap : = VarNameToExpMap.empty)

and postProcess(p: program) =
  ...for each module m in p
    for each variable v in m of the form __rematVar__*
      let SOME exp = VarNameToExpMap.find(rematMap,v)
      in prepend the binding val v = exp to m end...

end //structure RematStrategy
```

FIGURE 6.33: Non-Scalar Rematerialization.

1.In this particular case, the call can be eliminated by a conventional optimization, i.e., inlining, but this is generally not the case.

```
1  structure DAE = struct
2  ...
3
4  fun optimize(f:fun):fun =
5    transformFun(f, analyzeFun(f))
6  ...
7  and transformCmd_22(c:cmd, assigns:assign_map):cmd =
8  ...
9    | seq(c, c’, lbl) =>
10      seq(transformCmd_21(c,aMap),transformCmd_23(c’,aMap),lbl)
11 ...
12 and transformCmd_21(c, a) = c
13 ...
14 end (* structure DAE *)
```

(a)

```
1  structure DAE = struct
2  ...
3  val __rematVar__732 =
4    assign(var(“x”, label 15),
5           primop(op_add,exprs(var(“y”, label 16),
6                         exprs(var(“z”, label 17),
7                         expr_none(label 18), label 19)), label 20), label 21)
8
9  fun optimize(f:fun):fun =
10   transformFun(f, analyzeFun(f))
11 ...
12 and transformCmd_22(c:cmd, assigns:assign_map):cmd =
13 ...
14   | seq(c, c’, lbl) =>
15      seq(__rematVar__732,transformCmd_23(c’,aMap),lbl)
16 ...
17 end (* structure DAE*)
```

(b)

FIGURE 6.34: Example: SCF-Style Non-Scalar Rematerialization.

(a) Result of partial evaluation with scalar rematerialization (b) Result of partial evaluation with non-scalar rematerialization

call can be replaced by a rematerialization variable (`__rematVar__732` on line 15 of figure 6.34(b)), provided the variable is initialized with the appropriate constant value (lines 4-7 of the figure).

Since the non-scalar rematerialization strategy replaces each eligible expression with a single variable, it may seem that the strategy guarantees that the run-time overhead associated with the rematerialized expression is guaranteed to be no more than that of the original expression. However, if we take into account the cost of initializing the rematerialization variable, it is no longer clear that the cost of the rematerialized expression outweighs its benefit. Consider the example of figure 6.34. The original expression in this case is the call to function `transformCmd_21` on line 10 of figure (a). The cost associated with the function call is essentially that of jumping to the location of the function and returning from the location (since the function just returns one of its arguments). With a naive implementation of SCF-ML, the result of rematerialization (figure (b)), on the other hand, has to pay the cost of executing the initializer expression of lines 4-7 at module load time. Note that this expression is substantially more complicated than a jump-and-return sequence. If we associate this cost with the rematerialization variable, therefore, it is possible that the rematerialized version costs more at run-time than the original version.

A solution to this problem is to implement the SCF-ML language so that all constant expressions are evaluated at compile time, and generated into the text segment of the program. The only significant cost at module load time, therefore, is to page in the text segment from disk, and this can be done in parallel with the execution of the program as a whole. David Tarditi's dissertation [67] discusses in detail how to implement this optimization in a compiler for Standard ML. In this work, we do not actually implement this optimization. Rather, when evaluating the performance of non-scalar rematerialization, we simulate this effect by executing each module we load a large number of times, so that the one-time module-load-time cost is insignificant relative to execution cost.

6.5 Summary and Related Work

We have described a set of techniques, which we collectively call improvement strategies, that are essential for effective partial evaluation of optimization programs in SCF. For each technique, we give examples to show why the technique is necessary for the task of partially evaluating optimizations, and describe algorithms that are suitable for the task. The strategies discussed in this chapter are instances of corresponding techniques that have long been advocated for improving partial evaluation (and program analysis in general). Below, we describe how the techniques of this section relate to existing work.

6.5.1 Specialization

Specialization, i.e., the replication of code (whether at the function granularity, which we call “function specialization”, or at the expression granularity, which we call “expression specialization”) for different analysis contexts, is a standard technique in partial evaluation [30, 56], whole program optimization in general [60, 25, 20] and intraprocedural optimization [41, 19].

When the domain of analysis is unbounded in size, a known problem with function specialization is termination: the number of different analysis contexts may not be bounded at analysis time, so that the analysis may seek to generate and analyze an unbounded number of code replicas. The standard solution to this problem is to either model the domain of the analysis so that it is finite, or to bound a priori the number of contexts the analysis is allowed to consider. In partial evaluation, it is often useful to consider (at least) the typically infinite domain of concrete values with full precision. Further, typical applications (such as specializing interpreters) depend in practice on producing a number of specialized replicas for which no a priori bound is known. The standard techniques mentioned above are therefore insufficient.

Finiteness analysis [28, 23, 37, 36], which identifies “finite” function parameters, i.e., those that are guaranteed to take on a finite set of values during concrete execution, is intended to handle both infinite domains and the lack of a priori bounds on number of specializations. The underlying idea behind this technique is that if there exists some well-founded partial order on the domain of values over which the programs in a language execute, and some “reducing” operations O in the language that are guaranteed to produce outputs that are less than their inputs as per this order, then any parameter that is computed from the program input purely by a chain of reducing operations is finite. It is impossible for the parameter to be computed by an infinite sequence of reducing operations since the corresponding sequence of values produces by the operations is bounded from below.

The finiteness analysis of SCF is strongly inspired by those mentioned above, with the key difference that it is aimed at detecting parameters finite under *abstract* execution rather than concrete execution. The above finiteness analyses are aimed at offline partial evaluation where the fully concrete values of one or more parameters of the program are known at specialization time; partial evaluation consists essentially of fully concrete execution based on just these inputs. In SCF however, the domain of values we have to handle is the abstract value domain (including, in particular, *fix* values), and the potentially infinite chains of operations we are interested in are abstract projection operations. However, because applying a sequence of abstract projection operations to *fix* abstract values can produce the original *fix* value, we cannot use the fact that the operations strictly “reduce” their input values. Instead we use the requirement that if *v* is the input value to the chain of operations, all operations in the chain must produce values that are in a finite set derivable from *v*.

Expression specialization, which replicates a program point for different statically computed environments, has a long history as “the trick” for enabling polyvariant specialization under dynamic control in partial evaluation [30, 16, 24], and as a mechanism for converting higher-order calls to functions to first-order calls (or inlined expressions) in functional [55, 44, 61] and object-oriented [10, 18] languages. SCF shares the general emphasis of most of these systems on obtaining as precise a representation as possible for the set of possible environments at a particular program point, and on ensuring that the tests inserted to distinguish between concrete contexts are not too cumbersome. However, the abstract domain used by SCF is considerably more expressive than those used by these systems, so deciding on the abstract environments to split into is potentially a more complex task. Further, the concrete values to be distinguished have a different structure (they are arbitrary ML values, as opposed to tuples of class variables, for instance), so the patterns inserted to distinguish between them have to be computed differently (they are ML patterns, as opposed to sets of class-variable tests).

6.5.2 Widening

Widening of abstract stores has long [54, 34, 15] been recognized as necessary for termination in the presence of infinitely tall lattices. However, the particular technique used has depended on the domain of the abstract analysis, and on the particular aspects of the domain to be represented with maximum accuracy. Representing sets of maps precisely is especially important in the abstract analysis of compilers. SCF takes special care, therefore, to preserve accuracy in maps while widening.

6.5.3 Rematerialization

The non-scalar rematerialization problem, i.e., that of deciding where to place code computing constant values without increasing the running-time of the program being specialized, is a special case of the problem of placing residual expressions to eliminate redundancy. In general, the problem may be formulated as that of performing partial redundancy elimination [35, 9] on the residual code produced by the partial evaluator.

Ruf [56], for instance, describes a two-step online partial evaluation scheme, where a program is first partially evaluated with the residual code represented as *dataflow graphs* which use multiple references to a single graph node to represent common sub-expressions. A separate code-generation step then generates code for the nodes at a point dominating their use in order to avoid redundant recomputation. Since SCF only rematerializes constant computations, it is safe (and simple) to hoist these computations to the top level of the program. Hoisting is potentially inefficient, since many of these code fragments may not need to be computed in a given run of the residual program. However, the trick borrowed from Tarditi [67] of “compiling” the constant fragments into the text segment allows SCF to avoid the direct cost of unnecessary computation (although if the text segment gets too large, there may be indirect cost in terms of unfavorable memory usage patterns).

7. Dead-Store Elimination

As discussed in section 3.3, the residualized code produced by partial evaluation contains many store operations that are dead, i.e., some parts of data structures built up in the residual code are not actually used. SCF contains an optimization, called dead-store elimination (DSE), to delete these operations. In this chapter, we describe how this optimization works.

DSE may be viewed as a generalization of traditional dead-assignment elimination, which typically eliminates assignments to variables. In addition to avoiding assignments to dead variables, DSE also eliminates stores to fields of non-scalar, possibly recursive, data structures. Previous work [40, 53] has investigated how to generalize dead-assignment elimination in this manner, and SCF adapts some of these techniques, in particular the representation of the abstract stores using a form of context-free grammars called *liveness patterns*. The novelty of DSE is its special machinery to avoid a particularly common and expensive class of store operations that pervade partially evaluated optimization programs: insertions to the map data structures that represent the abstract store used by the optimization. Previous work has discussed general techniques to eliminate dead stores into recursive structures, but these techniques prove to be ineffective in handling this important class of stores in practice.

To help understand the problem with stores to maps, figure 7.1 below reproduces the example from section 3.3 that shows a partially evaluated version of dead-assignment elimination[1]. As previously discussed, partial evaluation results in specialized versions of

```
...
and analyzeCmd3 (c, lSet, aMap) =
case c of
  assign(v, e, l)      =>
     let val lv = live
     in (analyzeExpr(e, LiveSet.delete(lSet, v)),
          AssignMap.insert(aMap, 3, live))
     end
...

and transformCmd3 (c, aMap) =
  case c of
    assign(v, e, lbl) =>
      c
```

FIGURE 7.1: Analysis and Transformation Functions After Partial Evaluation.

the `analyzeCmd` and `transformCmd` functions. In principle, the specialized analysis function `analyzeCmd3` analyzes a command and updates the part of the abstract store (represented by the map `aMap`) related to this command. The specialized transformation function `transformCmd3` reads this same part of the abstract store and produces a transformed version of the command. In figure 7.1, however `transformCmd3` clearly does not read the abstract store map `aMap` even though the `analyzeCmd3` function writes the map (on line 7). This is a common situation, and one we seek to exploit: the part of the transformation function that reads the abstract store is folded away by the partial evaluator because the read is guaranteed to return a single fixed value (the value `live` in this case). The corresponding write to the abstract map is therefore unnecessary, and can be eliminated. Since updates of the abstract store are typically the most expensive part of what an optimization does, eliminating a significant number of such writes can result in big savings.

The eliminator is structured as a backwards whole-program abstract interpretation. In section 7.1 below, we discuss the structure and meaning of *liveness patterns*, the domain over which the abstract interpretation takes place. In section 7.2, we present the analysis itself. Section 7.3 discusses related work.

7.1 Liveness Patterns: The Domain of Abstract Interpretation

Traditional dead-assignment elimination (DAE) analyses abstract the store as a mapping from variables defined in the program to liveness values, i.e., either *live* or *dead*. If the abstract store maps variable *x* to value *dead* at a particular program point, the implication is that for all concrete traces of the program, the store location associated with variable *x* is guaranteed not to be read downstream. A value of *live* would imply that the store location may be read downstream. Traditional implementations of DAE represent the mapping from variables to liveness values as a set of *live variables*: a variable is in the live-variables set if and only if it would map to the value *live* in the abstract store map.

Traditional versions of DAE do not seek to model non-scalar values with much accuracy. In SCF, however, we are quite interested in avoiding stores to fields of non-scalars.

1.Note that we are applying partial evaluation and dead-store elimination to the dead-assignment elimination optimization, i.e., the program being optimized is itself an optimizer.

```
1 fun foo x0=
2 let  x1  = x0+1
3      x2  = x0*10
4      x3  = x0-2
5      x4  = (x1, x2)
6 in case x4 of (x5, _) =>
7    x5
8 end
```

(a)

```
1
2{x0}
3{x0, x1}
4{x0, x1, x2}
5{x1, x2}
6{x4}
7{x5}
8{return_val}
```

```
1 fun foo x0=
2 let  x1 = x0+1
3      x2 = x0*10
4      ~~x3 = x0-2~~
5      x4 = (x1, x2)
6 in case x4 of (x5, _) =>
7    x5
8 end
```

(b)

```
1
2{x0}
3{x0, x1}
4{x0, x1}
5{x1}
6{x4$^{-1}$}
7{x5}
8{return_val}
```

```
1 fun foo x0=
2 let  x1 = x0+1
3      ~~x2 = x0*10~~
4      ~~x3 = x0-2~~
5      x4 = (x1, 0)
6 in case x4 of (x5, _) =>
7    x5
8 end
```

(c)

FIGURE 7.2: Scalar vs. Non-Scalar Dead-Store Elimination.
(a) Candidate for dead-store elimination (b) Applying dead assignment elimination to `foo`. Live variables (left) and optimized function (right) (c) Applying dead-store elimination to `foo`. Live fields (left) and optimized function (right).

In particular, as mentioned above, we wish to avoid map insert operations, and AST subtree composition operations. SCF therefore models fields of non-scalar values (and maps in particular) in much more detail than a traditional DAE. Just as DAE uses sets of live variables to represent the abstract store, DSE uses sets of live fields to provide a more detailed abstraction of the store.

Figure 7.2 illustrates the difference between the DSE and DAE. Figure 7.2(a) specifies a function `foo`, some sub-computations of which are dead. In particular, note that the binding of variable `x3` on line 4 is dead, since `x3` is not used downstream. As shown in figure 7.2(b), DAE is able to identify and eliminate this dead binding. On the left of the figure, we show the set of live variables computed (in the traditional backward pass) by

DAE at corresponding points in `foo`. For instance, the analysis computes that at the program point preceding line 5 of `foo`, `x1` and `x2` are the only live variables. The two variables are deemed live since they are fields of `x4`, which is used on line 6. DAE is unable to track the fact that only the first field of `x4` is used on line 6, so that the store of `x2` into the second field does not imply that `x2` is used downstream and therefore live. Figure 7.2(c), shows how a DSE optimization may represent and exploit the partial liveness of `x4`. The key difference in representation is that DSE reasons about the liveness of *fields of values represented by variables* as opposed to *variables*, and as shown on line 6, has a way to represent the fact that only the first field of `x4` is live (using notation `x4`$^{-1}$). The key benefit, of course, is that variable `x2` is now seen to be dead at line 5, so that its binding (on line 3) may be eliminated in addition to that of `x2` (on line 4).

SCF represents sets of live fields indirectly using a representation called *liveness patterns*. In what follows, we first introduce *field projectors*, a representation for individual fields of SCF-ML values, and then describe an interpretation of liveness patterns as sets of field projectors.

7.1.1 Field Projectors

In figure 7.2(c), we used (without formally defining) the notation `x4`$^{-1}$ to represent the first field of the tuple-value represented by `x4`. We now define precisely a notation for representing arbitrary fields of non-scalar values of interest to SCF.

Figure 7.3 specifies the syntax and semantics of field projectors, along with examples. As per figure 7.3(a), there are four kinds of fields we specify; as per figure 7.3(b), the field represented by a projector may be understood in terms of the result of applying the projector to an incoming concrete value. The identity field (line 1) of a value is the value itself. The projector $c^{-1}\ \iota$ (line 2) represents the *c*-variant of a value, with the tag-label stripped. For instance, if some variable has values *cons(23, nil)* and *nil*, the field represented by projector $cons^{-1}\ \iota$ will have values *(23, nil)* and undefined respectively. The projector ι^{-i} (line 3) represents the *i*'th field of a tuple value. For instance if a variable has value *(23, nil)*, the field ι^{-2} will have value *nil*. Finally, the projector $\iota[h]$ represents the map value corresponding to map key *h*. For instance, given variable with value *[("bar",*

```
1  f ∈ FieldProjector (Π)  ::=  ι        //Identity field
2                           |   $c^{-1}$ f   //Constructor field
3                           |   $f^{-i}$      //Tuple field
4                           |   f[h]     //Map field
5  h ∈ Herbrand Universe               //SCF-ML concrete values (sec 3.2.2.1)
```

(a)

```
1  ι h                                        = h       //Identity projection
2  ($c^{-1}$ f) (c h)                             = f h     //Project constructor bodies
3  ($f^{-i}$) ($h_1$,…,$h_i$,…,$h_n$)                     = f $h_i$    //Project tuple elements
4  (f[$h_i$]) [($h_1$,$h'_1$),…,($h_i$,$h'_i$)…,($h_n$,$h'_n$)]= f $h'_i$ //Project map elements
```

(b)

```
1  ($cons^{-1}$ ι)(cons(1, cons(2, nil))) = (1, cons(2, nil))
2  ($cons^{-1}$ $ι^{-2}$)(cons(1, cons(2, nil))) = cons(2, nil)
3  ($cons^{-1}$ ($cons^{-1}$ $ι^{-1}$)$^{-2}$)(cons(1, cons(2, nil))) = 2
4  ($const^{-1}$ ι[var "x"]) ([(var "x", const 1), (var "y", non-const)]) = 1
```

(c)

FIGURE 7.3: Syntax and Semantics of Field Projectors.
(a) Syntax (b) Semantics (c) Examples: Applying field projectors.

1), ("foo", 7)] (a map represented as an association list), the field *ι ["foo"]* will have value *7*.

Field projectors can be combined, i.e., fields are allowed to have their own fields. Figure 7.3(c) shows examples of compound field projectors. Line 4, for instance, shows a field projector that represents the first field of the map entry corresponding to key *var "x"*.

In what follows, we will sometimes abbreviate "field projector" as "field".

7.1.2 Liveness Patterns: Syntax and Semantics

In the previous section, we presented a representation for individual fields of SCF-ML values. DSE represents a *set* of fields at each program point, so that any store operation that writes to a field not in this set may be eliminated. SCF-ML uses a representation called *liveness patterns (LP's)* to represent sets of fields. We use this special representation for sets of fields (rather than simply placing field projectors in a conventional set data structure) because the LP's provide a more compact and readable alternative.

Figure 7.4(a) specifies the syntax of LP's. Figure 7.4(b) specifies the translation function τ that maps a LP to the set of field projectors represented by it. Finally, figure 7.4(c) shows some example LP's and the set of fields they represent.

```
1  l ∈ LivenessPattern::=       L                 //live pattern
2                            |  D                 //dead pattern
3                            |  ($l_1$,...,$l_n$)        //product pattern
4                            |  $c_1$ $l_1$ |...| $c_n$ $l_n$  //sum pattern
5                                  s.t. $c_i \neq c_j$ if $i \neq j$
6                            |  [v, l]            //map pattern
7
8  v ∈ AbsValue       //SCF-ML abstract values (sec 3.2.2.2)
9  c ∈ constructors = ConstructorName ∪ Int ∪ Bool ∪ String
```

(a)

```
   τ:LivenessPattern -> $2^{FieldProjector}$

1  τ L                          = Π
2  τ D                          = {}
3  τ ($l_1$,...,$l_i$,...,$l_n$)         = $\cup_{i=1...n}$ {$f_i^{-i}$| $l_i \neq$ D ∧ $f_i$ ∈ τ $l_i$}
4  τ ($c_1$ $l_1$|...|$c_i$ $l_i$|...|$c_n$ $l_n$)  = $\cup_{i=1...n}$ {$c_i^{-1}$ $f_i$ | $f_i$ ∈ {ι} ∪ τ $l_i$}
5  τ[v, l]                      = {f[h]| h ∈ $\underline{v}$ ∧ f ∈ {ι} ∪ τ l}
```

(b)

```
1  τ(D, D)                               = {}
2  τ(D, cons D)                          = {$(cons^{-1}\ ι)^{-2}$}
3  τ(D, nil)                             = { $(nil^{-1}\ ι)^{-2}$}
4  τ(D, unary D | binary D)              = { $(unary^{-1}\ ι)^{-2}$,$(binary^{-1}\ ι)^{-2}$}
5  τ(D, L)                               = {$f^{-2}$| f ∈ Π}
6  τ([Int 1| Int 21| Int 1332, const D])= {$const^{-1}$[Int i]|i ∈{1,21,1332}} ∪
7                                             {ι[Int i]|i ∈{1,21,1332}}
8  τ(uni(uni(uni D | bin D))))           = {$uni^{-1}$ $uni^{-1}$ $uni^{-1}$ ι,
9                                             $uni^{-1}$ $uni^{-1}$ $bin^{-1}$ ι}
```

(c)

FIGURE 7.4: Syntax and Semantics of Liveness Patterns (LP's).

(a) Syntax (b) Semantics translating from liveness patterns to sets of field projectors (c) Examples: Sets of field projectors represented by liveness patterns.

The LP *L* (short for Live) stands for the set *Π* of all possible field projectors (line 1 of figure 7.4(b)). If a value if designated live, then it and any of its fields may be read by a downstream computation. The LP *D* (short for Dead) stands for the empty set of field projectors (line 2): if a value is designated dead, it is guaranteed that none of its fields will be read downstream. The set of fields represented by a product liveness pattern (line 3) is the union of the set of fields represented by the components of the product.[1] The set of fields represented by a sum pattern (line 4) is the union of the fields represented by its summands. A detail is that if *c l* is a summand, the field $c^{-1}\ \iota$ is a field represented by the sum, regardless of whether *l* is dead: a downstream computation may check the tag on a tagged

1.In what follows, we write patterns of the form `c()` as `c`.

value without inspecting its components. Finally, a map pattern, which is a pair of an abstract value *v* representing the domain of the map and a LP *l* representing the range of the map, represents the set of fields corresponding to indexing incoming maps with keys from $\underline{v}$, and those derived from accessing the fields corresponding to *l* of the resulting values.

Figure 7.4(c) gives examples of liveness patterns, and the sets of fields they correspond to. A product pattern whose components are dead (line 1) represents the empty set of fields. A subtle point here is that we could make this pattern represent the set $\{\iota\}$, i.e., we may want to account for the case where although the fields of a tuple value are not read, the structure itself is significant. However, the typing discipline of ML makes it unnecessary to distinguish this case. If a value has a product type, then the number of fields and their types are fixed *statically*; there is no need for a dynamic check to confirm this, so that the structure of a tuple by itself is never interrogated in an SCF-ML program.

As per line 3 of figure 7.4(c), zeroary patterns (such as *nil*) have interpretations as fields. Line 5 shows a product LP whose second component is *L*: the LP translates to the set of fields all of which are second components of a tuple, but internally may have any possible structure. Line 8 gives a simple example of why the LP representation is more compact than that of explicitly maintaining a set of fields: the LP, because it is essentially a grammar as opposed to an enumeration, is able to factor the prefix `uni(uni(...))` of the fields.

7.1.3 The Lattice of Liveness Patterns

The LP's of the previous section form a lattice under the partial order $\leq$ (read "is less dead than") defined in figure 7.5(a). It is straightforward to verify $l \leq l'$ iff $\tau\ l \supseteq \tau\ l'$, i.e., a liveness pattern is less dead than another if the set of fields it represents contains the set of fields represented by the other.

Figure 7.5(b) specifies a meet function *M* over this lattice. The meet function is symmetric in its arguments (line 11). It approximates conservatively the greatest lower bound (g.l.b.) for this lattice, in the sense that $M(l_1, l_2) \leq g.l.b.(l_1, l_2)$. Equivalently (since set union is the g.l.b. function over the lattice of the set under ordering $\supseteq$), $\tau\ M(l_1, l_2) \supseteq \tau\ l_1 \cup$

$$\begin{array}{ll}
L \le l \le D & \\
(l_1,\dots,l_n) \le (l'_1,\dots,l'_n) & \Leftrightarrow \forall_{i \in 1\dots n}.\ l_i \le l'_i \\
c\ l \le c\ l' & \Leftrightarrow l \le l' \\
l_1 \mid\dots\mid l_n \le l'_1 \mid\dots\mid l'_m & \Leftrightarrow \forall_{j \in 1\dots m}.\ \exists_{i \in 1\dots n}.\ l_i \le l'_j \\
[v, l] \le [v', l'] & \Leftrightarrow \underline{v}' \subseteq \underline{v} \wedge l \le l'
\end{array}$$

(a)

$$\begin{array}{ll}
M\ L\ l & = L \\
M\ D\ l & = l \\
M\ (l_1,\dots, l_n)\ (l'_1,\dots l'_n) & = (M\ l_1\ l'_1,\dots,M\ l_n\ l'_n) \\
M\ (c_{11}\ l_1 \mid\dots\mid c_{1m}\ l_n)\ (c_{21}\ l_1 \mid\dots\mid c_{2n}\ l_{2n}) & = c_{31}\ l_{31} \mid\dots\mid c_{3p}\ l_{31} \\
 & \quad \text{s.t. } c_{3i} \in \{c_{11},\dots,c_{1m},c_{21},\dots,c_{2n}\} \wedge \\
 & \quad (c_{3i} = c_{1j} = c_{2k}) \Rightarrow l_{3i} = M\ l_{1j}\ l_{2k} \wedge \\
 & \quad (c_{3i} = c_{1j} \ne c_{2k}) \Rightarrow l_{3i} = l_{1j} \wedge \\
 & \quad (c_{3i} = c_{2k} \ne c_{1j}) \Rightarrow l_{3i} = l_{2k} \\
M\ (l_1[v_1])\ (l_2[v_2]) & = (M\ l_1\ l_2) \\
 & \quad [\texttt{AbstractValue.meet}(v_1,v_2)] \\
M\ l_1\ l_2 & = M\ l_2\ l_1
\end{array}$$

(b)

FIGURE 7.5: The Lattice of Liveness Patterns.
(a) The partial order "is less dead than" on the LP domain (b) The meet function *M* on LP's.

$\tau\ l_2$: the set of fields represented by the meet of two LP's contains the union of the sets represented by the LP's individually.

An inconvenient aspect of the lattice is that its height is not bounded. The lattice includes descending chains such as $L < \dots\ c\ c\ c\ D < c\ c\ D < c\ D < D$. Furthermore, it is possible to generate such an unbounded sequence of liveness values during DSE. From the point of view of the dataflow analysis that uses the lattice, this unboundedness implies that the analysis will have to use a *widening function*, much as the partial evaluator did for the abstract value lattice in chapter 6. We present the widening function used by SCF in the next section.

7.1.4 Helper Functions

Figure 7.6 specifies the liveness pattern module. In addition to the *LP* datatype and the *meet* function, the module contains four auxiliary functions that operate on liveness patterns. Two of these functions are defined in figure 7.6, the other two in figures 7.7 and 7.8.

The first helper function is *widen* (lines 5-11 of figure 7.6), which takes two LP's l_{new} and l_{old} and returns a third pattern l_{res}. If DSE has analyzed a program point n^1 (typically n

```
structure LPModule =
type LP =                          //see definition of LivenessPattern in figure 7.4(a)
fun meet(lp:LP, lp:LP):LP =    //see definition of meet function M in figure 7.5(b),

and widen(L:LP, _:LP):LP = L
| widen(|(ls_new)|, |(ls_old)|) = let ls' = ListPair.map widen (ls_new,ls_old) in |(ls)| end
| widen([v, l], [v', l']) =
  let   v'' = if AV.mustBeEqual(v,v') then v else AV.bot(mkNewId())
        l'' = widen(l,l')
  in [v'', l''] end
| widen(l, l') = if identical(l,l') then l else L

and makeLPFromPattern: (SCF_ML.pattern * LPMap.map -> LP)
//see figure 7.7

and makeLPFromAbstValue(AV.bot _: value):value =
//see figure 7.8

and projectVariant(s:string, |c_1 l_1 |...|c_n l_n|) = if s = c_i then SOME l_i else NONE

and identical(l:LP, l':LP):bool = //true iff l and l' are structurally identical
end //LPModule
```

FIGURE 7.6: The Liveness Pattern Module.

= 10 in SCF) times, the *widen* function will be invoked on the LP computed for that program point on every subsequent analysis of the program point. LP's l_{new} and l_{old} are produced by two successive iterations of the fixpoint loop at a particular program point; l_{new} is the latest pattern, and l_{old} the penultimate one.

The widening function satisfies the criterion that after the widening threshold is crossed, the LP associated with a program point will change at most once. It is impossible, therefore, for the LP associated with any program point in the program to descend indefinitely down an unbounded chain in the lattice, and (as long as the flow functions are monotonic w.r.t. to the LP lattice), the dataflow analysis performed by DSE is guaranteed to converge.

A simple way to accomplish the criterion above would be for *widen* to return *L*: ***fun** widen(l,l')=* ***if*** *equal(l,l')* ***then*** *l* ***else*** *L*. This simple strategy is too conservative. In particular, recall from the example of figure 7.1 that a key task of DSE is to eliminate dead writes to the map(s) representing the abstract store in the optimization being analyzed. In

1.We call *n* the *widening threshold*.

typical implementations of optimizations, these maps are threaded throughout the program, typically as part of a tuple whose other element is the sub-tree of the AST being optimized. Note, for instance that as per line 2 of figure 7.1, the argument of the `analyzeCmd` function is a tuple whose first element is the command `c` being analyzed, and the remaining two arguments (`lSet` and `aMap`) are representations of the abstract store; similarly, as per line 11, the argument of the `transformCmd` function is a tuple whose first element is the command `c` being transformed, and the second is the map `aMap` representing the abstract store. To eliminate stores to these threaded maps in the face of widening, it is important that the widening function preserve detailed liveness information on the threaded map(s).

Lines 6-10 of figure 7.6 show how SCF maintains non-trivial information about threaded maps even in the face of widening. As per line 7, a tuple LP is widened by widening its individual components. If a threaded map is part of a tuple threaded through the optimization, therefore, as long as the map LP within the tuple pattern is widened not-too-conservatively, the LP for the map will be threaded back through the program. As per lines 7-10, a map LP is widened so that at least in the case that the LP computed by successive rounds of analysis after the widening threshold remains unchanged, the map remains unchanged. As per line 11, for all other LP's, if the new LP is different from the old, the entire LP is widened to *L*. Overall, the strategy has the effect that it drastically widens only those components of the tuple LP being passed through that change across fixpointing iterations.

The fourth helper function (line 35), *projectVariant* takes as argument a string *s* and a sum liveness pattern *l*. It optionally returns the liveness pattern *l'* such that *s l* is a summand of *l'*.

The second helper function, *makeLPFromPattern* (figure 7.7) takes as arguments an SCF-ML pattern, and a *liveness map* from identifiers to liveness patterns, and returns a liveness pattern that represents the set of live fields of any value matched by the pattern, given the access patterns specified by the liveness map. Consider, for example, the pattern *cons(x,y)* with the liveness map *[x->L, y->(L,D), z->cons(L,L)]*. The map implies that all parts of variable *x* are live, and that the fields y^{-1} and y^{-2} are live and dead respectively.

Given these three pieces of information, we may conclude that for any value matching the pattern, the fields $cons^{-1}\ \iota^{-1}$ and $cons^{-1}\ (\iota^{-1})^{-2}$ are live, i.e., that the fields denoted by the LP *cons(L,(L,D))* are live. In practice, replacing each identifier in the pattern with the LP attributed to the identifier in the liveness map (line 2), and replacing all wildcard matches with the LP *D* (line 1), yields the required result LP. The rest of the function (lines 3-8) traverses the pattern looking for identifiers and wildcards to replace.

A slight complication arises when the pattern is a binding pattern of the form *x* ***as*** *<pattern>*. Recall that SCF-ML allows case expressions such as **`case`** `foo()` **`of`** `x` **`as`** `cons(y, _) => y::x`, which prepends the first element of a list to the list itself. The complication is that if the pattern being matched in the previous paragraph were *z* ***as*** *cons(x,y)* (with the same liveness map), then we know from the *cons*... part of the pattern that the value being matched has LP *cons(L,(L,D))* as before, but the identifier *z* implies (since *z* has LP *cons(L,L)* in the map) that the value has LP *cons(L,L)*. We combine the two pieces of information by meeting the two LP's (line 8) to get aggregate LP for the value, in this case *cons(L,L)*.

The third function, *makeLPFromAV,* takes an abstract value and returns a LP that represents (a superset of) the fields of each concrete value represented by that abstract value. The LP returned has a structure very similar to that of the abstract value it represents. For instance, the abstract value *cons(3, cons(**1**, nil))* results in liveness pattern *cons(3, cons(L, nil))*. Note that the abstract value ***1*** is turned into the liveness pattern *L*: since the former represents the set of all possible values, the latter represents the set of all possible fields. LP's, however, are simpler in basic structure from abstract values in three basic ways, as reflected in the corresponding lines of the function:

```
1  fun makeLPFromPattern(|_|: SCF_ML.pattern, lpm:LPMap.map):LP = LP.D
2  | makeLPFromPattern(|x|, lpm)        = case LPMap.find(lpm,x) of SOME l => l | _ => LP.D
3  | makeLPFromPattern(|c|, _)          =|c()|
4  | makeLPFromPattern(|c p|, lpm)      = let l = makeLPFromPattern(p,lpm) in |c l| end
5  | makeLPFromPattern(|(ps)|, lpm)     =
6     let ps' = List.map (fn p => makeLPFromPattern(p, lpm)) ps in |(ps')| end
7  | makeLPFromPattern(|x as p|, lpm)   =
8     LP.meet(makeLPFromPattern(p, lpm), makeLPFromPattern(|x|, lpm))
```

FIGURE 7.7: The *makeLPFromPattern* Helper Function.

- It is possible for different alternates in *alt* abstract values to have the same constructor, whereas the different alternates of a liveness pattern must have different constructors (line 5 of figure 7.4(a)). For instance, *cons (3, nil) | cons (4, nil) | nil* is a valid abstract value, whereas the LP that summarizes (a superset of) the fields represented by this abstract value would be *cons(3|4, nil) | nil*. The conversion function therefore merges alternates with the same tag (lines 5-6) before recursively processing the summand abstract values (line 7).

- It is possible to have a recursive (*fix*) abstract value, whereas DSE provides no recursive LP's. The translation therefore conservatively attributes LP *L* to *fix* abstract values (line 9): all fields of conforming concrete values are assumed live.

- Abstract map values have an associated list structure, whereas map LP's are represented by a pair. Abstract values seek to maintain the correlation between particular domain and range elements of the map, whereas LP's do not. DSE therefore first collapses the domain and range of the incoming map into abstract values (line 11). These two values determine the domain and range respectively of the resulting map LP.

7.2 The Abstract Interpreter

The interprocedural part of DSE (not shown because it is a standard context-insensitive widening-based worklist scheme) associates with every function being analyzed one LP representing its return value, and one representing its argument. The *return LP* of a func-

```
1   fun makeLPFromAbstValue(AV.bot _: value):value = |L|
2   | makeLPFromAbstValue(AV.tuple(vs, _))      = |(List.map makeLPFromAbstValue vs)|
3   | makeLPFromAbstValue(AV.tagval(c, v, _)) = let l = makeLPFromAbstValue v in |c l| end
4   | makeLPFromAbstValue(AV.alt vs)            =
5      let   [c_1',..,c_m'] = List.map (fn c _ => c) vs
6          vs'              = List.fold (fn (|c v|, [...,c v',...] => [...,c (AV.meet(v,v'),...])) [c_1' top,...,c_m' top] vs
7          [l_1,...,l_n]    = List.map makeLPFromAbstValue vs'
8      in |l_1|...|l_n| end
9   | makeLPFromAbstValue(AV.fix(v,_))          = |L|
10  | makeLPFromAbstValue(v as AV.map _)       =
11     let   (v_key, v_val) = AV.collapseMap v in |[v_key, makeLPFromAbstValue v_val]| end
12  | makeLPFromAbstValue |[Int|Bool|String] k|= |[Int | Bool | String] k()|
```

FIGURE 7.8: The *makeLPFromAbstValue* Helper Function.

```
1 fun foo x1 =
2 case x1 of (x2, x3) =>
3    x2
4 end
```

```
1 fun foo x1 =
2 case x1 of (x2, _) =>
3    x2
4 end
```

FIGURE 7.9: Interprocedural Analysis Example.
Input for intraprocedural analysis (l). A pruned version of the function (r).

tion captures the fields of the return value accessed by computations downstream of the function. The *argument LP* captures the fields of the argument value accessed by the function body and all downstream computations. The analysis begins by assuming that the optimization program as a whole has return LP *L*, i.e., that every field of any return value of the program may be used. LP's are propagated from callees to callers in a fixpoint loop. Termination is guaranteed by widening the LP's associated with a function after it is processed a fixed number of times using the *widen* function of the previous section.

During execution, DSE maintains a map (defined by the module *LPMap*) between function names and liveness patterns (line 2 of figure 7.10) and a worklist *wl* (line 1) of functions to be processed along with their best known argument and result patterns. While processing each function, it consults the global collecting semantics map *lvm* (line 3 of figure 7.10) that maps map expressions to their set of live keys as described in section 4.4.4.

Given the return LP for a particular function, the intraprocedural part of DSE computes the argument LP for the function, and a pruned version of the function body. Given function `foo` of figure 7.9, for instance, and a return LP *L*, the intraprocedural analysis deduces that foo has argument LP *(L,D)*. In particular, note that the case expression of line 2 specifies that `x1` must have two fields, of which the second is unused (and is therefore dead). The pruned function body (shown on the right of the figure) is different from the original because it no longer binds the dead variable `x3`.

The intraprocedural analysis works by recursively associating a LP with each subexpression of the function body. The LP for a subexpression specifies which fields of the values produced by the subexpression may be used downstream. For each subexpression *e*, the backward pass returns a pair containing the expression resulting from pruning out dead sub-expressions of *e*, and a map (which we will call the *liveness map*) from free variables of *e* to LP's that represent the live fields of those variables. This map corresponds to the live-variables set in traditional dead-code elimination. Analyzing sub-expression `x2` of line 3 of figure 7.9, for instance, results in the liveness map *[(x2,L)]*; the case expression as a whole gives liveness map *[(x1,(L,D))]*. The bulk of the complexity of the DSE optimization lies in the way particular types of sub-expressions are treated. Figures 7.10 through 7.16 describe the different cases. We discuss each case below.

7.2.1 Dead Expressions

If the expression being analyzed has liveness pattern *D* (figure 7.10, lines 6-7), we may conclude that no field of the expression will be read downstream. In this case, if the expression has type *t* (written *e:t* in the figure), we replace the expression by a simpler expression of type *t*. The function *makeReplacementExpr* computes this simpler expression. If *t* is a product type, the replacement expression is a tuple consisting of replacement sub-expressions derived from the multiplicands. In the case that it is a sum type, the replacement sub-expression is simply a zeroary variant of that type; in case the type does not have a zeroary variant, we introduce one.

For example, an expression of type *list = cons of int * list | nil* will be replaced with the expression *nil*. An expression of type *int * list* could be replaced with expression *(0, nil*). The latter case exposes a complication. For instance, if the original expression were the variable *x*, then the replacement expression in this case is more complicated, not less, than the one it replaces! We finesse this issue by relying on the non-scalar rematerialization strategy of the previous chapter: we assume that all constants can be generated statically in the text segment so no runtime cost is incurred in constructing them. An alternate strategy, which we have also explored in SCF, is to never attempt to replace expressions of tuple type with a simpler expression; instead, we simply analyze such an expression using liveness pattern *(D,...,D)* (where the number of *D*'s correspond to the arity of the tuple), and thereby attempt to simplify the components of the tuple.

The second component of the returned tuple, the liveness map, is the empty map in this case: a dead expression does not constrain the liveness of variables or their fields.

```
val wl:DSEWorkList.worklist = ref DSEWorkList.empty
structure LPMap = ... //Map from IDs to liveness patterns
val lvm:LabelAbstractValueMap.map = ref LabelAbstractValueMap.empty
...
//Dead store elimination on expressions
fun DSE_e (|e:t|: SCF_ML.expr, D:LP):(expr * LPMap.map) =
   (makeReplacement t, LPMap.empty)

| DSE_e (|x|, lp) = (|x|, LPMap.empty.insert(x,lp))

| DSE_e (|(es)|, L) =
let   (es', lpms) = List.unzip (List.map (fn e => DSE_e(e, L)) es)
      lpm = List.fold mergeMap LPMap.empty lpms
in (|(es')| , lpm) end

| DSE_e (|(es)|, |(ls)|) =
let   (es', lpms) = List.unzip (ListPair.map (fn (l,e) => DSE_e(e, l)) (ls,es))
      lpm = fold mergeMap LPMap.empty lpms
in (|(es')| , lpm) end

| DSE_e (|c e|, |L|) = let (e', lvm) = DSE_e(e, L) in (|c e'|, lvm) end

| DSE_e (|c e|: t, cls as |c_1 l_1|...|c_n l_n|) =
case List.find (fn |c l| => true| _ => false) cls of
   SOME |c l|    => let (e', lvm)= DSE_e(e, l) in (|c e'|, lvm) end
   | NONE        => (makeReplacement t, LPMap.empty)

| DSE_e (e as |p _|, l) = DSE_primop(e, l)

| DSE_e (|case e of ms|, l) =
let   (ms', l', lpm') = DSE_ms(ms, l)
      (e', lpm) = DSE_e(e, l')
in (|case e' of ms'|, mergeMap(lpm, lpm')) end

| DSE_e (|f e|, l) =
let l_arg     = case LPCache.find(!lpc, l) of SOME ci => ci#argVal | NONE=> |D|
      _       = DSEWorkList.add(!wl, MAY(f, l_ret))
   (e', lpm) = DSE_e(e, l_arg)
in (|f e'|, lpm) end
| DSE_e (|k|, _) = (|k|, LPMap.empty)

| DSE_e (e_c as |cf (fn x => e) e'|, l) = DSE_cf(e_c, l)

//Merge two liveness pattern maps; meet liveness patterns shared between the maps
and mergeMap(lpm:LPMap.map, lpm:LPMap.map):LPMap.map =
LPMap.unionWith (fn (lp,lp') => LPModule.meet(lp,lp')) (lpm, lpm')

and makeReplacement (ts as |t_1* ...*t_i*...* t_n|:SCF_ML.type):SCF_ML.expr =
      let es = List.map makeReplacement ts in | (es)| end
| makeReplacement t = zeroaryVariant t
```

FIGURE 7.10: The Function DSE$_e$ for Dead Store Elimination on SCF-ML Expressions.

7.2.2 Variables

If the expression being analyzed is a variable *x* (line 9), since from the previous case we know that *x*'s LP *l* is not *D, i.e., x* is not completely dead, we conclude that *x* cannot be further simplified.

To record that all the fields of *x* represented by LP *l* are live, we return a liveness map *[x->l]*.

7.2.3 Tuples

If the expression is a tuple, the LP for the expression must either be *L* (denoting that every field of every value that the expression evaluates to is live), or a product LP (note that the case where the LP is *D* is already handled by section 7.2.1). In the former case (lines 11-14), we analyze every component of the tuple expression using LP *L*, and construct a new tuple expression and liveness map from the results. In the latter case, lines 16-19, we analyze each component of the tuple expression using the corresponding component of the tuple LP instead.

If a field is live in some field of a tuple, it is live in the tuple as a whole: we therefore simply merge the corresponding liveness maps to get the map for the tuple as a whole (lines 13 and 18).

7.2.4 Constructors

If the expression is a constructor application *c e*, the LP must either be *L* or a sum LP.

In the former case (line 21), we analyze the body *e* of the expression using LP *L* (to acknowledge that any field of *e* may be live), and tag the resulting expression *e'* with tag *c* to produce the pruned expression. The liveness map from *e* is the same as that of *c e*, since applying the constructor does not affect the liveness of any variables in the expression (or their fields).

In the latter case (lines 23-26), we first identify the summand *c l* that shares the same tag as the expression, and analyze *e* with LP *l* (so as to constrain only the fields of *e* denoted by *l*) (lines 24-25). The return values for the entire expression follow straightforwardly. If no such summand exists, the entire expression is dead, and replaced by a compact replacement expression (line 26) as in the dead expression case above; the liveness map is of course empty.

7.2.5 Primitive Operations

Primitive operations (primops) are analyzed via the DSE_{primop} function (invoked on line 28 of figure 7.10, defined in figure 7.11). The interesting case is when the primop in question is a map find, insert or equality operation. For all other cases (line 29 of figure 7.11), we simply optimize the argument of the primop under the assumption that all its arguments are live (i.e., using LP *L*),[1] and reconstitute the primop using the pruned argument expression, while noting the live fields of the argument.

7.2.5.1 Map Insertion

If the argument is a *map_insert* operation, we seek to determine if the operation is dead by checking if the potential map fields into which the insertion takes place are dead as per the LP for the map. If so we avoid the map insert operations. If not, we keep the operation and recursively optimize its subexpressions.

For instance, suppose we know that over any execution of a program, a map insert operation `map_insert(`e_m`,`e_k`,`e_v`)` writes to keys *{1, 454, 6443}, i.e.,* e_k may evaluate to one of these values.[2] Suppose also that the LP for the resulting map is *[12|13, L]*. Since the latter LP indicates that no downstream computation accesses keys other than *12* and *13* of the map, and the insert operation does not write to any of these keys, we can eliminate the map insert operation. In particular, we can replace the `map_insert` operation with the expression e'_m representing the optimized pre-insertion map. Line 2 of figure 7.11 consults the map *lvm* for the set of live keys for insert operations. Line 3 extracts live keys from the LP. Line 6 checks if the two sets are disjoint. Line 7 returns e'_m if they are indeed disjoint.

The question now arises as to what LP's to use when optimizing the subexpressions e_m, e_k and e_v in the above example. Take expression e_m first. Since e_m represents a

1.Note again that the case where the expression is dead is handled on lines 6 and 7.
2.e_m, e_k and e_v are meta-variables representing SCF-ML expressions.

```
and DSE_primop(|map_insert(e_m,e_k, e_v):Map.map|:SCF_ML.expr, l:LP):(SCF_ML.expr* LPMap.map)=
let  v_k_val      = case LabelAbstractValueMap.find(!lvm, SCF_ML.labelOf e_k) of SOME v => v
     (v_k_live,l')  = case l of L => (bottom(newId()), L)||[v, l]| => (v, l)
     v_k          = if AV.mustBeSingleton v_k_val then AV.subtract(v_k_live, v_k_val) else v_k_live
     (e_m', lpm_m)= DSE_e(e_m,[v_k, l'])
in if AbstValue.mustBeDisjoint(v_k_val,v_k_live) then
     (e_m', lpm_m)
   else
     let  ((e'_k,lpm_k), (e'_v, lpm_v))= (DSE_e(e_k,LPModule.makeLPFromAbstValue v_k ), DSE_e(e_v,l'))
          lpm                 = LPMap.merge(lpm_m, LPMap.merge(lpm_k,lpm_v))
     in (|map_insert(e'_m,e'_k, e'_v)|, lpm) end
end

| DSE_primop(|map_find(e_m,e_k)|, l) =
let  v_k          = case LabelAbstractValueMap.find(!lvm, SCF_ML.labelOf e_k) of SOME v => v
     l_v          = case LPModule.projectVariant("SOME", l) of SOME l' => l'
     (e_m', lpm_m)= DSE_e(e_m,[v_k, l_v])
     l_k          = LPModule.makeLPFromAbstValue v_k
     (e_k', lpm_k) = DSE_e(e_k,l_k)
     lpm          = LPMap.merge(lpm_m,lpm_k)
in (|map_find(e'_m,e'_k)|, lpm) end

| DSE_primop(e as |map_equal(e_m1,e_m2)|, l) =
let  (e'_1, lpm_1) = DSE_e(e_m1,[1, L])
     (e'_2, lpm_2) = DSE_e(e_m2,[1, L])
     lpm          = LPMap.merge(lpm_1, lpm_2)
in (|map_equal(e'_1, e'_2)|, lpm) end

| DSE_primop(|p e|, _) = let (e', lvm) = DSE_e(e, L) in (|p e'|, lvm) end
```

FIGURE 7.11: The Function *DSE$_{primop}$* for Optimizing Primops, Including Map Operations.

map, we need to build a LP of the form *[v, l]* for it, where *v* represents live keys of the map, and *l* the live fields of the values stored in the map. A conservative choice for the set *v* of live keys for `e`$_\mathtt{m}$ is to use the same set as for the `map_insert` operation as a whole: if a key *k* is not live downstream of the `map_insert` operation, since the `map_insert` operation itself does not make any additional keys live, *k* cannot be live in the map being inserted into. In the above example, for instance, we may conclude that at most the keys *{12, 13}* of the map represented by `e`$_\mathtt{m}$ are live. Similarly, a conservative choice for the LP *l* representing the values stored in `e`$_\mathtt{m}$ is the LP for the map resulting from the `map_insert` operation as a whole.

A small optimization is that in the particular case that the `map_insert` operation can insert exactly one concrete value, we can perform a "strong update": we can optimize (line 5 of figure 7.11) `e`$_{\mathtt{m}}$ using the LP for the post-insertion map with this key removed (line 4). For instance, if the possible values of `e`$_{\mathtt{k}}$ above were *{12}*, then we know that only the key *13* of `e`$_{\mathtt{m}}$ is alive: key *12*, even if defined pre-insertion, is re-defined before its first use.

Now, consider optimizing subexpression `e`$_{\mathtt{k}}$ (we only have to do this if the insert operation as a whole is not dead). We need to deduce a LP with respect to which to optimize `e`$_{\mathtt{k}}$. We again make a conservative estimate in this case. In the example above, since the result of inserting the key `e`$_{\mathtt{k}}$ into the map `e`$_{\mathtt{m}}$ is a map with LP *[12|13, L]*, we can assume that the only concrete values produced by `e`$_{\mathtt{k}}$ that are live are *12* and *13*. Correspondingly, it is sufficient that the LP for optimizing `e`$_{\mathtt{k}}$ specify that every field of concrete values *12* and *13* is live. The LP *12|13* does the trick. Given that the insert expression as a whole inserts keys into a map expression, and that it has LP *[v,l]*, a general technique for obtaining the LP for `e`$_{\mathtt{k}}$ therefore is to generate an LP that includes all the fields of all the concrete values that conform to *v*. This is precisely what we do via the call to *makeLP-FromAbstValue* on line 9.

Finally, consider optimizing subexpression `e`$_{\mathtt{v}}$. As LP for this subexpression, we simply (and conservatively) use the LP *l* for the range of the map expression as a whole (line 9). Since *l* includes all fields of all map range values possibly used downstream, it will in particular contain all live fields of values that `e`$_{\mathtt{v}}$ evaluates to.

7.2.5.2 Map Reads

We now describe how map read operations are processed. The basic intuition is that if a particular read operation reads a key from a map, then that key should be live in the LP for that map.

For instance, suppose we have a read operation `map_find(e`$_{\mathtt{m}}$`,e`$_{\mathtt{k}}$`)` with LP l_v = *SOME (12|77)*.[1] Suppose also that the collecting semantics map *lvm* attributes live keys v_k = *"a"|"c"|"m"* to this expression . We want an LP *[v,l]* for expression `e`$_{\mathtt{m}}$. First, focus on

1.Recall that the `map_find` operation has type `('a, 'b) map * 'a -> 'b option`. Thus, `map_find([(7,123)], 7)` returns value `SOME 123`, whereas `map_find([(7,123)], 12)` returns `NONE`.

v, the set of live keys in the map. Since (as per v_k) we read at most the values *“a”*, *“c”* and *“m”*, we conclude that at least these keys should be live. In general, we can simply use the abstract value v_k denoting the read keys as the set v of keys required to be live in the map. Now consider the LP l for representing the values stored in the map. Since (as per l_v) downstream computations are interested in at most the values *12* and *17* stored in the map, we conclude that it is sufficient to store these values in the map. Thus l is the result of projecting away the *SOME* tag from LP l_v, as per lines 15-17 of figure 7.11.

Finally, we need an LP with respect to which expression $\mathtt{e}_\mathtt{k}$ can be optimized. Ideally, we would like to identify just those keys of the map that result in the live fields denoted by l_v. For instance, if we knew that $\mathtt{e}_\mathtt{m}$ evaluates to abstract map *(must[(“a”,12),(“b”,17)], may [(“c”,99|100)])*, then since we know from l_v above that only the values *12* and *17* are live downstream, we could conclude that key *“c”*, which maps to dead value *99*, is itself dead. However, SCF maintains much simpler information about the map: it only maintains the abstract value, v_k, representing the set of keys with which the map is read. Since a map key cannot be live if that key is never read, we can use v_k as a conservative estimate of the set of live keys. In particular, we say that every field of every concrete value that conforms to v_k is live. We use the function *makeLPFromAbstValue* to generate the LP representing these fields (line 18 of figure 7.11). Line 19 performs the actual optimizing of $\mathtt{e}_\mathtt{k}$.

Merging liveness maps and reconstituting the map expression (lines 20-21) is routine.

7.2.5.3 Map Equality Tests

The third primitive operation of interest on maps is the equality test on maps. Given expression `map_equal(`$\mathtt{e}_{\mathtt{m1}}$`,`$\mathtt{e}_{\mathtt{m2}}$`)` and its LP l we need to determine the liveness patterns for the expressions $\mathtt{e}_{\mathtt{m1}}$ and $\mathtt{e}_{\mathtt{m2}}$ that represent the argument maps to the operation. To do so, we need to answer the question: given that we want to compare two maps for equality, which parts (i.e., keys and values) of these maps would we need to access in order to determine equality (and are therefore live upstream)? In the absence of any further information, we must resort to saying that every part of every key or value of these maps may contribute to determining equality. We therefore ascribe (lines 24 and 25 of figure 7.11) the liveness pattern *[**1**, L]* to each of the maps: the ***1*** indicates that every key in the maps may be accessed, and the *L* indicates that every part of every value for these keys may be accessed.

A possible concern at this point is that these values may be so conservative as to render the DSE pass ineffective. After all, the above approach implies that any time a map equality test appears in the residual optimization program (we expect at least one of these for each loop in the incoming program), most live upstream inserts to this map will also be live. The fact that these inserts cannot be eliminated as dead would seem to jeopardize our goal (mentioned in the introduction to this chapter) of removing as many map insertion operations as possible.

In practice, however, we are saved by the fact that optimization programs tend to thread two maps representing the abstract store: the first is the traditional abstraction of the store at a given program point, and the second is the “sticky representation” of the store, i.e., a map from all relevant program points to the latest relevant dataflow fact at that point. Equality testing is only performed between the former kind of map, whereas the latter (which is essentially an annotation of the program with dataflow facts) is only used in the transformation part of the optimization: transformations check whether the dataflow fact at a point requires a program transformation at that point. Even if many of the former kind of map insert operations are not removed, virtually all inserts into the latter map that are not read by the transformation code are removed.

In the DAE optimization of figure 3.7, for instance, the set `lSet` gives the set of live variables at the current program point, whereas the “sticky” map `aMap` maps all assignment commands analyzed thus far to the liveness of that command. Equality testing is performed only on `lSet` (figure 3.7, line 61).

7.2.6 Case Expressions

We now discuss (starting with figure 7.10, lines 30-33) how to compute the liveness map and pruned expression for a case expression e_c = *case* e_g *of* p_1 => e_1 | ... | p_n => e_n (recall that each individual pattern/expression pair m_i = p_i => e_i is called a *match*). We first describe how the individual matches m_i are processed, followed by the guard expression

```
1 case x1 of
2   unary x2        => unary x2
3 | binary(x2, x3) => binary(x2,x2)
4 end
```

```
1 case x1 of
2   binary(x2, _)  => binary(x2,x2)
3 end
```

FIGURE 7.12: DSE of Case Expressions.
Original expression (l); pruned expression (r).

e_g, and end by describing how to combine these component results into the results for the case expression as a whole.

7.2.6.1 Example

Before diving into details, we present an example (figure 7.12). The expression on the left of the figure is a case expression that needs optimization. Assume that the LP for the expression is *binary L*, i.e., downstream computations only access the *binary* variant of the result of the case expression. Note that intuitively, if the *unary* variant of the result is indeed unused, then the match of line 2 (which produces only this variant) is dead and can be removed. Further, since the variable `x3` bound in line 3 is unused, its binding can be removed. Finally, since the guard expression `x1` is live (since there is at least one match against it) and cannot be further simplified, we leave it unchanged. The result is the pruned expression on the right of the figure.

It remains to show how to produce the liveness map for the expression: which variables (and in particular, which of their fields) used in the expression are live? Examining the pruned expression on the right side of the figure (since we can disregard dead parts of the initial expression), we see that there are two candidate variables, `x1` and `x2`. `x2` is bound within the expression before any of its uses, and is therefore disqualified from the live set. `x1` is free within the expression and therefore live, but the case expression only reads the first field `x2` of the *binary* variant of this variable. Finally, since the expression as a whole returns `binary(x2,x2)` and the overall LP for the expression is *binary L*, we know that all of `x2` is live inside the match. We conclude therefore that only the field $(\texttt{binary}^{-1}\ \iota)^{-1}$ of `x2` is live, or equivalently, create a liveness map binding `x1` to LP *binary(L,D)*.

```
1  fun DSE_m(|p =>e|: SCF_ML.match, l:LP):(SCF_ML.match * LPMap.map * LP * bool)=
2    let  (e’, lpm)        = DSE_e(e, l)
3         (p’, l’, lpm’)  = filterPattern(p, lpm)
4         isLive          = not (isReplacementVal e’)
5    in (|p’ => e’|, lpm’, l’, isLive) end
6
7  and filterPattern(p:SCF_ML.pattern, lpm:LPMap.map): (SCF_ML.pattern * LP *LPMap.map) =
8    let  ids          = getIdsFromPattern p
9         p’           = List.fold (fn (id, p) => case LPMap.find(lpm,id) of
10                                            SOME L=> p
11                                            | _ => removeIdFromPattern(p,id))
12                                   p ids
13        lpm’         = List.fold (fn (id, lpm’) => LPMap.remove(lpm’, id)) lpm ids
14        (l’, lpm’’)  = LPModule.makeLPFromPattern(p’, lpm’)
15   in (p’, l’, lpm’’) end
16
17 and removeIdFromPattern(p:SCF_ML.pattern, x:SCF_ML.id): SCF_ML.pattern =
18 //If identifier x is bound in pattern p, replace the binding instance with the wildcard pattern “_”
19
20 and isReplacementVal(e: SCF_ML.expr): boolean =
21 //true if e is an expression produced by the makeReplacement function of figure 7.10, false otherwise
```

FIGURE 7.13: The Function DSE_m for Optimizing Case Expression Matches.

7.2.6.2 Optimizing Matches With DSE_m and DSE_{ms}

Function DSE_m of figure 7.13 describes how individual matches are optimized.

To analyze each match, we need to first decide what LP to analyze it with. In our example, if the case expression as a whole has LP *binary L*, it is clearly reasonable to use this LP for analyzing each match of the case expression (figure 7.14, line 2). This is conservative because if a value is definitely dead downstream of a case expression, then it must in particular be dead downstream of each match in the case.

Optimizing a match phrase *m* (of the form *p => e*) produces four results (line 1 of function DSE_m in figure 7.13): the pruned version *m’ = p’ => e’* of the match, the liveness map *lpm’* indicating live fields of the match, an LP *l’* representing live fields accessed by the pattern *p* of the map, and a boolean *isLive* that is true if any part of the value of *e’* is possibly live, and false otherwise. Below, we describe, in turn, how each of these results is computed.

1. We compute the pruned expression *e’* and liveness map *lpm* corresponding to expression *e* (line 2, figure 7.13) by a recursive call to DSE_e. We compute, in the helper

```
1  fun DSE_ms(ms: SCF_ML.match list, l:LP):(SCF_ML.match list * LP * LPMap.map) =
2    let  (ms, lpms, ls, les) = List.unzip (List.map (fn m => DSE_m(m,l)) ms)
3         l                  = List.fold LPModule.meet LPModule.D ls
4         lpm                = List.fold LPMap.merge LPMap.empty lpms
5         msLive             = ListPair.fold (fn (m, true, msLive')=> m::msLive' | _=> msLive')
6                                        [] (ms, les)
7    in (msLive, l, lpm) end
8
9  and DSE_m ... // See figure 7.13
```

FIGURE 7.14: The Merge Function *DSE_{ms}* for Optimizing Case Expression Matches.

function *filterPattern*, the pruned pattern *p’* as follows. If any of the variables bound in *p* don’t appear in *lpm*, we deduce that it is unnecessary to bind them (since they are dead), and remove them from *p* (lines 8-12). The result is *p’*. In our example, when pruning the match of figure 7.12(l), line 3, since `x3` is dead in the expression `binary(x2,x2)`, we replace it with a wildcard to get the pattern of line 2 of figure 7.12(r).

2. If *e* is not inferred dead (and therefore replaced by a simpler expression), we set *isLive* to true, else false (line 3).
3. To compute *lpm’*, we note that the pruned pattern *p’* binds some of the variables in *lpm*. These variables are therefore dead outside the match phrase. In our example, although `x2` is live inside the match of line 3 of figure 7.12(l), it is dead outside, since it is bound by the match pattern. We therefore remove from *lpm’* all variables that are bound in *p’* (line 13). This step is analogous to killing assigned variables from the live-variables set in conventional dead assignment elimination.
4. Finally (line 14), we invoke the helper function *makeLPFromPattern* (defined earlier in section 7.1.4) to compute the live fields accessed by each pattern, including the effect of bound variables. Intuitively, the resulting LP *l’* represents the set of fields of the guard expression e_g that are accessed as a result of the pattern match *p*.

Merging the results of individual matches is straightforward (figure 7.14). We apply the appropriate meet operators to the LP’s (line 3) and liveness maps (line 4) resulting from each match. For future reference, note that the resulting LP represents the set of all fields of the guard expression that may be accessed by any of the matches, i.e., all live fields of the expression. We accumulate the pruned matches that result into a list of matches, with the optimization that if a match is dead, we omit it from the list (line 5).

7.2.6.3 Optimizing the Guard Expression

Having computed the set of all live fields of the guard expression in the previous section, it is a simple recursive call to DSE_e (figure 7.10, line 32) to optimize the guard expression e_g. Intuitively, since e_g is consumed solely by the patterns p_i of the matches, its live fields are just the union of the fields accessed by these patterns (or of the fields of variables bound by the patterns).

7.2.7 Constants

Scalar constants cannot be further pruned, and don't result in new live variables (figure 7.10, line 40).

7.2.8 Curried Functions

Recall that SCF-ML has two built-in functions, `map_map` and `map_unionWith`, on map operations that each take an anonymous function and either one or two maps as arguments. The former applies the function to every range value in the argument map. The latter produces a union of two maps, applying the anonymous function to merge range values whenever the maps being combined have a common key. See figure 3.5 for examples of how these functions are used. We now discuss how DSE works for these two built-in functions.

7.2.8.1 Example

Consider the example expression on the left of figure 7.15. Suppose the expression as a whole has LP *[1|22|37, binary L]*.[1] This LP indicates that every live value in the range of

```
map_map
  (fn SOME x  => binary(x,x)
      | NONE  => nil)
  m
```

```
map_map
  (fn SOME x => binary(x,x))
  m
```

FIGURE 7.15: DSE of Curried Expressions.
Original expression (l); pruned expression (r).

the map is a *binary* variant. Since the range values of the map are produced by application of the anonymous function of lines 2-3, we can conclude that only the parts of this function that produce *binary* variants of this function are live. In particular, the case that produces the *nil* variant on line 3 is dead. The resulting pruned expression is the one on the right of figure 7.15.

The second result required from DSE of the expression is its liveness map: which of its variables are live, and what are their LP's, i.e., which fields of these variables are live? Since the efficacy of DSE depends critically on being able to track the live fields of maps, it is actually more important that we get as precise a result as possible for the LP of the map being operated upon, than to prune (as in the previous paragraph) the curried expression. It is clear from the figure that the only variable free in the expression, and therefore possibly live outside it, is `m`, which represents the incoming map. What can we say about the liveness of fields of this incoming map, given our knowledge of the LP of the result map?

Consider the liveness of map keys. Since the map operation itself does not use any of the keys of the incoming map (it uses purely the values in the range of the map, and those of variables bound outside the expression), we conclude that the incoming map has no more live keys than the result map. In our example, we conclude that the value *1|22|37* conservatively approximates the live key set of the incoming map. Note that the same reasoning holds for the union operation (since the anonymous function for this operation also operates purely on range values of the incoming maps); of course, in the case of unions, there are two incoming maps, each of whose domains are approximated by that of the outgoing one.

Now consider the liveness of map range values. The range values of the result map are created by applying the anonymous function to those of the input map. In our example, the live fields of the range values of this result map are captured by the LP *binary L*. The anonymous function that generates these values has the form *(fn SOME x => binary(x,x) | NONE => nil)*. By requiring that this expression produce results with live fields represented by *binary L*, we conclude that *x* must have LP *L* too, and that the input to the anon-

1.Since both built-ins result in a map, their LP's must both be either a map LP, *L* or *D*.

ymous function has LP *SOME L*. Abstracting away from the example, the LP for the range values of the incoming map is computed by using the range LP of the result map as the result LP of the anonymous function, and using the traditional DSE backward pass to compute the LP for the formal parameter of the function.

7.2.8.2 The Function DSE_{cf} for Optimizing Curried Functions

Figure 7.16 specifies the function DSE_{cf} that specifies the details of the algorithm underlying the example above. The function works via the helper function *optimizeCurriedMapOperation*, which takes six arguments: *opr*, the particular primitive map operation being optimized, the variable *x* that represents the formal parameter of the anonymous function, the expression e_b that is the body of the anonymous function, the expression e_m that generates the map (if *opr* is *map_map*) or the pair of maps (if *opr* is *map_unionWith*) to be operated on, the LP *l* of the resulting map, and the number *i* of argument maps (*i* is 1 for the *map_map* operation and 2 for *map_unionWith*).

We first tease apart the domain (into variable v_{map_dom}) and range (into l_r) of the result map (line 9) from the LP *l* for the whole expression. The complication here is that if the result is *L* (i.e., we simply know that all parts of the result are live), we need to infer that all possible keys of the map may be live (by using the abstract value *bottom* for it) and all possible range fields are live (by using LP *L* for it). We then proceed as described in the

```
1  and DSE_cf(|map_unionWith (fn x => e_b) e_m|:expr, l:LP):(expr* LPMap.map)=
2       optimizeCurriedMapOperation(|map_unionWith|, x, e_b, e_m, 2)
3
4  | DSE_cf(|map_map (fn x => e_b) e_m|, l)=
5       optimizeCurriedMapOperation(|map_map|, x, e_b, e_m, 1)
6
7  and optimizeCurriedMapOperation(opr:SCF_ML.primop, x:id, e_b:expr, e_m:expr, 1:LP, i:int):
8       (expr * LPMap.map) =
9  let  (v_map_dom,l_r)    = case l of L => (bottom(newId()), L)| |[v, l]| => (v, l)
10      (e'_b,lpm_b)       = DSE_e(e_b, l_r)
11      l_b                = case LPMap.find(lpm_b, x) of SOME l => l| _ => D
12      l_map_rng          = case opr of map_map => l_b | map_unionWith => LPModule.meet(l_r, l_b)
13      l_map              = [v_map_dom, l_map_rng]
14      l_m                = case i of 1 => l_map | 2 => |(l_map, l_map)|
15      (e'_m,lpm_m)       = DSE_e(e_m, l_m)
16      lpm'_b             = LPMap.delete(lpm_b,x)
17 in (|opr (fn x => e'_b) e'_m|, LPMap.merge(lpm'_b, lpm_m)) end
```

FIGURE 7.16: The Function DSE_{cf} for Optimizing Curried Function Applications.

preceding example. To compute the range value of the incoming map(s), we invoke (line 10) DSE on the body e_b of the anonymous function, and look (line 11) for the LP (if any) bound to formal *x*. As discussed in the example above, for the *map_map* operation, l_b then represents the liveness of range values.

A complication is that for *map_unionWith* operations, the anonymous function is only applied to range values that have a common key. For instance, the expression `map_unionWith (fn (x,y) => x + y) (m1,m2)`, with incoming maps *m*1 and *m2* bound to values *[(1,2)]* and *[(3,4)]* produces the result map *[(1,2), (3,4)]*. In this case, the anonymous function is not applied at all, since the two maps have no keys in common. Conservatively, therefore, to derive LP's for the incoming map ranges, given the LP for the result map, we need to account for the fact that the fields may or may not have been created by the anonymous function. For *map_unionWith* operations, in order to get the range LP for the incoming maps, we meet (line 12) the l_b (which is computed assuming that each range value is produced by application of the anonymous function) and l_r (which assumes the function was not applied). Also, since we cannot determine during our backward pass which of the two input maps contained each live field, we need to conservatively assume that either could have done so. In our algorithm, we therefore produce a tuple of identical LP's for the incoming pair of maps (line 14).

Since the incoming maps are produced by the expression e_m, now that we have the LP l_m for the incoming maps, we can optimize e_m with a recursive call to DSE_e (line 15).

It remains to create a liveness map for the expression as a whole. We do this by meeting the liveness maps from the two components of the expression, that from the anonymous function (lpm_b) and that from e_m (lpm_m) (line 17). Before doing so, we account for the fact the formal parameter of the anonymous function is not free (and therefore, dead) in the expression as a whole (line 16).

7.3 Summary

In this chapter, we described the dead-store elimination algorithm (a whole program context-insensitive, flow-sensitive abstract interpretation) that SCF uses to eliminate dead computations from partially evaluated optimization programs. Compared to traditional dead-assignment elimination, the novelty of this algorithm is that it keeps track of the liveness of individual fields of data structures, through the use of *liveness patterns.* Liveness patterns were originally introduced by Liu *et al.* [40], and Reps *et al.* [53]. In fact, the form of liveness patterns introduced there is more powerful than ours in some ways. In particular, when describing the liveness of recursive structures, they can describe certain infinitely deep but highly regular kinds of liveness patterns that a value may exhibit. Our contribution relative to these works is a pragmatic one: we have noticed that the high regularity that recursive liveness patterns capture is not exhibited by the key recursive structures (maps) in our application. Instead, we introduce a custom liveness pattern for maps that is intended to track the highly irregular patterns of liveness that arise in practice in a tractable and effective manner. We combine these specalized liveness patterns with information computed by the partial evaluation pass to more effectively remove dead map operations.

8. Evaluation

The goal of SCF is to make it easy to produce effective staged compilers. In this chapter, we evaluate the extent to which SCF succeeds, and the contribution of its components towards its performance.

We have implemented a prototype of SCF in Standard ML. We provide an SCF-ML front-end to allow specification of optimizers. We also provide a C front-end (which parses C programs into abstract values) to specify functions, called “input functions” below, whose optimization is to be staged. We have staged pipelines containing three traditional dataflow optimizations: constant propagation, copy propagation and dead-assignment elimination. Compared to our experience hand-writing staged versions of these optimizations for DyC [24], using SCF to automatically implement staged versions these optimizations has certainly been far easier in design, implementation and debugging: the burden of writing optimizations in SCF-ML and invoking the automatic stager is much lower than writing a specialized stager. The remaining issues, which can be evaluated by measurement, are whether automatic staging as in SCF is effective, and what the contributions of the individual techniques in SCF are.

The most direct way to establish the broad effectiveness and applicability of automatic staged compilation à la SCF would be to demonstrate significant reduction in total execution time (including late-stage optimization overhead), for a variety of plausible compiler pipelines over a broad, representative set of benchmark programs and their inputs. We could show, for instance, that SCF can effectively exploit information available only at run time by comparing the total execution time for the program with and without a (SCF-based) run-time optimization stage. Similarly, to show how the quality of SCF output compares to that of hand-staged schemes, we could compare total execution times for programs that use SCF-staged versus hand-staged run-time stages.

From the point of view of this dissertation, such an approach, although comprehensive, is prohibitive in terms of engineering effort. Implementing an entire compiler pipeline in SCF-ML (especially a non-trivial compiler backend), preparing for staged compilation a comprehensive set of C programs as benchmarks, and evaluating the performance of pre-existing staging techniques on these inputs, are each tasks that can take teams of engineers years of work.[1] The goal of this evaluation, therefore, is more modest. We provide strong evidence that at least for a few useful C programs, and for a pipeline consisting of three conventional intraprocedural optimizations that were staged (each at great engineering cost) by hand in a well-known previous system, SCF can produce (at little engineering cost) a staged version of the pipeline capable of yielding significant end-to-end speedups. In this restricted context, we also analyze the contributions of the various techniques comprising SCF to net speedup.

In section 8.1, we describe our evaluation framework and our measures of effectiveness. In section 8.2, we discuss the overall effectiveness of SCF. In section 8.3, we analyze the contribution of individual techniques. We summarize the results in section 8.4.

8.1 Evaluation Framework

We discuss in section 8.1.1 below what system configuration (and related parameters) we measure in order to establish SCF's effectiveness. In section 8.1.2, we discuss the inputs under which these measurements are made.

8.1.1 System Configuration and Parameters Used in Measurements

Figure 8.1 shows how SCF is intended to be used to support two-stage dynamic compilation, where the first three phases of the compilation pipeline are staged. For pragmatic reasons our actual implementation differs somewhat from the configuration shown. The two stages, labeled 1 and 2, are the traditional static-compile-time and run-time stages respectively. The part of the compiler pipeline to be staged (labeled *unstaged pipeline* in the figure) consists, in this case, of our three staged optimizations, i.e., namely, constant propagation (labeled *CnP*), copy propagation (*CpP*) and dead assignment elimination (*DAE*). The stager takes as input this pipeline of optimizations, as well as an abstract value representing the input to the pipeline. In this case, the input abstract value is a pair of which the first element is *f*, the function to be optimized, and the second is *C*, a list map-

1.In work directly preceding this thesis, the author spent two years in a team of three full-time graduate students accumulating one of the most comprehensive sets of benchmarks for evaluating a staged compilation system based on hand-staged optimizations [24]. However, even that work did not entail developing an entire compiler pipeline (although we did modify most of one), or directly testing competing systems.

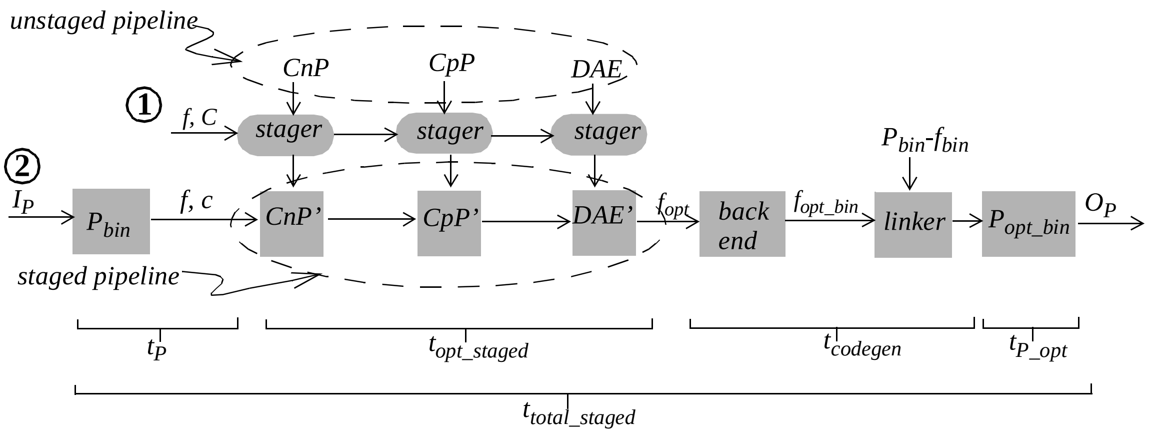


FIGURE 8.1: Staged Dynamic Compilation Using SCF.

ping the arguments of this function to sets of their possible run-time constants as discussed at the end of section 3.2.3. Execution of the stager produces a specialized version of the pipeline, labeled *CnP'* through *DAE'* in the figure.

At run time (stage 2), the binary version P_{bin} of program *P* (which is invoked with some input I_p, and of which function *f* is a part) is executed until *f* is invoked for the first time. At this point, the concrete value *c* corresponding to abstract value *C* is available, since *c* is derived from the actual parameters to be passed to *f*. *P* can therefore yield control to the staged version of the optimization pipeline (labeled *staged pipeline* in the figure) with argument *(f,c)*. The invocation generates an optimized version f_{opt} of function *f*. A traditional back end phase consisting of a scheduler, code generator and assembler performs machine specific transformations on the resulting function and produces a binary version f_{opt_bin} of the function. Finally, a linker links f_{opt_bin} to the rest of the binary image of the program (labeled P_{bin}-f_{bin}) to obtain a new, optimized executable for the program (P_{opt_bin}). All future invocations of function *f* result in execution of the optimized version f_{opt_bin}. For future reference, let t_{total_staged} (as indicated at the bottom of the figure) be the total time, inclusive of all optimization overhead, spent executing the program.

Figure 8.2 shows three baseline configurations that the automatically staged compiler may be compared against. The simplest configuration (figure 8.2(a)) is a conventional single-stage compiler that is used to produce an executable P_{bin}. Since compilation happens entirely before program execution, this configuration ignores the information *c* available

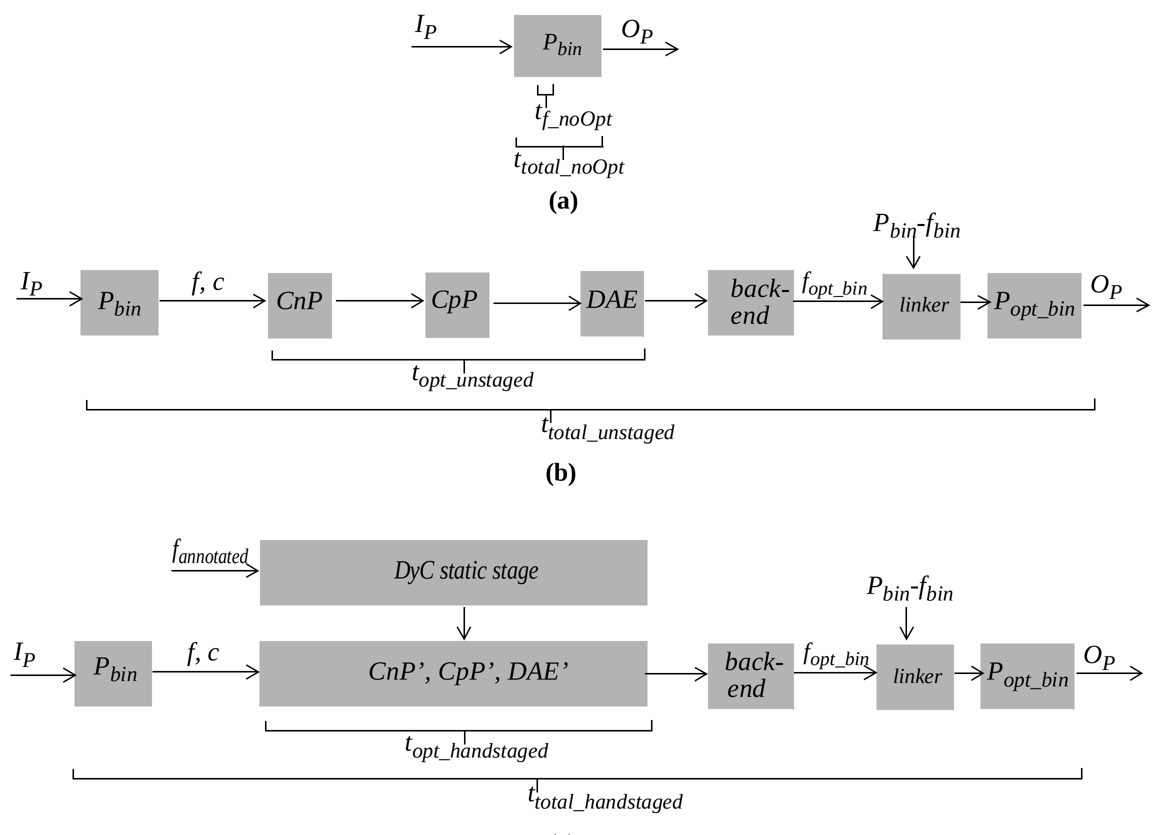


FIGURE 8.2: Baseline Configurations for Evaluating SCF.
(a) Single stage, no dynamic compilation (b) Single stage, all dynamic (c) Two stages, hand-staged dynamic compilation

only at run time. Let t_{total_noOpt} be the time taken to execute P_{bin} on I_P. An alternative (figure 8.2(b)) is to again use a single-stage compiler, but to execute the compiler entirely at run time. A final option (figure 8.2(c)) is to use a hand-staged compiler. Let $t_{total\_unstaged}$ and $t_{total\_handstaged}$ respectively be the total time (including optimization overhead) to execute the program P in the last two cases.

A direct approach to validate the SCF approach to staged compilation compared to no late-stage optimization, unstaged late-stage optimization, and hand-staged late-stage optimization would be to compare t_{total_staged} to t_{total_noOpt}, $t_{total\_unstaged}$ and $t_{total\_handstaged}$ respectively. For the purposes of this dissertation, however, the direct approach has a distinct disadvantage: it requires that the backend and linker phases of the compiler being staged be implemented. Implementing high-quality versions of these phases entails a substantial amount of work, as mentioned previously. We therefore settle for a more indirect demonstration of SCF’s effectiveness, as described below.

As illustrated at the bottom of figure 8.1, the end-to-end execution overhead t_{total_staged} can be broken into the following components: the time t_P spent executing the unoptimized version of the program, the time t_{opt_staged} spent executing the staged parts of the optimization pipeline, the time $t_{codegen}$ spent executing the backend and the linker, and the time t_{P_opt} subsequently spent executing the run-time-optimized version of the program. Staged compilation results in a net speedup if the time t_{loss} lost in late-stage compilation ($t_{loss} = t_{opt\_staged} + t_{codegen}$) is less than the time t_{gain} gained executing the optimized binary P_{opt_bin} instead of the unoptimized P_{bin} ($t_{gain} = t_{total\_noOpt} - (t_P + t_{P\_opt})$). An ideal technique would maximize t_{gain} and minimize t_{loss}.

Regarding t_{loss}, given that we have no way of affecting $t_{codegen}$, we will focus on quantifying the extent to which SCF can reduce t_{opt_staged}. Specifically, we will measure the *staged optimization speedup* ($t_{opt\_unstaged}/t_{opt\_staged}$): since $t_{opt\_unstaged}$ is unaffected by staging techniques, increasing this ratio will clearly imply a decrease t_{opt_staged}.

Regarding t_{gain}, we will focus on the *asymptotic speedup*, $s_a = t_{total\_noOpt}/(t_P + t_{P\_opt})$; again, since t_{total_noOpt} and t_P do not depend on staging technique, maximizing the asymptotic speedup clearly maximizes t_{gain}. In line with convention, we will actually measure the asymptotic speedup of the function f to be optimized dynamically (and not that of the program P as a whole): $s_a = t_{f\_noOpt}/t_{f\_opt}$, where t_{f_opt} is the time spent in the dynamically optimized version of function f, whereas t_{f_noOpt} is the time spent in the unoptimized version.

From the previous two paragraphs, it should be clear that we need to measure four timings: to compute staged optimization speedup, we need t_{opt_staged} and $t_{opt\_unstaged}$, and to compute asymptotic speedup, t_{f_noOpt} and t_{f_opt}.

We measure t_{f_noOpt} by instrumenting the unoptimized binary P_{bin} to record the time spent in function f; t_{f_noOpt} is a fraction of the total execution time t_{P_noOpt} as shown in figure 8.2(a). The remaining three numbers are measured as shown in figure 8.3.

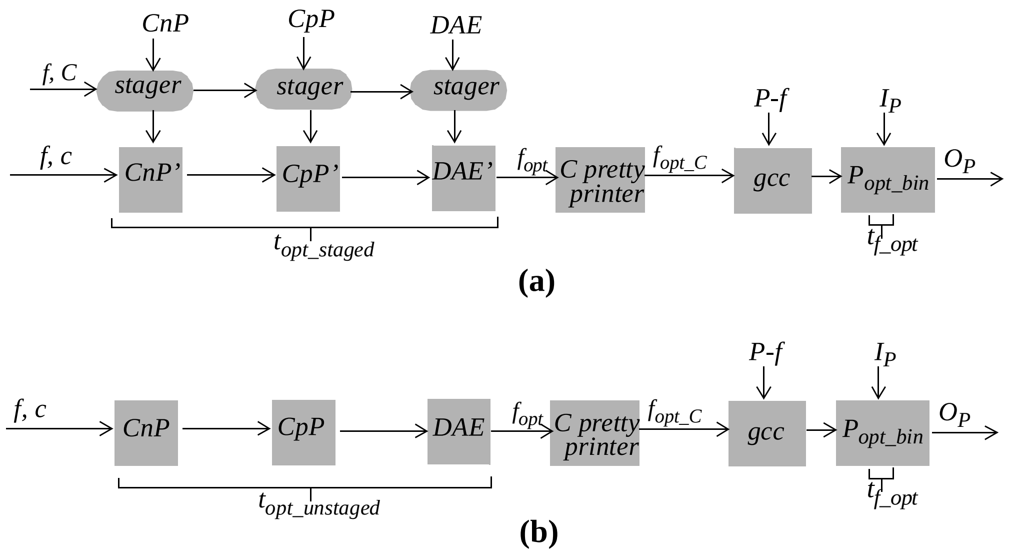


FIGURE 8.3: Configurations Used in this Evaluation.

Figure 8.3(a) takes the pipeline from figure 8.1, and essentially replaces the compiler backend with a full compiler, i.e., *gcc*. The compiler takes as input C source (generated by a pretty printer from SCF-ML abstract values, and written f_{opt_C} in the figure), and produces an executable. Since we have factored out the effect of the compiler backend (by avoiding a direct end-to-end measurement), we are free to insert an arbitrarily slow (but highly effective) backend. We simulate the interleaving of program execution and optimization by directly feeding the input *(f,c)* (which is ordinarily produced by late-stage execution of the program being optimized) to the optimization pipeline. We record the time spent by the staged pipeline generating the input to the backend; this is t_{opt_staged}. Figure 8.3(b) does the same with the unstaged pipeline of figure 8.2(b). The time spent executing the unstaged version of the sequence of three optimizations is $t_{opt\_unstaged}$.

In both cases, the result of the pipeline as a whole is the same (P_{opt_bin}): the staged optimizer produces exactly the same result as the unstaged variant, hopefully at much lower overhead. We instrument this binary to measure time t_{f_opt} spent in function *f*.

Table 3: Inputs to Staged Pipeline

No.	Input Function (*f*)	Description of *f*	Abstract Values (*C*) and Concrete Values (*c*) to Which Staged Arguments of *f* are Bound in the Early, Late Stages
1	`mul_add`	`mul_add` from figure 2.7: computes *a* * *x* + *y*; *a* is fixed at run time	stage1: *a* = 'Int stage2: *a* = 1
2	`mul_add`		stage1: *a* = 0 | 1 stage2: *a* = 1
3	`mul_add`		stage1: *a* = 3 | 1 stage2: *a* = 1
4	`dotproduct`	Finds the dot product of two vectors *v1* and *v2* of size *s*; *v2* and *s* are fixed at run time	stage1: *v2* = 'Int, s = 'Int stage2: *v2* = [0, 1, 7], *s* = 3
5	`dotproduct`		stage1: v2 = 'Int, s = 3 stage2: *v2* = [0, 1, 7], *s* = 3
6	`doconvol`	Convolves 2-D image matrix *i* with a 2-D convolution matrix *c*; *c* is fixed at run time (from the `pnmconvol` program of the `netpbm` library)	stage1: *c* = 'Int stage2: *c* = [[1, 0, 1], [0, 1, 0], [1, 0, 1]]
7	`doconvol_1d`	1-D version of above	stage1: *c* = 'Int stage2: *c* = [0, 1, 0]
8	`main_loop`	Main loop of the Dinero cache simulator; invokes routines for finding, fetching and updating cache entries; cache configuration parameters fixed at run time	stage1: Cache configuration parameters: i-cache size in kilobytes, *i* = 'Int d-cache size in kilobytes, *d* = 'Int i/d cache associativity, *a* = 'Int ... stage2: *i* = 8 *d* = 8 *a* = 1 ...
9	`main_loop_f`	As above, also with cache fetch routine inlined	
10	`main_loop_f_u`	As above, also with cache update routine inlined	
11	`main_loop_f_u_f`	As above, also with cache find routine inlined	

8.1.2 Inputs to the Optimization Pipeline

Having discussed which optimizations we stage and what aspects of the stager we measure, we now discuss the staged inputs to the optimization pipeline. Recall that (as shown in figure 8.3), the pipeline is fed an abstract version *(f,C)* of the optimization inputs at the early stage (an abstract pair of which the first element is the function *f* to be optimized and the list *C* mapping formals of *f* to abstract values), and the concrete version *(f,c)* at the late stage. In the unstaged case, we just use the late stage inputs (*f*,*c*). In this section, we discuss the actual functions *f* and the corresponding constants *C* and *c* used in our evaluation.

Table 3 lists the functions *f* and the inputs *C* and *c*. We use as tests four functions: the very simple `mul_add` function used as an example in chapter 2, and three other functions, `dotproduct`, `doconvol` and `main_loop`, which were used in our previous study on hand-staged optimization [24]. We consider different variants of these functions, along with variants of input abstract value *C* in each case. A combination of a function variant and its abstract value is an input value *configuration*; each configuration is a row in table 3, with eleven configurations in all. Below we discuss the configurations for each function.

1. For the `mul_add` function, we examine three possible values for the early stage input *C*. In configuration 1, all that is known is that parameter `a` of the function will have *some* constant value at run time. In configuration 2, we assume early-stage information that `a` will have value either 0 or 1 in the late stage. In configuration 3, we assume early-stage values 1 or 3 for `a`. The different early-stage input values are included to illustrate that, unlike conventional compilers, the performance of staged compilers depends on the quality of information available at the early stage. In all three configurations, we assume that the late-stage value of `a` is 1.

2. The `dotproduct` function takes two arrays `v1` and `v2` of length `s` as input and computes their dotproduct. In configurations 4 and 5, we assume that `v2` is a known early to be a constant array determined only at run time (in fact, we use annotations of the style used in DyC [24] to indicate that not only is the pointer `v2` a constant, but so are dereferences off it). In configuration 4, we assume that `s` is only known to be *some* fixed integer at run time, whereas in 5, we assume that it is the particular integer 3. In either case, we intend that at run time, the dotproduct loop is unrolled, and the values of `v2` corresponding to each iteration treated as a constant.

3. The `doconvol` function is extracted from the `pnmconvol` program of the `netpbm` [47] image manipulation library. It has two variants.

   The first, configuration 6, convolves input matrix `i` with respect to input matrix `c`. Both matrices are two dimensional. The convolution involves a quadruply nested loop: the outer two levels of nesting iterate over the elements of `i`, whereas the inner two iterate over those of `c`. We assume that `c` is known early to be a constant matrix determined only at run time. We intend that the doubly nested loop over the convolution matrix `c` is unrolled fully at run time.

   The second, configuration 7, is a convolution over one-dimensional matrices `i` and `c`.

In this case, the inner loop is doubly nested, the outer loop over `i` and the inner one over `c`. We intend that the singly nested loop over the convolution matrix `c` is unrolled fully at run time.

4. The `main_loop` function is the inner loop of a cache simulator [27]**.** It loops through the entries from a file containing a trace of memory addresses and cache queries (each query is a *fetch* or an *update*), and simulates performing the query at a given virtual address. A common sub-step for both fetches and updates is the *find* step which computes the physical location in the cache that the given virtual address maps to.

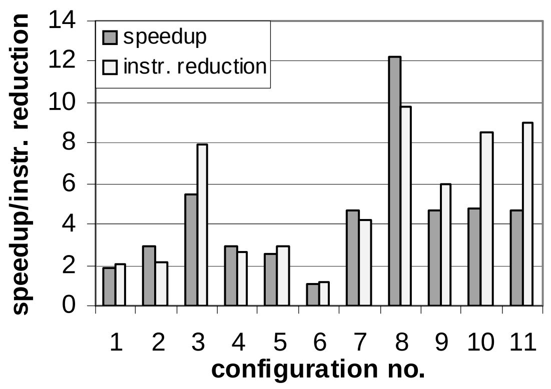
   Typically, the structure of the cache is fixed at the beginning of a simulation run, yielding the possibility that the function can be optimized with respect to this constant value. Further, we can stage the optimization, since we know at stage 1 that the variables representing the structure will have some constant value, as detailed in the rightmost column for configurations 8 through 11.

   The function as originally implemented has three helper functions `fetch`, `update` and `find` that implement the corresponding queries. The run-time constant computations are spread out over these three functions. In order to get maximum benefit from intraprocedural optimization, it is necessary to inline all three callee functions into the parent `main_loop` function. Configurations 8 through 11 differ in the number of these callee functions that are inlined. We choose to examine the four increasingly complex configurations separately in order to gauge how both asymptotic speedup and staged optimization speedup vary with input function size.

8.2 Overall Effectiveness of SCF

In this section, we examine the end-to-end effectiveness of SCF using the two metrics discussed in the previous section: staged optimization speedup (in section 8.2.1) and asymptotic speedup (in section 8.2.2).

All measurements in this section and the next were performed on a lightly loaded 350 MHz Pentium processor with 256MB RAM and 8kB L1 instruction and data caches and a 512kB L2 cache. Times reported are user times.



config #	speedup	reduction in instructions
1	1.9	2.1
2	2.9	2.2
3	5.5	7.9
4	2.9	2.6
5	2.5	2.9
6	1.1	1.2
7	4.7	4.2
8	12.2	9.8
9	4.7	6
10	4.8	8.5
11	4.7	9

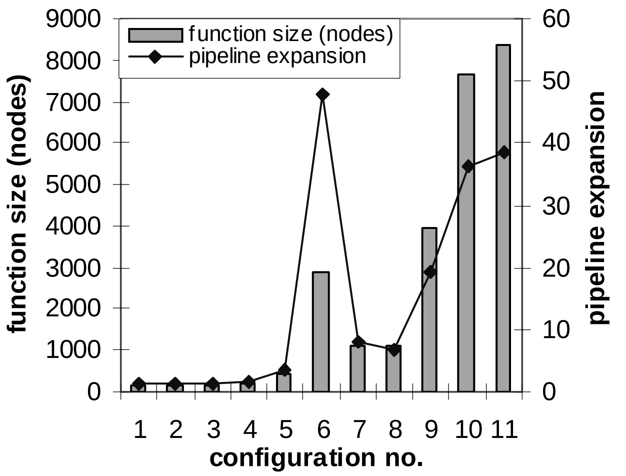
FIGURE 8.4: Speedup of Staged Optimizer Relative to Unstaged Optimizer.

8.2.1 Staged Optimization Speedup

Figure 8.4 shows the staged optimization speedup (i.e., the ratio of the time taken to execute the unstaged version of the compiler to that of the staged one) for each of the 11 input configurations described in the previous section. The figure also shows the reduction in the number of operations executed by the optimizer (i.e., the ratio of the number of operations executed in the unstaged version of the compiler to that of the staged one), as counted by a concrete interpreter for the optimization program. The chart on the left is a graphical representation of the table on the right. A few key points are worth noting.

First, the staged pipelines are significantly (up to an order of magnitude) faster than their unstaged versions in most cases. Thus, automatic staging via SCF is capable of significantly reducing run-time compilation overhead relative to conventional variants of optimization pipelines.

Second, the speedup due to staging may be quite sensitive to the particular abstract values provided at each stage. Comparing configurations 2 and 3, for instance, even though the two configurations differ only in that the former binds argument `a` of function `mul_add` to *0*|*1* and the latter to *1*|*3*, the speedup in the latter case is more than thrice that in the former. The reason is that since the product of any value with *0* is *0*, constant propagating a potential *0* value results in a chain of computations that potentially need to be folded away (to *0*). A staged constant propagator that handles these extra potential cases



conf #	func. size (nodes)	pipeline expansion
1	147	1.4
2	148	1.4
3	148	1.3
4	198	1.6
5	439	3.6
6	2880	47.7
7	1097	7.9
8	1084	6.7
9	3960	19.2
10	7656	36.3
11	8396	38.5

FIGURE 8.5: Pipeline Expansion Factor and Input Function Size.
Function size indicates the size of the function *before* staging. Pipeline expansion factor is the ratio of the size of the specialized optimization pipeline to the unspecialized version.

needs to perform more checks than one that does not, resulting in extra compile-time overhead.

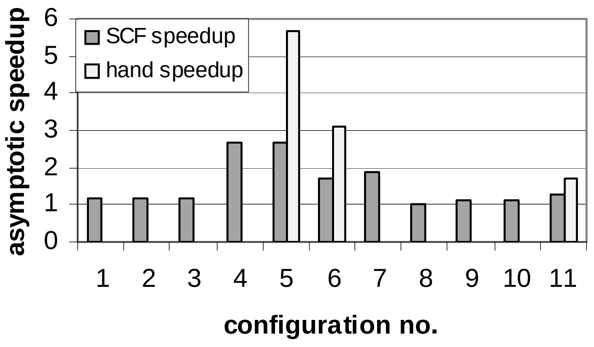
Third, large reductions in number of instructions executed do not always translate to correspondingly large gains in execution time. For instance, comparing configurations 10 and 11 to configuration 8, we would expect to get speedups of roughly 10× in the former cases, as in the latter case. However, the actual speedup is half of that expected. Figure 8.5, which presents the pipeline expansion, i.e., the ratio of the size of the staged pipeline *in the final stage* to that of the unstaged version, provides a possible reason for this anomaly. The staged pipelines for configurations 10 and 11 occupy roughly six times as much space as that of configuration 8. It is very likely that these pipelines perform poorly in the (small) hardware cache on our machine.

Fourth, the size of the staged compiler usually grows linearly in the size of the input program, rather than exponentially, as is the theoretical worst case described in section 6.2.1.2. Figure 8.5 shows that this is roughly true for all configurations but configuration 6 (note that although the function in configuration 6 is roughly three times the size of that in configuration 3, the expansion factor is six to seven times the size). The reason is that the theoretical worst case size increase applies when the input function has deeply nested loops. The convolution routine in configuration 6 contains a four-way nested loop; even bounded unrolling of the nested recursion required for analyzing this loop resulted in noticeable exponential blowup. Configurations 9 through 11 seem to yield sub-linear increase in size; this is mainly because the stager is able to determine at the early stage that the functions being optimized in these configurations are for the most part not amenable to late stage optimization, so that the corresponding residual code is for the most part folded away or removed as dead. Configuration 6 suffers in this respect. In that case, the loop body being unrolled consists of a load of a run-time constant of a value that is then multiplied and added to non-constant values. Given that the actual value of the constant is not known statically, the stager needs to accommodate the possibility that the value could be zero, one or a power of two. The residual code thus maintains analysis and transformation code to handle these possibilities.

8.2.2 Asymptotic Speedup

Figure 8.6 shows the asymptotic speedup of the optimized functions produced at runtime. For three of the configurations (5, 6 and 11), the corresponding ratio for hand-staged systems is included under the label "hand speedup". The numbers for the hand-staged system are taken directly from our previously published results [24] and therefore have caveats as discussed below. Three points are especially worth noting.

The staged optimizations do provide noticeable speedups. In a sense, this is not surprising since prior work on hand-staged systems [24, 14] has already shown that the optimizations in our pipeline are effective in speeding up input functions. However, most optimizations have versions with different levels of aggressiveness, e.g., a constant propagator may or may not reduce multiplies by powers of two to shifts, or fold multiplies by



conf #	SCF speedup	hand speedup
1	1.2	
2	1.2	
3	1.2	
4	2.7	
5	2.7	5.7
6	1.7	3.1
7	1.9	
8	1	
9	1.1	
10	1.1	
11	1.3	1.7

FIGURE 8.6: Asymptotic Speedup of Compiled Functions.

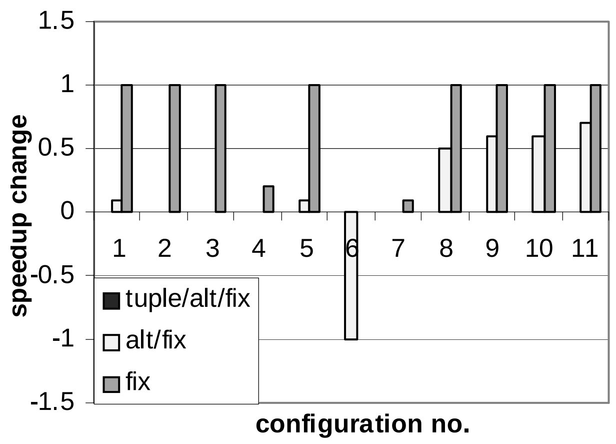
zero to the constant zero. Our measurements demonstrate that the particular SCF-ML specification of the three optimizations in our pipeline is aggressive enough to achieve good speedups on at least three of the input programs previously examined in the literature (`mul_add` is a micro benchmark particular to the current evaluation).

Configurations 8 through 11 illustrate a peculiarity of staged compilation that we discussed in the previous section. Fewer opportunities for late-stage optimization may actually result in greater staged compiler speedup, if the stager is able to determine conclusively in the early stage that the opportunities do not exist.

The speedup due to the hand-staged pipeline is significantly greater than that achieved by SCF. One possible reason for this gap is that the optimizations as specified in the SCF pipeline may not be as aggressive as that in the hand-staged pipelines. Another is that the two sets of speedup numbers were obtained on different hardware systems, and the utility of a given optimization can vary widely across systems. We reproduce the hand-staged numbers here simply to give an indication, however imperfect, of how the automated result compares with hand-staged versions.

8.3 Contributions of Staging Techniques to Compiler Speedup

We now analyze the extent to which various techniques in the stager contribute to the speedup of the staged compiler. The analysis proceeds by disabling a technique (or a group of techniques that work together) and recording the staged optimization speedup with the resulting stager. If the original speedup was *s* and the new speedup *s'*, we report the *fractional speedup change* (abbreviated "speedup change" below), *(s'-1)/(s - 1)*. For instance, if the original speedup were 1.4 and the new speedup 1.2, then the speedup change would be (1.2 - 1)/(1.4 - 1) = 0.5. If the new speedup is below 1, i.e., it is a slowdown, then the fractional speedup change is negative. For instance if *s'* is 0.6 (with *s* as before), then the fractional speedup change is -1. If the new speedup is the same as the old speedup, the speedup change is 1. Intuitively, the speedup change ascribed to a technique is the fraction of the original speedup that is lost by disabling the technique.



config #	orig. speedup	tpl/alt/fix	alt/fix	fix
1	1.9	1	1.1	1.9
2	2.9	1	1	2.9
3	5.5	1	1.1	5.5
4	2.9	1	1	1.3
5	2.5	1	1.1	2.5
6	1.1	1	0.9	1
7	4.7	1	1.1	1.3
8	12.2	1	6.7	12.2
9	4.7	1	3.2	4.7
10	4.8	1	3.4	4.8
11	4.7	1	3.6	4.7

FIGURE 8.7: Contributions from Variants of the Abstract Value Representation.

We present our results in three subsections. The subsections correspond to the benefits derived from abstract value implementation techniques, improvement strategies and dead store elimination respectively.

8.3.1 Contributions from Abstract Value Implementation Techniques

SCF uses a more detailed representation of abstract values (defined in figure 5.2) than most online partial evaluation systems. In what follows, we examine in succession the effect of removing *tuple*, *alt* and *fix* variants of SCF abstract values, of disabling value ID's, and of using a simpler representation than that based on *may* and *must* lists for abstract maps.

8.3.1.1 Removing tuple, alt and fix forms

Figure 8.7 shows the effect of removing various combinations of the *tuple*, *alt* and *fix* variants. By "removing" *alt* and *fix* variants, we mean that the stager is forced to use the abstract value *bottom* where it would ordinarily use one of these variants. By "removing" the *tuple* form, we mean that whenever the stager would ordinarily create a *tuple* abstract value that has even one field that is non-singleton, we replace the whole tuple with bottom; a fully concrete tuple would continue to be represented as before. In effect, we gauge the impact of these variants by using the conservative value *bottom* instead of them. Since many traditional online partial evaluation systems use a three-level abstract value hierarchy consisting of *top*, concrete values and *bottom*, this is an interesting study.

The table to the right of figure 8.7 gives, for each of the 11 input function configuration, the staged optimization speedup achieved by the full-featured version of SCF, the version (labeled "tpl/alt/fix") where all three variants (i.e., *tuple*, *alt* and *fix*) are removed, the version (labeled "alt/fix") where *alt* and *fix* variants are removed, and the version (labeled "fix") where only the *fix* variant is removed. The chart on the left shows the corresponding speedup changes.

Not surprisingly, disabling *tuple* values (along with *fix* and *alt*) leads to speedup dropping to 1 in every case. The reason is that since disabling non-singleton tuples sets them to bottom, then in particular, the abstract input tuple *(f, C)* to the optimization pipeline will be bottom for all eleven configurations. For instance, configuration 1 ordinarily has input *`(int mul_add(){...}, (..., CONSTANT(`**`1`**`)))`* as in figure 3.13(a). If we only represented fully concrete tuples explicitly, then (since this tuple contains non-concrete value ***1***), the configuration would have input ***1***. A *bottom* abstract input value at the early stage means precisely that the stager has no information whatsoever about the abstract input at this stage, so that the specialized value of each optimization is identical to the original one.

Disabling just *fix* and *alt* values produces more interesting results. Intuitively, this restricts early-stage information available to SCF exclusively to fully concrete sub-fields of tuples. We describe the significant effects below.

- There is no significant difference in performance between configurations 2 and 3, which only differ in the *alt* values that parameter `a` is bound to. In the absence of *alt* values, we would simply assume a value of *bottom* for this parameter in both cases. The magnitude of the speedup for configurations 1 through 3 is also close to one: the stager ordinarily produces *alt* forms for almost every statement in the `mul_add` function (figure 3.13); setting these to bottom loses almost all useful information at the early stage. Configuration 5 (optimizing a loop statically unrolled 3 times) also relies on extensive use of the *alt* form because the stager cannot rule out a variety of options; its speedup also therefore drops close to one.

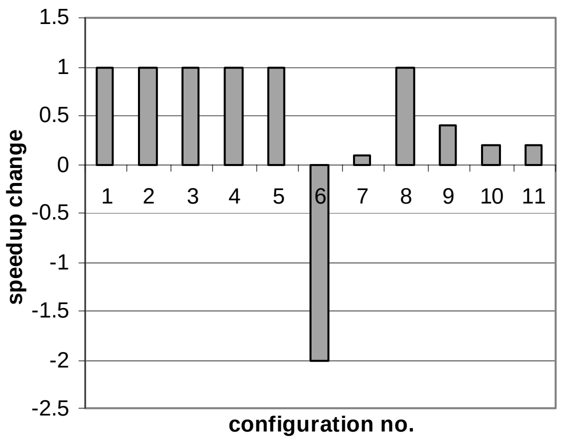
- In cases where the early stage needs to reason about loops being unrolled, and the degree of unrolling is statically unknown (configurations 4, 6 and 7), the stager ordinarily uses *fix* values to represent the early results of unrolling. In all three configurations, the loop to be unrolled dominates the body of the function being optimized, so that setting this *fix* value to bottom results in loss of information about most of the function, and therefore a precipitous drop in stager speedup. Note that although the histogram seems to imply an especially dramatic fall in performance on configuration 6, this is an artifact of the relative fractional speedup change we use: actually, the speedup drops from a modest 1.1 to a small slowdown of 0.9.

- Somewhat surprisingly, in the largest benchmark (configurations 8 through 11) most of the body of the function is provably unaffected by the optimizations, so the lack of *alt* and *fix* forms has much less impact. Of course, here we are making a virtue out of necessity: if much of the body is provably unaffected by optimization then presumably the code produced by late-stage optimization will not be much faster than that without the optimization.

Disabling just *fix* values only has an effect in the configurations where *fix* values are used. As mentioned above, this is in configurations 4, 6 and 7. In these cases, speedup all but disappears. In all other configurations, speedup is unaffected.

8.3.1.2 Dropping Abstract Value ID's

Figure 8.8 shows the effect of dropping ID tags from abstract values. In all programs that require fixpoint analysis of loops, performance drops significantly. The reason is that fixpoint analysis requires abstractly executing an SCF-ML map `equal` operation to test termination of the fixpoint loop (e.g. line 61 of figure 3.7). As explained in section 5.3.7.3, map equality invoked on non-singleton abstract maps will always result in an abstract value *(`true`|`false`)* in the absence of value ID's, so that the termination test of the fix-



config #	orig. speedup	new speedup
1	1.9	1.9
2	2.9	2.9
3	5.5	5.5
4	2.9	2.9
5	2.5	2.5
6	1.1	0.8
7	4.7	1.3
8	12.2	12.2
9	4.7	2.4
10	4.8	1.9
11	4.7	1.7

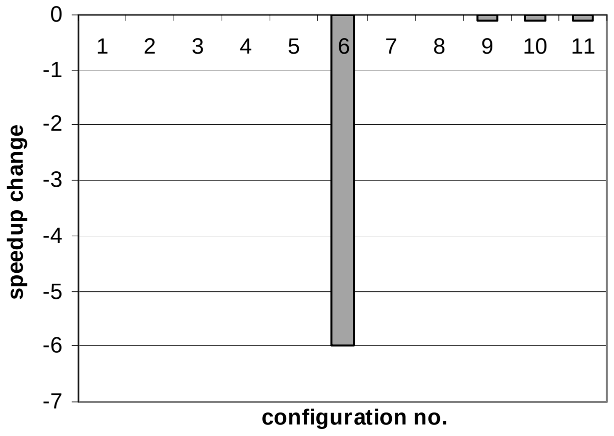
FIGURE 8.8: Contributions from Abstract Value ID's.

point loop may always be false during partial evaluation. At some point, the partial evaluator is forced to declare a possible infinite loop and widen its abstract state.

A somewhat subtle point to note here is that the map `equal` instructions as in line 61 of figure 3.7 are only executed when a `while` loop *possibly in the input function at the late stage* is being analyzed. Consequently, loops in the input function that are to be fully unrolled in the late stage (so that there is no possibility at all that they will be loops at the late stage) do not require abstract execution of the map `equal` test during abstract interpretation at the early stage. In the latter cases, therefore, value ID's, which are intended to facilitate equality comparisons, are not relevant. Configurations 4 and 5 (the fully unrolled `dotproduct` functions) therefore escape the performance penalty. Configurations 6 and 7 contain both unrolled loops and non-unrolled ones, so that accurate equality testing is important in them. All the loops in 9 through 11 are of the non-unrolled variety, so they experience signficant reduction in speedup. As will be seen later, these latter configurations do not lose all speedup because of the complementary role played by SCF's widening techniques. Configurations 1 through 3 contain no loops and are therefore unaffected by value ID's.

8.3.1.3 Dropping "must" and "may" Lists in Maps

Figure 8.9 shows the result of weakening the special representation that SCF uses for representing abstract maps. In particular, the stager is modified so that it does not separately



config #	orig. speedup	new speedup
1	1.9	1
2	2.9	1
3	5.5	1.1
4	2.9	1
5	2.5	1
6	1.1	0.4
7	4.7	1.1
8	12.2	1
9	4.7	0.7
10	4.8	0.7
11	4.7	0.5

FIGURE 8.9: Contributions from *must* and *may* Lists in Abstract Maps.

keep track of the definitely-known mappings using the *must* list (figure 5.2, line 12) and the possible mapping using the *may* lists. Instead, the stager represents a map by a pair of values *(k,v)*, where *k* is the meet of the keys in the original *must* and *may* lists, and *v* the meet of the corresponding values.

Not surprisingly, losing the ability to keep track of mappings precisely destroys speedup in every case. Recall that all our optimizations consist of an analysis step that sets the fields of a map representing the abstract store at each node of the function being optimized and a transformation step that consults this map for each node. Essentially, SCF is unable to simplify the transformation code for any optimization because information for every node in the AST is conflated with that for every other node. Most importantly, this results in map lookup operations that cannot be folded away because they do not return singleton values (both in analysis and transformation code) and map write operations that cannot be eliminated by dead-assignment elimination because the downstream transformation code may read these writes.

Unfortunately, dropping precise map representations does not affect unrolling of the optimization over the incoming AST, so that the specialized optimizations still occupy much more space than the unspecialized version. As with configuration 6, this double whammy of ineffective optimization and code bloat can result in significant slowdowns, not just lack of speedups.

8.3.2 Contributions from Improvement Strategies

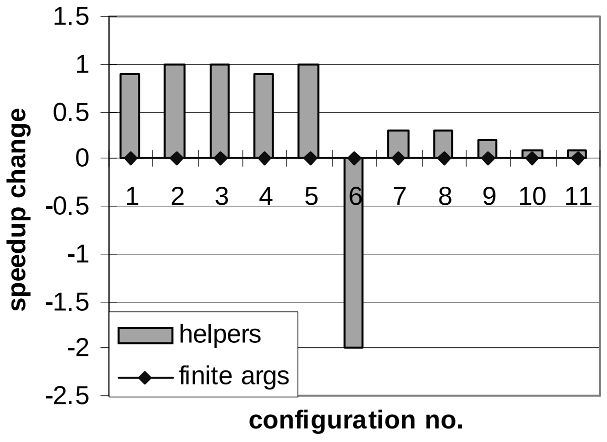
We now turn to the efficacy of improvement strategies used in SCF. The four improvement strategies we examine are function specialization (described in section 6.2.1), expression specialization (section 6.2.2), widening (section 6.3) and non-scalar rematerialization (section 6.4).

8.3.2.1 Function Specialization

To understand the effect of function specialization techniques we performed two experiments. In the first case (labeled "finite args" in figure 8.10), we omitted finiteness analysis, and therefore did not specialize on finite arguments. SCF's use of finiteness analysis is detailed in section 6.2.1.1. In practice, we implemented this by using the finiteness key *bottom* for all functions in the optimization program. In the second case (labeled "helpers"), we omitted specializing non-finite functions, which are typically helper functions, for each calling context. The need for specializing helper functions is detailed in section 6.2.1.2. We implemented this by using a single finiteness key *bottom* for all non-finite functions (i.e., with finiteness patterns of *I* for their arguments).

Not specializing on finite arguments (and therefore not unrolling the optimization over the incoming input function) essentially has the effect of performing no specialization at all. Note that we continue to specialize on *k*-deep context chains (with *k* = 2), so that all function call chains `optimize(...)-> ... f ... -> f'` (which may contain at most two instances of functions `f`) result in unique specialized versions of function `f'`. For instance, we generate two instances of the `analyzeCmd` function in the constant propagation optimization of figure 3.7, one corresponding to the call from the `analyzeFun` function (line 18, figure 3.7), and the other corresponding to a recursive call (e.g. line 31). The second instance contains a recursive call to the first one, since further unrolling is forbidden given the value of *k*.

Not surprisingly, blind specialization based purely on call chains (i.e., not specializing on individual nodes of the incoming AST) is insufficient for producing good specialization: there are tens to hundreds of instance of various types of phrases (expressions, commands, etc.) in any non-trivial function, and (if *k* = *2*) only two specialized versions of the functions that process them. Each version is therefore invoked more times than the (gener-



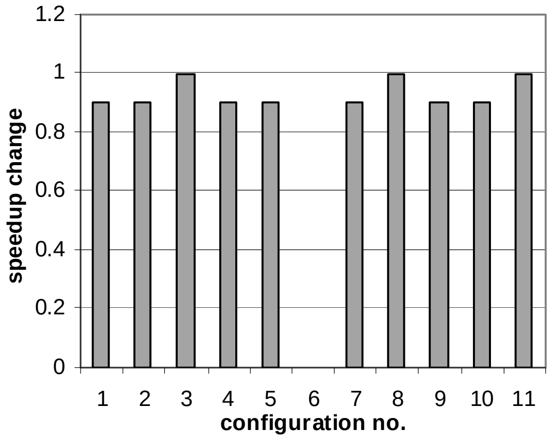
config #	orig. speedup	finite args	helpers
1	1.9	1	1.8
2	2.9	1	2.9
3	5.5	1	5.4
4	2.9	1	2.8
5	2.5	1	2.5
6	1.1	1	0.8
7	4.7	1	2.2
8	12.2	1	4.7
9	4.7	1	1.8
10	4.8	1	1.5
11	4.7	1	1.4

FIGURE 8.10: Contributions from Context-Sensitivity Strategy.

ous) widening threshold of $n_1 = 12$ times that SCF uses, resulting in aggressive widening of argument and return values to *bottom*. The net effect is the same as staging with input abstract value bottom: there is no speedup (or slowdown) relative to no staging, although staging time (not shown) is considerably longer, since widening needs to happen first. As shown in the curve labeled “finite args” in figure 8.10, the result is uniformly a fractional speedup change of zero.

Figure 8.10 also shows the effect of not specializing non-finite “helper functions” in the context of their calling contour key chain. Recall that helper functions are non-finite functions, i.e., ones with no finite arguments. In practice, for the three optimizations we are staging, there are only two helper functions: `meet`, which perform the relevant map meet operation in all three optimizations, and `constAsLatticeElement`, which converts an SCF-ML constant value into a constant propagation lattice value in the constant propagation optimization. Only the `meet` function affects the value of the abstract map threaded through the optimization: `constAsLatticeElement` does not read or write the abstract store. A consequence is that configurations that do not invoke the `meet` function are minimally impacted by not specializing helpers, whereas those that do (in particular, the ones with branches and/or loops) are affected significantly.

The `meet` function is invoked when the incoming function contains a branch. As described in section 6.2.1.2, the case that is especially expensive is when the incoming function contains an early branch followed by many downstream branches. The resulting



config #	orig. speedup	exp. spec.
1	1.9	1.8
2	2.9	2.8
3	5.5	5.5
4	2.9	2.8
5	2.5	2.3
6	1.1	1
7	4.7	4.5
8	12.2	11.7
9	4.7	4.5
10	4.8	4.4
11	4.7	4.6

FIGURE 8.11: Contributions from Expression Specialization.

conflation of abstract stores at branches can result in significant performance loss, as shown in configurations 6 through 11.

8.3.2.2 Expression Specialization

Figure 8.11 shows the effect of disabling expression specialization (see section 6.2.2), i.e., not introducing discriminators to case expressions in order to split the abstract environment under which the body of the case expression is specialized. In our benchmarks, expression specialization does not have a significant effect. Although it is not difficult to come up with examples where expression specialization should make a significant difference, it seems that these opportunities are certainly not ubiquitous across all instances of staged compilation. Note that the fractional speedup change metric exaggerates the effect on configuration 6 in the histogram because the speedup was relatively small to begin with.

8.3.2.3 Widening

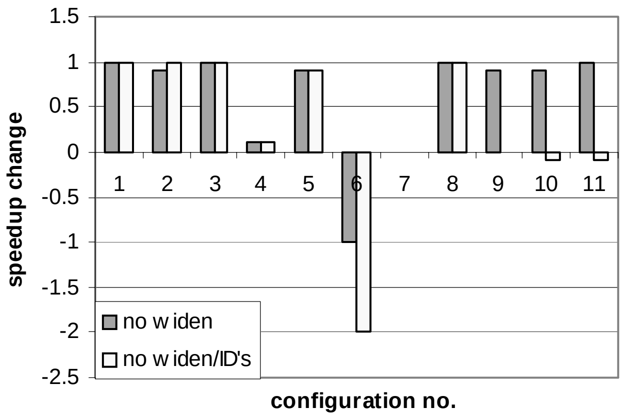
The careful widening strategy in SCF (see section 6.3) ensures that partial evaluation of optimizations terminates in a finite number of steps, while preserving accuracy in common cases. In particular, the widening ensures that maps (which, in general, represent the abstract store that is propagated throughout the optimization) are treated carefully. In the rest of this section, we refer to “careful widening” simply as “widening”.

Widening is essential in SCF, because programs specialized on finite arguments (i.e., "unrolled" over incoming AST's) may still contain recursion. Combined with the fact that the abstract value lattice has unbounded height, this recursion may result in non-termination. In the common case, recursion arises from two sources. First, optimization programs iterate to fixpoint in order to analyze loops in the programs they are optimizing; in SCF-ML the iteration is implemented as a recursive function call to functions such as `analyzeWhile` in the implementation of dead-assignment elimination (figure 3.7). Second, an abstract AST may contain recursive (or *fix*) abstract values, which correspond to infinitely large sub-trees of the incoming AST.[1] "Unrolling" optimizations over such AST's also results in recursive calls (e.g., figure 6.28).

The above sources of recursion make different requirements of widening. Assuming precise equality tests (facilitated by abstract value ID's), the *k*-deep-contour-chain-based specialization in SCF is typically sufficient for fully unrolling the recursive analysis of `while` loops. In this case, we expect each contour to be analyzed less than *k* times, so that the widening threshold for the contour (which is typically greater than *k*) is not exceeded, and widening is not required. When analyzing *fix* forms, however, there is no *a priori* bound on the number of times a contour is analyzed, so that the widening is typically expected to be invoked. Of the configurations we are evaluating, configurations 1, 2, 3 and 8 have no loop in them (so that widening is irrelevant), 5 has an unrolled loop that is, however, unrolled fully at the early stage (so that widening is again irrelevant), 4 has an unrolled loop represented as a *fix* form (so that widening is always relevant), 6 and 7 have both unrolled and non-unrolled loops (widening should be even more relevant when abstract value ID's are disabled in this case), and 9 through 11 have only non-unrolled loops (so that widening is relevant only when abstract value ID's are disabled).

Figure 8.12 shows the impact of disabling widening. The data labeled "no widen" is the case where only intelligent widening is turned off, i.e., dumb widening to bottom is used instead. The data labeled "no widen/ID's" is the case where both widening and abstract value ID's are turned off. Turning off just widening causes catastrophic loss in precision in the configurations (4, 6 and 7) that contain non-unrolled cases. However, its

1. In our examples, *fix* values are a result of unrolling loops at the early stage.



config #	orig. speedup	no widen	no widen/ID's
1	1.9	1.9	1.9
2	2.9	2.8	2.9
3	5.5	5.5	5.4
4	2.9	1.1	1.1
5	2.5	2.4	2.4
6	1.1	0.9	0.8
7	4.7	1.1	1
8	12.2	12.1	12.2
9	4.7	4.5	1.1
10	4.8	4.6	0.8
11	4.7	4.7	0.7

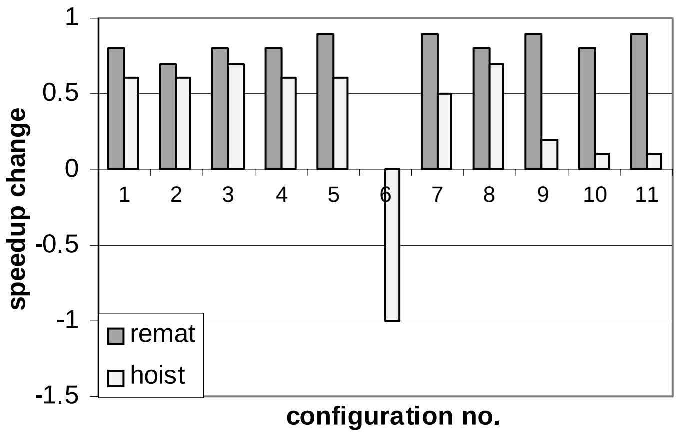
FIGURE 8.12: Contributions from Widening Strategy.

impact is not significant in the configurations that have only non-unrolled loops. In this case, it is only when abstract ID's are *also* turned off, so that widening is forced, that the contribution of widening becomes clear. Comparing configurations 9-11 across figures 8.8 and 8.12, it is clear that widening can compensate to some extent for the loss of precision from discarding value ID's. In figure 8.12 ("no widen/ID's" case) where widening is discarded, speedup change essentially falls to zero, whereas in figure 8.8, where widening (but not value ID's) is retained, we still see a noticeable speedup. Not surprisingly, widening has no significant impact on the configurations (1, 2, 3, 5 and 8) which have no loops.

8.3.2.4 Non-Scalar Rematerialization

Non-scalar rematerialization (see section 6.4) replaces SCF expressions that evaluate to a (possibly non-scalar, e.g., AST nodes) constant value with an expression that synthesizes just that specific constant. Given that these new expressions can themselves be much more expensive than the ones they replace, we need an additional optimization, that of generating them into the text segment at compile time (called "hoisting" below) so that no runtime cost is paid when evaluating them.

Figure 8.13 shows the results (labelled "remat") of disabling non-scalar rematerialization. In summary, disabling rematerialization has a modest effect. This is in line with our observation that much of the time spent in optimization is spent in accessing the abstract store (i.e., writing into and out of maps), and not as much in generating the trees. In many cases, the optimization generates trees by using unchanged parts of the incoming AST by



config #	orig. speedup	remat	hoist
1	1.9	1.7	1.5
2	2.9	2.4	2.1
3	5.5	4.7	4.1
4	2.9	2.5	2.1
5	2.5	2.3	1.9
6	1.1	1	0.9
7	4.7	4.2	2.7
8	12.2	10.2	9.1
9	4.7	4.2	1.6
10	4.8	4.1	1.4
11	4.7	4.3	1.3

FIGURE 8.13: Contributions from Non-Scalar Rematerialization and Hoisting.

reference (for instance, a dead code eliminator may simply reuse entire statements by reference as in line 79 of figure 3.7), so that the original expression is not as expensive as it would be if it re-assembled every sub-tree of the incoming AST.

Figure 8.13 also shows the results (labeled “hoist”) of performing non-scalar rematerialization without the hoisting optimization. In effect, we force the stager to generate all constant values including, in particular, entire sub-trees known at static compile time. Since the number of map operations eliminated remains the same, in most cases we still see a speedup. However, forcing all constant values to be built from scratch (when the original program may have constructed them by references to existing values) substantially reduces the efficacy of staging. The drop is most noticeable in configurations 9, 10 and 11, which process large AST’s, most of which can be proven to be constant (and untransformed) at static compile time.[1] Constructing these AST’s not only adds to the number of operations relative to the version without rematerialization, but also adds substantially to the size of the residual programs, which were already extremely large (see figure 8.5).

Configurations 9 through 11 benefit especially strongly from hoisting. The reason is that, as mentioned previously, configurations 1 through 8, have many potential (but not definite) opportunities for optimization. At the early stage, therefore, the structure of most

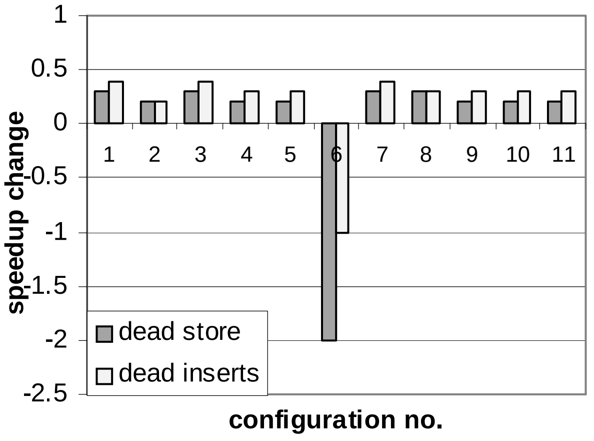
1.As in previous cases, although the *fractional* speedup change for configuration 6 is quite dramatic, the baseline speedup was only 1.1 in that case.

of the late-stage AST is not fully concrete. These parts of the AST are therefore not rematerialized. Correspondingly, the total benefit attributable to intelligent rematerialization is not as high for these configurations as for configurations 9 through 11.

8.3.3 Contributions from Dead-Store Elimination

Figure 8.14 shows the effect (labeled “dead store”) of turning off dead-store elimination (DSE) altogether. DSE is the subject of chapter 7. One of the key innovations of DSE as implemented in SCF, is the use of a written-keys map computed by the partial evaluator that indicates, for each map insert operation in the residualized optimization, the set of possible keys that may be written by that operation. SCF is able to remove map insert operations by checking whether this set of keys has a null intersection with the downstream list of possibly live map keys (section 7.2.5.1). The result of not using the written-keys map is labeled “dead inserts” in figure 8.14.

Two key trends stand out. First, disabling dead assignment elimination leads to significant losses in stager speedup. Surprisingly, however, not all speedup is lost: one may expect that removing the map-operation-eliminating and AST-construction-eliminating effects of DSE would remove all speedup or even result in slowdowns. On examining the specialized code, the primary reason is that a significant number of map read operations are folded away by the partial evaluator because they yield fully concrete results. A lesser effect is that some sub-trees, although unnecessarily constructed, are rematerialized at compile time in the text segment, so that their cost is not paid at run time.



config #	orig. speedup	dead store	dead inserts
1	1.9	1.3	1.4
2	2.9	1.4	1.3
3	5.5	2.2	2.9
4	2.9	1.4	1.6
5	2.5	1.3	1.5
6	1.1	0.8	0.9
7	4.7	2	2.4
8	12.2	4.6	4.9
9	4.7	1.8	2
10	4.8	1.7	2.1
11	4.7	1.6	2.2

FIGURE 8.14: Contributions from Dead-Store Elimination.

Second, not being able to eliminate map insert operations has almost as bad an effect as not performing any dead store elimination. To understand why, recall (see figure 7.1 for an example) that most staged optimizations (and certainly the three actually staged in this study) can be divided into an analysis part, which writes a map from labels in the incoming AST to lattice values, and a transformation part, which reads this map to perform appropriate transformations of the AST. Typically, if for any given node in the AST, the read (in the transformation code) from the map can be folded away, then the upstream write (in the analysis code) into the map becomes dead. If the write corresponding to a node becomes dead, then often all the analysis code that went into computing the written value also becomes dead, so that in summary, the map accesses, analysis logic and AST accessors for a particular node all become dead (and removable). Unfortunately, most of this benefit vanishes if we cannot eliminate the intermediate map write operation. As the figure shows, however, DSE still has some modest benefit even in this case: the “dead inserts” numbers are slightly higher than the “dead store” case. A look at the generated code reveals that many tree-construction and traversal operations that either build or extract parts of trees that are not used downstream (mostly because their results have been rematerialized downstream) can still be avoided.

8.4 Summary

We provide evidence that SCF can produce staged optimization pipelines that can generate significantly optimized code. The generated optimizations can exploit run-time information at substantially lower run-time overhead than their conventional variants. Asymptotic speedup of the generated functions ranges from 1× (no speedup) to 2.7×. Speedup of staged compilation ranges from 1.1× to 12.2×. A pleasing result is that the staged compiler speedup is often high when not much run-time optimization is feasible: in the common case that not much optimization is possible, staged compilation can determine statically that this is so, and avoid paying the run-time overhead of trying to optimize.

Many of the techniques included in SCF have a noticeable effect on its ability to produce faster late-stage compilers. Except for expression specialization, which did not have significant impact on our benchmarks, disabling each of the techniques we examined resulted in loss of at least half the speedup for at least one (and frequently more) of the input configurations. Not surprisingly, some design choices, such as representing at least *alt* variants of abstract values, *must/may* lists in abstract maps, and specializing on arguments (via finiteness analysis), were essential for any speedup on any configuration.

9. Conclusions

In this chapter, we examine the main contributions of this thesis, present a critique of the work and sketch possible directions for future work.

9.1 Contributions

This dissertation makes three main contributions. First, it introduces an architecture for staging individual optimizations and pipelines that requires far less work from compiler writers than traditional approaches. Second, it provides an implementation of this architecture that cleanly and effectively combines two existing powerful but notoriously difficult-to-use techniques, partial evaluation and dead store elimination. Third, it reports measurements of an implementation of this architecture that show that it can yield staged compilers with good performance at very low per-optimization engineering overhead.

SCF unifies and automates a variety of existing approaches to staging individual optimizations. Existing approaches involve writing a special staged version of each traditional optimization to be staged. Staging an optimization therefore requires considerable additional engineering effort specific to that optimization. In this dissertation, we show that it is possible to get many of the effects of the specially designed staged optimizations by applying one generic function (called the *stager*) to generic versions of these optimizations written in a domain specific language (SCF-ML); figure 3.2 illustrates this approach at a schematic level. Since it is far easier to re-implement a conventional design in this domain-specific language than to design and implement staged versions of optimizations, this viewpoint has the potential of making staged optimization more widely usable.

Through the use of a uniform representation (regular tree grammars) to represent early-stage inputs and outputs of optimizations, SCF further shows how the infrastructure for staging a single optimization can be composed with little effort to yield one that can stage pipelines of optimizations. Figure 3.1 captures this approach. Existing approaches specify early-stage information using optimization specific annotation languages, and provide little support for staging pipelines of optimizations: changing the order of optimizations in the staged pipeline either did not make sense (as in swapping compiler backends with intermediate optimizations), or implied writing an entire new staged pipeline corresponding to the new sequence of opportunities. Since most compilers rely heavily on pipelined optimizations for effectiveness, making it easy to construct and use staged pipelines further lowers the barrier to useing staged optimization.

A large literature exists both on partial evaluation and dead-store elimination, the two constituent technologies of the stager. Both techniques are known to be difficult to use. In terms of power and challenge of use, partial evaluation is comparable to theorem proving: many useful problems can be formalized in terms of these techniques, but both techniques are intractable with respect to many common problems that are nominally formulated in terms of them. Good solutions often require extensive use of heuristics, and often user involvement. Similarly, dead-store elimination belongs to a family of techniques, of which alias analysis is the canonical example, that is widely applicable in principle, but for which scalable, effective implementations require domain-specific heuristics. Developing an instance of these two techniques that automatically and effectively applies to a class of useful problems is therefore a significant challenge.

The key technical contribution of this dissertation is a set of implementation techniques (representation choices and custom heuristics) suited for partial evaluation of optimization programs, and for dead-store elimination of residual optimizers produced by the partial evaluator. Below, we list the key implementation techniques, and describe why they are particularly effective when specializing optimizers. Although the particular version of these implementation techniques used in SCF is often novel with respect to traditional designs of partial evaluators, the more important contribution is that taken together, they result in effective staging of intraprocedural optimizations.

- **Optimization specification language.** Many optimizations may be naturally written in a functional language. This allows SCF to require that optimizations are written in a purely functional, first-order subset of ML, thus skirting issues related to side effects and control-flow analysis.

- **Finiteness analysis.** Optimizations tend to be compositional over their input programs. This makes it easier for SCF to determine, via its finiteness analysis, which functions to specialize, and which of their arguments to specialize them with respect to.

- **Regular-tree-grammar based abstract domain.** The compositional nature of transformation functions also motivates the effort involved in using the highly accurate regular-tree-grammar based domain such as the one proposed in this dissertation. For partial evaluation of general programs, having an extremely detailed representation for abstract values is often futile because complex patterns of recursion in the function being interpreted very quickly confuse the partial evaluator; for most non-singleton inputs, the abstract store soon degenerates to the conservative abstract value *bottom* as result. Transformation functions, because they compositionally replace sub-trees of the incoming ASTs with transformed versions, are typically easier to reason about. As a result, they often yield intricate tree-grammars that describe their results.

- **Non-scalar rematerialization strategy.** The compositional structure of optimizations, combined with the fact that only select parts of the incoming AST are transformed, results in large sub-trees of the resulting ASTs being determined statically. The non-scalar rematerialization strategy proposed in this dissertation, in particular the technique (borrowed from the functional language implementation literature) of hoisting constant non-scalars to the text segment, is especially effective here.

- **Built-in abstract maps.** Optimizations tend to use certain data structures extensively, e.g., maps and sets. This enables SCF to provide pre-defined variants of these data structures. The specializer understands their semantics, allowing it to model accurately this large class of complex computations.

- **k-call-instance-deep calling contours.** A common class of recursive calls in optimizations, fixpoint loops to process recursive commands, is known to often terminate within a (typically small) fixed number of iterations independent of the program being optimized. This provides a natural and effective bound on the degree of context sensitivity to be used in analyzing these calls, motivating the k-call-instance-deep specialization approach proposed in this dissertation. This specialization approach differs from more conventional bounded-call-chain based specialization because it uses a bound on the number of times a function is specialized on a particular finite argument: it is the cooperation between the finiteness analysis and the context-sensitivity strategy that provides the desired level of specialization.

- **Highly context-sensitive analysis**. Optimizations are typically not very large (hundreds to thousands of lines, say), and input functions being analyzed do not typically consist of very highly nested loops (this is the worst case that results in exponential space and time blowup), so it is not impractical to use extensive specialization and highly context-sensitive algorithms, as SCF does.

- **Integrated partial-evaluator and dead-store eliminator.** Optimizations use a single datastructure, the abstract store, to propagate the results of analyzing various parts of the incoming AST in a structured manner. In particular, an optimization typically analyzes a sub-tree of the AST, stores the result of this analysis in a slot of the abstract store indexed by the sub-tree, and retrieves the result from the store when transforming the same sub-tree. In the common case that the partial evaluator folds away this last read operation, the dead-store eliminator in SCF is able to deduce that the corresponding write operation is dead, by comparing the set of downstream live keys with the "written-keys" set provided by the SCF partial evaluator for every map-write operation. The communication of these written keys from the partial evaluator to the dead store eliminator is a novel and critical aspect of SCF.

9.2 Critique

The work described in this dissertation can stand improvement along a variety of dimensions:

- **Relevance.** Staged compilation, as highlighted in this dissertation, is aimed at efficiently exploiting information available only at a late stage (such as run time) to optimize programs. The main kind of run-time information described in this dissertation is the value of variables that have run-time constant (or quasi-constant) values. Unfortunately, although strenuous efforts by various groups over the last decade have shown that such value-specific optimization can improve the time to execute a variety of real programs, it still seems unlikely that the vast majority of programs will benefit from it. Further, it is unclear that the performance benefits that accrue are worth the engineering cost of building the staging system. Although the techniques advocated in this dissertation may reduce the incremental cost of building a staged compiler, even the one-time cost of building the stager is still not clearly worth the benefit of increased performance in some, but not most, programs. This begs the question of whether staged compilation is destined to be a niche concept relevant only to a small extremely performance-conscious community with plenty of run-time constants in their programs.

In fact, recent trends in software engineering present an opportunity to increase the scope and relevance of staged compilation. These trends provide both more compelling kinds of run-time information than values of variables, and more compelling analyses and transformations (we give this combination the generic name “rewrites” below) than those that relate to a reduction in raw execution overhead for select C programs. An important emerging class of run-time information is the identity of dynamically linked parts of a program: dynamic assembly of programs is increasingly common in modern programs, and these programs are assembled in a number of stages. An important emerging class of rewrites is the analysis of whole programs to confirm that various soundness and security properties hold, and (if necessary) insert run-time tests to enforce these properties. Another increasingly important class of rewrites (because of the increasing adoption of languages such as Java and C#) is that of optimizations necessary for efficient implementation of high-level languages.

Putting these two trends together, staged compilation should be well positioned to verify whole programs for soundness and security in a staged way: it should be possible to partially evaluate these rewrites (and accompanying translations) with respect to fragments of programs available at the early stage, leaving for run-time execution only parts of the rewrites possibly affected by the structure of the program as a whole. Automatically staging rewrites may be especially valuable, because rewrites for security and soundness are often written by compiler users (as opposed to compiler-writers), who may find it far too technically demanding to hand-write staged versions of the rewrites.

- **Scalability.** Unfortunately, if whole programs are to be staged, the approach detailed in this dissertation is not quite the right one. In particular, one of the results of staging is the original optimization unrolled over the incoming code fragment being analyzed, so that the size of the resulting specialized optimizer increases at least linearly in the size of the incoming fragment. Already, when the fragment is a function of 300 or so lines of code, the specialized optimizations produced are too large to fit in modern processor caches. If the fragment is a program, which can easily be 30,000 to 30 million lines of code, it is possible that the specialized programs will begin to tax the memory requirements of processors.

- **Incrementality.** Because specialized optimizers in SCF are often far larger than the incoming optimizers, applying the stager in more than two stages is not necessarily beneficial. The original intention when designing SCF was that optimizations would be staged incrementally across multiple stages. Each stage would receive as inputs a more precise description (using the tree-grammar formalism) of the optimization inputs than the previous one, and the most specialized version of the optimization from the previous stage. Each stage would produce an even more specialized version of the optimization to be consumed by the next stage. The hope was that, by using the most specialized version of the optimization at any stage, we could avoid re-doing the work that was “folded away” in the previous stage, and thus perform only an incremental amount of work.

  In practice, because the specialized optimizations are so much larger than the unspecialized ones, and because the specialized optimization still traversed large pre-known parts of incoming ASTs to get at newly-known parts, using the much more compact original optimization at later stages is still faster than using the specialized version. Especially if whole programs are to be analyzed using staged optimization, it is critical to develop staging algorithms that pass on much less work (and much smaller structures) for later stages to process.

- **Applicability.** Although SCF-ML is a Turing-complete functional language, and therefore capable of specifying any optimization, the heuristics in SCF are all tuned towards staging optimizations that are compositional, in that nodes in the incoming AST are processed strictly by examining the results of recursively processing its sub-nodes. Unfortunately, certain important conventional optimizations, such as register allocation (where a global graph-coloring problem that is decidedly non-compositional needs to be solved), scheduling (where transformed sub-nodes of ASTs may be moved to be their sibling or cousin nodes) and partial redundancy elimination (with similar behavior to scheduling) have non-compositional structure in their traditional formulation. To allow staging of entire pipelines, it is important to understand how to handle (at least important) non-compositional phases. Two possible directions are to investigate compositional versions of these optimizations (for instance "linear scan" versions of register allocators that perform no global graph coloring optimizations are quite effective), and to investigate special hand-written stagers for just these optimizations (thus abandoning pure automatic staging).

- **Robustness.** Another disadvantage of the expressivity of SCF-ML is that even optimizations that have compositional structure can be written in ways that are difficult to analyze. For instance, users may even define and use their own recursive datatypes to represent the abstract store instead of using the built-in version, all but guaranteeing that no benefit will be derived from staging. As another example, the use of recursive helper functions may also result in widening and catastrophic degradation of staged results.

  Given that SCF relies fairly heavily on the incoming SCF-ML programs conforming to a "stager-friendly" style (which, in its defense, is a very natural style for writing optimizations), it is important to have either an automatic style checker (which ideally recommends how to fix programs with bad style), or to replace SCF-ML with a domain-specific language such that all optimizations specified in it are guaranteed to stage well.

- **Evaluation.** The suite of optimizations and input programs used for evaluating SCF-ML, discussed earlier in this dissertation, is intended to establish the potential of staging realistic problems on a pipeline of conventional optimizations. The suite is undoubtedly too limited, both in terms of optimizations staged and inputs optimized, to comprehensively establish the value of automatic staging. Part of the problem here is the sheer engineering cost of specifying a complete compiler pipeline for staging, and of preparing inputs for these pipelines: in effect, the performance study requires the specification of an entire compiler, and the porting of a variety of substantial programs to this compiler! A comprehensive study would therefore be appealing only if the anticipated benefits of automated staging exceed this cost. In particular, as per the "relevance" bullet earlier, the ability to perform whole-program optimization, correctness verification and security checks may serve as the necessary motivation.

- **Manageability.** The work described in this paper is the latest in a series of staged compilation systems built at the University of Washington [8, 24, 50]. The author of this dissertation was heavily involved in designing and building all three systems. Although the work from these projects has been well received (each system resulted in a publication at a high-quality venue, and the publication on the second system earned an "influential paper" citation), all three projects were only modestly successful in teasing out and developing as free-standing contributions the key technical innovations that enabled them. More bluntly, we produced one major publication per system, not one per major technique in the systems, leading to the question of whether the systems contained innovations that could stand by themselves. Given that the systems were extremely expensive to build (each taking many person-years of effort)[1], it is useful to consider whether anything could be done to increase the cost-benefit ratio of

1. Costs include the amount of labor involved (counting just graduate student time, roughly fifteen person years over seven years), and the size and sophistication of the systems built (easily 200,000 lines of C, Perl, assembler and Standard ML code all told, with more than 30,000 lines of code in SCF, the smallest system).

these complex projects.

The author’s experience with SCF suggests that one of the more effective ways to reduce the cost of system implementation and to focus on technically innovative aspects of such projects is to formalize the problem early, often and compactly. When building extremely large systems, it is critical to formally specify what is being built. Ideally, the formal specification should be abstract and compact enough so that the soundness of the approach is clear, and more importantly, the innovative parts of the system are in sharp focus. It is important to formulate the system as much as possible in terms of existing approaches, so that it is clear how the “innovative” parts truly add to the existing knowledge in the field. Advisors can better gauge from the formalization whether there is enough “meat” to support multiple students and publications. Graduate students should be facile with the formalization before implementing. In fact, it should be possible to implement the system to reflect the specification as closely as possible.

SCF does not have a particularly good formalization in this sense. Specifically, the formalization is not compact or sufficiently abstract; it is certainly difficult to be convinced of the soundness of SCF by looking at its (incomplete) specification in this dissertation! However, specifying SCF as an instance of the existing partial evaluation and dead-store elimination techniques (with their conventional decomposition into sub-problems such as the various improvement strategies), allowed us to both adapt existing work on these techniques, and also to be clear about the significance of the sub-problems in the scheme of things. For instance, given that improvement strategies for partial evaluation can rarely be viewed as substantial free-standing contributions, we made the decision to focus on solutions to these sub-problems (such as finiteness analysis) that were sufficient for SCF to succeed reasonably well, and not to strive for heroic solutions that could be presented as free-standing contributions to partial evaluation theory.

Finally, having used “in anger” a modern programming language (SML/NJ) with extensive support for abstraction, typing and modularization, it is clear to the author that specifying a program in such a language (as opposed, for instance, to C, in which the earlier systems were developed) both forces and aids in developing a formal view of the system.

9.3 Future Work

The critique of the previous section suggests a course for future work. Most importantly, the scope of staged compilation should be broadened to include whole program rewrites, whether those aimed at performance optimization or, perhaps more importantly, those aimed at enforcing adherence of programs to various restrictions, e.g., those for correctness, security, and style. The key staged information to be accommodated would be the structure of the program being assembled. In a nutshell, staged compilation may enable automatic and efficient whole-program checking and enforcement of properties in the face of dynamically assembled programs.

Realizing the above vision requires first that a clearly useful set of rewrites that benefit from whole-program structure, and that cannot be solved by interface matching alone, be identified. Good candidate rewrites would be those critical for efficient implementation of advanced languages, and rewrites for policy enforcement.

Next, staging will need to be more efficient, both in terms of time and space costs. It will be useful to investigate implicit representations of rewrites, which maintain specialized versions of the data structures being interpreted by the rewrites rather than of the code that interprets the data structures. For instance, a staged points-to analysis (or any set-based [26] analysis) may communicate specialized points-to (or more generally, set-constraint) graphs between stages rather than entire "unrolled programs". It will further be useful to investigate incremental representations, where instead of communicating entire specialized rewrite programs (albeit in implicit form) between stages and analyzing the entire specialized input program for the stage, only the "part that does useful work" and the "part that contributes new information" respectively are communicated. Another interesting approach to efficiency could be approximation. SCF produces specialized optimizations that have exactly the same semantics as their unspecialized variants, even if the possible run-time benefit is not very high. It may be possible to systematically transform an optimization into less aggressive variants by using conservative lattice-bottoms if the expected gain from optimization is not very high. Finally, it will be useful for staging to be modular: when two parts of a program that have each been built up in many stages are finally composed, it should be possible to exploit as much as possible the partial results from the two parts.

Given that dynamic compilation will likely always be the key motivator for staged compilation, it is important to understand how the compiler backend (register allocation, scheduling, assembly, codegen, linking) that generates executable code from the above rewrites could be staged. Register allocation and scheduling are two optimizations that can sometimes impact program performance substantially and have aspects somewhat different from conventional dataflow optimizations. Ideally, it should be possible to develop versions of these optimizations that stage fairly well and produce code of acceptable though perhaps not optimal quality. At the very least, it is important to identify and integrate extremely fast unstaged versions of these optimizations, so that the gains of staging are not exceeded by the loss of slow backend compilation.

Finally, it is important to formalize the above work so that both soundness and novelty of the approach are clear. The emphasis here should be on being precise while abstracting away irrelevant detail. The key challenge is to identify the important and general underlying operations, define a compact notation to express them, and specify the entire staged infrastructure in this notation. The result should ideally be something that a practitioner can appreciate and understand within a few days of poring over the specification. Such a specification can serve as an important managerial and motivational tool.

The allure of staged compilation has always been that of having your cake and eating it too: performing high quality optimization at late stages while transferring most of the cost of doing so to the earlier stages. Recent work, including this dissertation, has shed considerable light on realizing this vision, although the broader goal of these efforts, i.e., value-specific dynamic optimization, has been of limited interest. On the other hand, the problem of efficient whole-program analysis of dynamically assembled program is an instance of the staged compilation problem that is of far broader interest. Applying and extending the insights of the staged compilation community to this problem could dramatically increase their relevance: much more cake to have and to eat!

References

[1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 280–290, June 1998.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[3] A. Aiken and B. R. Murphy. Implementing regular tree expressions. In *Proceedings of the Fifth Conference on Functional Programming Languages and Computer Architecture*, pages 427–447, Berlin, West Germany, Sept. 1991.

[4] A. Aiken and B. R. Murphy. Static type inference in a dynamically typed language. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, Jan. 1991.

[5] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994. DIKU report 94/19.

[6] A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[7] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1991.

[8] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, May 1996.

[9] P. Briggs and K. D. Cooper. Effective partial redundancy elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 159–170, June 1994.

[10] C. Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, Mar. 1992. Technical Report STAN-CS-92-1420.

[11] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, CO, Nov. 1999.

[12] A. L. Chow and A. Rudmik. The design of a data flow analyzer. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 106–113, June 1982.

[13] H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. Tree

automata techniques and applications, 1998.

[14] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, Jan. 1996.

[15] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Jan. 1977.

[16] O. Danvy and J. Palsberg. Eta-expansion does the trick. *ACM Transactions on Programming Languages and Systems*, 18(6):730–751, 1996.

[17] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 35–46, May 2000.

[18] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 1995 European Conference on Object-Oriented Programming*, LNCS 952, Aarhus, Denmark, August 1995. Springer-Verlag.

[19] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computing*, 30(7):478–490, July 1981.

[20] R. Fitzgerald, T. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. *Software: Practice and Experience*, 30(3):199–232, Mar. 2000.

[21] C. Flanagan and M. Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 235–248, June 1997.

[22] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, Jan. 1996.

[23] A. J. Glenstrup and N. D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Ershov Memorial Conference*, volume 1181 of *Lecture Notes in Computer Science*, pages 273–284. Springer, 1996.

[24] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 293–304, May 1999.

[25] D. Grove. *Effective Interprocedural Optimization of Object-Oriented Languages*.

PhD thesis, University of Washington, Oct. 1998.

[26] N. Heintze. Set-based analysis of ML programs. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 306–317, Orlando, FL, June 1994.

[27] M. Hill and A. Smith. Experimental evaluation of on-chip microprocessor cache memories. In *Proceedings of the International Symposium of Computer Architecture*, pages 158–166, June 1984.

[28] C. K. Holst. Finiteness analysis. In *Proceedings of the Fifth Conference on Functional Programming Languages and Computer Architecture*, pages 473–495, Berlin, West Germany, Sept. 1991.

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

[30] N. D. Jones, C. K. Gomarde, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, NY, 1993.

[31] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of Lisp-like structures. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 244–256, Jan. 1979.

[32] R. Kelsey, J. Rees, and W. Clinger. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1), Aug. 1998.

[33] D. Keppel. *Runtime Code Generation*. PhD thesis, University of Washington, 1996.

[34] G. A. Kildall. A unified approach to global program optimization. In *Conference Record of the First ACM Symposium on Principles of Programming Languages*, pages 194–206, Oct. 1973.

[35] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[36] C. S. Lee. Finiteness analysis in polynomial time. In M. V. Hermenegildo and G. Puebla, editors, *Proceedings of the Ninth Internatinal Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 493–508, Madrid, Spain, Sept. 2002. Springer-Verlag.

[37] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Conference Record of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–92, Jan. 2001.

[38] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN ’96 Conference on Programming Language Design and Implementation*, pages 137–148, May 1996.

[39] M. Leone and P. Lee. Optimizing ML with run-time code generation. Technical

report CMU-CS-95-205, School of Computer Science, Carnegie Mellon University, December 1995.

[40] Y. A. Liu and S. D. Stoller. Eliminating dead code on recursive data. In *Proceedings of the Sixth International Static Analysis Symposium*, LNCS 1694, pages 211–231, Venice, Italy, Sept. 1999. Springer-Verlag.

[41] J. L. Lo and S. J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 151–162, June 1995.

[42] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[43] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[44] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283, Jan. 1996.

[45] M. Mock, C. Chambers, and S. J. Eggers. Calpa: A tool for automating selective dynamic compilation. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, Monterey, CA, Dec. 2000.

[46] B. R. Murphy and M. S. Lam. Program analysis with partial transfer functions. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 94–103, 2000.

[47] Netpbm web page. ftp://wuarchive.wustl.edu/graphics/graphics/packages/NetPBM/.

[48] F. Noël, L. Hornof, C. Consel, and J. L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, May 1998.

[49] Y. G. Park and B. Goldberg. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 116–127, June 1992.

[50] M. Philipose, C. Chambers, and S. Eggers. Towards automatic construction of staged compilers. In *Conference Record of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2002.

[51] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 109–121, June 1997.

[52] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages, pages 49–61, Jan. 1995.

[53] T. W. Reps and T. Turnidge. Program specialization via program slicing. In *Dagstuhl Seminar on Partial Evaluation*, pages 409–429, 1996.

[54] J. C. Reynolds. Automatic computation of data set definitions. In *IFIP Congress (1)*, pages 456–461, 1968.

[55] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.

[56] E. Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, February 1993. Technical report CSL-TR-93-563.

[57] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, June 1995.

[58] E. Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, May 2000.

[59] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, Jan. 1996.

[60] O. Shivers. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[61] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991. CMU-CS-91-145.

[62] F. Smith, D. Grossman, J. G. Morrisett, L. Hornof, and T. Jim. Compiling for template-based run-time code generation. *Journal of Functional Programming*, 13(3):677–708, 2003.

[63] V. C. Sreedhar, M. Burke, and J.-D. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 208–218, May 2000.

[64] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, Jan. 1996.

[65] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java just-in-time compiler. *IBM Systems Journals*, 39(1):175–193, February 2000.

[66] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[67] D. Tarditi. *Design and Implementation of Code Optimiziations for a Type-Directed Compiler for Standard ML*. PhD thesis, School of Computer Science, Carnegie Mellon University, Dec. 1996. Technical Report CMU-CS-97-108.

[68] D. W. Wall. Register allocation at link time. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 264–275, July 1986.

[69] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, Jan. 1980.

VITA

Matthai Philipose was born in Ernakulam, India. He received a B.S. from Cornell University in 1994, an M.S. from the University of Washington in 1996 and a Ph.D. from the University of Washington in 2005, all in computer science. He is currently a member of the research staff at Intel Research in Seattle.