Towards Automatic Construction of Staged Compilers

Matthai Philipose, Craig Chambers and Susan J. Eggers

Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle, WA 98195-2350 USA

{matthai, chambers, eggers}@cs.washington.edu

Abstract

Some compilation systems, such as offline partial evaluators and selective dynamic compilation systems, support staged optimizations. A staged optimization is one where a logically single optimization is broken up into stages, with the early stage(s) performing preplanning set-up work, given any available partial knowledge about the program to be compiled, and the final stage completing the optimization. The final stage can be much faster than the original optimization by having much of its work performed by the early stages. A key limitation of current staged optimizers is that they are written by hand, sometimes in an *ad hoc* manner. We have developed a framework called the *Staged Compilation Framework* (SCF) for systematically and automatically converting single-stage optimizations into staged versions. The framework is based on a combination of aggressive partial evaluation and dead-assignment elimination. We have implemented SCF in Standard ML. A preliminary evaluation shows that SCF can speed up classical optimization of some commonly used C functions by up to 12× (and typically between 4.5× and 5.5×).

1 Introduction

Information relevant to program optimization 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 [16, 9, 6, 10, 11, 31, 12], and run-time compilers can optimize based on dynamic program behavior [19] or target platform characteristics [5], 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 analyzed) 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 re-loaded, it is run a large number of times. Correspondingly, for each instance of load-time optimization, we expect many instances of run-time optimization. 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 functionality to speed up optimization has proven to be effective in many cases.

We are investigating a complementary approach to speeding up late stages, called *staged optimization*, in which early-stage cycles are used 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), our 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 later-stage parts that performs only the analysis 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

POPL '02, Jan. 16-18, 2002 Portland, OR USA
 ISBN 1-58113-450-9/02/01...$5.00

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 [15**,** 21, 8, 25]. 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 a single optimization i.e. partial evaluation. In spite of the fact that a vast literature exists on staging partial evaluation, building these systems was quite challenging.

Our previous system, DyC [15], usefully staged optimizations beyond partial evaluation i.e. copy propagation and dead-assignment elimination. Our experience convinced us that hand-writing staged versions of optimizations is substantially more difficult and error-prone than writing their unstaged counterparts. The principal 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 run time, 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.

To address this difficulty, we present in this paper an initial proof-of-concept system called SCF (for "Staged Compilation Framework") that can automatically and mechanically construct a staged compiler from the implementation of a single-stage optimizer. The compiler writer simply writes normal program optimizations that (typically) input a program and output a transformed version of the program. 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 be aware that the optimization is to be staged. At any stage, given approximate information about the inputs to the optimization, a compiler *user* (who may be distinct from the *writer*) may feed the optimization and the information to a program called the *stager*, which will automatically produce a version of the optimization specialized for the approximate information, and an approximate representation of the possible results of the optimization on the approximate input.

The design of the system is based on two insights. First, the approximate information, describing both inputs to and results of a staged optimization, may be viewed uniformly as the *set of possible concrete values* assumed by the formal parameters and the return values respectively of the optimization function, at later stages. In SCF, we represent these sets of values using an augmented form of *regular tree expressions* [3]. Second, the effect of a hand-written staged optimization on its approximate input is similar to *systematic specialization of 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 an aggressive online partial-evaluation forward pass composed with a dead assignment elimination reverse pass.

The devil is in the details. For general inputs, it is notoriously difficult [29] to design effective online partial evaluators or dead assignment eliminators (which, as Reps pointed out [27], are very similar to program slicers) that have reasonable termination behavior. Focusing on specializing compiler optimizations has some key advantages over specializing arbitrary programs:

- Optimizations tend to be compositional over their input programs. This allows our finiteness analysis, which determines which arguments to specialize, to be very effective.
- Many optimizations may be naturally written in a functional language. This allows us 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.
- Optimizations tend to use certain data structures extensively, e.g., maps and sets. This enables us to provide pre-defined variants of these data structures. Our specializer understands their semantics, allowing it to model accurately this large class of complex computations.
- A common class of recursive calls in optimizations, fixpoint loops to process recursive commands, is known to often [1] terminate within a (typically very small) fixed number of iterations independent of the program being optimized. This provides a natural and effective bound on the degree of context sensitivity used by the specializer.
- Optimizations are typically not very large (hundreds to thousands of lines, say), so that it is not impractical to use extensive specialization and highly context-sensitive algorithms.

Exploiting these advantages has enabled us to design and build an effective automated staged compilation system.

This paper makes three main contributions:

1. It motivates and formulates the design of a staged compiler in terms of systematic specialization of an unstaged compiler.
2. It describes a set of techniques for specializing compiler optimizations so as to get a substantial speedup.
   Novel aspects of these techniques include a very expressive domain for our online partial evaluator, a simple but effective finiteness analysis over this domain to determine arguments for specialization, a context sensitivity strategy tailored to the concrete behavior of the input program, and a dead assignment elimination algorithm that co-operates with the partial evaluator to enable effective dead assignment elimination through commonly used data structures.
3. It evaluates an implementation of the design, targeted at staged compilation of C programs. The evaluation shows our techniques can produce staged compilers that are several times faster than their unstaged versions.

The rest of the paper is structured as follows. Section 2 presents a brief overview of the design of SCF followed by a detailed example of how our system works. Sections 3 and 4 describe the framework in some detail, with particular attention to novel aspects necessary for effective specialization. Section 5 presents an evaluation of our the framework. Section 6 discusses related work, and section 7 presents a summary and future work.

2 Overview and example

Section 2.1 presents a high-level schematic of SCF. Section 2.2 then presents a detailed example to illustrate inputs and outputs of various parts of the framework.

2.1 High-level description of SCF

Figure 1(a) illustrates the compiler writers' view of SCF. They write a set of optimizations O_i some subset of which may be sequenced to form a conventional optimizing compiler pipeline $O_1..O_n$ that takes in a program P to produce an optimized program P_{opt}. Each optimization O_i is a program transformer that takes in an arbitrary ML data structure P_i (typically a tree representation of the procedure to be optimized and possibly additional context information about the properties of the procedure's formal parameters, the target machine characteristics, or the program's class hierarchy) and produces a transformed ML data structure P_{i+1}. Writing these optimizations should be the same amount of work as writing an optimization pipeline in a regular unstaged compiler.

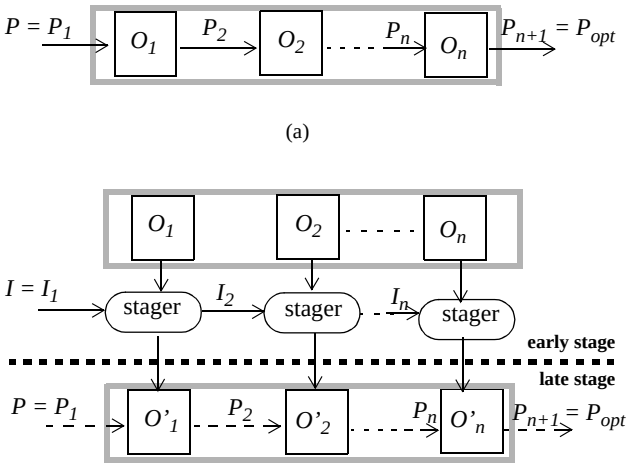

Figure 1: High-level views of SCF

Figure 1(b) illustrates the compiler users' view of staging an initial pipeline of optimizations $O_1..O_n$ with respect to a partial description I of any of its eventual inputs P, to produce a specialized pipeline $O'_1..O'_n$. The partial description I defines the set of possible inputs (i.e., programs and context information) on which the specialized pipeline might be invoked. The specialized pipeline can then be run on any input P that is a member of the set described by I, to produce a corresponding optimized program P_{opt}.

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 I_i have been; the work that remains will finish the optimization when it is finally given the complete program and context information P_i. When staging a whole pipeline, a pipeline of specialized optimizations is produced. The specialized pipeline $O'_1..O'_n$ can be run just like the original pipeline $O_1..O_n$, with exactly the same result P_{opt}, as long as its input P is a member of the set of expected inputs described by I.

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 I_i of the optimization's possible inputs to produce the *source code* of the specialized optimizer O'_i and a partial description I_{i+1} of the optimization's possible outputs. In the second stage, the specialized optimization pipeline $O'_1..O'_n$ can be *run* on an input program P to produce an optimized program P_{opt}.

2.2 An example

We show in this section how SCF may be used to stage a compiler containing $n = 3$ optimization phases, with constant propagation, copy propagation and dead-assignment elimination as optimizations O_1, O_2, and O_3. We will apply this compiler to staged compilation of the function `mul_add` in figure 2. Say this function is in file `mul_add.c`. Also, for concreteness, say the early stage is separate compile time and the late stage is run time.

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

Figure 2: Function to be optimized

2.2.1 Inputs to SCF

SCF has two kinds of inputs: the concrete optimizations O_i provided by the compiler writer(s), and the input information I_1 about the function to be optimized, provided by the compiler user.

The compiler writer must write the three optimizations in the pipeline in the SCF-ML language. Figure 3 shows how dead-assignment elimination may be written in SCF-ML. We assume that this module is defined in a file `dae.scf-ml`. The optimization is implemented as an analysis pass (function **`analyze_fun`**), followed by a transformation pass (function **`transform_fun`**). The analysis function computes, using a threaded set (`lSet`) of live variables, a map (`aMap`) which indicates whether each assignment in the incoming AST is live or dead. The transformation consults the map to prune out the dead assignments.

As the figure shows, the optimization may be specified quite naturally using SCF-ML. This has been our experience with the two other intraprocedural dataflow-based optimizations we implemented as well. SCF-ML provides built-in support for map and set datastructures. The `Set` and `Map` functor applications in the figure serve merely as syntactic sugar to facilitate introduction of the corresponding pre-defined data structures while ensuring that the optimization remains a valid ML program.

For purposes of this example, we assume that the constant propagation optimization (implemented, say, in module `CoP` and file `const_prop.scf-ml`) has a slightly non-traditional interface. Say the optimization uses the traditional constant-propagation lattice with elements that belong to the sum type `lattice_val = non_const | const of int | unknown`. Say, however, that in addition to the body of the function f to be optimized, the entry function `CoP.optimize` of the constant propagator takes as argument a bindings list 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., `CoP.optimize: AST.Fun * (CoP.lattice_val list) -> AST.Fun`.

Finally, we assume the copy propagation pass is implemented as a conventional function-to-function transformer in module `CyP`, in file `copy_prop.scf-ml` i.e., `CyP.optimize: AST.Fun-> AST.Fun`.

We now turn to how the compiler *user* stages a pipeline containing the above three optimizations. The user is interested in compiling and running a C program, of which the `mul_add` function of figure 2 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 infrequently 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 $O_1..O_n$ 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 4 shows the ML expression the user would have to evaluate (at static compile time) to achieve this end. The expression essentially constructs a representation `I1` of the early-stage partial description of the value of the argument to the first optimization in the pipeline i.e. constant propagation, parses in the SCF-ML programs corresponding to the three optimizations, and uses the stager on the optimizations `Oi` in a sequential manner to produced specialized optimizations `Oi'`. Note that the values `I2` and `I3`, which represent the possible results of executing optimizations `O1` and `O2` are used as inputs to stage the next optimization in the pipeline. Finally, the optimization programs output by the stager are written back to disk.

```
structure DAE = struct
open AST    import declarations of input program representation
datatype Liveness = dead | live
(* LiveSets hold the set of live variables at each point *)
structure LiveSetMod = Set(type value=Var)
type LiveSet = LiveSetMod.set
(* AssignMaps record whether each assignment is live or dead *)
structure AssignMapMod = Map(type key=Label
                             type value=Liveness)
type AssignMap = AssignMapMod.map
fun optimize(f:Fun):Fun =
  transform_fun(f, analyze_fun(f))
and analyze_fun(func(_,_,c,_):Fun):AssignMap =
  let val (_, aMap) =
    analyze_cmd(c, LiveSetMod.empty, AssignMapMod.empty)
  in aMap end
and analyze_cmd(c:Cmd, lSet:LiveSet, aMap:AssignMap)
              :(LiveSet * AssignMap) =
  case c of
    assign(v, e, lbl) =>
      let val lv = if LiveSetMod.member(lSet, v)
                   then live else dead
      in (analyze_expr(e, LiveSetMod.delete(lSet, v)),
          AssignMapMod.insert(info, lbl, lv)) end
  | seq(c,c',_) =>
    let val (lSet,aMap) = analyze_cmd(c',lSet,aMap)
    in analyze_cmd(c,lSet,aMap) end
  | return (e,_) => (analyze_expr(e, lSet), aMap)
  | while_do(e, c, _) =>
      let val (lSet', aMap') = fix(lSet, aMap, e, c)
      in (analyze_expr(e, lSet'), aMap') end
  | ... other Cmd cases here ...
and analyze_expr(e:Expr, lSet:LiveSet):LiveSet =
  case e of
    var_ref(v, _)   => LiveSetMod.add(lSet, v)
  | primop(op,es)   => analyze_exprs(es, lSet)
  | ... other Expr cases here ...
```

```
and analyze_exprs(es:Exprs, lSet:liveSet):LiveSet=
  case es of
    exprs(e,es)  => analyze_exprs(es,analyze_expr(e))
  | exprs_none   => lSet
and fix(live_fix:LiveSet, live_entry:LiveSet,
        assigns:AssignMap, test:Expr, body:Cmd)
      :(LiveSet * AssignMap) =
  let val live_head = analyze_expr(test, live_fix)
      val (live_back, assigns1) =
        analyze_cmd(body, live_head, assigns)
      val new_live_fix = meet(live_entry, live_back)
  in
    if LiveSetMod.equal(live_fix, new_live_fix)
    then (live_head, assigns1)
    else fix(new_live_fix, live_entry,
             assigns, test, body)
  end
and meet(live1:LiveSet, live2:LiveSet):LiveSet =
  LiveSetMod.union(live1, live2)
and transform_fun(func( fname, formals, c, lbl):Fun,
                  aMap:AssignMap):Fun =
  func(fname, formals, transform_cmd(c, aMap), lbl)
and transform_cmd(c:Cmd, assigns:AssignMap):Cmd =
  case c of
    assign(v, e, lbl) =>
      case AssignMapMod.find(assigns, c) of
        SOME dead => skip(lbl)(* replace with empty cmd *)
      | SOME _ => c            (* leave unoptimized *)
      end
  | seq(c,c',lbl) =>

seq(transform_cmd(c,aMap),transform_cmd(c',aMap),lbl)
  | while_do(e, c, lbl) =>
      while_do(e, transform_cmd(c), lbl)
  | return _ => c
  | ... other Cmd cases here ...
end (* DAE structure *)
```

Figure 3: Implementation of dead-assignment elimination in SCF-ML

One part of figure 4 requires further explanation. The call to `toAbsValue` converts a concrete ML value containing the special construct `SomeInt` into an abstract value `I1` which represents a set of concrete values. `SomeInt` should be considered a special 0-ary variant of type `int`. In this case, the set of concrete values represented by `I1`, written |`I1`| is {`(P1, [CoP.non_const,CoP.non_const,CoP.const`(i)`])` | i ∈ Integers}, where `P1`, as defined in figure 4, is simply an ML data structure representing the function of figure 2.

At run time, when the C program containing `mul_add` is about to invoke it with some concrete value of argument `a`, it first invokes

```
let val P1: AST.func = AST.parse "mul_add.c"
    val I1: AbsValue = toAbsValue
                         (P1, [CoP.non_const,
                               CoP.non_const,
                               CoP.const(SomeInt)])
    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', I2) = stager (O1, I1)
    val (O2', I3) = stager (O2, I2)
    val (O3', _ ) = stager (O3, I3)
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 4: Invoking the stager at static compile time

the specialized optimizers (written to the files suffixed "`.staged.scf-ml`" in figure 4) by calling the ML-function of figure 5 with the value of `a` as argument, and then jumps to the array of machine code returned by this function call.

A few points are worth noting about the function of figure 5. First, note that it is a function specifically generated (at static compile time) to allow its caller to optimize function `mul_add` with respect to the late-stage value of `a` and the pipeline discussed above. This "stub-function" is typically generated as part of the expression in figure 4 (but not shown there for clarity). Second, although for clarity we show the run-time 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 so that run-time parsing could be avoided. Third, note that the last action of the parser is to generate machine code for the optimized `mul_add` function via the `codegen` function. This illustrates a useful level of flexibility provided by our framework: a staged pipeline may be preceded or succeeded by unstaged optimizations. Finally, we should clarify

```
fun optimize_mul_add_on_arg_a(a:int): code array =
let val P1  = AST.parse "mul_add.c"
    val P2  = CoP.optimize
                (P1,[CoP.non_const,CoP.non_const,
                     CoP.const a])
    val P3  = CyP.optimize P2
    val Popt= DAE.optimize P3
in CGen.codegen Popt end
```

Figure 5: Run-time interface to staged optimizers

```
(* I2: After staged constant propagation on I1 *)
int mul_add(int x, int y, int a) {
  int u = x | (x * 'Int);                    (a)
  int w = u + y;
  return w; }
(* I3: After staged copy propagation on I2 *)
int mul_add(int x, int y, int a) {
  int u = x | (x * 'Int); (* cmd 1 *)       (b)
  int w = (u | x) + y;    (* cmd 2 *)
  return w; }             (* cmd 3 *)
```

Figure 6: Structure of abstract result values

that the language inter-operability support required for this exchange of values between a C program and its ML compiler is not yet integrated into SCF.

2.2.2 Structures produced by SCF

To get a better feel of how SCF-ML works, we now examine values produced by SCF when it is used. Figure 6 presents the structure of values `I2` and `I3` of figure 4. In practice, these values are ML values of the `AbsValue` type defined in the next section: figure 6 is a human-readable representation of these values. `I2` is an abstract value that, intuitively, represents the set of ASTs that result from concretely executing the constant propagation optimization on each input value in |`I1`| above. `I3` is related similarly to `I2`. In what follows, we assume that the constant propagation phase folds all expressions in the input program that evaluate to constant values as usual, and that it also performs the extra simplification step of replacing all expressions `1*e` (or `e*1`) with `e`.

Now consider the input value `(P1, [CoP.non_const, CoP.non_const, CoP.const(1)])` ∈ |`I1`|, i.e., the possibility that at the late stage, argument `a` has constant value 1. The result of constant propagation on this input would be the same AST as `P1`, except that the expression `x*a` of figure 4 would be folded to `x`. On the other hand, any other input `(P1, [CoP.non_const, CoP.non_const, CoP.const(`i`)])` s.t. i ∈ Integers-{1} would result in an output program where `x*a` is folded to {`x*i` | `i`∈ Integers-{1}}, which is contained in the set {`x*i` | `i`∈ Integers}. This latter set is written `x*'Int`. Finally, the expression that results from combining the two cases (i.e. `a` = 1 and `a` <> 1) is written as `x|(x*'Int)` in figure 6(a).

Figure 6(b) represents, in turn, the effect of performing copy propagation on the ASTs in |`I2`|. In the case that the incoming AST has assignment `int u = x`, the copy will propagated to the later use of `u`. Otherwise, no copy will be propagated. The two alternatives are captured by the transformation of expression `u+y` in figure 6(a) to `(u|x)+y` in figure 6(b).

We now turn our attention to the specialized optimizers produced by SCF. Figure 7 shows the specialized optimizer `O3'` that results from specializing the dead-assignment elimination pass `O3` (which was specified in figure 3) with respect to the abstract value `I3` (specified in figure 6(b))[*].

Intuitively, the specialized optimization is the result of making a specialized version of the appropriate analysis and transformation function from figure 3 for each AST node in the input in figure 6(b), and running a dead-assignment elimination pass on the resulting SCF-ML program. In figure 7, we use comments in bold to indicate the correspondence between specialized code fragments and commands from figure 6(b). We precede each sequence of code with a comment with the number (see figure 6(b)) of the command the code is supposed to analyze or transform. Since the analysis part of dead-assignment elimination is a reverse analysis, the code

[*] Aside from variable renaming and removal of some extraneous copies between temporaries, this is the code produced by SCF.

```
fun optimize f =
let (* Specialized version of analyze_fun *)
    val func(_,_,c,_) = f
    val (lSet, aMap) =
          (LiveSetMod.empty, AssignMapMod.empty)
    val seq(c1_2,c3,_) = c
  (* cmd 3 analysis(body folded away) *)
  (* Eliminated dead code in italics below *)
    val return(e0) = c3
    val var_ref(v0,_) = e0
    val lSet = LiveSetMod.add(lSet,v0)
    val seq(c1,c2,_) = c1_2
  (* cmd 2 analysis *)
    val assign(_,e1,_) = c2
    val primop(_,es0,_) = e1
    val exprs(e3,_,_) = es0
    val var_ref(v1,_) = e3
    val lSet = LiveSetMod.add(lSet, v1)
  (* cmd 1 analysis *)
    val assign(v2,_,lbl) = c1
    val lv = if LiveSetMod.member(lSet, v2)
             then live else dead
    val aMap = AssignMapMod.insert(aMap, lbl, lv)
  (* Specialized version of transform_fun *)
    val func(fname, formals, c, lbl) = f
    val seq(c1_2,c3,lbl1_2_3) = c
    val seq(c1,c2,lbl1_2) = c1_2
    val assign(_,_,lbl1) = c1
  (* cmd 1 transform *)
    val temp_c1 =
        case AssignModMap.find(aMap,lbl1) of
          live => c1 | dead => skip(lbl1)
  (* cmd 2 transform (body folded away)*)
    val temp_c2 = c2
    val temp_c1_2 = seq(temp_c1,temp_c2,lbl_1_2)
  (* cmd 3 transform (body folded away) *)
    val temp_c3 = c3
    val temp_c1_2_3 = seq(temp_c1_2,temp_c2,lbl_1_2_3)
  in func(fname,formals,temp_c1_2_3,lbl) end
```

Figure 7: Dead-assignment elimination after specialization

analyzing commands is produced in the reverse order to which the commands appear in the input program.

The specialization enables two important classes of optimizer instructions to be eliminated:

1. Since a separate specialized fragment of code is produced for each node, the case test on node type may often be folded away. For instance, the code for analyzing each command (function `analyze_cmd`) in figure 3 begins with a case statement selecting among the different command variants. However, command 2 of figure 6(b) is known to be an assignment command. Consequently the code for analyzing command 2 begins with the unconditional binding `val assign(v,e,_) = c2`, the case statement having been folded away. Such folding is a standard effect of partial evaluation.
2. Instances of operations on data structures (variables `lSet` and `aMap` in our case) representing the abstract store may be eliminated if the results of the operation are not used downstream. In figure 7, one sequence of instructions thus eliminated is in bold italics. To see why this sequence is dead, note that since this fragment of code is specialized to the command `return w;`, the operation `LiveSetMethod.add(lSet,v0)` on `lSet` adds the variable `w` to `lSet`. Even though the set `lSet` is accessed twice later (via `add` and `member` operations, marked in bold in the figure), none of these operations depends on the presence of variable `w` in `lSet`. The `add` and `member` operations are on variables `u|x` and `u` respectively, and so do not depend on the presence of `w`

```
fun stager(p:Program,v:AbsValue):
        (Program, AbsValue) =
let (cs:(SCFLabel, AbsValue) map, p',v') = PE p v
in (DAE p' cs,v') end
```

Figure 8: High-level structure of the stager

in `lSet`. The `add` operation can therefore be eliminated. The first two operations in the sequence are clearly dead if the third is.

The italicized code is only an example of the dead assignment elimination required to obtain figure 7. For instance, the analysis of figure 3 performs an update of the live-variables set at every use of a variable in the incoming function. Even though commands 2 and 3 contain five uses of the variables `u`, `x` and `y` (see figure 6(b)), the specialized analysis code for these commands contain just one update of the live-variables set (all other updates are found to be dead).

This form of pruning cannot be effected by standard partial evaluators, and requires a novel flavor of dead assignment elimination.

3 The partial evaluator

As described in the previous section, the stager performs partial evaluation followed by dead-assignment elimination. Figure 8 specifies its high-level structure. `PE` is the partial evaluator and `DAE` is the dead assignment eliminator. Types *Program* (optimization programs written in Core SCF-ML) and *AbsValue* (abstract values), which have so far been described intuitively, will be defined precisely below. The role of the extra map `cs` will also be explained. The rest of this section describes the key aspects of `PE` and section 4 describes `DAE`.

3.1 Overall structure

The partial evaluator in SCF is an online partial evaluator. Given a program and an abstract value (representing a possible set of concrete values of the arguments of the program), online partial evaluation (PE) abstractly interprets (with a collecting semantics; we call this abstract interpretation *abstract execution* below) the program on the abstract value. For each subexpression *e* abstractly executed in some abstract environment *E*, in addition to the abstract value *v* that would be produced by abstract execution, PE also produces a *residual* subexpression *e'* that produces the same result as *e*, under any concrete environment that is an instance of *E*. Typically, this is achieved by "folding away" *e* into the constant expression $|v_0|$ in the case that *v* represents the singleton set of concrete values $\{v_0\}$.

Traditionally, the result of the online partial evaluator is a residualized program and an abstract value representing the result of abstract execution. SCF extends this by recording the value of each residual subexpression it generates. The resulting map from expression labels to abstract values is one of the outputs of the SCF partial evaluator. This map is named `cs` in the definition of the `stager` function above. The map is useful for precise dead-assignment elimination, as will be explained in section 4.

Aside from the residualization strategy which decides what code to fold and how, the challenge of partial evaluation is essentially that of accurate abstract execution. The two issues key to effective online partial evaluation in SCF are the same as those for accurate abstract execution:

- designing a domain that is expressive enough to represent certain sets of concrete values accurately, but on which operations are fast in the common case, and
- achieving a degree of context sensitivity in the abstract execution of function calls sufficient to maintain accuracy while guaranteeing termination on all inputs, and achieving termination quickly in the common case.

$P \in Program$ ::= $g1,..., gn$
$g \in FunDef$ ::= $f\ x = e$
$e \in Expr$ ::= $x \mid c(e1,...,en) \mid c^{?}\ e \mid c^{-k}\ e \mid$ *if e1 e2 e3* $\mid$ *let x = e1 in e2* $\mid f\ e \mid p\ e$
$p \in Primop$ ::= *map_insert, map_find, map_equal,* + , -, ...
$f \in FunName$,
$c \in Constructor = FunName \cup Int \cup Bool$,
$x \in Variable$

Figure 9: Core SCF-ML

In what follows, we discuss in detail how our design addresses these issues. In section 3.2, we introduce a simple core language to which SCF-ML may be de-sugared. In section 3.3, we present the augmented regular tree expression domain used as the abstract domain in SCF. In section 3.4, we present SCF's context-sensitivity strategy.

3.2 Core SCF-ML: The input language for PE

SCF-ML programs are desugared into the *Program* datatype of the core language of figure 9. It is a fairly standard untyped, first order, call-by-value, purely functional language. $c(e1,...,en)$ generates *n*-ary tuples with tag *c*. $c^{?}\ e$ checks whether the value *v* that *e* evaluates to is tagged with *c*, and $c^{-k}\ e$ projects the *k*'th field of *v* if *e* evaluates to a tuple tagged with *c* (and is undefined otherwise). The language provides built-in primitives for map operations. All expressions are labelled with labels of type *SCFLabel = Int* (not shown in figure 9). Given Core SCF-ML expression *e*, *labelOf e* returns *e*'s label.

The concrete execution domain for this language is the Herbrand Universe *H*, given by the smallest set satisfying the equation:

$$H = C_0 \cup \{c(t_1,...,t_{arity(c)}) \mid c \in Constructor \wedge t_i \in H\}$$

$C_o \subset Constructor$ is the set of 0-ary constructors, which includes integer and boolean constants.

For concreteness, assume that built-in maps have an association list based implementation*:

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

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

```
''a set = (''a, boolean) map
```

Built-in maps and sets therefore have a concrete representation in *H*.

3.3 Augmented regular tree expressions: The domain of the partial evaluator

Since the partial evaluator performs a collecting-semantics abstract interpretation of Core SCF-ML programs, its domain *AbsValue* is simply 2^H, the powerset of the above concrete domain. Much prior work in abstract interpretation and set-based program analysis [28, 20, 2, 17] has gone into formalisms for representing this domain. We choose as our representation a version of the Regular Tree Expression (RTE) representation of Aiken and Murphy [3]. The syntax and semantics of our version of RTEs, simply called "abstract values" below, is in table 1. On the left of the figure is the syntax of abstract values *v*. These may include variables α. On the right is the set, written $\Psi(v,\sigma)$, of concrete values represented by *v* under a substitution σ from variables to abstract values. We say that *t conforms to v under* σ iff $t \in \Psi(v,\sigma)$. If abstract value *v* has no free occurrences of variables α, we say that *v* is *closed*. For closed *v*, we

* Strictly speaking, untagged tuples are disallowed in core SCF-ML, but these can be simulated using tagged tuples.

$v \in AbsValue ::= v' \# id$ $id \in Int$ $v' ::=$	*Set represented by RTE:* $\Psi(v',\sigma) \in 2^H$
0	$\{\}$
1	H
$c(v_1,...,v_n)$	$\{c(t_1,....,t_m) \mid t_i \in \Psi(v_i,\sigma)\}$
$v_1 \mid ... \mid v_n$	$\Psi(v_1,\sigma) \cup ... \cup \Psi(v_n,\sigma)$
fix $\alpha.\ v$	*least fixpoint of* $T = \Psi(v, \sigma[\alpha \rightarrow T])$
α	$\sigma[\alpha]$
map (*must*$[(u_1,u_1')...(u_n,u_n')]$, *may*$[(v_1,v_1')...(v_1,v_m')])$	$\{[(t_1, t_1') ... (t_n, t_n')] \in map \mid$ $(\forall (s_i = \Psi(u_i,\sigma), s_i' = \Psi(u_i',\sigma)).$ $\exists (t_j,t_j').\ \{t_j\} = s_i \wedge t_j' \in s_i')$ $\wedge$ $(\forall (t_j,t_j')\ s.t.\ t_j \notin \Psi(u_1 \mid ... \mid u_n,\sigma).$ $\exists (s_i = \Psi(v_i,\sigma), si' = \Psi(v_i',\sigma)).$ $t_j \in s_i \wedge t_j' \in s_i')\}$

Table 1: Augmented regular tree expressions (abstract values): Syntax and semantics

say that *t conforms to v* iff $t \in \Psi(v,[])$, where [] is the empty substitution.

Our abstract values do not provide for some of the conventional variants of RTEs such as intersection and negation RTEs. On the other hand, they have two unconventional features.

3.3.1 Identity tags

First, each abstract value is tagged with an integer identity tag. The intention is that the partial evaluator maintain the following *identity tag invariant*: if, during abstract execution, the abstract values corresponding to two expressions have the same tag at some program point, then on any concrete execution of that program point along the paths abstractly executed thus far , the two variables are guaranteed to have the same concrete value. To understand this better, consider the function:

fun foo x = (*x* = *x*)

It is clear by inspection of this function that it can only ever return the value *true*. However, if *foo* is abstractly executed with *x* bound to abstract value **1** (i.e., all possible concrete values) in the absence of identity tags, the equality test will evaluate to the abstract test **1** = **1**. This test will be interpreted as $\{t = t' \mid t \in \mathbf{1} \wedge t' \in \mathbf{1}\}$, which evaluates to {*true, false*}. In the presence of identity tags, the abstract test will have the form **1***#i* = **1***#i* for some tag *i*. The identity tag invariant then allows the abstract equality operation to evaluate to the singleton {*true*} (i.e the abstract value *true#j* where *j* is a fresh tag).

Accurate abstract execution of equality is particularly important for PE of optimization programs, because fixpoint loops guarded by equality tests are common in these programs. Inaccurate equality tests cause the partial evaluator to loop excessively on the fixpoint loops, and to use widening strategies that finally terminate the looping, but lose information.

3.3.2 Abstract maps

The second distinctive feature of our abstract values is the special representation for sets of maps. 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* under some substitution σ must map the single member of $\Psi(v,\sigma)$ to a member of $\Psi(v',\sigma)$. 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 $\Psi(v,\sigma)$ and *t'* is in $\Psi(v',\sigma)$. 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.

As an example, consider abstractly executing the following function:

foo m = let m' = map_insert (*m*,*1*,*7*) *in map_find* (*m'*,*1*) *end*

Suppose, during abstract execution, the incoming map *m* may be one of the concrete maps *t*=[(*1*,*2*)] or *t'*=[(*3*,*2*)]. A possible abstract value for *m* is *map*(*must* [], *may* [(*1*|*3*,*2*)]). The *map_insert* operation could result in the abstract value *map*(*must* [(*1*,*7*)], *may* [(*1*|*3*,*2*)]). The conforming maps now are [(*1*,*7*)] and [(*1*,*7*),(*3*,*2*)] which are as expected given *t* and *t'* above (the occurrence of the key *1* in the *must* list shadows that in the *may* list). The shadowing semantics allows SCF to return the singleton abstract value *SOME*(*7*) as the result of the final *map_find* primitive operations.

Without special support for maps, we would have to partially evaluate user-defined function definitions for map accessors with respect to the RTE(s) representing the incoming map(s). Faced with this task, all partial evaluators and abstract interpreters [2, 29, 20, 28] of which we are aware produce inaccurate results (at best *NONE*|*SOME*(*7*)) on the above example. In partially evaluating optimization programs, it is particularly important to perform abstract operations on maps and sets as accurately as possible, since these data structures are often used to represent the abstract store and are therefore threaded throughout the optimization. Inaccuracies are therefore propagated globally, and tend to compound catastrophically.

The details we have presented in this section on our special representation for abstract maps are meant not so much to specify "the right way to do abstract maps" as to present a design that seems sufficient (where the more conventional approach was not) for the task of partially evaluating optimization programs.

3.3.3 Internal representation and implementation

We have discussed the semantics desired of our abstract value representation, but not described how to implement operations such as (widening) meets, equality tests and abstract versions of the operations of figure 9. Aiken and Murphy report exponential-time lower bounds for many of these operations over RTEs [4]. They recommend an internal representation of RTEs called a *leaf linear system*, and discuss heuristics that are fairly effective in keeping overheads low in practice for their application.

In SCF, we do not use an internal representation of abstract values different from that implied by the definition of table 1. We are willing to sacrifice some missed optimization opportunities in order to have a tractable implementation. We adopt simple canonical forms where easy, such as constructing an arbitrary total ordering over abstract values and using this ordering to represent the elements of a set of abstract values and the bindings in a map in a fixed order. We then use linear structural traversals to implement equality, ordering, and greatest lower bound over abstract values. We use structural traversals even for recursive abstract values, and accept the possible (conservative) loss of accuracy since in our domain we do not expect to encounter different recursive descriptions of the same abstract value. For widening meets, we use heuristics similar to those suggested by Aiken and Murphy [3].

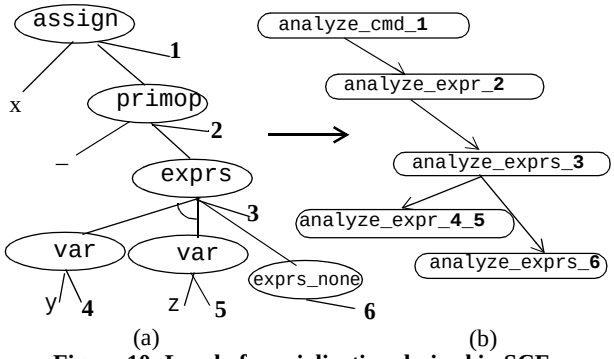

Figure 10: Level of specialization desired in SCF

3.4 Context-sensitivity strategy

To enable precise abstract execution during PE, the stager specializes aggressively the functions of the Core SCF-ML optimization program. The primary goal of this specialization is to replicate and then specialize the logical "flow functions" (and "transformation functions") in the program for each node in the input program, in effect unrolling the optimization program over the input program. Figure 10 shows how the specialization works.

On the left (figure 10(a)) is the abstract value
assign(*x*, *primop*(-, *exprs*(*var*(*y*,*4*)|*var*(*z*,*5*), *exprs_none*(*6*), *3*),*2*),*1*)
This abstract value represents the set of concrete commands
{*assign*(*x*, *primop*(-, *exprs*(*var*(*z*,*5*), *exprs_none*(*6*), *3*),*2*),*1*),
assign(*x*, *primop*(-, *exprs*(*var*(*y*,*4*), *exprs_none*(*6*), *3*),*2*),*1*)}.
The integers 1 through 6 in the abstract value (highlighted in bold in figure 10(a)) constitute the label field of the commands and their subexpressions. On the right (figure 10(b)) is the corresponding specialized function call graph we expect SCF to generate when partially evaluating the dead-assignment analysis from figure 3 with respect to this abstract command.

Whenever we partially evaluate a call to analysis function *f* where one of the formals[*] (called a *finite argument*) is bound to an abstract sub-tree *v* of the original abstract function that was input to the analysis, we produce and analyze a specialization *f*_|*v*| of *f* (|*v*| is a unique textual representation for *v*; in figure 10, as a mnemonic aid, we have chosen |*v*| to be the label of the phrase that *v* represents). The pair (*f*,*v*) is called a *specialization key.* If a function *f'* called from a specialization *f*_|*v*| does not have its own finite argument (e.g., the helper function `meet` in figure 5), it is partially evaluated with key (*f'*,*v*), i.e., helper functions are replicated for every node.

To ensure termination, we need to ensure that the number of abstract sub-trees generated as values of finite arguments during PE is finite. In SCF, specializable arguments are discovered by a simple *finiteness analysis*. To begin the analysis, the argument of the entry function of the optimization is assumed to be finite (in the sense that abstract execution will see only a finite number, in this case one, of abstract values bound to this formal). The finiteness analysis proceeds by deducing that any value that is the result of a sequence of projection operations on a finite value is itself finite.

The rules used by this finiteness analysis are a simple variant of those proposed previously in the context of offline partial evaluation [18]. The main complication in adapting the analysis to the online case is in showing that the rules are sound in the case of online partial evaluation. In particular, we need to show that for any *abstract* value *v*, there exists a finite set $V=\{v_1,...,v_n\}$ of abstract values such that if we perform any sequence $p_1, p_2,...,p_m$ of *abstract* projection operations on *v*, the result *v'*, if defined, is guaranteed to be in *V*. The corresponding finiteness argument is easy in the case of offline partial evaluation, since the projections in that case are performed on *concrete* values, and a simple montonicity argument about the size of concrete values resulting from projections is sufficient to establish finiteness.

The level of context sensitivity provided by specializing on finite arguments is not quite enough for accurate PE of optimization programs. For iterative analysis of some input program nodes, we often wish to specialize each call of the recursive function separately, even though each recursive call has the same input program node argument and hence has the same specialization key. For example, the `fix` function in the dead-assignment elimination example of figure 3 iteratively analyzes `while_do` nodes, returning the final fixpoint solution to its dataflow equation. If we merged each of these recursive calls, then our analysis would meet each of the intermediate stages of the iteration into a combined summary of the whole loop, causing a loss of precision.

Our solution is to maintain *call chains of specialization keys* rather than individual specialization keys as the specialization context. Every time abstract execution encounters an abstract call with specialization key (*f*,*v*), it appends the caller's chain of specialization keys to the key to obtain the callee's complete specialization context. We ensure termination by requiring that at most *k* instances (where *k* is fixed for each program being partially evaluated) of a specialization key occur in a specialization context: when we find ourselves building a chain with *k+1* instances of some key, we use the shortest suffix of the chain with *k* instances of the key instead as the specialization context. Since we expect iterative analyses to reach fixed-point in only a few recursive calls per level of loop nesting, we typically use a small value (2 or 3) for *k*.

The addition of call chains makes the worst-case running time (and size) of the analysis exponential in the size of the result of performing finite-argument-based specialization on an optimization program (or equivalently, in the size of the incoming abstract 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. With a tree-shaped callgraph, call-chain-based specialization costs drop from exponential to linear. Also, the size of both (intraprocedural) optimizations and their inputs tend not to be very large. Our experimental results so far indicate that the exponential blow up does not affect us in most cases.

4 Dead-assignment elimination

As discussed in section 2.2.2, the residualized code after PE contains many dead assignments, and SCF contains an optimization designed to delete these assignments. Figure 11 specifies key details of dead-assignment elimination (abbreviated as DAE below) in SCF. The main interesting features of the optimization are the machinery it provides to track liveness across data structures (maps in particular) and its use of the collecting semantics of a program (see figure 8) to identify dead operations in it.

The analysis is structured as a backward abstract interpretation over the domain of *liveness patterns* (written *LP*'s below), defined at the top of figure 11. Intuitively, a liveness pattern is a generalization of the lattice values *live* and *dead* associated with scalar variables in traditional dead-assignment elimination. In the traditional setting, if a variable is *live* at a program point, we conclude that the variable may be read downstream of the program point. However, if the variable is not a scalar (i.e., it has more than one field), we are often interested in *which* fields of the possible values contained in the variable are read downstream, so that stores to these fields may be eliminated if possible. Liveness patterns allow us to express the liveness of sub-fields of values produced by expressions as follows. A LP of **1** for some expression at some program point indicates that

[*] In general, more than one formal may be specializable, and the specialization key used in that case is the tuple of corresponding abstract values.

```
l   ∈ LivenessPattern (LP) =
       1| 0 | c(l1,...ln) | {l1|...|ln} | [l1,l2]
cs  ∈ CollectingSemanticsMap= (Label,AbsValue) map
lm  ∈ LPMap    = (Variable, LivenessPattern) map
DAE_e : Expr → LivenessPattern →
        CollectingSemanticsMap → Expr * LPMap
DAE_e e 0 cs = (| c_dead |, []) (* dead expression *)
DAE_e |x| l cs = (|x|, [x → l] )
DAE_e  |c^-k e| l cs =
  (* l is k'th place of arity(c)-sized tuple *)
  let l'= c(0,0,...l,...0,0)
      (e', lm) = DAE_e e l' cs
  in (|c^-k e'|, lm) end
DAE_e |if e1 e2 e3 | l cs =
  let (e1, lm1)       = DAE_e e1 1 cs
      ((e2, lm2),(e3, lm3)) =
                    (DAE_e e2 l cs, DAE_e e3 l cs )
  in (|if e1 e2 e3|, meet lm1 (meet lm2 lm3)) end
DAE_e |let x = e1 in e2| l cs =
  let (e2, lm2) = DAE_e e2 l cs in
    if e2 = |c_dead| then (|c_dead|, []) else
    let v = find lm2 x in
      if v = NONE orelse v = SOME 0 then (e2, lm2)
      else
        let (e1, lm1) = DAE_e e1 cs in
          (|let x = e1 in e2|,
           meet lm1 (delete lm2 x))
  end end end
DAE_e |map_find e_m e_k| l cs =
  let l_k      = toLivenessPattern
                   (find cs (labelOf e_k))
      (e_m, lm_m) = DAE_e e_m [l_k , l] cs
      (e_k, lm_k) = DAE_e e_k l_k  cs
  in (|map_find e_m e_k|, meet lm_m lm_k) end
DAE_e |map_insert e_m e_k e_v| [l,l'] cs =
  let v_k        = find cs (labelOf e_k)
      (e_m, lm_m) = DAE_e e_m [l , l'] cs
  in
    if mustBeDisjoint l v_k  then (e_m, lm_m) else
    let (e_k, lm_k) = DAE_e e_k  l cs
        (e_v, lm_v) = DAE_e e_v  l' cs
    in (|map_insert e_m e_k e_v|,
        meet lm_m (meet lm_v lm_k))
  end end
DAE_e ...
(* Handle other kinds of expressions & LPs*)
```

Figure 11: Dead-assignment elimination (DAE)

any field of a value it evaluates to may be read downstream. A LP of **0** indicates that no field of a value it evaluates to is read downstream. A LP of *c*(*l1*,...*ln*) requires the value to have the form *c*(*v1*,...,*vn*), where each value *vi* satisfies LP *li*. A LP {*l1*|...|*ln*} requires the value satisfies at least one of LP's *li*. Finally, [*l1*,*l2*] requires the value to be a map such that every key of the map satisfies *l1* and every range value satisfies *l2*.

The interprocedural part of DAE (not shown) associates every function being analyzed with one LP representing its return value, and one representing its argument. The return LP of a function captures the requirements placed on the return value by callers of the function. The argument LP captures requirements placed by the function body on the argument values passed to the function. The analysis begins by assuming that the optimization program as a whole has return LP **1**, 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 to **1** the LP's associated with a function after it is processed a fixed number of times.

The intraprocedural part of DAE (figure 11) associates a LP with each subexpression of the function body. The LP specifies which

```
meet: LPMap → LPMap → LPMap
meet lm lm' =
  (* Take the union of maps lm and lm' . If a
  variable x maps to LP's l and l' in lm and lm'
  respectively, bind x to meetLP l l' (see below)
  in the resulting map *)
meetLP: LP → LP → LP
meetLP l l' =
  (* Let V and V' be the sets of all concrete
  values matching l and l' respectively. Return a
  LP that matches all of V ∪ V' *)
mustBeDisjoint: LP → AbsValue → Boolean
mustBeDisjoint l V =
  (*false if a concrete value v ∈ V may match l*)
toLivenessPattern: AbsValue option → LP
toLivenessPattern (SOME V) =
  (* return a LP that matches every concrete value
  v ∈ V *)
```

Figure 12: Helper functions for DAE

parts of the values produced by the subexpression may be used downstream. To start things off, the LP of the function body expression is set to the return LP of the function. For each subexpression *e*, the backward pass returns a pair containing:

1. the expression resulting from pruning out dead subexpressions of *e*, and
2. a map from variables in *e* to LPs.

Figure 13 is an example of how DAE works. Consider (figure 13(a)) an invocation of DAE_e on the body of function *foo* (in which $e_{[i]}$ indicates that subexpression *e* is labelled with integer *i*), with LP *l* = **1** and collecting-semantics map *cs*. Say *tr* is a 3-ary constructor.

The value **1** of *l* indicates that any part of any concrete value produced by the body of *foo* may be required downstream. We process the let-bindings in *foo* from the inside out. Since the result of *foo* is the result of the *map_find* operation on map *m'*, we deduce that all concrete values produced by *map_find* may be required downstream as well. We conclude therefore that the *map_find* operation is not dead and cannot be pruned away, and also that map *m'* may have any concrete value in its range (since downstream uses do not constrain the range). Further, since we are performing a lookup on *m'* with key $y_{[5]}$, and map *cs* indicates that at this program point $y_{[5]}$ may only have values 44 or 46 on any concrete execution, we conclude that the domain of *m'* need only contain 44 or 46. Putting requirements on the domain and range together, we get a LP of [{44 | 46}, **1**] for *m'*. We conservatively set the LP of *y* to its value {44 | 46} in *cs*, since in a correct program, *y* cannot be required to produce values other than those it can evaluate to in any concrete execution.

fun foo a =
let m = tr^-1 a in
let x = tr^-2 a in
let y = tr^-3 a in
let m' = map_insert m x[2] 10 in
map_find m' y[5]
end end end end

l = **1**

cs = [*2* → {*22*|*32*},
5 → {*44* | *46*}, ...]

(a)

fun foo a =
let m = tr^-1 a in
let y = tr^-3 a in
let m' = m in
map_find m' y
end end end

(b)

Figure 13: DAE example

Config. No.	Input Function	Description of Input Function	Abstract Values to Which Staged Arguments of *f* are Bound
1	`mul_add`	`mul_add` from figure 2: 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 dotproduct of two vectors *v1* and *v2* of size *s*; *v2* and *s* are fixed at run time	stage1: *v2* = *some 1-D array*, s = 'Int stage2: *v2* = (0, 1, 7), *s* = 3
5	`dotproduct`		stage1: *v2* = *some 1-D array*, 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* = *some 2-D array* stage2: *c* = [[1, 0, 1], [0, 1, 0], [1, 0, 1]]
7	`doconvol_1d`	1-D version of above	stage1: *c* = *some 2-D array* 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 *b* = 1 ...
9	`main_loop_f`	As in 8, with routine for fetching cache entries inlined	
10	`main_loop_f_u`	As in 9, with routine for updating cache entries inlined	
11	`main_loop_f_u_f`	As in 10, with routine for finding cache entries inlined	

Table 2: Description of benchmark inputs

We next process the innermost let binding, *let m' = ...* . Given that *m'* is only required to contain keys 44 or 46, and that (looking up the value corresponding to label 2 in *cs*) the insert operation only ever inserts (via key $x_{[2]}$) keys 22 or 32, we conclude that the inserted binding is dead and so the insert is unnecessary. We replace the whole insert expression |*map_insert m* $x_{[2]}$| with its subexpression |*m*| (which just computes the map the being inserted into). This simplification results in the let binding *let m' = m in ...*, which simply copies *m* into *m'*, and we correspondingly set the LP of *m* to that of *m'*, i.e., to [{44 | 46}, **1**].

We now process the binding *let y* = tr^{-3} *a*. Since we have found above that *y* has LP {44 | 46} we conclude that tr^{-3} *a* must have the same pattern, i.e., it is only required to have concrete values 44 or 46, and that consequently the third field of *a* is also only required to have these values, i.e., it has LP {44 | 46}. Similarly, from the binding *m* = tr^{-1} *a* of *m*, we conclude that the first field of *a* has LP [{44 | 46}, **1**]. Finally, given that *x* did not have a LP associated with it above (because its use in the *map_insert* was pruned away), the second field of *a* remains unconstrained (i.e., has liveness value **0**) and the dead binding *let x* = tr^{-2} *a in...* is eliminated. The resulting liveness pattern for *a* is *tr*([{44 | 46}, **1**], **0**, {44 | 46}). The resulting pruned body of *foo* is in figure 13(b).

The interaction between collecting semantics and the dead-assignment elimination to accurately model partial deadness of maps (and thereby set) data structures is critical to effective removal of dead code. Even with very aggressive specialization, if we treat map and set accessors as user-defined functions, the resulting analysis would not be accurate enough for our purposes.

5 Evaluation

We have implemented a prototype of SCF in Standard ML [24]. We provide an SCF-ML front-end to allow specification of optimization programs. 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.

As discussed in the introduction, the main goal of SCF is to enable the *easy* construction of *effective* staged compiler pipelines. We have so far 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 [15], using SCF to stage automatically unstaged versions of these optimizations has been far *easier* in design, implementation and debugging. The remaining issue, which can be evaluated by measurement, is whether automatic staging as in SCF is *effective*: how do automatically staged optimization pipelines compare with unstaged pipelines and with hand-staged ones? In this section, we examine this issue.

In all experiments below, we stage an optimization pipeline consisting of the constant propagation, copy propagation and dead-assignment elimination passes, i.e., the pipeline of figure 4. This pipeline is used to optimize the various input functions over two stages, as specified in table 2. In the second stage, we provide fully

Config. No.	Compiler Speedup	Reduction in Instructions Executed	Input function size		Increase in Compiler Size (final stage pipeline/ unstaged pipeline)	Compiled Code Speedup (speedup due to hand staging)
1	1.9×	2.1×	4 loc	148 nodes	1.4×	1.2×
2	2.9	2.2	4	148	1.4	1.2
3	5.5	7.9	4	148	1.3	1.2
4	2.9	2.6	8	198	1.6	2.7
5	2.5	2.9	8	198	3.6	2.7 (5.7)
6	1.1	1.2	102	2880	47.7	1.7 (3.1)
7	4.7	4.2	50	1097	7.9	1.9
8	12.2	9.8	39	1084	6.7	1.0
9	4.7	6.0	150	3960	19.2	1.1
10	4.8	8.5	292	7656	36.3	1.1
11	4.7	9.0	322	8396	38.5	1.3 (1.7)

Table 3: Benchmark results

concrete values of pipeline inputs, and concretely execute the pipeline on these inputs to produce a concrete output, i.e., the optimized version of the input function specified as the pipeline input. Since we do not currently have a code-generation pass in our compiler pipeline, we pretty-print the optimized input function as a C function, and use a conventional C compiler (gcc) to compile the function and link it into its calling C program to produce an optimized C program.

Given this framework, we focus on answering the following two questions:

1. How fast are the staged optimizers? In particular, how much faster are the staged optimizers produced by SCF, when concretely executed, than the original (i.e., completely unstaged) pipeline?*
2. How effective are the staged optimizers? In particular, how much faster is the code optimized with the staged optimizers than without the optimization?

5.1 Speedup of run-time stage of compiler

To answer question 1 above, we measured the overhead of concrete execution of the two versions of the pipeline. We compiled both versions of the pipeline into machine code and executed them directly on the hardware, and measured overhead in microseconds (all times are minimum user times on a lightly loaded 350MHz Pentium-based machine with 256MB RAM and 8kB L1 instruction and data caches and a 512kB L2 cache). In order to get a less machine-dependent picture, we also executed the pipeline on an instrumented Core SCF-ML interpreter, and measured the number of instructions executed†. Column 2 of table 3 shows the ratio of the time taken to execute unstaged pipelines to that to execute staged ones. Column 3 shows the ratio of abstract instructions executed by the unstaged optimizer to that executed by the staged version. A few key points are worth noting:

- 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 producing fast staged compilers.
- 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. Column 5, which presents 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.
- 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 needs to perform more checks than one that does not, resulting in extra compile-time overhead.
- 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 3.4). This is evident from comparing columns 4 and 5 of table 3. Each entry in column 4 of the table represents the total size of stager input programs in the penultimate stage. The one anomaly is configuration 6. The convolution routine in configuration 6

* Ideally, we would also like to compare the overhead with that of a hand-staged pipeline. Unfortunately, since the latter overheads are reported in terms of late-stage compile cycles per machine instruction generated, and we do not yet stage a code generation phase, we cannot provide this number.

† We only show the decrease in map-manipulation instructions here because they account for most (over 90%) of the compiler execution time

contains a four-way nested loop. Specializing the recursive calls that process this loop results in code bloat.

5.2 Speedup of compiled program

To answer the question 2 above, we executed the optimized and unoptimized versions of the C program containing the input function and compared the times spent in the input function in the two cases. In the case of the `mul_add` and `dotproduct` input programs, because time spent in the input functions was below the resolution of the timer, we invoked the input function a large number of times and averaged over the invocations to get the overhead for a single call. Column 6 of table 3 presents, for each configuration, the ratio of time spent in the unoptimized version of the input function for that configuration to that spent in the optimized version. Where available, the corresponding ratio for hand-staged systems is included in bold parentheses. Two 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 [15, 8] 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 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.
- The speedup due to the hand-staged pipeline is significantly greater than that achieved by the SCF pipelines. 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 do not know the precise reason yet.

6 Related work

This work is motivated by our previous work on DyC [15], which included a hand-staged optimization pipeline consisting of partial evaluation, zero/copy propagation, and dead-assignment elimination. The system demonstrated that staging the latter optimizations can yield substantial speed up at low late-stage overhead in realistic programs. However, the technique used for staging was impractical for staging pipelines with many optimizations. The stager for each optimization communicated the effects of the optimization to downstream stagers using the standard technique of annotating the outgoing program with "action annotations" [7], which are specific to each optimization. Each staged optimization must therefore know the semantics of the action annotations used by all predecessor optimization stages, and how they may interact. By replacing optimization-specific action annotations with a uniform regular-tree-expression-based description of sets of programs, our current work enables staging of arbitrary optimizations to be implemented mechanically in a generic way, and enables whole pipelines of arbitrary optimizations to be staged mechanically.

Work on staged dynamic partial evaluation [14, 21, 8, 25] has focused on adapting offline partial evaluation to support a run-time stage. These systems have developed a suite of techniques to increase late-stage performance while reducing late-stage overhead. In particular, the optimized program produced by the late stage is in executable machine code format (instead of the usual source format), the late stage is restricted to a single non-iterative pass (trading optimization quality for speed), late optimization of parts of the input program may be performed on demand or conditionally, and the stager may be parameterized by policy and program information. Our staging framework can potentially be extended to support all these techniques. Sperber and Thiemann have shown how to compose a code generator with a specializer in a staged way, by viewing them both as catamorphisms [30]. Though the generic optimizations we seek to compose and stage are not catamorphisms, it is possible to automatically create "pessimistic" versions of many optimizations that are catamorphisms. Marlet *et al.* discuss using multi-level staging to speed up late-stage partial evaluation [23], but they assume that the incremental information available at each stage is the concrete value of some argument of the function being partially evaluated, rather than possibly just a more refined set of possible values of the argument, as our system supports.

Accurate online partial evaluation is key to our system. A vast literature exists on this topic. Ruf provides a good overview [29]. Handling recursive functions accurately but finitely is widely identified as the key problem, one with provably no general solution. Common solutions are the use of finiteness annotations [33] to guarantee that arguments have a finite number of abstract values, and finiteness analyses (similar to ours) [18, 33] to automatically detect such arguments. Our predefined map and set datatypes may be viewed as a variation on the former technique (the optimization writer uses a predefined library of functions whose properties are known to the partial evaluator).

One issue that Ruf does not address is the definition of accurate domains for PE. This issue is discussed extensively in the program analysis literature, however [28, 20, 2, 17]. Formalisms such as tree grammars, graph grammars, set constraints and tree expressions have been studied in this context, and in many cases the domains used by the analyses have been more expressive than ours. However, none of these analyses have been context sensitive, an aspect we have found crucial to our application.

Dead-assignment elimination through partially-dead data structures has been studied by Liu and Stoller and by Reps [22, 27]. Both these works support recursive liveness patterns, a capability missing in SCF. We have deferred adding recursive patterns until we find a need for them in our application. We have found so far that for our purposes, custom description of particular datatypes (e.g., maps) is much more effective than providing more expressive generic liveness patterns. Neither of the other techniques allows the dead-assignment analysis to consult the results of partial evaluation, a technique we found critical to obtaining good pruning of dead code.

7 Summary and future work

We have presented a framework for constructing staged optimizations that allows the optimizations to be written independently as unstaged dataflow-based optimizations, composed into pipelines of arbitrary length in arbitrary order, and specialized automatically over compilation stages. The key to modular composability of optimizations in the face of staging is the use of a uniform, but expressive, regular-tree-expression-based representation to communicate information between optimizations within a stage. We achieve automatic staging by applying an extended form of online partial evaluation followed by post-specialization dead-assignment elimination. We have implemented and evaluated a prototype of the framework, which demonstrated that our techniques are capable of producing fast, effective late-stage compilers.

This work establishes a baseline staged compilation system that produces exactly the same optimized code as one where the optimizations were run solely during the final stage, with the specialized compilers targeted for high speed while maintaining the behavior of the original unspecialized optimizations. However, these quality and speed constraints can lead to overly large and expensive late-stage optimizers. In addition to investigating staging of additional optimizations and developing front-ends for new input

languages, our future work will study alternative trade-offs among optimized code quality, late-stage compiler size, and late-stage compiler speed. For example, we plan to study principled ways to sacrifice some optimization opportunities in order to make the late-stage compilers require only linear passes. We also plan to investigate having the stager produce specialized program representations instead of specialized compilers; the former can be much more compact than the latter. Finally, we wish to study having the late-stage optimizations accept and return only the parts of the program being optimized that weren't known to the earlier stages, to minimize their data traversal and construction costs, and to allow fusion of adjacent optimizations to eliminate intermediate data structures [32].

Acknowledgments

This work has been supported by ONR contract N00014-96-1-0402, ARPA contract N00014-94-1-1136, NSF Grant CCR-9975057, and NSF Young Investigator Award CCR-9457767. We thank Brian Grant and Markus Mock for joint work and discussions about dynamic compilation and staging, and Josh Redstone for help with formatting the paper. We thank the anonymous referees for useful feedback, especially that on improving the presentation.

References

[1] A. Aho, R.Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] A. Aiken and B. R. Murphy: Static type inference in a dynamically typed language. In *Symposium on Principles of Programming Languages*, pages 279–290, Jan. 1991.

[3] A. Aiken and B. Murphy. Implementing Regular Tree Expressions. In J. Hughes, editor, *5th ACM Conference on Functional Programming Languages and Computer Architecture*, number 523, Cambridge, MA, USA, August 26-30, 1991. Springer.

[4] 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.

[5] V. Bala and E. Duesterwald. Dynamo: A transparent runtime optimization system. In *Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

[6] C. Chambers, J. Dean, and D. Grove. Whole-program optimization of object-oriented languages. Technical Report TR-96-06-02, Department of Computer Science and Engineering. University of Washington, June 1996.

[7] C. Consel and O. Danvy. From interpreting to compiling binding times. In *3rd European Symposium on Programming*, LNCS 432, pages 88–105. Springer-Verlag, May 1990.

[8] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Symposium on Principles of Programming Languages*, pages 145–156, Jan. 1996.

[9] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *OOPSLA'96 Conference Proceedings*, pages 83–100, Oct. 1996.

[10] A. Diwan, E. Moss, and K. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *OOPSLA'96 Conference Proceedings*, Oct. 1996.

[11] M. Fernandez. Simple and effective link-time optimization of modula-3 programs. In *Conference on Programming Language Design and Implementation*, pages 103–115, June 1995.

[12] 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.

[13] R. Glück and J. Jorgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, 1997.

[14] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Annotation-directed run-time specialization in C. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 163–178, June 1997.

[15] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An evaluation of staged, run-time optimizations in DyC. In *Conference on Programming Language Design and Implementation*, pages 293–304, May 1999.

[16] M. Hall, J. Mellor-Crummey, A. Carle, and R. Rodriguez. Fiat: A framework for interprocedural analysis and transformation. In *The Sixth Annual Workshop on Parallel Languages and Compilers*, Aug. 1993.

[17] N. Heintze. Set-based analysis of ML programs. In *ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.

[18] C. Holst. Finiteness analysis. In *Functional Programming Languages and Computer Architecture*, LNCS 523, pages 473–495. Springer-Verlag, Aug. 1991.

[19] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Conference on Programming Language Design and Implementation*, pages 326–336, June 1994.

[20] N. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Symposium on Principles of Programming Languages*, pages 244–256, Jan. 1979.

[21] 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.

[22] Y. A. Liu and S. D. Stoller. Eliminating dead code on recursive data. In *Static Analysis Symposium*, pages 211–231, 1999.

[23] R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In *Conference on Programming Language Design and Implementation*, pages 281–292, May 1999.

[24] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[25] 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.

[26] T. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Symposium on Principles of Programming Languages*, pages 322–332, Jan. 1995.

[27] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, pages 409–429, Schloss Dagstuhl, Wadern, Germany, 12–16 1996. Springer-Verlag, New York, NY.

[28] J. C. Reynolds. Automatic computation of data set definitions. In A. J. H. Morrell, editor, *Information Processing 68*, volume 1, pages 456–461, Amsterdam, 1969. North-Holland.

[29] E. Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, February 1993. Technical report CSL-TR-93-563.

[30] M. Sperber and P. Thiemann. Two for the price of one: Composing partial evaluation and compilation. In *Conference on Programming Language Design and Implementation*, pages 215–225, June 1997.

[31] F. Tip, C. Laffra, P. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. In *OOPSLA'99 Conference Proceedings*, pages 292–305, Oct. 1999.

[32] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[33] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In *Conference on Functional Programming Languages and Computer Architecture*, LNCS 523, pages 165–191. Springer-Verlag, 1991.